Model-Driven Software Development

Cross-Platform App Development and Further Applications of Domain-Specific Languages

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Wirtschaftswissenschaften
durch die Wirtschaftswissenschaftliche Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Christoph Rieger

Münster, 2019

Dekanin:	Prof. Dr. Theresia Theurl
Erstberichterstatter:	Prof. Dr. Herbert Kuchen
Zweitberichterstatter:	Prof. Dr. Dr. h.c. Dr. h.c. Jörg Becker
Datum der Disputation:	9. Mai 2019

Abstract

Smartphones have become ubiquitous devices and the ecosystem of mobile apps continues to thrive. This trend is amplified by the emergence of new mobile or wearable devices and further devices such as smart TVs have also become app-enabled. However, platform-specific functionality and user interface guidelines require repetitive implementations to reach a large number of users. Cross-platform approaches set out to reduce the development effort but are usually restricted to smartphones and tablets.

The dissertation overcomes these limitations by analysing the challenges of app-enabled devices, categorising device classes, and assessing the applicability of current cross-platform approaches. Because the model-driven paradigm is particularly suited to solve the issues of pluri-platform development, the MAML framework is proposed as graphical domain-specific language (DSL) to facilitate the creation of business apps for a diverse audience of technical users and domain experts. In addition, the design of DSLs is scrutinised to ease language development by considering the perspectives of modularisation, preprocessing, and cross-cutting concerns.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ANTLR	Another Tool For Language Recognition
API	Application Programming Interface
AR	Augmented Reality
AST	Abstract Syntax Tree
BIS	Business Information System
BPM	Business Process Management
BPMN	Business Process Model And Notation
CIM	Computer-independent Model
CRDT	Conflict-Free Replicated Data Types
CRF	Cameleon Reference Framework
CRUD	Create/Read/Update/Delete
CSS	Cascading Style Sheets
CTT	ConcurTaskTree
DSL	Domain-specific Language
EMF	Eclipse Modeling Framework
EPC	Event-driven Process Chain
FFI	Foreign Function Interface
FSS	Fish School Search
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GPL	General-purpose Language
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HPC	High-performance Computing
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IFML	Interaction Flow Modeling Language
IoT	Internet Of Things
IS	Information Systems
JS	JavaScript
LOC	Lines Of Code

MAML	Münster App Modeling Language
MDA	Model-Driven Architecture
MDSD	Model-driven Software Development
MRT	Media Richness Theory
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
NCLOC	Non-comment And Non-blank Lines Of Code
NRR	Number Of Related Requirements
OS	Operating System
OT	Operational Transformation
PAIS	Process-aware Information System
PIM	Platform-independent Model
PM	Platform Model
PSM	Platform-specific Model
PWA	Progressive Web App
QVTo	Query/View/Transformation Operational
SDK	Software Development Kit
SPA	Smart Personal Agent
SUS	System Usability Scale
TAL	Traceability Analysis Language
TICM	Traceability Information Configuration Model
TIM	Traceability Information Model
TIMM	Traceability Information Meta Model
UI	User Interface
UML	Unified Modeling Language
UX	User Experience
VR	Virtual Reality
WCP	Workflow Control Pattern
WFM	Workflow Management

Part I

Research Overview

1

Introduction

This chapter gives an overview of the dissertation. Section 1.1 presents the motivation for applying model-driven software development techniques in the context of mobile apps. Next, the main research questions are introduced in Section 1.2 before Section 1.3 describes the outline of this thesis.

1.1 Motivation

Since the advent of the iPhone in 2007 [App07], ubiquitous smartphones have affected communication, information retrieval, and work practices in everyday life. Within one decade, the average time spent with mobile devices has surpassed other digital media usage in the US [eMa17], and worldwide mobile device traffic on websites is almost even with desktop devices [Sta19b]. A significant share of this activity stems from mobile applications – *apps* – created by third-party developers and distributed via app stores. In Germany, mobile apps account for 89% of the total time spent with mobile devices [Sta17b]. The app ecosystem facilitated a trend towards task-oriented and interoperable software and has also gained economic importance. With a growing number of 205 billion app downloads on the major app stores in 2018 [Sta17a], the mobile app business is projected to generate $188.9 billion as global revenue by the year 2020 [Sta16].

Trends such as *Bring-your-own-device* [Tho12] demonstrate the importance of mobile devices not only for private use but also in enterprise contexts. Therefore, so-called *business apps* support business processes and provide value to company-internal collaborative work or customer-facing departments such as marketing or after-sales services [MEK15]. Although several mobile platforms such as Nokia's Symbian or Windows Phone disappeared or have fallen into insignificance, Android developed by Google and Apple's iOS have formed a duopoly of mobile operating systems which both need to be considered when developing mobile apps for a broad audience. This directly affects the cost of mobile development because both platforms vary with regard to programming languages, user interface design, and typical interaction patterns. In addition, the introduction of new languages such as Swift (for iOS apps) or Kotlin (for Android), and the platform fragmentation resulting from different versions of the same operating system installed on older devices (especially on Android [Dob12]) require continuous attention by developers.

Consequently, cross-platform approaches set out to reduce the development effort by writing code only once and executing it on different platforms through transformation or interpretation. Several approaches have been proposed, including mobile-optimised web pages (so-called web

apps), custom runtime environments, and code generation [El-+17]. Advantages and drawbacks of each approach have been studied in the past [UB16] and improved solutions have been proposed which will affect mobile app development practices in the future [Man18]. For example, *Accelerated Mobile Pages* [Goo19b] are designed to deliver content across devices with high performance using a subset of the Hypertext Markup Language (HTML), *Progressive Web Apps* [BMG18] to provide an app-like behaviour for web apps, or *Instant Apps* [Goo19f] to loosen the installation requirement of native apps.

Traditional cross-platform approaches are focused on smartphone and tablet devices. However, in recent years many new – often mobile or wearable – devices have emerged, including smartwatches and augmented/virtual reality headsets. In addition, further devices such as smart TVs and smart personal assistants (e.g., Amazon Alexa) offer extensible functionality through apps. model-driven software development (MDSD) represents a flexible cross-platform approach that is suitable to tackle the heterogeneity of user interfaces and user interactions. MDSD frameworks let users *model* software artefacts on a high level of abstraction and apply transformations to generate executable code or interpret models at runtime.

1.2 Problem Statement

The main area of research in this thesis combines the fields of model-driven software development with the domain of mobile computing using a design science research approach [Pef+07] to contribute theories and software artefacts on model-driven cross-platform app development.

Challenges creating apps for multiple platforms are manifold, both from a technical perspective and the resulting user experience. In particular, the landscape of devices and operating systems for which apps can be created has quickly evolved but terminology regarding "cross-platform" development often remains vague with respect to the devices and features which are supported by a specific framework or researched in academia. Taking the plethora of current and upcoming devices into account, the corresponding challenges of device heterogeneity, as well as near-native performance and appearance [Biø+19], lead to the definition of the first research question:

RQ1 How can cross-platform app development be extended beyond the common scope of smartphones and tablets and to what extent do the requirements differ from current cross-platform approaches?

Novel frameworks such as React Native [Fac19], NativeScript [Pro19], or Flutter [Goo19e] combine different cross-platform approaches and partially generate apps for performance reasons. MDSD fully embraces generative development and enhances it with domain-specific abstractions. In contrast to frameworks that narrowly focus on developers, model-driven apps may be co-developed by mixed teams of software engineers and people from business departments or citizen developers [LvV14] – potentially including future users. Prerequisites and opportunities in the domain of business apps are highlighted by the second research question:

RQ2 How can the user-centred and model-driven creation of business apps be facilitated for a diverse audience of technical users and domain experts?

Finally, the development environment itself needs to be taken into consideration in order to decrease the effort of creating domain-specific languages and surrounding tools such as code generators and editor components. Non-functional requirements or *external quality attributes*, e.g., maintainability, scalability, and usability promote an efficient development process [WB13]. In addition, cross-cutting concerns exist both in language design and the generated software such that an architectural alignment between different levels of abstraction can be investigated to manage the complexity of a model-driven framework. These aspects constitute the third research question:

RQ3 How can components of a model-driven approach be optimised with regard to non-functional considerations in order to create maintainable and usable development frameworks?

1.3 Outline

The motivation and research questions formulated in this chapter are complemented by the research design applied to answer these questions as presented in Chapter 2. Chapter 3 then presents the status quo of cross-platform app development and defines the concept of *app-enabled* devices. Subsequently, the landscape of devices is categorised using a taxonomy (Section 3.2) and challenges for cross-platform development across device classes are highlighted (Section 3.3). Also, an extensive criteria catalogue is compiled to holistically assess cross-platform development frameworks according to company- or project-specific needs (Section 3.4).

Chapter 4 deals with the implementation of such a framework using model-driven techniques. After summarising relevant fundamentals in Section 4.1, Section 4.2 presents the textual MD² framework for creating cross-platform business apps, and Section 4.3 describes and evaluates the newly developed graphical domain-specific language (DSL) called MAML. Moreover, Section 4.4 expands on the topic of novel app-enabled device classes and proposes a model-driven approach to jointly model apps for heterogeneous devices together with a prototypical implementation.

In Chapter 5, non-functional aspects of DSL design are investigated in several application domains, including modularisation of textual DSLs (Section 5.2), preprocessing to generate high-performance programs (Section 5.3), and traceability of artefacts in the software development process (Section 5.4). Chapter 6 concludes the research overview with a synopsis and discusses limitations as well as opportunities for future research. Finally, Part II presents 18 publications (P1-P18) which expand on the topics presented in the research overview.

2

Research Design

This section gives an overview of the research design chosen to answer the research questions presented in Section 1.2. First, the design science research methodology which is applied to design and evaluate software artefacts in the field of model-driven software development is presented in Section 2.1. Because domain-specific languages as part of the MDSD approach focus on user-centred notations, an overview about evaluating software usability is provided in Section 2.2.

2.1 Design Science Research

At the intersection of people, organisations, and technology, the field of Information Systems (IS) deals with the implementation and usage of IT artefacts according to tasks, goals, and (business) needs of people within an organisational context of strategies, culture, and processes [SMB95]. The technological facets comprise both existing and planned systems with regard to, e.g., infrastructure, applications, architectures, and development capabilities. In contrast to approaches based on behavioural science, *design science research* considers the creation and evaluation of IT artefacts while focusing on the utility of the technological solution to the identified organisational problem [Hev+04]. March and Smith distinguish between four types of IT artefacts [MS95]:

- a *construct* defines a specialised term or vocabulary in a particular domain,
- a *model* is a purpose-oriented representation of relationships among constructs and, therefore, created on a suitable level of abstraction to describe situations or artefacts[1], and
- a *method* denotes an algorithm or process and can be seen as a tool to perform tasks on models, e.g., to analyse or transform them
- an *instantiation* represents a concrete implementation such as an information system in its environment. It may apply constructs, models, and methods but also precede their formalisation when implicitly applied artefacts are later derived from the successful instantiation.

This thesis contributes several IT artefacts, for example, constructs to clarify notions in the domain of cross-platform development (cf. Section 3.3), DSLs that provide a formal description of models, methods such as transformations for code generation, as well as prototypical instantiations demonstrating their feasibility.

[1] This terminology also applies to users of DSLs who create *models* in this sense of purpose-driven representations of abstract domain concepts which are later transformed into concrete implementations.

In contrast to the routine creation of systems, design science research uses rigorously applied methods to address "important unsolved problems in unique or innovative ways or solved problems in more effective or efficient ways" [Hev+04]. Both purposes can be found in this dissertation: On the one hand, recent technological developments such as wearable devices require gathering knowledge and best practices on how to leverage appropriate techniques for heterogeneous devices, e.g., using model-driven approaches (cf. Chapter 4). On the other hand, applying such techniques to established domains allows for improvements such as reduced development effort (cf. Section 3.2) or performance optimisations (cf. Section 5.3).

Based on seven guidelines of design science proposed by Hevner et al. [Hev+04], Peffers et al. developed a design science research methodology for IS in order to "understand essential elements of empirical IS research" [Pef+07] and provide a structured process of performing the same. The iterative process contains six main steps as visualised in Figure 2.1 [Pef+07]:

1. **Problem identification and motivation:** Firstly, researchers need to clarify the problem which should be addressed by the future solution, possibly by decomposing complex problems for better conceptualisation. In particular, the value of a solution to this problem needs to be justified in order to interpret the researcher's understanding, reasoning, and final results.
2. **Objective definition** Subsequently, the general problem formulation needs to be transformed into objectives that describe feasible characteristics of a potential solution in quantitative or qualitative terms. Because of the incremental process, not all objectives may be defined – or even known – upfront, but objectives can evolve together with the iteratively developed solution.
3. **Artefact design and development:** The actual artefact, i.e., construct, model, method, or instantiation, is designed and developed in this step. Particular attention is paid to the embedded contribution of the designed artefact with regard to the solution objectives.
4. **Demonstration:** To assess the utility of the developed artefact, its *effectiveness* in solving the identified problem needs to be demonstrated by applying it to artificial or real-world instances of the problem through experiments, case studies, proofs, or other activities.
5. **Evaluation:** Then, the *efficiency* of a solution is analysed and measured. The observed results are compared to the solution objectives defined earlier, e.g., using empirical surveys, client feedback, or simulations. After this evaluation, results can be communicated or trigger a new iteration to improve upon the design of the developed solution.
6. **Communication:** Finally, the overall process from problem definition, relevance and novelty of the research, rigour of design, artefact demonstration, and evaluation results needs to be communicated to practitioners and researchers. The audience can be informed through research papers, journals, or other common mediums of discourse in the community. Again, feedback on communicated results can lead to a new iteration of the research process by refining objectives for a solution or improving the efficiency of the design.

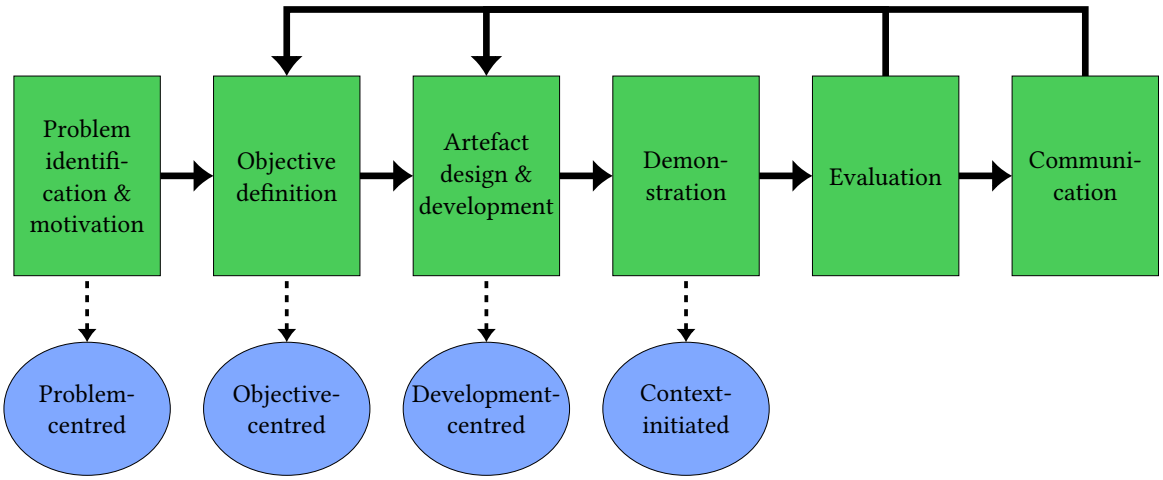


Figure 2.1: Design Science Research Process (adapted from [Pef+07])

The first four stages offer possible entry points – i.e., based on identified problems, desired objectives, practical developments, or during application of an artefact – in order to initiate research that extends along the proposed process sequence [Pef+07]. Throughout the projects and research topics of this dissertation, all four entry points have occurred to initiate research on software artefacts and development processes in the realm of MDSD. For example, recent technological developments triggered problem-centred research on classifying novel devices (cf. Subsection 3.2.2), whereas development-centred observations of parallel programs initiated work on MDSD approaches to optimise performance (cf. Section 5.3).

Entry points and artefact types highlight the user-centric focus of design science research within the triad of people, organisations, and technology. From a user perspective, an important aspect of utility is usability. Therefore, some artefacts presented in this dissertation were evaluated to this effect and the following section explains particular methods of usability research methodologies.

2.2 Particularities of Usability Evaluation

User experience (UX) can be defined as a "person's perceptions and responses resulting from the use and/or anticipated use of a product, system or service" [Int11]. Assessing the usability before releasing a software product is, therefore, important for the successful application in the field. This particularly applies to innovative fields that require a qualitative approach to explore suitable designs [Cho14; SLT09]. According to Nielsen [Nie93], usability can be decomposed into five attributes:

Learnability Users should be able to achieve rapid progress with a new system.

Efficiency A high level of productivity should be sustained once a user is accustomed to the system.

Memorability Casual users should be able to remember the system over a period of time in which the system is not actively used.

Errors A system should exhibit a low error rate during regular usage and allow users to easily recover from them.

Satisfaction In addition, users should experience a subjective satisfaction while using the system, sometimes also called the *joy of use* [HBB01].

Appropriate methods for usability evaluation depend on the progress of the development. In general, software artefacts can be tested in various stages from paper sketches to wireframes, design mockups, software prototypes, or the final product. Usability testing techniques include observations, interviews, focus groups, and questionnaires [Nie93]. With regard to domain-specific languages, research on usability either compares different types of notations (e.g., comparing textual to graphical modelling [SH09]) or compares alternative representations (e.g., general-purpose and domain-specific graphical DSLs [Żył15]). However, quantitative metrics on DSL usability are scare and not widely-used [Die+15].

Besides traditional expert interviews, a particular variant of observational studies applied in this dissertation is the think-aloud study in which participants individually perform a series of tasks in the system under test. While being observed by the experimenter, the participant is urged to verbalise thoughts, comments, and issues regarding the chosen approach to solve the tasks [Nie93]. In contrast to more sophisticated observation-based techniques with intimidating videotape recording or head-mounted eye-tracking devices, the think-aloud method reflects the actual rationale of thinking within a comfortable environment [RD90].

To gather quantitative data, several usability questionnaires have been proposed, e.g., the User Experience Questionnaire developed by SAP [LSH06], IsoMetrics [GH99], System Usability Scale (SUS) [Bro96], or ISONORM [Into6]. In this dissertation, the latter two questionnaires are employed with different purposes. One the one hand, the SUS questionnaire developed in 1986 has been applied in many contexts. It represents a fast – yet effective – approach to determine usability characteristics and can be used to evaluate prototypical states of development (e.g., a paper-based evaluation of a graphical notation is presented in Subsection 4.3.5). Each participant answers ten questions using a five-point Likert-type scale between strong disagreement and strong agreement, which is later converted and scaled to a [0;100] interval [Bro96]. In order to facilitate the interpretation of results – e.g., what threshold constitutes an acceptable level of usability – an adjective rating scale was proposed by Bangor et al. [BKM09]. The ISONORM questionnaire represents a more detailed assessment of a system's usability according to the ISO 9241-110 standard on ergonomics of human-system interaction. Its 35 questions on a scale between -3 and 3 are subdivided along the seven requirements of usability according to the ISO: suitability for the task, suitability for learning, suitability for individualisation, conformity with user expectations, self-descriptiveness, controllability, and error tolerance [Into6]. Thus, it allows for a more detailed interpretation of usability issues in actual systems.

3

Cross-Platform App Development

In this chapter, the main concepts, requirements, and challenges of cross-platform development for mobile apps are presented. After reviewing different cross-platform approaches (Section 3.1), the landscape of *app-enabled* devices is categorised (Section 3.2) and challenges of jointly developing software for multiple device classes beyond the "traditional" scope of smartphones and tablets are highlighted (Section 3.3). Moreover, a criteria catalogue for evaluating cross-platform development frameworks is presented (Section 3.4).

3.1 Foundations on Cross-Platform App Development

Cross-platform development deals with the increased efficiency of creating software products for multiple platforms using a common code base [San+14]. In this context, a platform can be defined as the underlying infrastructure that provides system functionality [SV06]. It should be noted that a platform is not necessarily equivalent to an operating system (OS). For example, a specific environment including libraries may constitute the platform on which a cross-platform approach relies. Also, different OS versions might be considered as distinct platforms if system-provided functionality, user interfaces, and typical usage patterns have evolved significantly in between.

Cross-platform app development applies this idea of a common code base for multiple target platforms to the domain of mobile applications and their ecosystem of small-scale, task-oriented, and interoperable software. To account for the multitude of devices to which these software artefacts can be applied, we denote an *app-enabled* device as one that "by its hardware and basic software (such as the operating system or platform) alone provides far less versatility than it is able to offer in combination with additional applications. Such apps are not (all) pre-installed and predominantly provided by third-party developers unrelated to the hardware vendor or platform manufacturer " [RM17]. Usually, research on cross-platform development exclusively targets smartphones and potentially tablets – sometimes even bluntly referred to as "smart devices" (e.g., [HEE13]) – and ignores the plethora of novel devices encompassed by our more general definition (cf. Section 3.2).

One decade after the introduction of the iPhone as the first app-enabled mainstream device [App08], only two major OSs – Android by Google and Apple's iOS – dominate the smartphone market with a combined share of 99.9% [Sta19a]. Previous competitors such as Microsoft (Windows Phone), RIM (Blackberry OS) now lead a niche existence. However, this does not diminish the

importance of cross-platform development because creating native apps for the duopoly of Android and iOS still duplicates the required development effort.

In order to achieve a common development process and use a single code base, different cross-platform approaches have been proposed in academia and practice, forming a continuum from purely web-based approaches to native apps. According to classifications by El-Kassas et al. [El-+17] or Majchrzak et al. [MEK15], five main approaches can be distinguished as depicted in Figure 3.1.

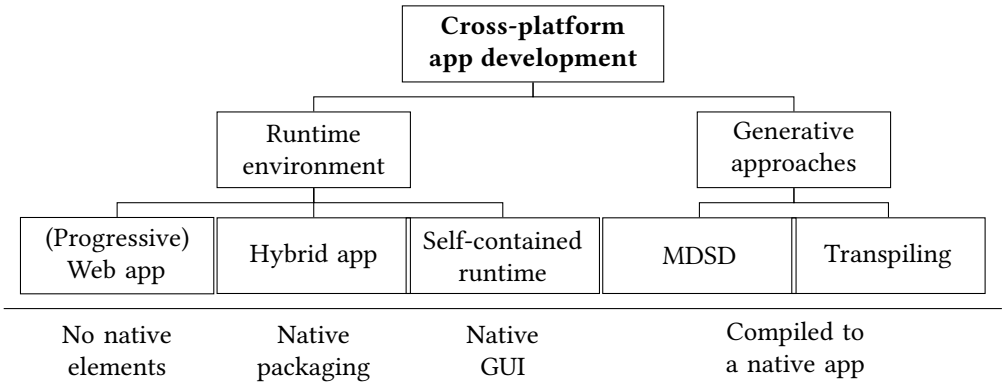


Figure 3.1: Categorisation of Cross-Platform Approaches (adapted from [MEK15])

(Progressive) Web apps: A mobile web app is developed using web technologies such as HTML, Cascading Style Sheets (CSS), and JavaScript (JS). It essentially represents a web application which is optimised for smartphone (and tablet) screen resolutions. Consequently, the app cannot be installed on the device but is executed within the platforms' browser. With the rising standardisation and support of various Application Programming Interfaces (APIs) by mobile browser environments in the past years, it is possible to access device features such as motion sensors and data storage [Bar19]. Recently, Progressive Web Apps (PWAs) were introduced by Google to improve traditional web apps with modern web technologies such as *service workers* (code running in a background thread to allow for asynchronous content updates and notifications) and *off-line capabilities* [Rus15; MBG18]. Together with a web app *manifest* (to provide metadata), PWA can be added to the home screen similarly to installed apps. This has evidently led to possibilities not previously available in the web platform, with Progressive Web Apps performing on a par with regular native apps [BMG18].

Hybrid apps: The hybrid app approach combines web-oriented development with the user experience of traditional app installation from an app store [El-+17]. Cordova [Apa19], the open-source core of the framework which became known as PhoneGap [Ado18], was an early representative of this approach that wraps HTML, CSS, and JavaScript files in an installable app container. As shown in Figure 3.2, these files are subsequently rendered using a WebView component – an embeddable browser window which hides its typical controls (such as address bar, bookmarks, and settings) [RM19]. Similar to web apps, standardised JavaScript APIs provided

by the respective browser engine can be used to access device features. The framework provides additional functionality including contact lists, Bluetooth, and network connectivity via so-called *bridge* components – i.e., JavaScript APIs acting as foreign function interface (FFI) between the WebView component and the underlying native code (in Java or Objective-C). By using the widely known web languages and packaging the resulting application as an installable app which can be used off-line, the hybrid approach has become popular amongst both practitioners and researchers. Further prominent frameworks include the Ionic Framework [Dri19], Onsen UI [Mon19], and Framework7 [Kha19].

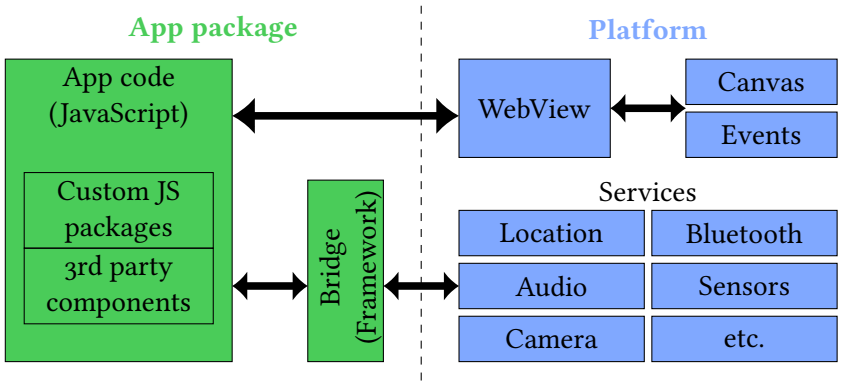


Figure 3.2: Hybrid App Architecture (adapted from [Lel17])

Self-contained runtime: In contrast to reusing platform components (such as the device's browser engine in the hybrid approach), apps built with this approach ship with a self-contained runtime component that needs to be developed separately for each target platform by the framework vendor. App developers can then use a common API to access the provided functionality. Often, frameworks of this approach expose bridges that allow for the invocation of FFI. Typically, the application code is written using a general-purpose programming language such as C# (Xamarin [Mic19]) or custom markup (Qt [The19b]), but it is also possible to develop runtime-based apps using JavaScript (e.g., Titanium [Axw19]) without using an embedded browser component. In general, the runtime interprets the app contents at runtime and maps User Interface (UI) elements and actions to corresponding native counterparts [Cor+18].

Model-driven software development: The model-driven paradigm has been used for many years in software engineering for the purpose of managing variability. Starting from a model representation of a system, the actual software artefact is derived through model transformations (cf. Section 4.1). In the context of mobile computing, it allows for the development of cross-platform apps using a higher layer of abstraction than programming code through the use of domain-specific languages (DSLs) or general-purpose modelling notations (such as UML). Subsequently, code generators (one per target platform) transform the platform-agnostic model into platform-specific source code, which can then be compiled. The resulting apps can, therefore, exploit the full potential of the platform as they are – ideally – indistinguishable from native apps.

Commercial frameworks include WebRatio Mobile [Web19], BiznessApps [Biz16], and Bubble [Bub19], whereas in academia the focus on DSLs is more prevalent (cf. Chapter 4).

Transpiling: Compilation-based approaches (so-called transpilers or cross-compilers [MEK15; El-+17]) aim at creating a native application by mapping features of an existing input application to the desired target platform. This can happen on the level of bytecode or high-level programming code. Because identifying implementation patterns from the low level of abstraction and substantial differences between the respective platforms, compilation-based approaches typically focus on specific parts of an application (e.g., the business logic layer) and need manual additions to replicate the full app functionality. Examples include XMLVM [ANP12] and J2ObjC [Goo19g].

Recently, this clear separation of approaches has become more blurred with newly emerging frameworks. In particular, a trend towards partly interpreted and partly cross-compiled apps is visible to combine the advantages of both approaches [Cor+18]. For example, React Native and NativeScript are open-source frameworks backed by large companies such as Facebook and Progress (cf. Figure 3.3) [Fac19; Pro19]. With NativeScript, established JS web frameworks such as AngularJS are used to develop apps together with view specifications using HTML and CSS. When building the app, the runtime interprets this code using the bundled V8 JavaScript engine [Pro19]. Other parts of the JS application are cross-compiled to native UI controls for the respective platform [Cor+18]. React Native has a similar approach following the reactive programming paradigm, i.e., apps are event-driven and changes to the application state propagate through dependent calculations managed by the execution model [Bai+13]. Again, the UI is cross-compiled to native widgets and business logic is interpreted by the JavaScriptCore engine (provided by the iOS platform and bundled with the App for Android). In both cases, device functionality can be accessed through the runtime APIs bridging calls to native service modules. Lastly, Flutter [Goo19e] is a framework developed by Google which uses the Dart programming language and

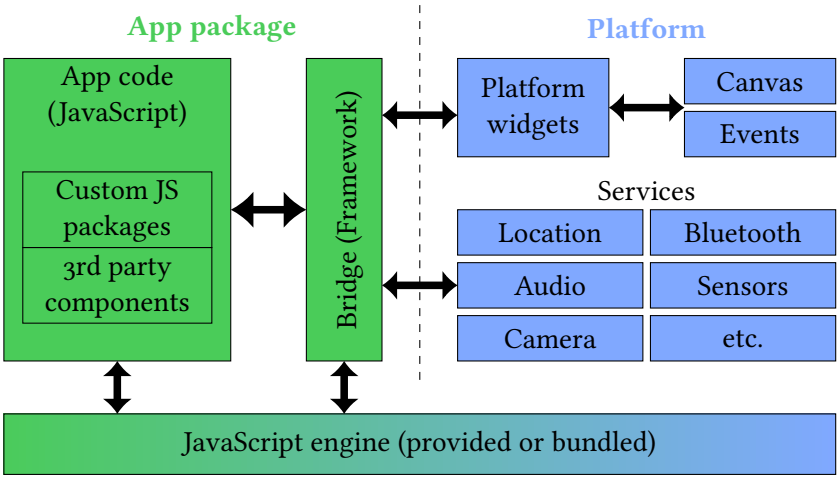


Figure 3.3: NativeScript and React Native Architecture (adapted from [Lel17])

compiles to native code ahead-of-time to eliminate the need for JavaScript bridges. Compared to the aforementioned approaches, Flutter additionally bypasses native platform widgets and ships with custom widgets which are directly rendered on the screen canvas as visualised in Figure 3.4.

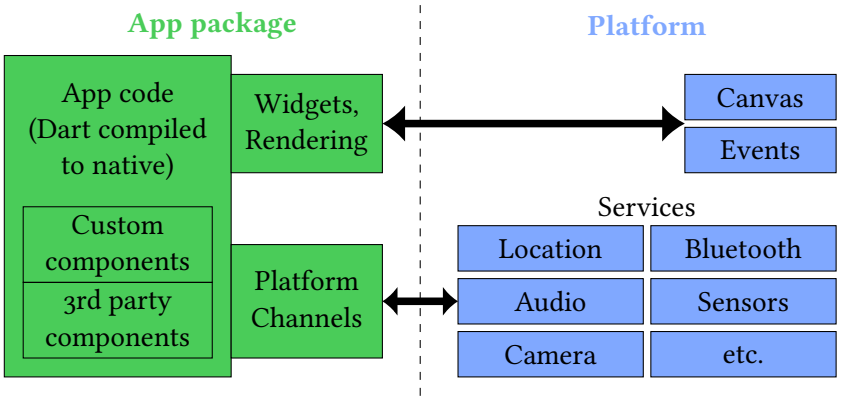


Figure 3.4: Flutter Architecture (adapted from [Lel17])

3.2 Categorising App-Enabled Devices

The approaches presented in the previous section highlight different options of creating apps for multiple platforms. Their area of application is typically limited to smartphones and tablets. However, given the wider notion of app-enabled devices, the landscape of current and upcoming devices first needs to be categorised.

3.2.1 Status Quo of App-Enabled Devices

Whereas the field of cross-platform app development for smartphones and tablets has been researched intensively [RM19] and different approaches aim to solve the challenges, many more devices have become app-enabled (cf. Section 3.1). In early visions of a world connected by ubiquitous mobile devices, the imaginable possibilities were simply categorised as tabs, pads, and boards according to their size [Wei91]. However, modern devices that follow our definition of app-enablement differ greatly in intended use, capabilities, input possibilities, computational power, and versatility, to name just a few aspects. So-called "smart devices" such as smartwatches and smart TVs have reached the mainstream market of consumer devices and exhibit double-digit sales growths over the past years [Sta18]. Furthermore, similar hardware in professional and consumer contexts, numerous sensor-driven devices for the Internet of Things (IoT), and possible variants concerning the physical embodiment of virtual assistants impede a precise denomination and discussion of related concepts [Iba+17]. In order to close this gap, a taxonomy for app-enabled devices was developed [RM17].

To make sure that we do not miss an existing taxonomy (or similar work), we conducted an extensive literature search. We focused on work from 2012 or later, where the first broader range of smartwatches such as the Pebble had already been presented. Together with the increasing variety in devices, new operating systems have appeared since then. Examples are Wear OS – formerly Android Wear – and watchOS, which focus on wearable devices [Goo19i; App19b] as well as webOS and Tizen, which address a wider range of smart devices [The19a; LG 19].

We, thus, created a search string from variations of keywords in the three domains of "app", "smart", "device" and "categorisation" for searching the Scopus database [RM18]. Although the keywords *application* and *system* represent relevant keywords in our context (e.g., cyber-physical system), we deliberately excluded them from our search for lack of specificity and differentiation from generic meanings such as *utilisation* or *structure*. Also, the *medical* area was excluded as these papers focus on individual app implementations for therapeutic purposes and do not contribute to the categorisation of app-enabled devices. A search on August 1st, 2017 yielded 1,268 results. Despite a plethora of related work in the individual fields, no systematic work exists that defines device capabilities, modes of app-enablement, notions of mobility of devices, or similar dimensions. Even overview papers typically focus on one category of devices [JM15].

To complicate matters, some papers mention that there are other *smart* devices than smartphones and tablets but do not go into detail. Only four papers went beyond a perspective on traditional mobile devices: Some authors focus on specific combinations of devices, including Neate et al. [NJE17] who investigate second screening applications that combine smart TVs with additional mobile devices, and Singh and Buford [SB16] who describe cross-device team communication apps for desktop, smartphones, and wearables. Regarding more generalised approaches, Queirós et al. [QPM17] focus on context-aware apps also suitable for novel mobile devices using the example of an automotive app. Finally, Koren and Klamma [KK16] considered the integration of heterogeneous Web of Things device types by adopting a middleware approach.

3.2.2 Taxonomy of App-Enabled Devices

Based on the literature review and the features of current and upcoming devices, our taxonomy classifies app-enabled devices with regard to the three dimensions *media richness of inputs*, *media richness of outputs*, and the *degree of mobility*. Each dimension represents a continuously increasing intensity and variability of the particular capability. The depicted gradations should be regarded as exemplary cornerstones – in contrast to a discrete checklist of technological features. Figures 3.5 to 3.7 visualise the three-dimensional categorisation of different device classes using two-dimensional projections for better readability.

Media richness of inputs describes the characteristic user input interface for the respective device class. Thereby, it captures how *humans* can interact with a device, whereas machine-to-machine communication is not considered.

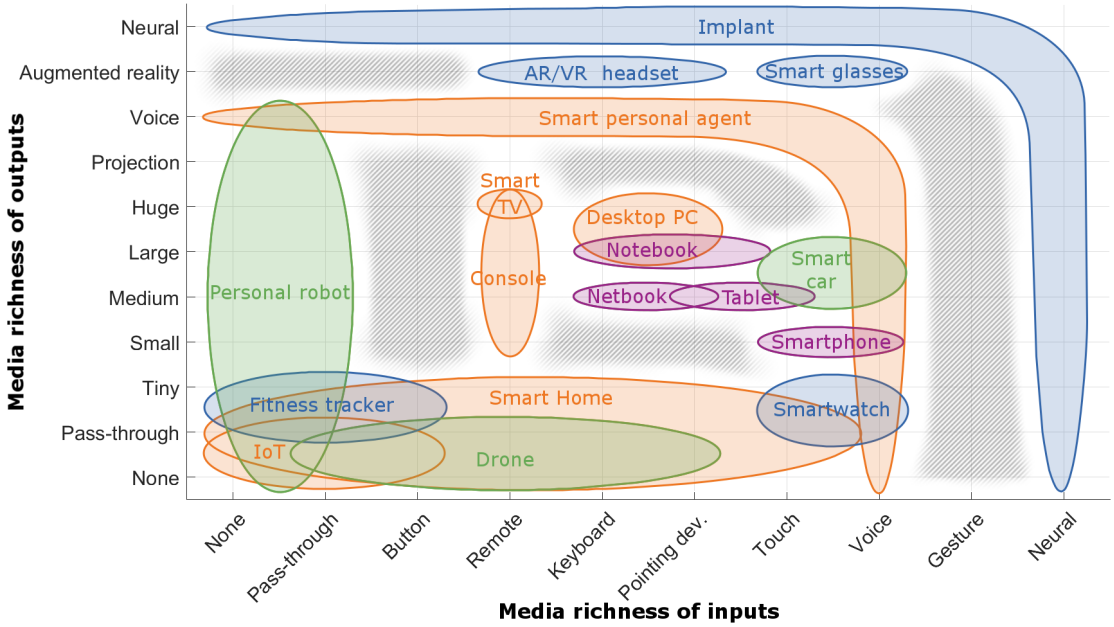


Figure 3.5: Matrix of Input and Output Dimensions (cf. [RM18])

None refers to fully automated data input through sensors and no manual activity by a user.

Pass-through represents the indirect manipulation through data exchange with an external device whose purpose is not solely to provide the user interface for the main device, e.g., remote handling through companion smartphone apps.

Buttons are physically located at the device and provide rather limited input capabilities (including switches and dials).

Remote controls refer to dedicated devices that are tethered or wirelessly connected to the app-enabled device (including also joysticks and gamepads). Technically, they merely make use of buttons, switches, dials etc. but provide a richer experience due to being decoupled from the target device.

Keyboards are also dedicated devices, but with more flexible input capabilities due to a large variety of keys. Input still is discrete.

Pointing devices refer to all devices to freely navigate and manipulate a two-dimensional Graphical User Interface (GUI), for example, mouse, stylus, and graphics tablet. While these devices technically still provide discrete input, the perception of input is continuous.

Touch adds advanced input capabilities on the device itself without a dedicated input device, allowing for more complex interactions such as swipe and multi-touch gestures.

Voice-based devices are not bound to tangible surfaces but are controlled without haptic contact.

Gestures allow for hands-free user interactions with the device in 3D space, for example using gloves or motion-sensing through gyroscopes, cameras, or lidars [LKN17; Bho13].

Neural interfaces can be expected to become the richest form of user inputs by directly tapping into the brain or nervous system of the human operator [ACC19].

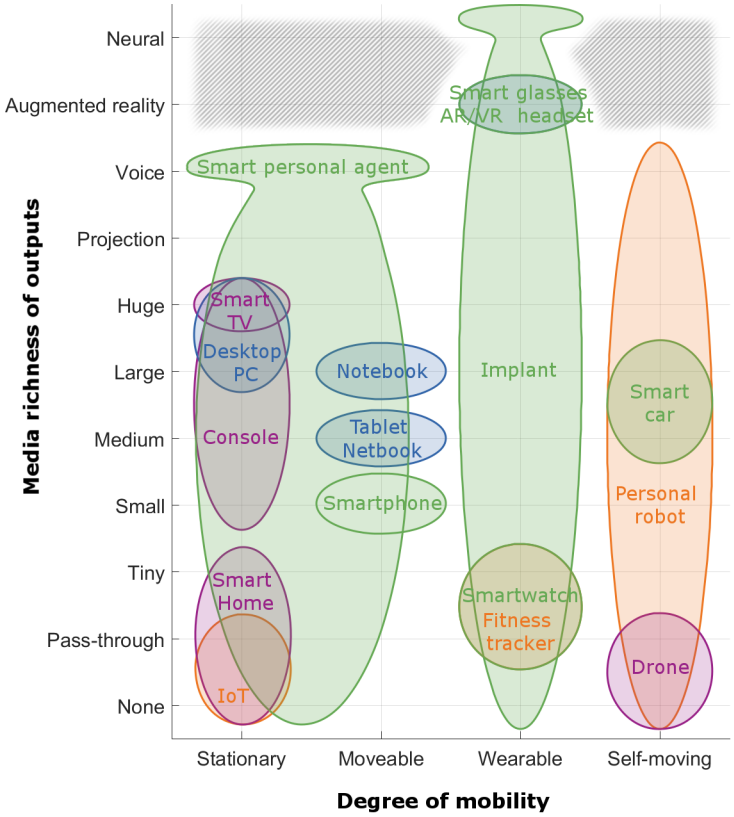


Figure 3.6: Matrix of Output and Mobility Dimensions (cf. [RM18])

As the second dimension, *media richness of outputs* describes the main human-directed output mechanisms for the respective device class.

None refers to the absence of user-directed communication. This applies to cyber-physical actuators with direct manipulation of real-world objects (e.g., switching on light).

Pass-through includes mechanisms that in general or in some situations do not produce a human-directed output of their own but pass it through to a connected managing device which retrieves information and handles user output.

Screen output is the prevalent form of user communication found in app-enabled devices. Although a clear subdivision is not possible, several classes are typically observed, ranging from tiny screen displays (<3”) to small screens such as for smartphones (<6”), medium screens for handheld devices (<11”), large screens (≤ 20”), and usually permanently installed huge screens >20”.

Projection refers to the disembodied visual output to a device-external 2D surface, e.g., a wall.

Voice-based output extends the disembodiment with auditive output to communicate with the user without visual manifestation.

Augmented reality includes virtual reality applications and hologram representations, further increasing the richness of device outputs by enriching or fully replacing the perceived reality around the user.

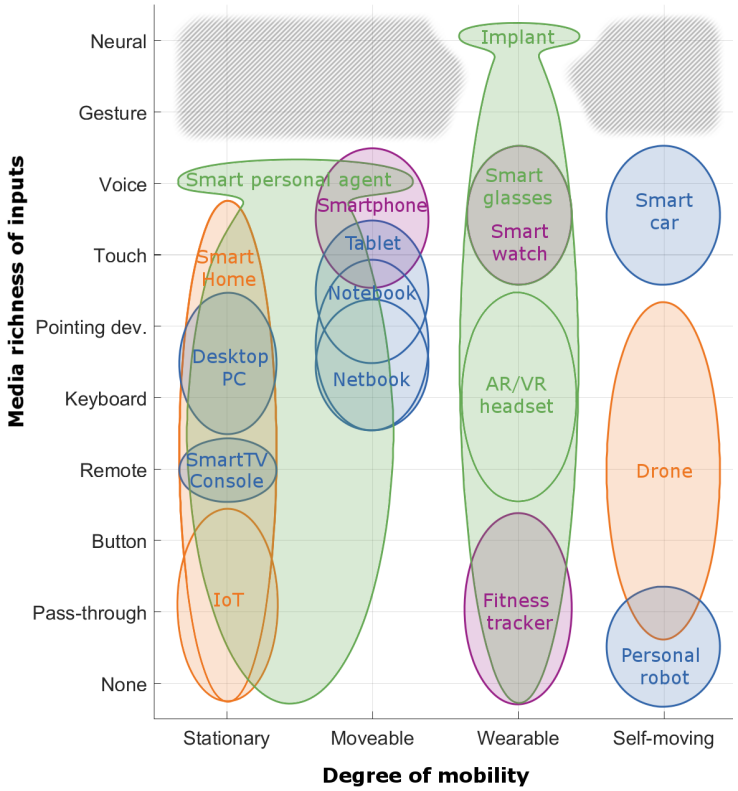


Figure 3.7: Matrix of Input and Mobility Dimensions (cf. [RM18])

Neural interfaces connect directly to the user in order to achieve a tightly coupled human-computer interaction.

Finally, the combination of input and output characteristics ignores different application areas of the respective device class. For example, intelligent switches and drones for aerial photography can both be remotely controlled and have no direct output, but can hardly be grouped as being in the same device class. Therefore, the *degree of mobility* describes the usage characteristics as the third dimension on a high level. Considering trends such as ubiquitous computing [Gub+13], this dimension also reflects the pervasiveness and integration of devices in everyday activities – from on-demand usage of stationary devices to always connected autonomous assistants.

Stationary devices are permanently installed and have no mobile characteristics during use.

Moveable devices can be carried to the place of use. This includes an "on-the-go" utilisation, such as a smartphone being used while walking.

Wearable devices are designed for ubiquitous usage and high availability through physical contact with the user. In contrast to "moveable", transporting the device is implicit and often hands-free.

Self-moving devices do not rely on human displacement (although directly or indirectly controlled by the user). Ultimately, autonomous devices represent the richest form of mobility.

The plethora of devices can then be categorised according to the three proposed dimensions.

As shown in Figure 3.5, many device classes can be assigned to distinct positions in the two-dimensional space of input/output media richness. However, it should be noted that the depicted positioning represents dominant interaction mechanisms within the device classes. For example, *smartphones* also have a few physical buttons but are mainly operated by touch input. Also, specialised or experimental devices with innovative features may provide alternative interaction mechanisms but do not constitute a distinct class of devices. For example, so-called pico projectors allowing image projection from smartphones and tablets are currently no typical means for output. Similarly, not all devices falling into a device class must necessarily implement all possibilities of that class. For example, the *smart personal agent (SPA)* Google Home typically allows for voice-based input, whereas Amazon's Echo product line also has a screen-based device.

Figure 3.5 reveals differences in the specificity of device classes by their represented size. Some of them fill specific spots in the diagram, either due to technical restrictions (*smart TVs* evolved from traditional remote-controlled TVs with large screens) or special purposes (*smart glasses* enable hands-free interaction and visualisation). Less specific device classes exist for two reasons. On the one hand, buzzwords such as *smart home* comprise every technology that relates to a specific domain, subsuming very heterogeneous devices – thereby serving rather as an umbrella term for a high-level discussion but having poor discriminating power with regard to actual device capabilities. On the other hand, underspecified device classes such as *implants* cover larger areas within the taxonomy due to their novelty and future hardware characteristics or interaction patterns are still uncertain.

Figure 3.6 depicts the combination of output media richness and mobility. Unsurprisingly, a general tendency towards large screen output for stationary devices can be observed. With increasing mobility, output capabilities develop in two directions. On the one hand, screen sizes tend to diminish, from small screens on *smartphones* to very limited *fitness tracker* screens and screen-less *drones*. On the other hand, output capabilities become richer and overcome traditional screen-based approaches due to recent technological innovations enabling intangible outputs, for instance, *Augmented Reality (AR)/Virtual Reality (VR) headsets* and voice-controlled *SPAs*.

Finally, Figure 3.7 visualises the relationship between input media richness and mobility. Usually, an increasing degree of mobility entails less physical input mechanisms with dedicated buttons and keys. This might be attributed to a lack of technological progress or practical use cases. Future devices or the evolution of existing device classes might fill some of these gaps. For example, augmented reality devices are still an active field of research and not yet common besides gaming applications. In addition, complex information output of smart devices usually requires equally sophisticated input capabilities as explained by media richness theory (MRT).

MRT describes a corridor of effective communication with matching levels of message ambiguity and media richness [DLT87]. For example, *IoT* devices have only rudimentary input capabilities but also give not much feedback in return. Keyboard and mouse are helpful means for interacting with medium to large screen output of *desktop computers* and *notebooks*, and SPAs provide advanced interpretation mechanisms that allow for natural language processing in everyday

situations. MRT also partly explains areas in the continuum with no assigned device class. Rich forms of user input such as gestures overcomplicate interactions for devices that have just, e.g., small screens and limited sensing and processing resources. On the other extreme, devices with barely a few buttons are not sufficiently flexible to manipulate large screens.

Of course, the rapid evolution of the device landscape and the constant influx of new products mandates a regular update of the categorised *device classes* for relevance and accuracy. In general, existing classes might extend towards adjacent areas or even converge. For example, convertibles represent hybrid devices between keyboard-based netbooks and touch-optimised tablets utilising docking or folding mechanisms. The evolution of one device class might even render another obsolete; this can currently be observed with all-purpose smartwatches taking over the market for fitness trackers specialised on health applications. Also, device classes might be subdivided in the future due to new technological developments or shift their position within the dimensions' continuum. However, the chosen level of abstraction implies that the taxonomy *dimensions* are intended to be rather static.

3.3 Challenges of Cross-Platform Apps Across Device Classes

Developing apps in a cross-platform fashion for more than one class of devices adds new challenges to the general problem of balancing platform commonalities and platform-specific look and feel [Fra+17]. These challenges can be grouped into four main categories [RK18b; RK19a].

Output heterogeneity: As described in Section 3.2, novel device classes vary greatly in terms of screen size (e.g., smartwatches ≤ 3" and smart TVs ≥ 20") or introduce innovative output mechanisms such as auditive output through text-to-speech synthesis or projection. Moreover, variability increases even for screen-based devices beyond "known" issues of mobile development such as device orientation and pixel density which can be solved to a certain extent through fluid or responsive layouts. Furthermore, new device classes exhibit completely different aspect ratios (e.g., ribbon-like fitness devices worn around the wrist) and form factors (e.g., round smartwatches) [RM18]. In order to develop apps for a wide variety of devices, the desired information output needs to be described on a high level of abstraction, independent of a particular screen layout.

User input heterogeneity: Correspondingly, novel app-enabled devices have different characteristics for user inputs which span from minimal interfaces with a few hardware buttons to GUIs, finger-operated touch screens, auxiliary input devices, and hands-free input via voice commands (cf. Section 3.2). Moreover, multiple input alternatives may be available and used in combination or depending on user preferences, usage context, or established interaction patterns of the respective platform. Again, this complexity calls for a high level of abstraction to manage the multitude of platform-specific event handlers. For instance, a developer may define intended user actions from a semantic perspective which facilitates the conceptualisation of an app and can later be mapped to specific input events.

Device class capabilities: Beyond user interfaces, hardware and software variability between different device classes are challenging. Of course, significant differences exist already within one device class (e.g., budget and high-end smartphones with different screen sizes and resolutions) but these challenges are aggravated when considering heterogeneous devices. For example, the miniaturisation in modern devices such as wearables limits computational power and battery capacity. Therefore, complex computations may be performed either on the device, offloaded to potential companion devices, or provided through edge/cloud computing [RZ15]. Also, sensing capabilities can vary widely. Suitable replacements for unavailable sensors need to be provided, e.g., using manual map selection instead of GPS sensors.

Multi-device interaction: Until now, cross-platform approaches often focus on providing equivalent functionality for *different users*. If at all, multi-device interaction is, therefore, achieved by common back-end components which communicate with the platform-specific apps through APIs. However, additional complexity arises from multi-device interactions *per user* because novel app-enabled devices usually do not replace smartphone usage but represent complementary devices which are used contextually. Consequently, device interactions may occur *sequentially* when a user switches to a different device depending on the usage context (e.g., location- or time-based) or personal preferences. For example, someone might use an app with smart glasses while walking and switch to the in-vehicle app when boarding a car. Alternatively, *concurrent* use of multiple devices for the same task is possible, for instance, in second screening scenarios in which one device provides additional information or input/output capabilities [NJE17]. To achieve a pleasant user experience, cross-platform development frameworks need to consider this additional complexity through device management as well as fast and reliable synchronisation to automatically update other devices based on the current application state. Whereas some device combinations are apt to peer-to-peer communication (e.g., smartwatches and fitness trackers are commonly connected to a smartphone device), a more generic approach might still involve back-end components.

To emphasise the difference in scope and the respective challenges and solution approaches compared to traditional cross-platform development, we propose the term *pluri-platform* development to signify the creation of apps *across* device classes, in contrast to cross-platform/multi-platform development for several platforms *within* one class [RK19b]. Consequently, pluri-platform development can be understood as a superordinate term for different approaches aiming to bridge the gap between multiple device classes by tackling the challenges of heterogeneous input and output mechanisms, device capabilities, and multi-device interactions. In contrast to cross-platform development, feature parity across all supported platforms is likely infeasible. Instead, the focus lies on a simplified app creation process, not just with regard to the representation of user interfaces but also the integration with platform-specific usage patterns and the interaction within a multi-device context. The related research fields of adaptive user interfaces and context-aware interfaces only account for a subset of requirements to achieve pluri-platform-development.

3.4 Evaluating Cross-Platform App Development Approaches

Although several works compare cross-platform development frameworks with different foci such as performance benchmarks or access to device functionality (cf., e.g., [El-+17; QGZ17; DM15]), a comprehensive catalogue of criteria on how to select an appropriate tool is lacking. The most extensive related work by Heitkötter et al. [HHM13] proposes 14 evaluation criteria grouped into infrastructure and development perspective. However, recent developments such as app-enabled devices beyond smartphones/tablets, the aforementioned pluri-platform development challenges, and the application of professional software development methodologies to the domain of apps are not considered.

3.4.1 Criteria Catalogue for Cross-Platform Development Frameworks

In order to guide the selection of app development frameworks, we propose an extension to the previous work which extends the original two perspectives to four [RM16]:

Infrastructure Using a cross-platform app development framework is inherently bound to preconditions, which can be summarised as the infrastructure a framework provides. Most fundamentally, this concerns the supported target platforms. Moreover, aspects of licensing and long-term prospects are considered.

Development A cross-platform framework is only as good as its utility for developing apps. Frameworks may offer further built-in support that can make development more rapid, support inexperienced developers, or both. Adequacy for development is bound to a host of criteria that all have a technical appeal and often decisive for programmers and software engineers.

App If an app is developed using a platform's native framework, it has access to all device features regarding sensors, user input, and device output. A development framework should ideally provide a near-native range of support for device features such that access is versatile and easy to employ. Also, the integration of business concerns with regard to an app as a product can be subsumed by this perspective. For instance, security is considered to be very important while becoming increasingly harder to overview for developers due to its multi-faceted nature [Wat+17].

Usage An successful app results from more than the sum of its functionality. Therefore, the usage perspective comprises many aspects that constitute non-functional (or: external quality) requirements in systems' design. Besides management aspects, this perspective embodies performance characteristics and how user-friendly an app is, including considerations of aesthetics, ergonomics, and efficiency.

We deem this distinction not only helpful for assessing a framework with regard to different stakeholders (such as developers, managers, and users) in mind but also to support a large variety

of devices. As already argued, the devices found in modern mobile computing are no more limited to smartphones and tablets and the choice of targeted devices – e.g., good smartphone support is mandatory, but compatibility with smartwatches would be nice – is merely one aspect of consideration.

Table 3.1 summarises the criteria of the perspectives; for detailed explanations, the reader is referred to [RM19]. Besides explaining what should actually be measured respectively expressed by a category, we also give its rationale – based on the literature whenever possible.

Table 3.1: Criteria Catalogue for Cross-Platform Framework Evaluation

Criterion	**Description**
(I1) License	Particularly for commercial development, a framework's license is important [Dal+13; CGG14; PSC12]. If the license is liberal concerning modifications of the framework, the impact of questionable long-term feasibility might be reduced. As part of the licensing, the pricing model needs to be considered [HHH15; SK13].
(I2) Supported Target Platforms	The supported platforms are a major concern [CGG14; PSC12]. Besides new device categories increasing the number of platforms [RM18], two versions of a platform might be different enough to consider them as distinct branches of development. This problem is worsened by the slow update behaviour of device vendors who, particularly for Android, maintain *forks* with custom modifications [Dob12].
(I3) Supported Development Platforms	App development in heterogeneous teams benefits from the flexibility of development platforms when developers can use accustomed hardware and software [PSC12]. In addition, the development of custom business logic and advanced configuration of the apps possibly differs from the actual app specification using one or multiple interoperable programming languages. Software in this sense does not only comprise the operation system but also development tools including the development environment.
(I4) Distribution Channels	Typically, platform- or vendor-specific app stores provide large repositories of apps, such as the Apple App Store and Google Play [JB13]. However, not all kinds of apps can necessarily be uploaded to all stores, for example, PWAs that are only discoverable via search engines [Rus15]. Cross-platform frameworks differ in the degree of compatibility with app store restrictions and submission regulations [SK13; DM15] as well as advanced features such as the rating of apps to improve app store ranking or support for rolling-out app updates [HHH15].
(I5) Monetisation	Monetisation possibilities include *paid* apps sold for a one-time fee, *freemium* apps which are initially free but require payment – usually via *in-app-purchases* – for advanced features, *paidmium* apps combining paid downloads and in-app purchases, and *in-app advertising* shown while using the app [Tan16; RM16]. Moreover, apps may be offered for free with the purpose of improving customer loyalty or fostering process automation [Meu+00].
(I6) Internationalisation	Apps are typically distributed globally but internationalisation and localisation techniques offer added value by broadening the base of potential users and by providing better-targeted functionality, for example, through multi-language support and conversion tools (e.g., for dates, currencies, and units) [SK13].

Criterion	Description
(I7) Long-term Feasibility	Depending on the framework, the kind of apps developed, their intended lifetime, and the situation in the developing company, a significant initial investment for research, fees, and training might be necessary and risk of vendor lock-in exists particularly for small companies with limited resources [ZZ12]. The *maturity* of a framework can be judged according to a long-lasting existence and a large community of developers. In addition, the *stability* can be assessed when looking at the history and future schedule of feature and security releases.
(D1) Development Environment	Rapid development is typically supported by the use of an integrated development environment (IDE). Ideally, a framework is not tied to a specific IDE in order to let the developer choose one with accustomed development workflows, thus lowering the set-up effort of dependencies such as runtime environments or SDKs [SK13].
(D2) Preparation Time	The learning curve of a framework should enable rapid progress of a developer in getting acquainted with the capabilities of a framework. The entry barrier is also influenced by the required technology stack and the number and kind of supported programming languages [XX13; CGG14]. Additionally, extensive documentation, "Getting Started" guides, tutorials, screencasts, and code examples make a framework more accessible.
(D3) Scalability	Particularly in large-scale or distributed development projects, apps need to scale through modular architectures. Ideally, more developers can be added to a project while the app's functionality grows [HHH15; PSC12] and support for layering allows for a higher level of specialisation is possible for developers.
(D4) Development Process Fit	A framework should be compatible with custom ways of developing software, from the initial boilerplate code to a *minimum viable product* and further incremented scope. Tailored views and specialised tools support modularising development to development roles in contrast to the work of full-stack developers typically encountered in small projects [Was10].
(D5) User Interface Design	A *What You See Is What You Get* (WYSIWYG) editor can be very helpful to design appealing UIs in faster time without repeatedly building and deploying the full app. However, reasonable support for platform-adequate designs is beneficial through responsive UIs or multiple layouts for specific ranges of screen sizes [EVP01].
(D6) Testing	User interface, business logic, and additional components of apps need to be thoroughly tested [HHH15; SK13]. In particular, challenges of mobile contexts such as physical movement, app interactions, or unreliable network connectivity need to be considered [MS15]. Also, monitoring the app at runtime through asynchronous error logging or remote debugging simplifies the detection of bugs.
(D7) Continuous Delivery	Frameworks vary with regard to the extent of deployment support, from creating source code to providing signed packages using installed platform Software Development Kits (SDKs), or using could-based build services [HHH15; SK13]. Particularly when agile development methods are applied, continuous delivery platforms help automate building, testing, and deploying an app to devices or app stores for publishing updated versions [HHH15].
(D8) Configuration Management	Often, apps do not exist in isolation but have multiple versions when considering multiple roles (such as user and administrator with different capabilities), theming (or branding for white-label apps), regional peculiarities (e.g., right-to-left script), or intended usage.

Criterion	Description
(D9) Maintainability	When an app is maintained for a shorter or longer period, over which the code base evolves, the reusability and portability across development projects can be evaluated [SK13]. Metrics such as lines of code (LOC) or cyclomatic complexity density can shed light on code complexity [GK91]. Also, qualitative aspects of code readability, the use of design patterns, the kind of in-code documentation, and similar aspects can be considered.
(D10) Extensibility	Although a framework covers a scope of *typical* apps which can be built with it, project-specific requirements may go beyond the provided features but do not represent common functionality to be added to the framework. Being able to extend the framework through third-party libraries or custom-built components is, therefore, preferred [HHH15; PSC12].
(D11) Integrating Custom Code	Even in cross-platform approaches, some applications require native code or third-party libraries to be run, for example due to non-extensible framework functionality (D10) or uniqueness for a specific platform [SK13; PSC12]. Also, the ability to integrate custom code allows for the successive migration of native legacy code to a cross-platform approach.
(D12) Pace of Development	While many of the above criteria have an influence on how rapid development will be, particularly influencing factors include the amount of boilerplate code necessary for functional app skeletons [Hei+14], Also, the availability of software functionality for recurring use cases (e.g., user authentication or data transfer) facilitates swift development progress.
(A1) Access to Device-specific Hardware	Access to device-specific hardware is vital for cross-platform frameworks to be versatile in different apps [HHH15; Dal+13; CGG14; SK13; DM15]. A multitude of sensors exists, including camera, microphone, GPS, accelerometer, and gyroscope as well as novel additions such as a heart rate monitor. In addition, cyber-physical systems may also offer actuators to interact with their environment (e.g., for vibrotactile feedback or light switches).
(A2) Access to Platform-specific Functionality	Modern mobile platforms provide various services that should be supported in order to create versatile apps. Features include a persistence layer, providing file system access and storage to a database, contact lists, information on the network connection, and battery status [HHH15; DM15].
(A3) Support for Connected Devices	Wearables and sensor/actuator networks of cyber-physical systems are often used as companion devices, typically coupled to a smartphone. The support of viable device combinations by frameworks should be assessed [Sey+15], either to access (sensor) data or to pre-render UI components for output on the coupled device.
(A4) Input Device Heterogeneity	Cross-platform frameworks need to make user input mechanisms (cf. Subsection 3.2.2) available to developers via simple-to-use APIs, consider the lack of input actions on individual devices, and respect platform- or device-specific patterns.
(A5) Output Device Heterogeneity	Similarly, multiple output possibilities exist depending on the device capabilities (cf. Subsection 3.2.2). Adaptability is even more challenging when considering device class specific context changes such as a day/night screen mode for in-vehicle apps [AK14; SAW94].
(A6) Application Life Cycle	In contrast to the development life cycle (D4/D7/D9), the app life cycle refers to states such as starting, pausing, continuing, and exiting an app [SK13]. Multithreading, continuously running background services, and notifications further extend the states in which an app is executed.

Criterion	Description
(A7) System Integration	Frameworks can support the integration with back-end systems with components for, e.g., data exchange protocols and serialisation [Dal+13]. Additionally, workflow-oriented use cases typically rely on collaboration from several user roles.
(A8) Security	Regarding security attributes [Par98], *confidentiality* can be supported by restrictive access permissions for platform and devices features raising user awareness and acceptance [Kel+12], *integrity* can be improved by using encryption for storing and transmitting sensitive data [Dal+13; HHH15], and *control* can be fostered by preventing code injections, cross-site request forgery, and similar attacks [HHH15].
(A9) Robustness	Following the strategy of *graceful degradation*, apps should include intelligent fallback mechanisms in case specific features are unavailable. Also, fault-tolerant and resilient mechanisms are important to ensure functionality on erroneous user inputs, missing permissions, or external influences such as network connection [Chu+12].
(A10) Degree of Mobility	Different degrees of intended mobility strongly affect app mechanics and emphasise features beyond infrastructure considerations. On a high level, stationary, mobile, wearable, and autonomous devices can be distinguished (cf. Subsection 3.2.2).
(U1) Look and Feel	A near-native look and feel of UI elements and platform-specific interaction patterns are vital for user adoption and satisfaction [SK13; Dal+13]. Elements, views, and interaction possibilities can be evaluated according to the respective human interface guidelines provided by platform vendors [Goo19c; App19a].
(U2) Performance	Performance considerations include, for example, app load time, app speed for changing views and computations resulting from user interactions, the speed of network access, and stability [DM15]. Moreover, resource utilisation such as CPU load, memory usage, or energy demand can be scrutinised [SK13; CGG14; Dal+13; CG15].
(U3) Usage Patterns	Apps should align with personal workflows for information processing such as sharing with other apps, saving to persistent storage, or receiving contextual notifications. Also, apps should provide an "instant on" experience in the face of interrupted usage such that users can continue where they left the app (including unsaved data).
(U4) User Authentication	Apps may have a purely local users, cloud-based accounts (e.g., enabling services such as synchronisation), or employ centralised user management with multi-device account management or role-based access rights [Kun+14]. The availability of user accounts also influences possible authentication mechanisms [LL15].

3.4.2 Assessment of Criteria Using Weight Profiles

On the one hand, this holistic view on cross-platform app development considerations enables fact-based and replicable assessments for each criterion with as little subjectivity as possible. On the other hand, frameworks are selected within an individual situation of project-specific contexts, personal preferences, and other situational requirements to account for during the framework selection. Therefore, criteria should be weighted instead of simply using the average score of the criteria as the overall assessment. This also serves the purpose of balancing different levels of technical depth, which is unavoidable given the heterogeneity of the criteria in our catalogue.

In addition, the weighting approach also helps to document the rationale behind specific tool selections for decision-making processes in large organisations in which executive managers are accountable for the selection decision.

To apply our weighting scheme, each of the 33 criteria receives a *weight* between 1 and 7 to avoid unbalanced assessments in which the majority of criteria has no overall impact on the resulting score. The total is 100 points, i.e., each point denotes a 1% weight. In case a criterion should be explicitly omitted in the assessment, assigning a weight of 0 is also possible. Each evaluated criterion is assigned a *score* from 0 (criterion unsatisfied) to 5 (criterion fully satisfied) for the framework under test. The overall score S of an assessment is then calculated as the weighted arithmetic mean of the criteria, i.e., as

$$S = \frac{\sum_{j=1}^{7} w_{i,j} * i_j + \sum_{j=1}^{12} w_{d,j} * d_j + \sum_{j=1}^{10} w_{a,j} * a_j + \sum_{j=1}^{4} w_{u,j} * u_j}{100}$$

$$\sum_{i} w_i + \sum_{d} w_d + \sum_{a} w_a + \sum_{u} w_u = 100$$

with i_j, d_j, a_j, and u_j being the criteria score from the infrastructure, development, app, and user perspective, respectively, and $w_{i,j}$, $w_{d,j}$, $w_{a,j}$, and $w_{u,j}$ the corresponding weights for each criterion. A final score of 0 would represent an absolutely dysfunctional framework, whereas 5 represents the all-encompassing tool. Using the weighted sum is deliberately chosen to ensure simplicity and understandability for all stakeholders in the assessment process. Essential requirements, such as weights being proportional to the relative value of criterion score changes, are fulfilled by using equal intervals for all scores across heterogeneous criteria [Hob80]. In contrast to more advanced techniques for multi-criteria analysis such as rank-based criteria assessment or pair-wise comparisons [BB96], the chosen approach is modular such that weights and scores can be defined by different experts in the respective domain. In addition, new frameworks can easily be added to the decision process without re-evaluating all combinations. This is particularly beneficial with regard to the large number of cross-platform frameworks and the constant evolution of newly appearing frameworks whilst others becoming obsolete.

Although the weighting approach is highly flexible, many developers and companies in need of a framework typically face one of a number of comparable settings. Therefore, along with our evaluation framework, we propose *weight profiles* as reasonable weighting for typical situations which can evolve with the general progression of the mobile computing field. They can be used unchanged to gain an overview from a quick assessment. Alternatively – or successively – they can be used as the foundation for an individual assessment. Referring to corporate contexts, weight profiles also allow for a divide-and-conquer proceeding in which narrow-area experts assess individual criteria, experts with a more general focus evaluate criteria categories, IT managers set the weights, and executive management finally makes the decisions.

Table 3.2 provides an exemplary *smartphone* weight profile, along with the weights of five

Table 3.2: Comparison of Frameworks and Device Class Weight Profiles (cf. [RM19])

			Smartphone Comparison					Criteria Weights (%)				
Criterion		**Weight (%)**	**Web Apps**	**PWAs**	**PhoneGap**	**React Native**	**Native apps**	**Tablets**	**Entertainment**	**Wearables**	**Vehicle**	**Smart Home**
I1	License	5	5		5	5	5	5	6	5	3	5
I2	Target Platforms	6	5	4	5	4	1	5	6	5	4	7
I3	Development Platforms	2	4		5	4	2	2	2	1	1	1
I4	Distribution Channels	2	5		3	3	4	2	3	4	3	3
I5	Monetisation	1	0		3	3	5	2	1	1	2	2
I6	Internationalisation	1	1		3	3	5	2	2	2	0	1
I7	Long-term Feasibility	5	5		5	3	4	5	3	3	5	5
D1	Development Environment	7	4		5	4	5	7	7	5	5	6
D2	Preparation Time	7	5		4	4	3	7	7	5	1	5
D3	Scalability	2	3		3	4	3	2	3	2	3	2
D4	Development Process Fit	2	3		3	3	2	2	3	1	3	2
D5	UI Design	4	3		3	2	4	4	5	5	6	2
D6	Testing	3	3		4	3	5	3	3	3	6	3
D7	Continuous Delivery	3	5		5	3	3	3	3	4	3	2
D8	Configuration Management	1	0		0	0	3	1	1	2	2	2
D9	Maintainability	2	2		4	4	2	2	2	1	3	2
D10	Extensibility	2	5		5	2	5	2	2	2	1	2
D11	Custom Code Integration	2	0		3	3	5	2	2	1	0	0
D12	Pace of Development	4	3		4	3	0	4	3	3	2	4
A1	Hardware Access	4	2		4	3	5	3	1	6	4	6
A2	Platform Functionality	5	2		4	3	5	5	3	2	2	3
A3	Connected Devices	3	0		2	2	5	2	1	5	3	7
A4	Input Heterogeneity	1	3		4	4	5	3	3	2	2	2
A5	Output Heterogeneity	1	3		4	4	5	1	1	6	3	4
A6	App Life Cycle	2	0	2	4	4	5	3	3	3	3	2
A7	System Integration	3	3		3	3	5	3	3	1	2	1
A8	Security	3	0		0	0	3	4	1	3	7	5
A9	Robustness	2	2		4	2	3	1	4	3	2	1
A10	Degree of Mobility	1	1		1	3	5	1	0	4	5	0
U1	Look and Feel	5	1	2	3	4	5	4	2	4	5	3
U2	Performance	4	2	3	2	3	5	3	6	3	3	2
U3	Usage Patterns	2	0	2	2	2	2	3	4	3	3	4
U4	User Authentication	3	0		0	0	1	2	4	0	1	4
Weighted Score			2.87	2.98	3.59	3.11	3.73					

additional profiles. Prior work has shown that cross-platform approaches often focus on developers' needs [Res14] but a balanced profile could focus on some major aspects as explained in the following. Open-source approaches are appreciated for their low license costs (I1). At the very least, recent iOS and Android versions must be supported to reach a large user base (I2) through the respective app stores (I4). Also, the long-term feasibility of the framework is of high importance due to the initial training effort to build up knowledge (I7). Accordingly, a proper development environment (D1), low preparation time (D2), and rapid development progress (D12) are essential to foster a timely completion of app projects. On the other hand, scalability (D3), development methodologies (D4), and advanced configuration management (D8) are usually of minor relevance unless aiming for complex and large-scale apps. However, a good user interface design process (D5) is more important for smartphone usage, together with extensive access to device hardware (A1) and platform functionality (A2). From the user perspective, smartphone apps need to provide a near-native appearance (U1) and a good runtime performance (U2).

As mentioned before, this generic smartphone profile can be modified to fit particular needs. If, for example, multiple apps are planned with strategic value for a company, the full weight could be given to I1 (License) and I7 (Long-term Feasibility), and possibly higher weights to D9 (Maintainability) and A7 (System Integration). At the same time, D2 (Preparation Time) could be assigned low weights due to long-lasting involvement.

A *tablet* profile with weights for app projects specifically targeting tablets is very similar to the smartphone profile and typically only deviates by 1. This reflects the current device landscape in which tablets and smartphones share many hardware characteristics (mostly distinguished by screen size) and often run the same OS. Although both device classes do not tend to converge, some devices are hard to assign to one specific class such that the term *phablet* was coined. If app development is planned for smartphones and tablets simultaneously, considering both weight profiles might tip the scales for matured frameworks which are almost equipollent concerning smartphone weights alone.

The focal areas of further proposed weight profiles in Table 3.2 are sketched in the following. An *entertainment* profile applies, for example, to AR/VR devices. In contrast to the aforementioned profiles, the app perspective is generally of reduced importance due to the reduced need for device access and connectivity but performance (U2), adequate support of input mechanisms (A4), and robustness (A9) matter more to create an immersive experience, e.g., in games or virtual museums [Bro19]. Also, a common look and feel (U1) is not necessarily aimed for when highly individual UIs are designed to make the app stand out from its competitors.

Wearables represent an increasingly common class of devices whose main development challenge results from the heterogeneity of device capabilities. Therefore, the creation of UIs (D5), access to device hardware (A1), and the output heterogeneity of app contents (A5) need to receive particular attention. Also, aspects of mobility (A10) and connectivity to other devices (A3) are of importance, whereas typical business integration criteria such as user management (U4), and system integration (A7) are of minor relevance for this type of innovative technology. Similarly,

apps for wearables are rather small-scale and do not require sophisticated development processes (D4) and scalable teams (D3).

App-enablement of cars is just at its beginning and uncertainty exists regarding the best approach to ensure security and reliability [Man+18]. Still, criteria weights for *vehicles* highlight a different set of considerations than the aforementioned profiles. Apps should be conceived with a long-term focus due to the slow dissemination of the technology (I7) but other infrastructure criteria are given less weight. Regarding development, UI design (D5) is particularly important to adapt to contextual factors (e.g., night mode) and match the look and feel (U1) of the in-vehicle infotainment system. Good testing support (D6) is essential to simulate apps before executing them in the real environment and achieve a maximum of security (A8).

Finally, due to the rise of consumer IoT devices, we deem a *smart home* profile to be utile. Even though this field is also still emerging, an initial assessment can be made. Due to the very multitude of devices, adequate support of possible target platforms (I2) is essential. For the same reason, special emphasis should be put on long-term feasibility (I7) at this stage. Development aspects can get comparatively less focus but for a powerful IDE and low preparation time, enabling a rapid start and low barriers for changing to another framework if necessary. However, IoT apps require profound hardware access (A1), good connectivity with a wealth of other devices (A3), and a high level of security (A8).

Several of the proposed weight profiles target device classes which have just recently emerged and no body of knowledge has formed yet. The weights might, therefore, need adjustments in the near future but Table 3.2 conveys the main objectives and influencing factors for these types of devices. Also, it should be noted that the weighting approach is not limited to enterprise app development settings. For instance, a custom set of weights could be devised to projects in app development classes by focusing on rapid ramp-up time (D2) and development progress (D12) while deliberately neglecting considerations such as distribution (I4) and monetisation (I5).

To sum up, the criteria catalogue presented in this section covers app development from the infrastructure, development, app, and user perspectives that allow for a frameworks selection based on objective assessments as well as flexible customisation through a weighting scheme.

3.5 Discussion

The artefacts developed in this chapter pave the path towards answering research question RQ1. While devising methods for extending cross-platform development beyond smartphones and tablets, our literature search revealed that both terminology and prior research does not sufficiently consider novel devices. Cross-platform development traditionally relates to the domain of mobile computing and focuses on smartphones and – often implicitly – tablets which arbitrarily limits the scope and opportunities of device ecosystems encountered today. In addition, new software platforms have appeared for these devices, some of which are either modifications of established operating systems for other device classes (for instance, Wear OS similar to Android

for smartphones [Goo19i]) or are newly designed to run on multiple heterogeneous devices (e.g., Tizen running on smartwatches and smart TVs [The19a]).

In order to consider the rapidly changing device landscape, developers need to shift their attention towards the broader view of *app-enabled* devices which, for example, includes wearables but also stationary devices which are extensible through third-party apps. Consequently, categorising the device classes using a taxonomy approach constitutes a prerequisite for the concept of *pluri-platform* development which is proposed to accentuate development activities *across* device classes. Although we highlight that the taxonomy will need to be updated in future, it can serve as a scheme for academic discourse on app development. With regard to challenges of developing apps beyond smartphones described in Section 3.3, the challenges of output heterogeneity, user input heterogeneity, and device class capabilities are already existent in traditional cross-platform development but grow in complexity when considering heterogeneous device classes. In addition, device interactions of multiple owned devices per user pose new challenges which cross-platform development approaches and frameworks do not yet consider.

Already from their technical foundations, the approaches for cross-platform development mentioned in Section 3.1 are varyingly suited to the specific challenges of pluri-platform development. Whereas *mobile web apps* – including Progressive Web Apps – are mobile-optimised web pages for screen-based browsers, most novel device types do not provide browser engines that allow for HTML rendering and the execution of JavaScript code. Consequently, this approach cannot currently be used for targeting a broader range of devices. *Hybrid apps* essentially use the same web technologies within WebView components. Even though the framework can provide additional functionality through APIs, the essential JavaScript engine is missing and hybrid apps are, therefore, neither suitable for pluri-platform development.

In contrast, apps using a *self-contained runtime* do not depend on the device's browser engine and vendors may develop runtimes for platforms from different device classes. Of the runtime environment approaches, this is the only one that supports developing apps across device classes. Although usually based on custom scripting languages, a runtime can also be used as a replacement for inexistent platform functionality. For example, CocoonJS [Lud17] recreated a restricted WebView and therefore supports the development of JavaScript-based apps also for Google's Wear OS platform. However, synergies with regard to input/output mechanisms and available hardware/software functionality across heterogeneous devices are limited by the runtime's API and its developer's ability to establish and maintain support for many platforms.

Considering generative approaches to cross-platform development, *model-driven software development* has several advantages from its use of textual or graphical models. Domain-specific concepts facilitate a high level of abstraction, for example by circumventing issues such as screen layout using declarative notations. From this platform-neutral model specification, arbitrary platforms can be supported by developing respective generators which implement a suitable mapping to native platform-specific source code. Finally, bridging device classes with *transpiling* approaches is technically feasible because transformations between platform-specific representa-

tions also output native source code. However, there is more to app development than just the technical equivalence of programs, which also explains the low adoption of this approach by current cross-platform frameworks. In particular, the device UIs across different device classes follow distinct interaction patterns, and substantial efforts would be required to identify and adequately transform them on the low level of platform-specific code. It is, therefore, unlikely that this approach is viable for bridging device classes beyond reusing generic components such as business logic.

To sum up, only self-contained runtimes and model-driven approaches are candidates for device class spanning app development. The latter additionally benefit from the transformation of domain abstractions to platform-specific implementations. Therefore, Chapter 4 focuses on the model-driven paradigm and the creation of cross-platform and pluri-platform business apps using domain-specific languages to provide a technical solution to RQ1.

3.6 Contributions to the Field of Research

With regard to cross-platform app development, this dissertation makes several contributions which can be subdivided into *theoretical* contributions of constructs, models, or methods (cf. Chapter 2), *empirical* contributions of transferable observations or evaluations, and *practical* results such as recommendations for real-world applications.

Theoretical The conceptualisation of cross-platform development is advanced by defining the new scope of *app-enabled* devices in contrast to previous notions of "mobile apps" or "mobile computing". Furthermore, a theoretical view on the challenges of app development across device classes and the adequacy of existing approaches is provided.

Empirical Within the multitude of novel devices in visionary concept studies or already available for purchase, this thesis contributes a taxonomy of device classes according to the dimensions of input/output media richness and degree of mobility in order to pinpoint recent technological developments and precisely discuss idiosyncrasies, drawbacks, and potential solutions of cross-platform approaches or frameworks.

Practical As practical research artefact, we provide an extensive criteria catalogue and weighted evaluation approach for framework selection that can be used by researchers and practitioners to assess development frameworks according to company- or project-specific needs.

These contributions are described and evaluated in greater detail within the following publications (cf. Part II):

- Christoph Rieger and Tim A. Majchrzak. "Towards the Definitive Evaluation Framework for Cross-Platform App Development Approaches". In: *Journal of Systems and Software (JSS)* (2019). DOI: 10.1016/j.jss.2019.04.001

- Christoph Rieger and Tim A. Majchrzak. "A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle". In: *Web Information Systems and Technologies*. Ed. by Tim A. Majchrzak et al. Springer International Publishing, 2018, pp. 202–220. DOI: 10.1007/978-3-319-93527-0_10
- Christoph Rieger and Herbert Kuchen. "Towards Model-Driven Business Apps for Wearables". In: *Mobile Web and Intelligent Information Systems (MobiWIS)*. ed. by Muhammad Younas et al. Springer International Publishing, 2018, pp. 3–17. DOI: 10.1007/978-3-319-97163-6_1
- Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-enabled Devices". In: *13th International Conference on Web Information Systems and Technologies (WEBIST)*. Porto, Portugal, 2017, pp. 332–339. DOI: 10.5220/0006353003320339
- Christoph Rieger and Tim A. Majchrzak. "Weighted Evaluation Framework for Cross-Platform App Development Approaches". In: *Information Systems: Development, Research, Applications, Education: SIGSAND/PLAIS EuroSymposium*. Ed. by Stanislaw Wrycza. Springer International Publishing, 2016, pp. 18–39. DOI: 10.1007/978-3-319-46642-2_2

4 Model-Driven Mobile Development

Based on the findings of the previous chapter, model-driven software development seems apt to the challenges of pluri-platform development. In this chapter, the model-driven paradigm is presented with regard to the targeted domain of so-called business apps (Section 4.1). Subsequently, we focus on two domain-specific languages which can be used to create source code for multiple platforms using a textual (MD²; Section 4.2) or graphical (MAML; Section 4.3) representation. In order to achieve pluri-platform development capabilities, a model-driven process is proposed and illustrated by the example of a code generator for smartwatches (Section 4.4).

4.1 Foundations on Model-Driven Mobile Development

The full potential of MDSD is tapped when using the approach in combination with a DSL. This section, therefore, highlights the foundations of model-driven software development from a technical perspective and also presents the domain of business apps.

4.1.1 Model-Driven Software Development

MDSD shifts the focus of software development activities from source code to models as the primary artefacts for developing software more efficiently. Abstract – but formally concise – models are used to specify the artefact, in contrast to earlier approaches such as round-trip engineering which operate on a similar level of abstraction as the resulting code. Models are then either interpreted at runtime or transformed into executable software through code generation. The concepts of model-driven and generative software development have gained attraction in academia and practice because of the expected benefits in development speed, software quality, and code reuse [SV06]. Therefore, model-driven approaches are particularly suited to repeated programming tasks such as software product lines with multiple variants of modular software components in different configurations [GS03] or cross-platform development approaches. In our work, we concentrate on cross-platform code generation in order to create native app source code for different target platforms from a common model.

Regarding instantiations of the model-driven paradigm, one common approach is the Model-Driven Architecture (MDA) which consists of four components [Arc01; Arc14]:

- On the highest level of abstraction, the *computer-independent model (CIM)* represents the domain concepts from a business perspective without technical considerations, e.g., Business Process Model and Notation (BPMN) for describing business processes (cf. next subsection).

- Subsequently, the *platform-independent model (PIM)* depicts interactions of the respective components in a logical system view. As an example, Unified Modeling Language (UML) profiles can be used to create a PIM.
- The *platform model (PM)* specifies the architecture and constraints of a target platform.
- By taking the PIM and PM into account, the *platform-specific model (PSM)* describes the software system for a specific platform, e.g., regarding database dialects.

Through a series of (not necessarily automated) model-to-model transformations, a CIM is first transformed into a PIM and, finally, into a PSM within the constraints specified in the PM.

Several general-purpose modelling notations exist for graphically depicting applications and processes, such as the UML with a collection of interrelated standards for software development. For example, the Interaction Flow Modeling Language (IFML) can be used to model user interactions in software systems and has been extended with additional elements for the domain of mobile apps [BKF14]. Process workflows can be modelled using, e.g., BPMN [Obj11] (cf. Subsection 4.1.3). Examples of technical notations in the domain of mobile applications include a UML extension for distributed systems [SW07] and an extension to BPMN for web service orchestration [BDF09]. However, a potential drawback of such notations is the balancing of readability and technical specificity: models are often superficial to suit generic modelling tasks or represent rather complex technical notations [Fra+06].

To maximise the potential of MDSD approaches, domain-specific languages (DSLs) can be used, i.e., computer languages that provide syntax and semantics for a variety of models within a scope of similar problems, the so-called *domain* [MHS05]. Usually, DSLs provide either a textual or graphical syntax to specify models (cf. Section 5.1) [Völ13]. Moody [Moo09] has pointed out guidelines for the cognitive effectiveness of visual notations, including the principles of semiotic clarity and semantic transparency (providing meaningful visual representations which correspond to semantic concepts), perceptual discriminability and visual expressiveness (using clearly distinguishable symbols), and complexity management through modularisation. Subsequent studies have revealed potential comprehensibility issues of existing graphical notations through effects such as symbol overload, e.g., for the WebML notation preceding IFML [Gra+15].

With regard to engineering a DSL, so-called language workbenches are software development tools to define, reuse, and compose languages using high-level mechanisms [Erd+13]. Thus, (domain-specific) languages can be designed and integrated with manageable effort, including validation, testing, and syntactic/semantic editing services within the IDE. For textual DSLs, Spoofax [Met19] and Xtext [Ecl19g] are common frameworks of which the latter is more feature-complete (a detailed list can be found in [Erd+13]). In the open-source Xtext framework, the DSL is created by specifying a grammar in Xtext's grammar language. Integrated with the Eclipse Modeling Framework (EMF), an Ecore meta model is derived from this together with an ANTLR parser [Ecl19a; Ecl19g]. Additionally, a plug-in is generated to provide modelling support such as syntax highlighting, auto-completion, and validation within the Eclipse IDE

[Bet13]. Regarding graphical DSLs, an additional challenge lies in the effort to create and maintain an editor component which complies with the meta model and enforces semantic and syntactic constraints. Within the Eclipse ecosystem, this can be achieved on multiple levels. The Graphical Editing Framework (GEF) [Ecl19c] simplifies the visualisation of graphical components, whereas the Graphical Modeling Framework (GMF) [Ecl19d] is built on top of this to provide a runtime infrastructure for EMF-based models. The Sirius project [Ecl19e] adds a layer of abstraction to this technology stack in order to declaratively configure the appearance and behaviour of graphical editor component and reduce the development effort of a graphical DSLs framework [Jäg+16; VMP14].

4.1.2 Model-Driven Business Apps

In order to adequately cover the features, variability, and interactions of mobile apps, the scope of the domain needs to be clearly defined. In contrast to gaming or messaging apps, this work focuses on so-called *business apps*. According to the definition by Majchrzak et al. [MEK15], a business app is not primarily designed to generate revenue through app store sales but rather serves the purpose of supporting business processes, improving company-internal or inter-organisational collaborative work, or fostering customer loyalty and marketing. In this sense, business apps are not exclusive to an enterprise context but applicable in different situations in which an app shares the following characteristics [MEK15]:

- The app is *form-based* and makes use of so-called widgets as standardised UI elements. This includes both content elements such as text fields, checkboxes, and buttons as well as layouts which can be nested to organise content in typical structures such as vertical/horizontal sequences of elements, tables, cards, or tabs.
- The app is *data-driven*, meaning it focuses on data input, manipulation, or output as its core purpose. A sequence of data processing steps reflecting a business process is an example of this, whereas games reacting to user inputs are not data-driven in this sense.
- Whereas the app is functional without a network connection, it is linked to one or several *back-end systems* in order to remotely store data, orchestrate devices, off-load complex computations, or utilise services from other information systems. Whereas this is not a mandatory characteristic, it aligns well with considerations such as security (e.g., protecting critical logic), flexibility (e.g., frequently changing calculations are centrally applied without updating all apps), and multi-device management (cf. Section 3.2).

Regarding related terminology, business apps are similar to business information systems (BISs) [dB14] but focus on mobile characteristics. Also, the Create/Read/Update/Delete (CRUD) principle may be associated with business apps [Ben+16] in that it focuses on the data-driven aspect. However, business apps do not just deliver a data management front-end but are capable of supporting business processes with custom-designed workflows, UIs, and business logic.

The superordinate term business process management (BPM) – i.e., concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes [Wes12] – encompasses business app implementations but additionally addresses the overall management and monitoring of such processes [van09]. Workflow management (WFM) system or process-aware information system (PAIS) are commonly used terms to denote the digital management and execution of workflows [van09]. In contrast, the emphasis of business apps does not lie on the management of complex workflows – potentially including the orchestration of automated web services – but on the coordination of distributed processes executed by humans. Also, the mobile domain exhibits different characteristics than centralised workflow management systems which typically operate on large-scale processes accessible through browsers in always-on settings and possibly with parallel streams of execution (cf. Subsection 4.1.3).

Multiple model-driven approaches to business apps have been proposed in the scientific literature. A detailed survey can be found in [UB16]. Only a few of them, such as Mobl [HV11], MobiDSL [Kej09], PIMAR [Vau+14], Xmob [LW13], or AXIOM [JJ14] cover the full spectrum of runtime behaviour and structure of an app; often providing a custom textual DSL for this means. Although the technical foundation of proprietary solutions cannot be assessed, configuring the parameters of app views within a more or less detailed emulator is also a form of graphical modelling. Companies considered leaders in the field of *low-code* or *no-code* app development [Res+18] such as OutSystems [Out19], Kony [Kon19] and further commercial products such as BiznessApps [Biz16] and Bubble [Bub19] promise codeless creation of apps using web-based user interface editors and templates. Other tools [Pro16] focus on individual components such as back-end systems or content management, or are limited to specific development phases such as prototyping. The commercial WebRatio Mobile Platform [Ace+15] is closest to our work by generating apps from graphically edited models for describing the data layer (using UML class diagrams), interactions (using IFML), and business logic (using a custom notation). However, our approach to graphical app development presented in Section 4.3 abstracts from concrete interface specifications by providing a DSL with a significantly higher level of abstraction representing the logical flow of actions.

4.1.3 Process Modelling Notations

Despite reducing development effort and complexity through domain-specific concepts, textual DSLs often feel like programming to non-technical users [Żył15]. Also, technical modelling notations are not easy to understand for domain experts with knowledge of the field of application [Fra+06]. In order to include those stakeholders and potential end users in the development of the app, graphical modelling notations need to be considered [ROS14; Nam+16].

Graphical notations for supporting business operations have been developed for several purposes, including business process compliance [Knu+13] and data integration [Hit19]. Specifically for mobile business applications, only few approaches exist. AppInventor targets programming

novices to create mobile applications by using a visual programming language of composable building blocks [Wol11], and Puzzle enables visual development within a mobile environment itself [DP12]. Furthermore, RAPPT represents a model-driven approach that mixes a graphical DSL for process flows with a textual DSL for programming business logic [Bar+15]. Although all of them aim at simplifying the actual programming work, they are often designed for – possibly novice – software developers and neglect non-technical stakeholders.

In contrast, process flows can be visualised using various general-purpose notations, for example, Business Process Model and Notation (BPMN) [Obj11], event-driven process chains (EPCs) [van99], YAWL [vt05], and flowcharts [Int85]. As analysed by Schalles [Sch13], notations describing behavioural aspects are generally more usable than system modelling approaches that focus on structural concepts. However, process modelling notations often lack the technical specificity required for automated processing. For example, the YAWL notation is constructed according to workflow management patterns but initially did not provide a data perspective and does not support the specification of task types that could be used for automated view templates [vt05]. The UML is the most prominent example of graphical modelling notations for software developers, providing a set of standards to define the structure and runtime behaviour of applications but requires additional models to cover all perspectives of an application [Obj15b]. In particular, the IFML notation (succeeding the previous WebML notation) allows for the specification of user interactions within a software system [Obj15a].

Domain-specific examples of process modelling for (mobile) applications are scarce. By integrating BPMN models with additional elements for representing the UI, Trætteberg and Krogstie [TK08] cover different application perspectives of desktop applications. Other approaches use existing modelling notations such as statecharts [Fra+15] or extend them for mobile purposes, e.g., UML to model contextual information in mobile distributed systems [SW07], or IFML with mobile-specific elements [BMU14].

4.2 MD²

The MD² framework is a model-driven approach to cross-platform app development and initially focused on smartphone and tablet devices. Using an Xtext-based textual DSL with a platform-agnostic syntax, business apps can be described and native source code is generated for the Android and iOS target platforms together with a central JavaEE-based back-end component [HMK13].

4.2.1 Language Overview

The MD² DSL is structured along the Model-View-Controller (MVC) design pattern [Gam+95] to specify a data model, view elements, and business logic, as depicted in Figure 4.1 [ME15].

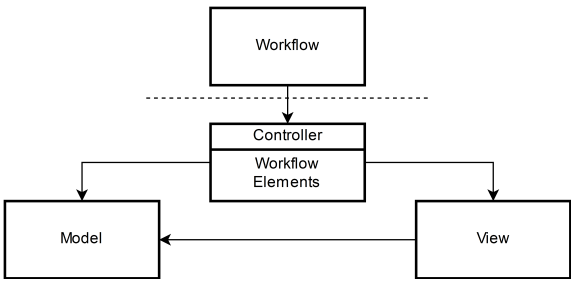


Figure 4.1: Architecture of MD² Models [RWK18]

Model *Entities* as main language elements are defined within the model layer. They can contain single- or multi-valued attributes of pre-defined data types such as `String`, `Integer`, and `Date` as well as references to other entities. Also, *enumeration* types can be defined.

View Concrete view definitions consist of declaratively described *content elements* such as input fields, labels, and buttons. These can be organised in nestable *container elements* such as scrollable linear layouts, grids, or tabs. Additional parametrisation allows for styling of elements and input validation.

Controller The controller layer handles the behaviour of the app using *events* and *actions* as central elements. Actions are defined to, e.g., transition between views, perform CRUD operations on the data, or access sensor data. These actions can be dynamically bound to different UI events, global events (e.g., network connection loss), or custom-defined conditions. Moreover, the controller maps view elements to content providers which encapsulate data objects of a certain entity type.

Workflow As the framework evolved, an additional workflow layer was added. It decouples the view-specific business logic from the more abstract domain concept of workflow elements to simplify the specification of app product lines and enable interactions across apps through user roles. Therefore, the modeller can define *workflow elements* to group sequences of view into logical process steps. *Workflow events* then trigger transitions between workflow elements and potentially distribute the execution to other users when role changes occur [Dag+16].

While creating apps, the models are continuously validated for syntactic and semantic correctness, and contextual feedback is provided in the IDE. Subsequently, a preprocessor component resolves, e.g., shorthand definitions, and simplifies the models [ME15]. For example, an *AutoGenerator* parser rule allows specifying just an entity type (more specifically, a *content provider* for an entity) from which a suitable view specification is inferred according to the contained attributes and their data types [HMK13]. The preprocessor removes these abstractions and replaces them with concrete view elements. Finally, the preprocessed models are passed to the code generators written in Xtend [Ecl19f] to create the respective native Java (Android) and Swift (iOS) source

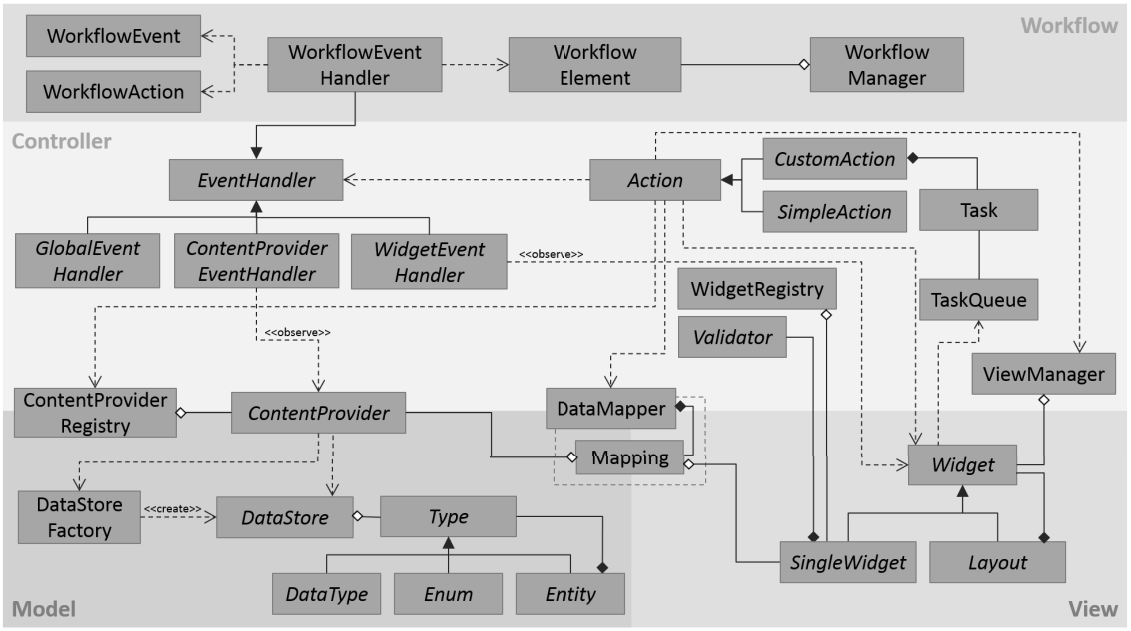


Figure 4.2: Revised Reference Architecture [Ern+16]

code. Code generation was chosen to avoid a performance overhead of runtimes interpreting models (cf. Section 3.1) and achieve a native look and feel adapted to UI guidelines and typical interaction patterns [MBG18].

4.2.2 Revised Reference Architecture for Business Apps

A common problem of model-driven software development is the "right" level of abstraction. Particularly for the view specification, abstract modelling is often limited to the lowest common denominator of all targeted platforms' features. At the same time, extensive knowledge of the platform-specific ecosystem (e.g., programming environment and libraries) is required by the framework maintainers in order to develop the necessary transformations. Evers et al. [EEM16] proposed a reference architecture to handle repeating architectural patterns and the complexity of mobile app components across multiple platforms with the purpose of reducing the development effort of new generator implementations.

In accordance with the process model for reference architecture creation presented by Nakagawa et al. [Nak+14], we revised this initial reference architecture through *information source investigation*, *architectural analysis*, *architectural synthesis*, and *architectural evaluation*. Based on feedback from several – albeit mostly student – developers working on MD², drawbacks of the rigid structure of the original reference architecture were identified. In addition, two implementations were analysed with regard to the formerly unspecified view layer. As a result, the reference architecture was improved regarding its *structural* characteristics and a *behavioural* perspective on interaction patterns within the app, as shown in Figure 4.2.

With regard to the overall structure, the additional workflow layer in MD² is now reflected in the MVC-architecture of apps. To deal with workflow paths of consecutive activities, a `WorkflowManager` component is introduced which integrates with the general architecture by emitting workflow events and can be controlled via a new type of action called `SetWorkflowElementAction`. Besides local handling of workflows, this component also transfers the state of a workflow that needs to be continued in another app.

Second, after analysing the major smartphone operating systems Android and iOS and applying the reference architecture to these platforms, commonalities could be identified to fill the missing view layer specification. The previous concept of `WidgetWrappers` serving as a common interface for all widgets proved to be inflexible in practice and caused unnecessarily complex structures (e.g., Android allows customising existing system-provided widgets directly). Therefore, enforcing too specific patterns is not advisable and the platform-specific transformation should choose an appropriate mode of interaction between controller logic and view elements.

Third, custom actions consist of atomic instructions such as data bindings which were directly translated into source code. However, adding these low-level *tasks* as separate objects in the resulting architecture has benefits concerning the platform-specific view life cycle. The Android platform, for instance, utilises built-in memory management which is allowed to free memory allocated to *activities* – Android's notion of views – which are invisibly paused while another activity is displayed. Although widgets are instantiated when an activity is created, it is not guaranteed that an object continuously exists in memory but it may be recreated on demand. Consequently, executing tasks which, e.g., register event listeners, may fail or require workarounds which are prone to memory leaks if not thoughtfully managed. Therefore, a `TaskQueue` component is added to the reference architecture as temporary storage for tasks that cannot be executed directly and which are triggered as soon as the target object is available (e.g., when an activity is reactivated).

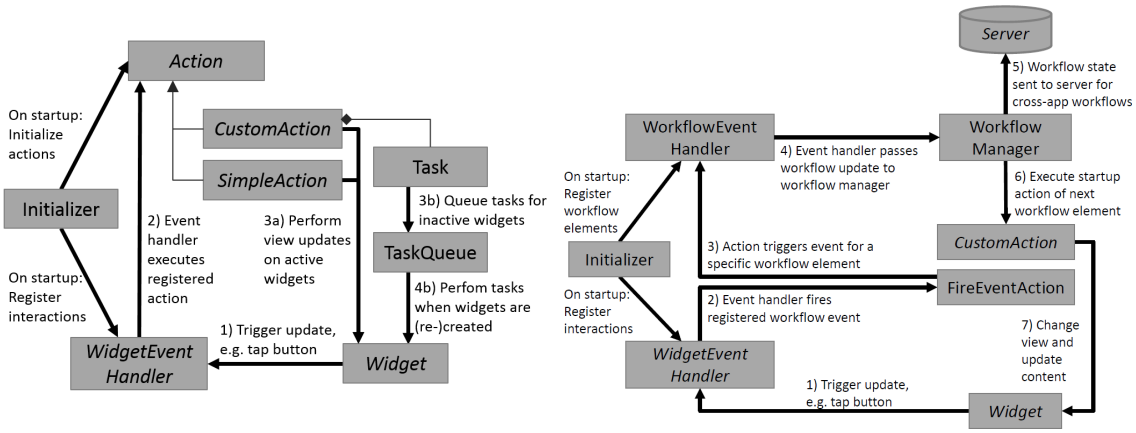
Over-generalisation is a problem when designing and applying a reference architecture for cross-platform purposes in a top-down fashion without considering the variability of target platforms. As a result, appropriate platform implementations are hampered by tedious re-implementations of existing features and harder to debug through unnecessarily complex object structures. Therefore, a reference architecture should be permissive with regard to different methods of adapting the concepts to platform-specific programming patterns. To support the varying availability of language constructs, three techniques for object-oriented programming languages are applicable to the reference architecture [FMO11]:

- *Glue* interfaces bridge potentially incompatible code parts by specifying an interface towards the rest of the application [FR07]. In general, this applies to all components of the reference architecture in Figure 4.2. As long as the interface towards the remaining objects is fulfilled, classes are not necessarily required. This leaves the flexibility to use unique platform capabilities or create custom adapters that delegate the desired actions. For example, the

`DataMapper` component may use platform-provided data structures for efficient bidirectional mapping between model entities and view elements to reduce development time but other implementations are likely suitable.

- In many cases, *inheritance* of existing classes is viable to reuse functionality of the platform and implement only the missing operations in order to comply with the reference architecture specification. For example, some platforms allow extending the native event system with custom events to limit implementation overhead, whereas others (e.g., the Swift programming language) requires custom implementations for event-related components.
- Also, *overloading* method implementations and using generic classes can be used to implement components within the reference architecture when supported by the platform.

These structural choices offer more flexibility for developers to align with platform-specific patterns. In addition, the interactions within components of the architecture are clarified in the revise reference architecture according to three control flow and data flow loops as depicted in Figures 4.3 and 4.4 [Ern+16]:



(a) Widget control flows

(b) Workflow control flows

Figure 4.3: Interaction Diagrams for Control Flows Within the Reference Architecture [Ern+16]

Widget control flow As explained before, business logic and view changes are modelled as actions. On start-up, the app `Initializer` component (cf. Figure 4.3 (a)) needs to register events and actions before displaying the initial view. Subsequently, the event-action loop is the backbone of the application. Widget changes are observed through gesture or value change listeners, and respective event handlers execute the specified actions such as view transitions. If the action target is an inactive view element, the respective tasks are queued in the `TaskQueue` component.

Workflow control flow Besides low-level view updates, the business process represented by workflow elements also integrates with the event-action loop (Figure 4.3 (b)). If a

widget event triggers a so-called `FireEventAction`, the workflow event handler notifies the `WorkflowManager` component to choose the next element along the workflow execution path. Depending on the user's role, the app either continues locally or the workflow instance is passed to another app. For local workflow steps, a view transition is called and the regular widget control flow takes over for executing the view's start-up actions.

Data flow In addition, data actions are also integrated into the event system (Figure 4.4). Whenever a data field is manipulated, input validation is applied to ensure data correctness and avoid unnecessary data flow cycles. If the input value is new and valid, the responsible event handler notifies the content provider about the changes. Because multiple UI elements can be bound to an entity attribute, the content provider also emits an update event which is propagated to all widgets registered in the `ContentProviderEventHandler`. Simultaneously, the data is stored in the local data storage on the device and – if modelled using a remote data store – sent to the back-end server. Similarly, the cycle can be started through an external data change which updates the data store and notifies content providers and widgets accordingly.

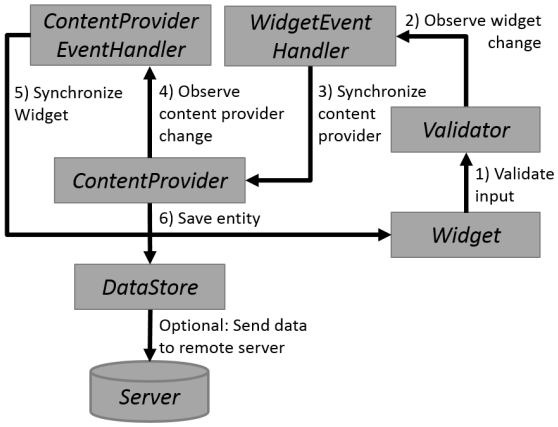


Figure 4.4: Interaction Diagrams for Data Flows Within the Reference Architecture [Ern+16]

To sum up, the improved reference architecture serves as a solid foundation for the development of business apps across different platforms and covers both structural and behavioural aspects of the apps. The architecture helped to develop the current generator implementations of MD². Although designed with smartphone platforms in mind, the same patterns can also be applied to the WearOS smartwatch generator as will be explained in Section 4.4.

4.3 MAML

Whereas MD² and similar textual DSLs significantly improve app development efficiency, the creation process still targets users with programming skills [MHS05]. However, business apps

typically focus on tasks to be accomplished by employees with deep domain insights and knowledge of process variations. Applying traditional software development practices may fall short of optimally supporting end users when software built according to centrally defined processes does not adequately reflect the actual process execution. By giving end users the opportunity to shape the software they use in their everyday work routines, continuously co-designing such systems may increase the adoption of the resulting application and possibly strengthen their identification with the company [Ful+06]. Mobile app development can bridge the organisational separation of IT and business departments to benefit from the incorporation of people from all levels of the organisation. Consequently, development tools should be understandable to both programmers and domain experts.

According to a prediction by the research company Gartner in 2014, more than half of all company-internal mobile apps would be built using codeless tools by 2018 [Rv14]. Whereas this seems overly optimistic in retrospect, a general trend towards *low-code* or *codeless* development is apparent by the increase of graphical tools to create business apps. However, current approaches often lack the functionality for holistic app modelling, either operating on a low level of abstraction using GUI builders or overly simplifying development by composing pre-defined view templates (e.g., [Mic19; Goo19a]).

In order to advance research on cross-platform development of mobile apps and investigate opportunities for digitising data-driven processes in organisations, we developed the Münster App Modeling Language (MAML) framework. Contrary to existing notations, MAML models have sufficient information to transform them into fully functional mobile apps. Besides the DSL grammar defined as an Ecore metamodel, MAML also comprises the necessary development tools to design models in a graphical editor built using the Sirius framework. In contrast to other frameworks such as the WebRatio Mobile Platform [Web19], all of the modelling work for MAML is performed in a single type of model, mainly by dragging elements from a palette and arranging them on a large canvas. The modelling environment was developed using the Eclipse Sirius framework [Ecl19e]. For generating source code, the input models are preprocessed and transformed into the MD² language (cf. Section 4.2) to reuse existing code generators for the creation of Android and iOS apps for smartphones and Wear OS smartwatches. This intermediate step is, however, automated and requires no intervention by the user.

The graphical DSL for MAML is based on five design goals [Rie18a]:

Automatic cross-platform app creation Most important, the whole approach is built around the key concept of codeless app development. To achieve this, individual models need to be recombined and split according to different roles (cf. Section 4.4) and transformed into platform-specific source code. As a consequence, models need to encode technical information such as data types and interrelations between workflow elements in a machine-interpretable way as opposed to an unstructured composition of shapes filled with text.

Domain expert focus MAML is explicitly designed with non-technical users in mind. Process modellers as well as domain experts are encouraged to read, modify, and create new models by themselves. The DSL should, therefore, not resemble technical specification languages drawn from the software engineering domain but instead provide generally understandable visualisations and meaningful abstractions for app-related concepts.

Data-driven process modelling The fundamental idea of business apps to focus on data-driven processes determines the level of abstraction chosen for MAML. In contrast to merely replacing manually programmed UIs with a visual editor for screen composition, MAML models represent a substantially higher level of abstraction. Users of the language concentrate on visualising the sequence of data processing steps. The concrete representation of affected data items is then automatically generated using adequate input/output UI elements.

Modularisation To engage in modelling activities without advanced knowledge of software architectures, appropriate modularisation is important to handle the complexity of apps. MAML embraces the aforementioned process-oriented approach by modelling use cases, i.e., units of functionality representing a self-contained set of behaviours and interactions performed by the app user [Obj15b]. Combining data model, business logic, and visualisation in a single model deviates from traditional software engineering practices which, for instance, often rely on the Model-View-Controller pattern [Gam+95]. But in accordance with the domain expert focus, the end user is unburdened from this technical implementation issue.

Declarative description MAML models consist of platform-agnostic elements, declaratively describing *what* activities need to be performed with the data instead of *how* to layout arbitrary UI elements. The concrete representation in the resulting app is deliberately unspecified to account for different device capabilities and usage patterns of each targeted platform. The respective code generator can provide sensible defaults for such platform specifics.

4.3.1 Language Overview

In the following, key concepts of the MAML DSL are highlighted using the fictitious scenario of a publication management app. An exemplary process to add a new publication to the system consists of three logical steps: First, the researcher enters some data on the new publication. Then, he can upload the full-text document and, optionally, revise the corresponding author information. Although potentially executed by different users, this self-contained set of activities is represented as one model in MAML, the so-called use case, as depicted in Figure 4.5 [Rie17].

A model consists of a *start event* (labelled with (a) in Figure 4.5) and a sequence of process flow elements towards an *end event* (b). A *data source* (c) specifies what type of entity is first used in the process and whether it is only saved locally on the mobile device or managed by the remote

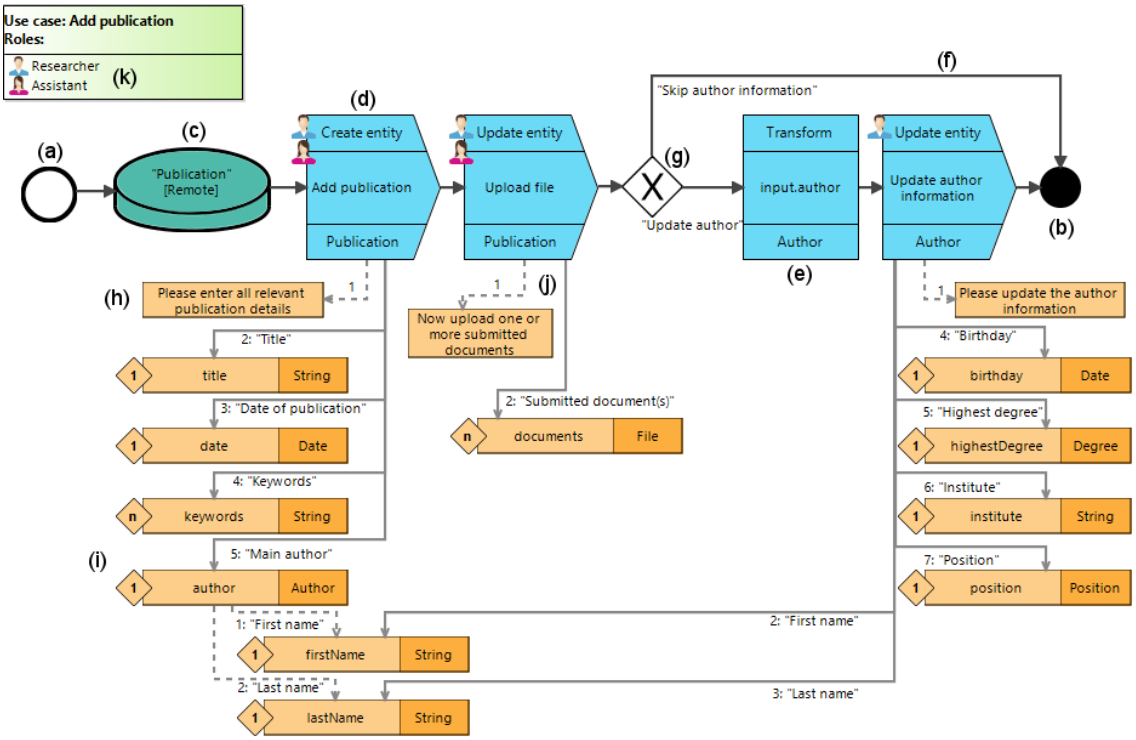


Figure 4.5: MAML Use Case for Adding a Publication to a Review Management System [Rie17]

back-end system. Then, the modeller can choose from pre-defined *interaction process elements* (d), for example, to *create/show/update/delete* an entity, but also to *display messages*, access device sensors such as the *camera*, or *call* a telephone number. Because of the declarative description, no device-specific assumptions can be made on the appearance of such a step. The generator instead provides default representations and functionalities, e.g., display a *select entity* step using a list of all available objects as well as possibilities for searching or filtering. In addition, *automated process elements* (e) represent steps to be performed without user interaction. Those elements provide the minimum amount of technical specificity in order to navigate between the model objects (*transform*), request information from *web services*, or *include* other models to reuse existing use cases.

The order of process steps is established using *process connectors* (f), represented by a default "Continue" button unless specified differently along the connector element. *XOR* (g) elements branch out the process flow based on a manual user action by rendering multiple buttons (see differently labelled connectors in Figure 4.5), or automatically by evaluating expressions referring to a property of the considered object.

The lower section of Figure 4.5 contains the data linked to each process step. *Labels* (h) provide explanatory text on the screen. *Attributes* (i) are modelled as the combination of a name, the data type, and the respective cardinality. In MAML, only those attributes are specified which will be used in a particular process step, e.g., used for the user interface or in web service calls. Although

this concept seems surprising to programmers expecting fully defined data models, it simplifies the integrated modelling of process flows and data. Attributes which are neither updated nor displayed do not contribute to the resulting app and are, therefore, ignored during modelling[1].

Data types such as `String`, `Integer`, `Float`, `PhoneNumber`, `Location`, etc. are already provided but the user can define additional custom types. To further describe custom-defined types, attributes may be nested over multiple levels (e.g., the "author" type in Figure 4.5 specifies a first name and last name). In addition, *computed attributes* (not depicted in the example) allow for runtime calculations such as counting or summing up other attribute values.

A suitable UI representation is automatically chosen based on the type of *parameter connector* (j): Dotted arrows signify a reading relationship, whereas solid arrows represent a modifying relationship. This refers not only to the manifest representation of attributes displayed either as read-only text or editable input field. The interpretation also applies in a wider sense, e.g., regarding web service calls in which the server "reads" an input parameter and "modifies" information through its response. Each connector also specifies an order of appearance and, for attributes, a human-readable caption which is derived from the attribute name unless manually specified.

Finally, annotating freely definable *roles* (k) to all interactive process elements allows for the coherent visualisation of processes that are performed by more than one person, for example, in scenarios such as approval workflows. When a role change occurs, the app automatically saves modified data and users with the subsequent role are informed about the open workflow instance in their app.

4.3.2 Data Model Inference

As described in the previous paragraphs, each process step within a model refers only to the attributes required as input or output of this particular step. Also, attributes may be connected to multiple process elements simultaneously or can be repeated at different positions to avoid wide-spread connections across the model. This effectively creates a multitude of partial data models. Nevertheless, a global data model of all data structures is a technical requirement for the generated application. Consequently, an inference mechanism is necessary to aggregate the scattered information and ensure consistency between partial models on multiple levels [Sut+16; EI10]: for each process element individually, then for the whole use case, and finally across multiple use cases as described next [Rie16; Rie17].

To infer a global view from multiple partial models, a schema-only and constraint-based approach on element level is suitable according to the classification by Rahm [RB01]. We focus on an additive proceeding which matches elements by name, although more sophisticated strategies such as fuzzy or ontology-based approaches may later identify "similar" attributes and uncover further modelling inconsistencies [LGJ07]. Similar inference problems are encountered

[1] Such attributes may, however, be required for manually written business logic after the code generation step. In this case, respective attributes can be likewise introduced in the generated source code.

when reverse engineering evolving metamodels [Liu+12] or searching for correspondences within models [Jav+08]. López-Fernández et al. [Lóp+15] proposed a related approach of building a metamodel in a bottom-up fashion from multiple model examples. The challenge for MAML data models lies in the unidirectional specification of relationships, i.e., a process element with a given data type "has a" relationship to an *attribute* of a specified type and cardinality, but there is mostly no explicit information on the opposite relationship.

Let M denote the set of use case models for which a coherent data model should be inferred. Also, D_m denotes the set of data types appearing within a concrete model $m \in M$. Then,

$$R_m \subset D_m \times String \times D_m \times \mathscr{P}(\mathbb{N}) \times \mathscr{P}(\mathbb{N})$$

denotes the set of modelled *relationship* tuples R_m between two data types. Within such a tuple $r_i = (s_i, rn_i, t_i, sc_i, tc_i)$, the value s_i represents the source data type of the relationship, rn_i the corresponding name, and t_i the target data type. A *cardinality* represents the possible number of values referred to by a relationship [Obj15b], using the notation $i..j = \{i, i+1, ..., j\}$ and n represents infinity[2]. sc_i signifies the source cardinality of the relationship and rc_i the respective target cardinality. Consequently, the annotated, directed graph $G_m := (D_m, R_m)$ represents the data model of m with data types as vertices and relationships as edges. In particular, the graph may contain multiple edges between the same pair of source and target data types, if the relationship names, the source cardinalities, or the target cardinalities differ.

In addition, let V_m denote the set of primitive types within a concrete model $m \in M$, representing atomic values which do not contain any relationship to other data types. Then,

$$P_m \subset D_m \times String \times V_m \times \mathscr{P}(\mathbb{N})$$

represents the set of *property* tuples. Accordingly, for a tuple $p_i = (ps_i, pn_i, pt_i, pc_i)$, the value ps_i represents the source data type that contains a property of the primitive type pt_i with the name pn_i and cardinality ps_i.

In MAML, a pre-defined list of primitive types (`Integer`, `Float`, `String`, `PhoneNumber`, `Location`, `Boolean`, `Date`, `Time`, `DateTime`, and `File`) is provided from which custom data types can be composed. A custom data type is specified either within the data source element, for a process element, or as part of an attribute element. For example, Figure 4.5 depicts the visual representations of a remote data source referring to the data type "Publication" (c), a "create entity" interaction process element for the same data type (d) as well as an attribute for the data type "Author" (i).

Interrelations of data types can then be observed either between a process element and an attached attribute (marked with (1) in Figure 4.6), between an attribute's custom data type and a nested attribute (2), or a transitive connection from a process element to an attribute through one or more computed attributes (e.g., (3) in Figure 4.6 which represents a "count" aggregation

[2] In MAML models, **1** specifies a 0..1 cardinality and **n** refers to 0..*n*.

operator). MAML does not differentiate between primitive types and data types with regard to their representation in the model. Therefore, the aforementioned options might translate to either a *relationship* or *property* of the originating data type.

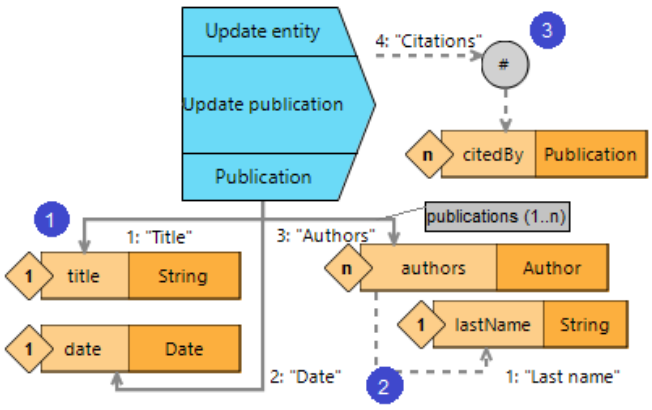


Figure 4.6: Attribute Relation Options in MAML Models (cf. [Rie16])

The cardinalities which can initially be assigned to a relationship depend on the type of association. Four main cases can be distinguished:

- **Primitive:** MAML attributes representing a primitive data type are directly converted into single- or multi-valued *properties* of the source data type. For example, the connection (2) in Figure 4.6 represents the property tuple $(Author, "lastName", String, 0..1) \in P_m$.
- **Unidirectional:** Each *relationship* is by default unidirectional and specifies a name and a cardinality. The source cardinality is unknown and, therefore, temporarily interpreted as unrestricted with $0..n$. In the example of Figure 4.6, the connection (3) translates to the tuple $(Publication, "citedBy", Publication, 0..n, 0..n) \in R_m$.
- **Bidirectional:** Bidirectional relationships between data types d_1 and d_2 are explicitly specified in the graphical model and provide both source and target cardinalities. In MAML, this is represented by an additional annotation (containing the name and cardinality for the opposite direction) along the connector, e.g., the "authors" relationship in Figure 4.6. To capture the bidirectional navigability in the graph, a second relationship is inserted in R_m with inverted order of source and target data types and cardinalities.
- **Singleton:** Data types of which only one element can be instantiated [Gam+95] are a variant of the unidirectional scenario. However, the otherwise unknown cardinality can be restricted to 0..1 for singletons. In MAML, they are created using a special singleton data source element within the process flow.

Two models are considered *compatible* if the combined constraints of both models for the data type, name, and cardinality consistency are satisfiable (cf. Subsection 4.3.3). For instance, the graph structure of two compatible MAML models in Listing 1 is visualised in Figure 4.7.

Formally, partial data models can be merged into a global data model G_g using an associative and commutative merging operation – allowing for iterative or simultaneous aggregation of all considered models – before identifying modelling inconsistencies as described in Subsection 4.3.3.

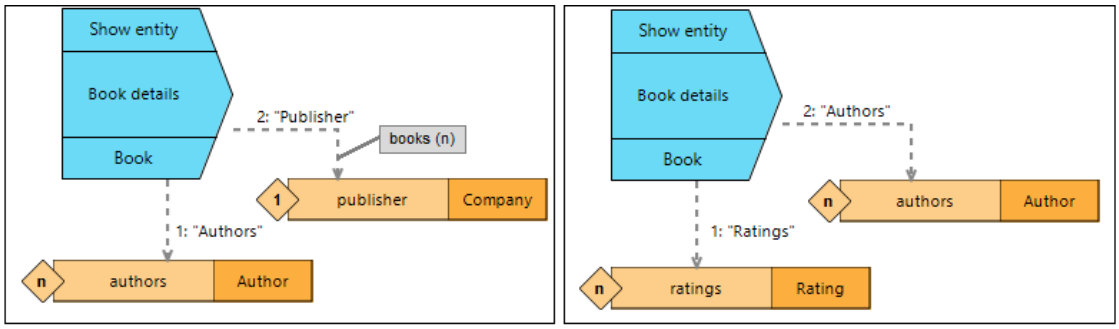


Figure 4.7: Compatible Partial MAML Models (cf. [Rie16])

Listing 1 Exemplary Data Structure for Two Compatible Models

$M = \{m_1, m_2\}$

▷ Data types

$D_{m_1} = \{Book, Author, Company\}$ $D_{m_2} = \{Book, Rating, Author\}$
$V_{m_1} = V_{m_2} = \emptyset$

▷ Properties

$P_{m_1} = P_{m_2} = \emptyset$

▷ Relationships

$r_1 = (Book, "authors", Author, 0..n, 0..n)$ $r_4 = (Book, "ratings", Rating, 0..n, 0..n)$
$r_2 = (Book, "publisher", Company, 0..n, 0..1)$ $r_5 = (Book, "authors", Author, 0..n, 0..n)$
$r_3 = (Company, "books", Book, 0..1, 0..n)$
$R_{m_1} = \{r_1, r_2, r_3\}$ $R_{m_2} = \{r_4, r_5\}$

First, all custom and primitive data types can trivially be aggregated from the partial graphs:

$$D_g = \bigcup_{m \in M} D_m \tag{4.1}$$

$$V_g = \bigcup_{m \in M} V_m \tag{4.2}$$

Second, relationships and properties from each partial model m are added to R_g if there is yet no relationship between both data types with a given name or cardinality. Because R_m and P_m are also sets of tuples, this is equivalent to taking the union of all relationship and property sets of the source models, respectively:

$$R_g = \bigcup_{m \in M} R_m \tag{4.3}$$

$$P_g = \bigcup_{m \in M} P_m \tag{4.4}$$

Applied to the example, the relationship r_5 is equivalent to r_1 and, therefore, ignored. The resulting graph structure is presented in Listing 2 and depicted as a UML class diagram in Figure 4.8.

Listing 2 Merged Data Structure of the Two Compatible Models in Listing 1

$D_g = \{Book, Author, Company, Rating\}$
$V_g = \emptyset$
$P_g = \emptyset$
$r_1 = (Book, "authors", Author, 0..n, 0..n)$ $\quad r_3 = (Company, "books", Book, 0..1, 0..n)$
$r_2 = (Book, "publisher", Company, 0..n, 0..1)$ $\quad r_4 = (Book, "ratings", Rating, 0..n, 0..n)$
$R_g = \{r_1, r_2, r_3, r_4\}$

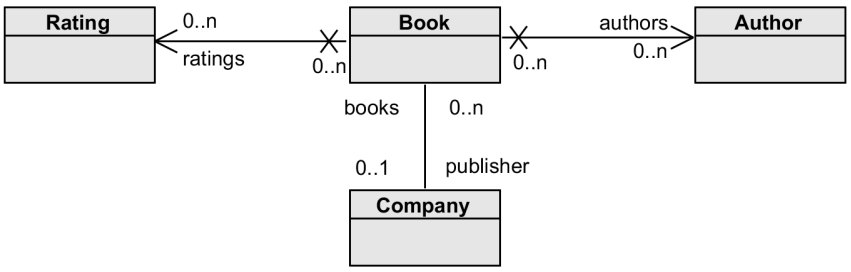


Figure 4.8: UML Class Diagram of the Merged Model (cf. [Rie16])

4.3.3 Identifying Modelling Inconsistencies

Whereas the previous merging step has aggregated all partial models, the resulting global data model might be invalid. For example, both partial models in Figure 4.9 are valid on their own and can be merged according to equations (4.1) to (4.4). However, they are called *conflicting* because merging those models results in a global model that is semantically invalid. Generally, three types of modelling errors can be observed [Rie16; RK18a]:

Name conflicts exist if the same name is assigned more than once to relationships and properties for the same source data type, violating

$$\forall r_i \in R_g, p_j \in P_g \mid s_i = ps_j : \qquad rn_i \neq pn_j \tag{4.5}$$

Type errors occur if any source data type in the graph has two properties or relationships of the same name pointing to different target data types, i.e., violating

$$\forall r_i, r_j \in R_g \mid s_i = s_j, \;\; rn_i = rn_j : \qquad t_i = t_j \tag{4.6}$$

$$\forall p_i, p_j \in P_g \mid ps_i = ps_j, \;\; pn_i = pn_j : \qquad pt_i = pt_j \tag{4.7}$$

Cardinality conflicts exist if two *relationships* with the same name differ with regard to their modelled cardinalities for any pair of data types, i.e., not conforming to any of the following:

$$\forall r_i, r_j \in R_g \mid s_i = s_j,\ \ rn_i = rn_j,\ \ t_i = t_j : \qquad sc_i = sc_j \tag{4.8}$$

$$\forall r_i, r_j \in R_g \mid s_i = s_j,\ \ rn_i = rn_j,\ \ t_i = t_j : \qquad tc_i = tc_j \tag{4.9}$$

A cardinality conflict also exists if two *properties* with the same name differ with regard to their cardinalities for any pair of primitive types, i.e., not conforming to

$$\forall p_i, p_j \in P_g \mid ps_i = ps_j,\ \ pn_i = pn_j,\ \ pt_i = pt_j : \qquad pc_i = pc_j \tag{4.10}$$

Type errors and name conflicts according to equations (4.5) to (4.7) cannot be resolved automatically and need to be corrected by the modeller. For instance, the inference mechanism cannot autonomously decide whether the ambiguous target data type of the "publisher" attribute in Figure 4.9 should be set to "Company" or "Person".

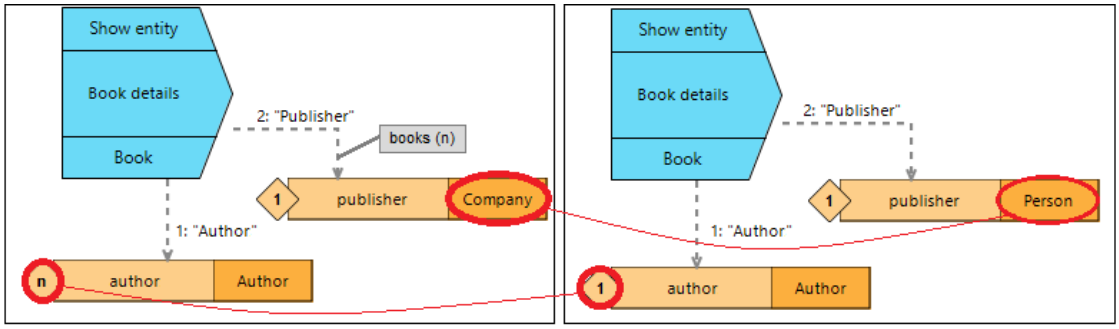


Figure 4.9: Conflicting Partial MAML Models (cf. [Rie16])

In the case of cardinality conflicts (i.e., relationships not satisfying equations (4.8) to (4.10)), the modeller should be warned, but an automatic resolution is possible. For each pair of relationships $r_i, r_j \in R_g$ with matching source data type, target data type, and name, the cardinality for each side of the association can be calculated as the union between the conflicting cardinalities to avoid invalidating existing data. The cardinality is calculated accordingly for each pair of properties $p_i, p_j \in P_g$ with matching source data type, target data type, and name. For the example of Figure 4.9, the target cardinality $0..n \cup 0..1 = 0..n$ is assigned to the "author" attribute.

As a result, the user does not need to specify a distinct global data model and consistency among each process step, and the use cases as a whole are automatically checked whenever models change to prepare for the subsequent code generation.

4.3.4 Model-to-Code Transformation

In addition to giving feedback on modelling errors, advanced modelling support attempts to provide guidance and overview to the user. For example, the current data type of a process element (lower label of (d) in Figure 4.5) is automatically derived from the preceding elements based on the inferred data model. Also, suggestions of probable values are provided when adding elements (e.g., known attributes of the originating type when adding UI elements) and the user is informed about potentially unwanted behaviour (e.g., from missing annotations). In order to generate apps, the proposed approach reuses previous work on MD² (see Section 4.1). The complete generation process is depicted in Figure 4.10.

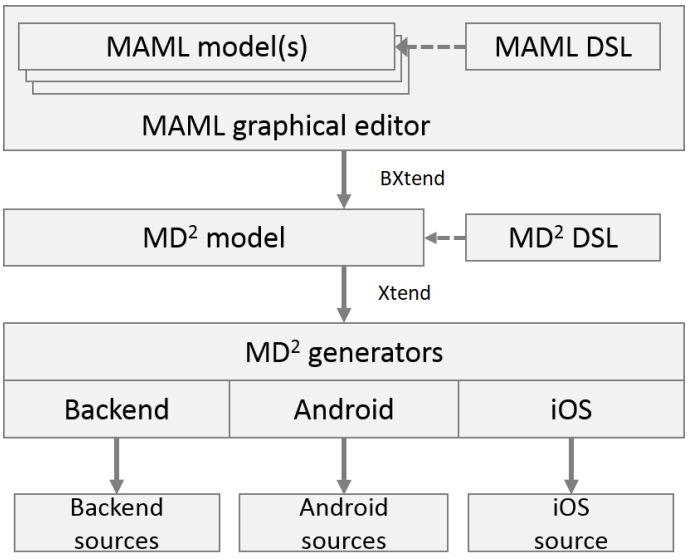


Figure 4.10: MAML App Generation Process (cf. [Rie18a])

First, model transformations are applied to transform graphical MAML models to the textual MD² representation using the BXtend framework [Buc18] and the Xtend programming language [Ecl19f]. Amongst other activities, all separately modelled use cases are recombined, the inferred global data model across all use cases is transformed into a dedicated persistence layer of data stores and content providers, and process flows are recombined according to the specified user roles. Second, the code generators of MD² are employed to create the actual source code for all supported target platforms. Optionally, a modeller has the possibility to configure the application in more detail within the intermediate MD² representation and modify default design decisions (e.g., view components) before the source code is generated for each platform. However, this two-step approach is not an inherent limitation of the framework. Newly created generators might just as well generate code directly from the MAML model or use interpreted approaches instead of code generation.

It should be noted that this proceeding differs from transformation approaches such as MDA

[Arco1] in that the intermediate representation is still a platform-independent representation but with a more technical focus.

Although the tooling around MAML is still in a prototypical state, it currently supports the generation of Android and iOS apps as well as a Java-based server back-end component. Also, a smartwatch generator for Google's Wear OS platform highlights the applicability to further device classes as discussed in Section 4.4. The screenshots in Figure 4.11 depict the generated Android app views for the first process steps of the MAML model depicted in Figure 4.5.

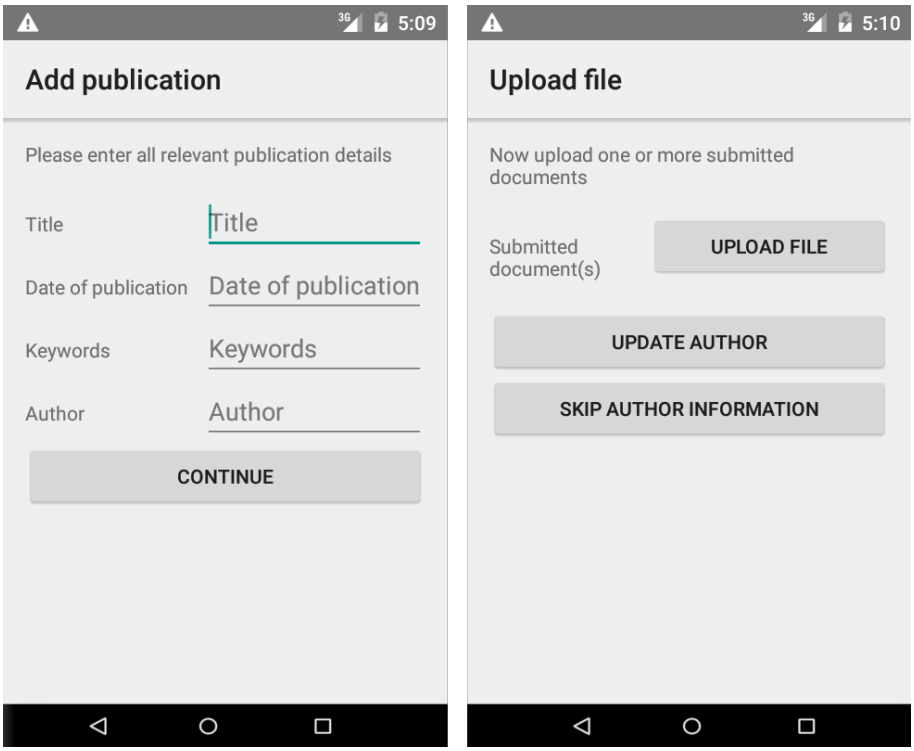


Figure 4.11: Exemplary Screenshots of Generated Android App Views (cf. [Rie18a])

4.3.5 Usability Evaluation

In order to assess MAML regarding its comprehensibility and usability, an observational study according to the think-aloud method was conducted (cf. Section 2.2). Within 26 individual interviews of around 90 minutes duration, participants performed realistic tasks with two notations while verbalising their actions, questions, problems, and general thoughts while executing these tasks. First, a MAML model and an equivalent IFML model were presented to the participants – in random order to avoid bias from priming effects [BCoo] – to assess the comprehensibility of such models without prior knowledge (none of the participants had previous experience with either IFML or MAML) or introduction to a specific notation. To quantify the readability, the short SUS questionnaire (cf. Section 2.2) was filled out for each notation. Second, a 10-minute introduction to the MAML notation was given and four modelling tasks were performed. The standardised

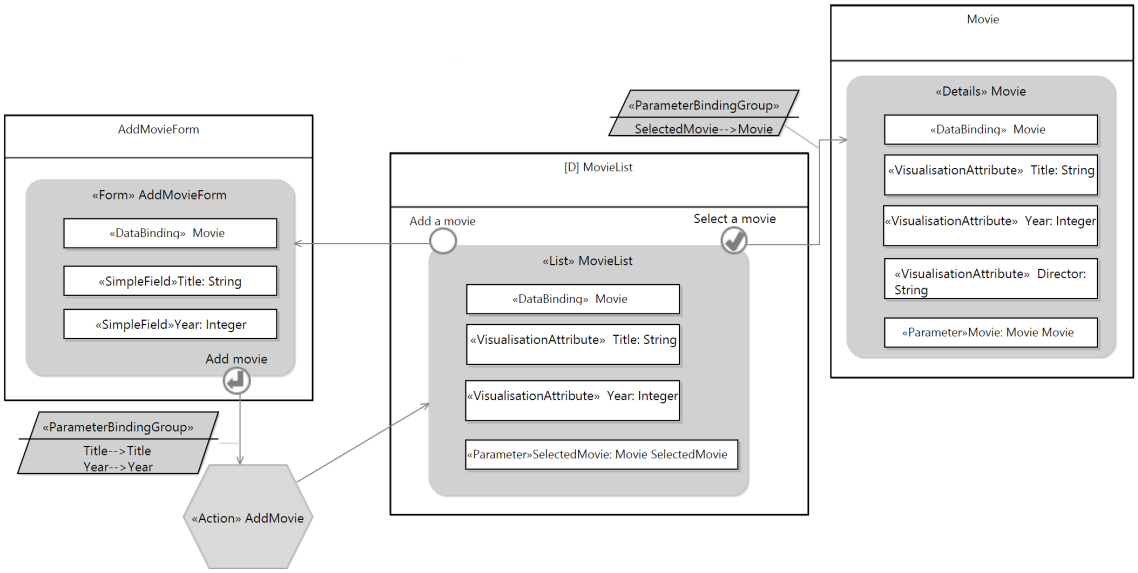


Figure 4.12: IFML Model to Assess the a Priori Comprehensibility of the Notation [Rie18a]

ISONORM questionnaire was used to collect more quantitative feedback. To capture the variety of possible usability issues, 71 observational features were identified and structured in six categories of interest: comprehensibility, applying the notation, integration of elements, tool support, effectiveness, and efficiency. In total, over 1500 positive or negative observations were recorded as well as additional usability feedback and proposals for improvement. To further analyse and interpret the results, the participants were categorised in three distinct groups according to their personal background stated in a preceding online questionnaire: 11 software developers have at least medium knowledge in traditional/web/app programming or data modelling, 9 process modellers have at least medium knowledge in process modelling (exceeding their programming skills), and 6 domain experts are experienced in the domain but have no significant programming or process modelling knowledge. According to Virzi [Vir92], five participants are sufficient to uncover 80% of usability problems. Whether this claim is debated (e.g., [SS01]), arguably the selected amount of participants in this study is reasonable with regard to evaluating the design goals and finding the majority of severe usability defects for MAML instead of a deep analysis of a production-ready software product.

Comprehensibility results

When assessing how well both notations are understood, the MAML model of the fictitious movie database (cf. Figure 4.5) was juxtaposed with the IFML model depicted in Figure 4.12. Participants were asked to describe the purpose of the overall models and the particular elements of the notation.

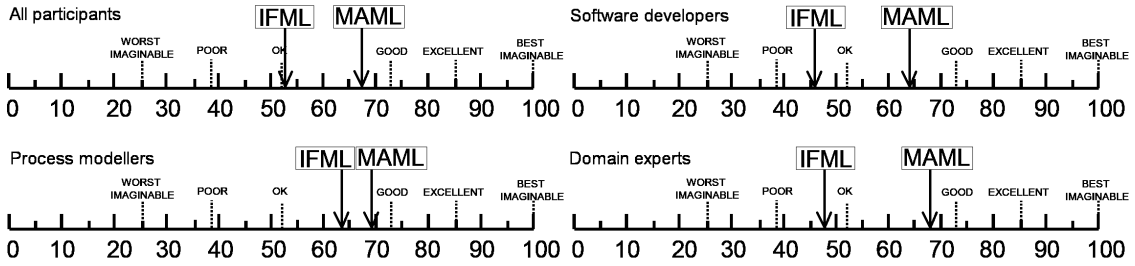


Figure 4.13: SUS Ratings for IFML and MAML (cf. [Rie18a])

The results of the SUS questionnaire depicted in Table 4.1 and Figure 4.13 show that MAML's scores are superior in total as well as for all three groups of participants individually. In addition, the consistency of scores across all groups supports the design goal of creating a notation which is well understandable for users with different backgrounds. As assumed, domain experts expressed the most drastic difference in comprehensibility of almost 20 points. Surprisingly, the absolute scores for IFML were worst for the group of software developers, although they are accustomed to other UML standards and diagrams. Reasons for the negative assessment of IFML can be found in the amount of "technical clutter", e.g., regarding parameter and data bindings, as well as perceived redundancies and inconsistencies at first sight.

Figure 4.14 provides further details on the distinction between domain experts and technical users (i.e., developers and process modellers together) by analysing the answers to the 10 questions of the SUS questionnaire (rescaled to a [0;4] interval; 4 denoting strong acceptance). With one exception, responses for MAML are higher than for the technical IFML notation. Moreover, domain experts reacted more positively when assessing the MAML notation as being widely usable (+1.17 compared to IFML), fast to learn (+1.17), and self-descriptive (+1.00). Consequently, the *understanding* of notational elements is a distinguishing factor for domain experts, which aligns well with the intention to use MAML for communicating with potential end users and closely integrate them in the app development process. From the perspective of technical users, the strongest deviations can be observed related to self-learnability (+1.05), perceived consistency (+0.80), and pace of learning (+0.70). Conforming with their technical background, these aspects emphasise the correct *application* of the notation which apparently is perceived as advantageous in MAML, too.

Table 4.1: System Usability Scale Comparison Between IFML and MAML

SUS ratings	**IFML**	**MAML**
All participants	52.79 (σ = 23.0)	66.83 (σ = 15.6)
Software developers	45.91 (σ = 23.6)	64.09 (σ = 17.3)
Process modellers	64.17 (σ = 19.0)	69.44 (σ = 12.0)
Domain experts	48.33 (σ = 24.5)	67.92 (σ = 18.7)

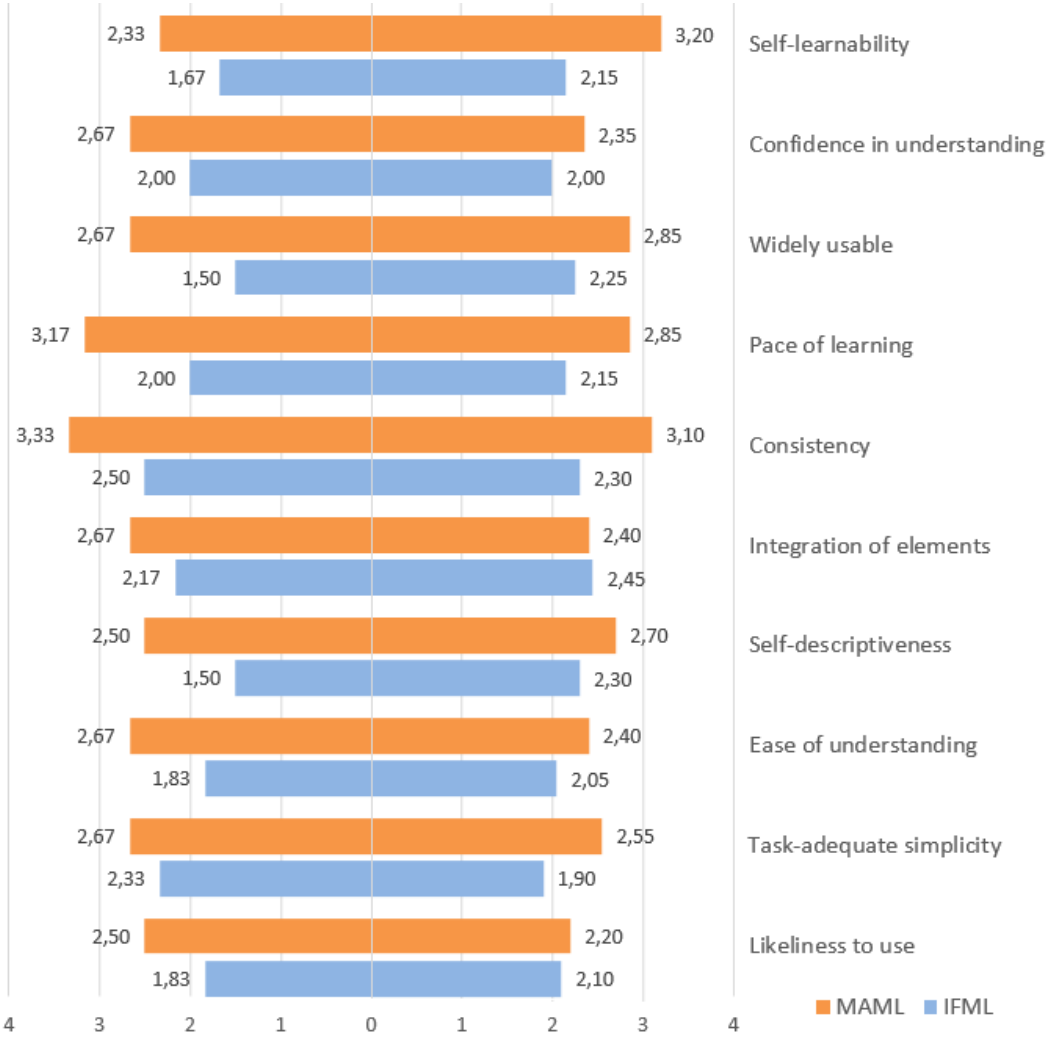


Figure 4.14: SUS Answers for Domain Experts (left) and Technical Users (right) [RK18a]

Considering also the qualitative observations, some interesting insights can be gained. According to the questionnaire results, most of the criticism for MAML is still related to the categories "easy to understand" and "confidence in the notation". IFML's approach of composing elements in screen-like containers for visually hinting at the outcome was often noted as more intuitive compared to MAML. This feedback is not unexpected because MAML's representation was deliberately chosen to be more abstract in order to cover a wide range of app-enabled devices. However, this is valuable feedback for the future, e.g., for improving modelling support by previewing the resulting UI using an additional simulator component. Some suggestions exist to improve readability, e.g., replacing the textual data type names with icons.

Three in four participants can also transfer knowledge from other modelling notations, e.g., to interpret elements such as data sources. All participants within the process modeller group immediately recognise analogies from other graphical notations such as BPMN and understand

the process-related concepts of MAML. Whereas elements such as data sources (understood by 75% of all participants) and nested attribute structures (83%) are interpreted correctly on an abstract level, comprehensibility drops with regard to inevitable technical aspects, e.g., data types (57%) or connector types (43%). Especially domain experts have difficulties in understanding the technical aspects of MAML without previous introduction, including concepts such as cardinalities (0%), data types (25%), and nested object structures (67%). Whereas omitting them in the DSL specification impedes the ability to generate executable code, all participants who initially ignore these intricacies are still able to visualise the process steps and main actions of the model.

Overall, the sample IFML model is often perceived as being a more detailed technical representation of MAML which supports the design goal of providing understandable models with an equipollent degree of specificity without creating the impression of technical clutter. However, the study is not supposed to discredit IFML but emphasises different foci of both notations: IFML is originally designed as a generic notation for modelling user interactions by technical users. In contrast, MAML uses domain-specific concepts to reduce the complexity of simultaneously specifying app development concerns, including data, business logic, and UI.

Usability results

After a brief ten-minute introduction of the language concepts and the editor environment, the participants were asked to solve four tasks covering many of MAML's features and concepts using the developed graphical editor. The tasks consisted of creating a simple model from scratch based on a textual description, replicating a model from the visual app outcome, extending a model with advanced language features, and altering a model in a multi-role context (for more details the reader is referred to [RK19b]). A concluding evaluation was performed using the ISONORM questionnaire with 35 questions on a scale between -3 and 3 in order to assess the usability according to the ISO 9241-110 standard [Into6]. As summarised in Table 4.2, MAML achieves positive results for every criterion, again for the participant subgroups and in total.

Taking the observations into account for qualitative feedback, these figures can be evaluated in more detail. First, the editor is handled without major problems (*suitability for the task* criterion). Although 37% of the participants describe MAML's process-oriented approach as uncommon or astonishing, 67% state to have an understanding of the resulting app while modelling. Also, 94% of the participants noticed a fast familiarisation with the notation, whereas domain experts are generally warier when using the software.

Self-descriptiveness not only refers to the aforementioned comprehensibility but also deals with the integration of different elements while modelling. As a positive example, 86% of the participants intuitively drag and drop role icons on process elements, and 71% of the participants apply the "error event" element correctly, although their handling was previously not explained. Self-descriptiveness is, however, more limited when dealing with technical issues. Side effects of transitive attributes are only recognised by 43% of process modellers and 25% of domain experts.

Consequently, additional modelling support and validation capabilities are needed to guide users towards semantically correct models.

Regarding *controllability*, very positive scores can be attributed to the simplistic design of MAML and its tools. In contrast to other notations, all modelling activities are performed in a single model by arranging elements on a canvas instead of switching between multiple perspectives. In addition, live data model inference (cf. Subsection 4.3.2), validation rules, and content suggestions provide sophisticated modelling support. Consequently, participants state that "the editor does not evoke the impression of a complex tool".

Similarly, the results for MAML's *conformity with user expectations* are also clearly positive, although the prototypical nature of the tools causes occasional performance issues. In particular, participants confirm the editor handling is easily internalised by the participants.

Regarding the criteria *error tolerance* and *suitability for individualisation*, scores are moderate but the prototype was not optimised for production-ready stability or performance. For example, participants appreciate the support of not being able to model invalid constellations due to syntactic and semantic validity checks but criticise the blocking of actions without further feedback on why a specific action is invalid. Users might benefit more from such contextual explanations than from traditional help pages but, unfortunately, the modelling environment Sirius is not yet able to provide this information.

Finally, the graphical approach to express mobile app content earns positive ratings for the *suitability for learning* criterion. The overall satisfaction and usability can be illustrated using quotes such as calling MAML "a really practical approach", and participants having "fun to experiment with different elements" or being "surprised about what I am actually able to achieve".

To sum up, MAML models are favoured by participants from all groups, despite differences in personal background and technical experience. Although the results and qualitative feedback

Table 4.2: ISONORM Usability Questionnaire Results for MAML

Criterion	**All participants**		**Software developers**		**Process modellers**		**Domain experts**	
	μ	σ	μ	σ	μ	σ	μ	σ
Suitability for the task	1.63	1.04	1.36	1.13	1.62	1.12	2.13	0.62
Self-descriptiveness	0.51	0.73	0.62	0.62	0.38	1.02	0.50	0.41
Controllability	2.10	0.83	2.20	0.63	2.02	0.63	2.03	1.41
Conformity with user expectations	1.78	0.52	1.85	0.47	1.64	0.47	1.87	0.70
Error tolerance	0.92	0.96	0.89	0.63	1.11	0.81	0.70	1.63
Suitability for individualisation	1.20	0.90	1.04	1.05	1.42	1.02	1.17	0.27
Suitability for learning	1.83	0.67	2.02	0.54	1.69	0.66	1.70	0.90
Overall score	1.43	0.49	1.43	0.46	1.41	0.53	1.44	0.59

highlight possible improvements, the study confirms MAML's design principle of creating an understandable DSL and usable editing environment for the purpose of mobile app modelling.

4.3.6 Interoperability of MAML and BPMN

While the MAML DSL can be used to incorporate end users in the development of mobile apps, the process-oriented design additionally offers the opportunity to integrate app development with traditional process management practices. In companies, business processes are often documented using notations such as the de-facto industry standard BPMN (cf. Subsection 4.1.3). However, such models usually lack a sufficient level of detail to be directly executable using workflow management systems, especially in mobile contexts [Rec10; Tuo+07]: Regarding the *data layer*, BPMN is control-flow oriented and mostly neglects data modelling issues such as data and attributes, even with the process-internal or persistent data states available in BPMN 2.0. Concerning *business logic*, many concepts of workflows are provided by BPMN but mobile-specific functionalities (e.g., sensor access), practical issues such as connectivity, and parallel execution of user actions on distributed devices also need to be considered. Furthermore, the *presentation layer* of software is not covered by BPMN for lack of UI elements. Consequently, it is not possible to holistically describe applications using solely BPMN and process models need to be regularly synchronised with the implemented software and actual execution of activities.

In contrast, MAML's representation is designed at the intersection of software development and process modelling. It would, therefore, be beneficial to consider a bidirectional transformation between both notations. Existing process models could be enriched with app-specific content and app models representing the course of activities could be converted back to BPMN for documentation and analysis. To assess the compatibility of both notations, the concept of Workflow Control Patterns (WCPs) can be used. Russell et al. [Rus+06] identified 43 workflow-related patterns as well as further collections on data and resource patterns [Rtv04; Rus+05]. An analysis of the BPMN 2.0 notation is provided by the Workflow Patterns Initiative [Wor11] whereas MAML's coverage of WCPs is discussed in [Rie18b]. In the following, the most important differences are highlighted together with a potential mapping between concepts.

Basic workflow patterns include the *Sequence* (WCP1) of activities, *Parallel Split* (WCP2) and *Synchronisation* (WCP3) to transition between single and parallel threads of control flows, an *Exclusive Choice* (WCP4) to execute one of multiple alternatives, and a *Simple Merge* (WCP5) to consolidate multiple alternative branches. In MAML, WCP1 is provided by the corresponding "process connector" construct. However, the aforementioned characteristics of mobile devices with unreliable communication preclude inter-workflow parallelism in which a process depends on multiple instances of a workflow. Regarding structural patterns, *Deferred Choice* (WCP16) represents a runtime choice between different branches of which the first executes. In MAML, this is possible to a certain extent: If multiple roles are assigned to one task, (only) the first user will execute the task. Patterns referring to a partial ordering of process steps are neither available

in BPMN nor MAML. Concerning cancellation patterns supported both by MAML and BPMN, current workflow instances can be stopped (*Cancel Case*; WCP20) and individual steps might be implicitly terminated if multiple devices accidentally resume the same instance after role-context changes (*Cancel Activity*; WCP19). As BPMN does not support the concept of workflow state, executing an activity until a state-dependent *Milestone* (WCP18) is not possible. In contrast, this behaviour can be represented in MAML through state-based conditions. Finally, both notations allow for *Arbitrary Cycles* (WCP10) with *Structured Loops* (WCP21) to repeatedly execute activities. Particular to MAML, a loop is either explicitly defined given a conditional expression before/after the activities are executed or implicitly applied to multiple elements chosen in a "select entity" process element.

Business processes already documented in BPMN models can often be used as a starting point for describing mobile applications and need to be enriched by other software development perspectives. To exploit the benefits of interoperability, a bidirectional model-to-model transformation was developed using based on the QVTo language [Ecl19b; Rie18b].

Many core elements of the notations have direct correspondences. For example, *start*, *end*, *timer*, and *error events* have the same semantics in both notations. Concepts such as user roles have different representations but a semantic correspondence between separately modelled *lanes* in BPMN and *participant* annotations in MAML. Regarding many-to-one correspondences, multiple *process element* types in MAML correspond to BPMN's generic *user task*. During the transformation, the more specific MAML task type (e.g., *Create entity*) is derived from the task description in BPMN via keyword-based matching. Further mappings of BPMN concepts include:

- *Sub-process* and *service tasks* with equivalent MAML elements (*Include* and *Webservice*)
- *Manual tasks* for application-external actions represented as *Display message* steps in MAML
- *Script tasks*, i.e., automated actions by the process engine, are only possible when providing the code as a web service
- *Call activities* for reusable global tasks are used for specific process elements such as location retrieval, camera, and phone calls

Because of BPMN's limited support for data modelling, detailed information on data types or nested attributes is lost during the transformation towards BPMN. However, basic information on data such as *data associations*, persistent *data stores*, and *InputSets*/*OutputSets* for an activity can be specified. Conversely, these elements are used to prepare MAML models with initial hints on data types. In particular, data stores associated with BPMN tasks are merged with MAML's data flow as depicted in Figure 4.15.

BPMN contains several elements without equivalent representation in MAML. For instance, *conversations* and *choreographies* are unsupported in MAML for reasons of missing inter-workflow communication. Furthermore, elements such as *multiple instance tasks*, *(parallel) multiple events*, and *message flows* cannot be mapped to MAML because of the inherently sequential design

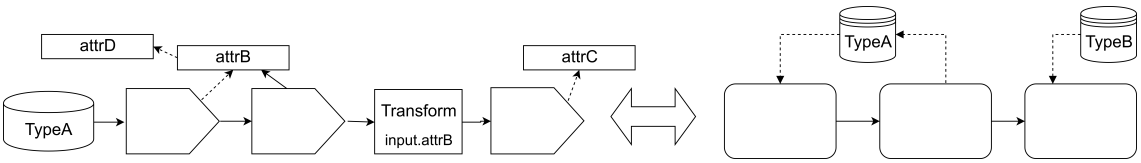


Figure 4.15: Data Transformation Between MAML (left) and BPMN (right) [Rie18b]

of smartphone apps. Complex control flows are omitted in MAML for reasons of concept representation and precision. For example, features equivalent to BPMN's *signal*, *escalation*, or *termination events* and respective *event-based gateways* need to have generalizable and intuitive user interactions in the app. However, more research on adequate representations within both the DSL and the resulting apps is required. In addition, some elements in BPMN such as *complex gateways*, *business rule tasks*, and *compensation* are less structured or delegate processing to a workflow engine but MAML models require a sufficient level of technical specificity in order to automatically generate functional source code.

Conversely, some elements in MAML have no adequate representation in BPMN. Besides the aforementioned data layer, data-dependent process variations unknown at design time (e.g., single or multiple selections) have no equivalent representation in BPMN. Also, UI-related elements such as labels, buttons, and field captions are not in the scope of BPMN and omitted during the transformation.

All in all, MAML covers 16 out of 43 workflow patterns at least partially [Rie18b] and is generally compatible with concepts of the BPMN notation with the exception of inter-workflow and parallel execution capabilities. The implemented model-to-model transformation demonstrates that MAML apps can be exported to BPMN models in order to document the implemented processes using this established notation, and existing BPMN models can be imported to MAML as starting point for adding data-related information and reducing the initial effort of generating apps.

4.4 Model-Driven App Development Across Device Classes

Arguably, not all types of apps are equally efficient on all types of devices (e.g., extensive input on small screens is tedious). However, our focus on the domain of business apps with workflow-like data manipulation activities can be transferred to common user experiences on many heterogeneous devices.

4.4.1 Model-Driven Process for Pluri-Platform App Development

In order to manage the variability of hardware and software capabilities, the complexity of developing and maintaining code generators for multiple platforms across device classes requires a structured approach beyond the generic model-driven process which does not further elaborate

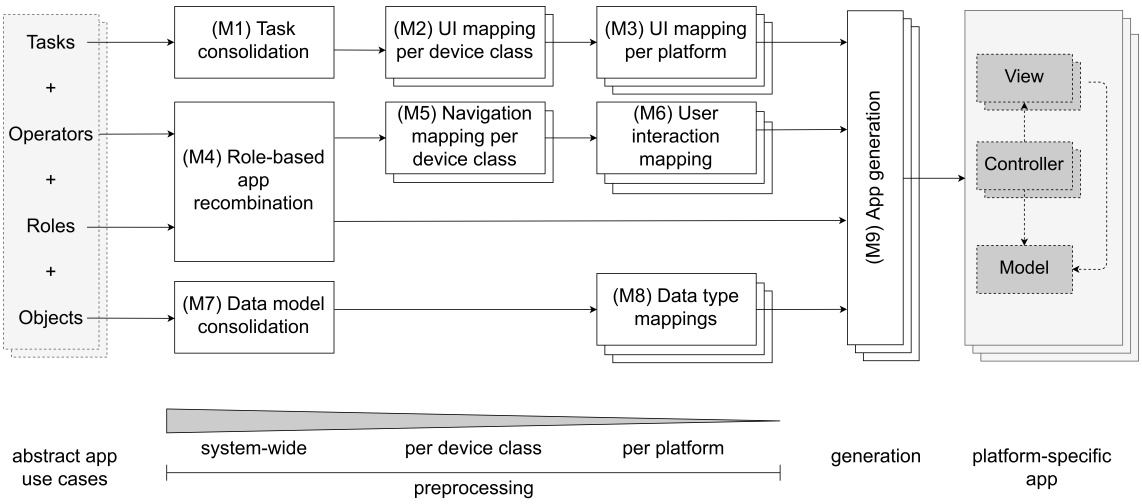


Figure 4.16: Process Model for Model-Driven Pluri-Platform App Development [RK19a]

on the transformation step from source to target representation. Therefore, we propose an extended model-driven process to develop business apps, as depicted in Figure 4.16 [RK19a].

For comprehensibility reasons, the domain-specific notation should be designed with modularity in mind and allow the specification of *use cases* or similar units of self-contained functionality. A process-oriented use case can also be interpreted from a *user task model* perspective [SCK07]. The ConcurTaskTree notation, for example, consists of four elements [Pat00]:

- the declarative *task* descriptions which together form the use case's functionality,
- *operators* defining the allowed sequences of executed tasks, representing an abstract notion of navigation actions available in each task,
- the user *role*s that are allowed to interact with the system (and might differ per task), and
- the data *objects* used within a model.

From such a descriptive representation, mappings need to be defined towards a platform-adapted app result. To handle the transformation complexity, we propose a step-wise refinement of input models. First, system-wide transformations need to be applied to consolidate multiple modular and independently modelled use cases with regard to functionality, role-based access restrictions, and the underlying global data model (M1, M4, M7). Second, the model is preprocessed according to each targeted class of devices concerning the presentation layer (M2, M5). The extent of these transformations highly depends on the degree of maturity of the respective device class. For example, layout patterns such as tabs or vertical content scrolling may be observed frequently on screen-based devices. Third, platform-specific preprocessing of UI and user interactions is required, similar to today's model-driven cross-platform approaches for smartphones (M3, M6, M8). The actual mapping of abstract UI elements to concrete widgets based on a choice of possible representations for the given task is only specified in this step. Finally, the code generation is executed which outputs the platform-specific app artefacts (M9). Software architectures such as

the Model-View-Controller (MVC) pattern or variants such as Model-View-ViewModel (MVVM) are suited as the high-level structure of the resulting applications [SW14]. In addition, available reference architectures can be used to reduce development effort and maintain consistency for platforms of the same device class (cf. Section 4.2) [Ern+16].

Using the proposed model, the complexity of a single code generator is reduced and the addition of features across existing platform generators can be simplified for framework developers. Instead of redundantly implementing the mapping of high-level constructs in each code generator, commonalities between platforms can be abstracted to earlier phases of the transformation chain and only the direct translation into source code remains to be implemented. For example, the generic need for back-and-forth navigation (M4) can be refined in the user interaction mapping (M6) to use swipe gestures for all smartwatches and adaptations to the code generators are limited to the respective event handling. Also, the implementation effort to support new platforms is drastically reduced by starting from preprocessed models for the same device class.

As a result, the refined model-driven process helps to tackle the challenges of pluri-platform app development presented in Section 3.2 [RK19a]:

Resolving output heterogeneity: Based on the descriptive use case models with a high level of abstraction, the main activity related to task consolidation (M1) is the specification of views from the conceived units of functionality. If required, generic model preprocessing and simplification activities are also performed, e.g., the resolution of shorthand definitions and references within the models. Subsequently, mapping the UI per device class (M2) is achieved through two types of transformations that adapt contents to typical device class patterns: On the one hand, the presentation of information can be *layouted* according to a range of screen sizes, for example by choosing an appropriate appearance – one could think of tabular vs. graphical representations – or leaving out complementary details. On the other hand, the content can be *re-formatted* according to usual interaction patterns, e.g., to present large amounts of information through scrolling, subdivided into multiple subsequent steps, or as a hierarchical structure [EVP01].

Finally, a platform-specific UI mapping (M3) defines concrete representations and the actual user interactions within a view. Activities within this mapping include the selection of suitable widgets, their arrangement within the previously specified layout container. To exemplify the difference to the previous device class preprocessing, an "item selection" task within a use case might be mapped to an abstract list selection for all tablet platforms. The concrete representation such as a horizontal card-based layout for Windows or a vertical list view for iOS is then specified in the platform-specific mapping stage.

Resolving input heterogeneity: User inputs are not only important for entering data but also to navigate within the app. The multitude of device class specific (e.g., remote controls for TVs), platform-specific (e.g., Android's hardware buttons), and even device-specific input events (e.g., Apple's 3D touch gestures) is a hurdle for efficient modelling. Instead, user inputs should be described as *intended actions* for completing a particular task. Based on the possible sequences

of tasks, navigation paths can be established (M5), including an initial task selection step when opening the app, navigation between views, and conditional process flows.

In the platform-specific mapping (M6), actions are subsequently transformed into actual input events. This is similar to the decoupling mechanism for user inputs in MVC architectures described in [CFS16] but on a higher level of abstraction. For instance, a "back" action can be linked to a hardware button, displayed in a navigation bar on the screen, bound to a right-swipe gesture, or recognised by a spoken keyword. For novel device classes in an early experimental stage such as smart cars, the meaning of specific input events then represents preliminary design decisions in accordance with vendor guidelines. As noted above, repetitive platform-specific transformations can, of course, be shifted to a more generic layer in the future when commonalities become apparent.

Managing device capabilities: Whereas tasks, operators, and roles have no interdependencies between use cases, a global data layer needs to be established from different data models (M7). Data model inference can validate the compliance of different use cases and also provide additional modelling support for the editing tools of the DSL (cf. 4.3.2). Subsequently, common non-primitive data types (e.g., dates or colours) need to be mapped to available platform concepts for output to the user and back-end communication (M8).

In the context of business apps, devices usually do not need to perform complex computations. If necessary, computations may be offloaded to remote computation providers. With regard to sensors, a progression of functionality should be aimed for in order to avoid the "least common denominator" problem of being restricted to the functionality available on *all* platforms [SV06]. Instead, the same app should function on many devices while providing the highest level of functionality achievable with the given hardware. For instance, location information is easily retrieved if a GPS sensor is available but generic approaches need to consider different sources of location information such as Wi-Fi or cellular networks, and provide adequate fall-back mechanisms such as the manual selection of addresses on a map.

Enabling multi-device interaction: For multi-user scenarios, the recombination of tasks (M4) modifies the sequence of tasks to cater to the interruption of activities and the automatic transmission of application state to the subsequent role. As a result of this decomposition and bundling of use cases, either one app is created that supports all user roles (depending on the logged-in user) or different apps are generated per role (e.g., separate apps for users and administration).

In addition, synchronisation is essential for both the sequential and concurrent usage of apps. In contrast to cross-platform apps, information not only needs to be propagated between devices after a given task has been completed. Intermediate states of data or even a live synchronisation capability of partial user inputs (e.g., each character input) are desirable to seamlessly switch between devices. Also, the workflow state itself must be captured in order to pass the current process instance to different users when role changes occur. Consequently, the back-end component

also needs to manage the relationship of users and devices (either via central device registration or tracking on which device the user is currently logged in).

4.4.2 Usability Evaluation for Pluri-Platform Development

The MD² framework used as the underlying infrastructure for the code generation of MAML models has been extended according to the model-driven process depicted in Figure 4.16 in order to investigate the practical opportunities and challenges of pluri-platform app development. Also, a new code generator for the Wear OS platform by Google [Goo19i] was developed which supports the creation of stand-alone apps for respective smartwatches. Consequently, the model-driven foundation of our framework now allows for the combined generation of smartphone and smartwatch source code using the same MAML models as input.

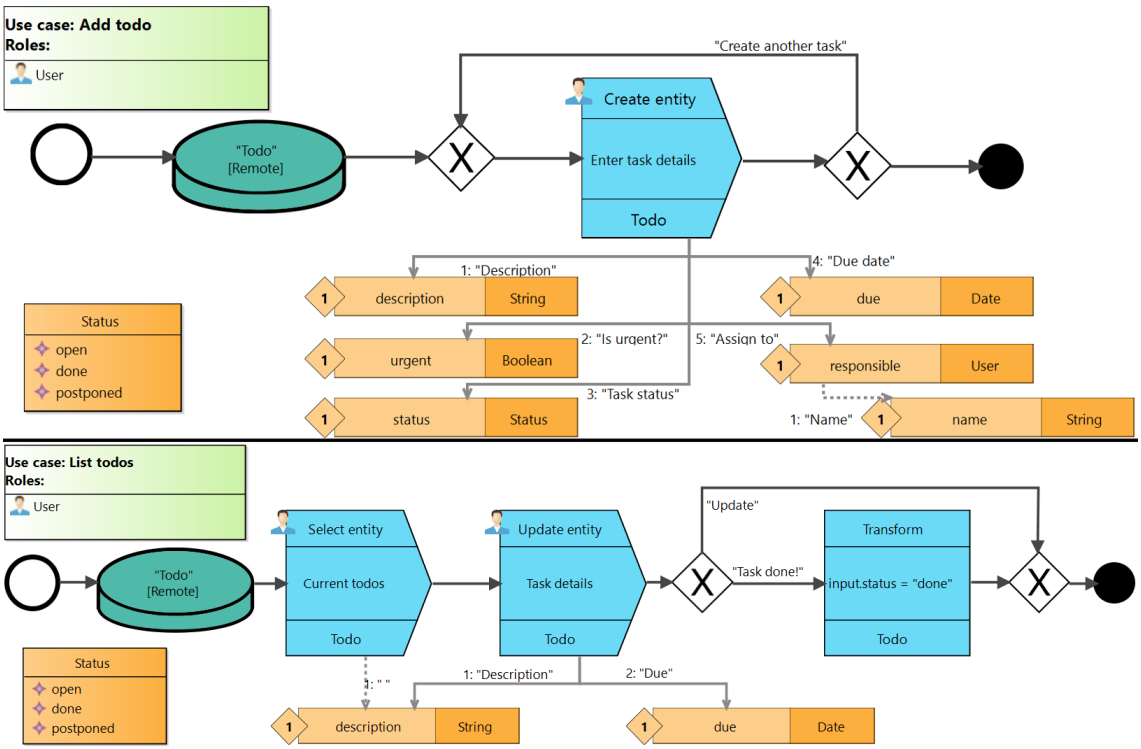


Figure 4.17: Use Cases for Adding and Displaying Items in a To-Do Management System [Rie17]

To assess the usability of the notation for pluri-platform apps, another study was conducted with 23 students from a course on advanced concepts in software engineering of an information systems master's program. Whereas designing applications using MDSD techniques is part of the course contents and knowledge of process modelling notations can be presumed, no previous experience with app development was expected to avoid a bias towards existing frameworks or approaches. This is supported by the low responses regarding experience in the development of

web apps (3.26), hybrid apps (4.35), and native apps (4.30) on a 5-point Likert scale (1.00 denoting maximum agreement).

Using a simple to-do management scenario depicted in Figure 4.17, a short five-minute introduction to the MAML notation was given to explain the two data-driven processes of creating a new to-do item in the system (upper model in Figure 4.17) and displaying the full list of to-dos with the possibility to update items and ticking off the task (lower use case). Subsequently, participants were asked to express their imagined conceptions of the resulting apps by sketching smartphone and smartwatch GUIs for these use cases.

A variety of smartphone interaction patterns could be derived from the sketches, for example, 65% of the participants represented repetitive elements as a vertically scrollable list as opposed to using a horizontal arrangement (17%). From the sketches that reveal navigation patterns, the master-detail pattern of the "list todos" use case of Figure 4.17 was conceived either via tapping on the element (42%), using an edit button (33%), pressing a hardware button such as the watch crown (8%), swiping to the right hand side (8%), or using a voice command (8%).

As regards the creation of new entities, 35% of the participants imagined a scrollable view containing all attributes, whereas 30% decided for separate input steps for each attribute, 9% utilised the available screen dimensions and distributed the attributes across multiple views with more than one attribute on each. In addition, 30% of the participants allowed the unstructured input of data via voice interface and 26% allowed voice inputs per attribute[3]. It can be said that a common perception of navigation within a smartphone app has not been established so far: From the identifiable navigation patterns, 53% relied on buttons to continue through the creation process whereas others depicted horizontal (20%) or vertical (13%) swipe gestures or hardware buttons (14%). Thus, model-driven transformations for smartwatches are for now best devised on the per-platform layer (M3/M6/M8).

(a) "Add Todo" Use Case

(b) "List Todos" Use Case

Figure 4.18: Generated Wear OS App for the System Modelled in Figure 4.17 (cf. [RK19a])

The SUS questionnaire was again used to triangulate the results with the initial study presented in Subsection 4.3.5. The resulting score of 66.85 ($\sigma = 12.9$) aligns very well with the previous results

[3]the total above 100% results from multiple alternatives combined in the same sketches

depicted in Table 4.1 and reinforces the validity of the previous study. Upon showing the generated app of the new code generator (depicted in Figure 4.18), the participants were asked about the relation to the use case models (again using a 5-point Likert scale). The participants agreed (2.04) that the generated smartwatch app suitably represents the process depicted in the MAML model. Furthermore, they supported the statement (2.39) that the resulting app is functional with regard to the to-do management scenario. The visual appearance of the smartwatch was rated merely with 3.3, which can be explained by the generic transformations and assumptions derived from the abstract process model. Also, the prototypical nature of our generator needs more refinements of the implemented widget representations.

Regarding the combined generation of apps for smartwatch and smartphone from the same model, the participants did not feel that the MAML notation makes app development unnecessarily complex (3.35) and agreed that having one notation for both app representations accelerates app creation (2.48). When asked about specific development effort, the students estimated the required time to build the MAML models in Figure 4.17 with 50 minutes on average, compared to a mean estimate of 27.3 hours when programming the application natively or with cross-platform programming frameworks. Although the development was not actually performed in this study, these estimates underline the possible economic impact of MDSD to reduce the effort for creating apps and, thus, achieve a faster time to market.

4.5 Discussion

By providing suitable transformations towards the MD² language and reusing its code generators, MAML achieves the design goal of codeless app creation following the high-level process depicted in Figure 4.10. In addition, a newly developed generator for the Wear OS smartwatch platform underlines the applicability of MAML for pluri-platform development by describing a mobile app as a process-oriented set of use cases. Consequently, the DSL reaches a suitable balance between the technical intricacies of cross-platform app development and the simplicity of usage through the high level of abstraction and can be used to create app source code for both device classes from the same input models. This demonstrates a functional framework capable of generating interoperable apps across device classes according to research question RQ1. In particular, the process presented in Figure 4.16 provides a generalised approach to the efficient development of pluri-platform apps by applying model-driven techniques.

With regard to research question RQ2 on addressing a diverse audience in the app creation process, using a domain-specific language and appropriate abstractions allows for the incorporation of different stakeholders who are hitherto barely involved in development projects beyond requirements engineering and acceptance testing. An observational study substantiates MAML's design goals of achieving wide-spread comprehensibility and usability of models for different audiences of software developers, process modellers, and domain experts. In comparison to the IFML notation, an equivalent MAML model is perceived as less complex and participants felt a high

level of control in solving their tasks. With regard to BPMN, many workflow concepts are shared by both notations, although MAML does not strive for replacing workflow engines. Especially the data layer is currently not well represented in BPMN and workflow engines such as Camunda [Cam19] use custom extension attributes in order to bypass these limitations. This flexibility of design also explains the decision to build a new DSL from scratch in contrast to extensions to existing notations such as previous research by Brambilla et al. that enriches IFML with mobile-specific elements [BMU14] or uses BPMN for web service orchestration [BDF09].

Instead, MAML's user-centric DSL allows to fully focus on the domain concepts of business apps and reduce the amount of technical specification. Also, automatic inference and modelling support within the IDE help with creating syntactically and semantically valid models with limited previous experience. Of course, the syntax should be aligned with intuitive representations for the target audience. For example, MAML draws on concepts of established process modelling notations such as BPMN to provide familiar representations for process modellers (cf. Subsection 4.3.6). Moreover, importing pre-existing process documentation allows for the reuse of models for app creation, e.g., for small enterprises with limited resources. Conversely, process documentation derived from the models always remains synchronised with the actually implemented processes.

The design and development of MAML and MD² over the course of multiple years according to the *build* and *evaluate* cycle of design science research [Pef+07] still leaves room for improvement. Besides performance optimisations, mixing textual and graphical DSL editors would be beneficial to formalise the use of expressions (e.g., for conditional branches of execution) and provide the modeller with features such as auto-completion. Such combinations of notations typically rely on different modelling perspectives instead of their integrated development. However, a recent collaboration between Sirius and Xtext is working on integrating their textual and graphical editors [KB17]. Also, concepts such as data filters or aggregations may be included in MAML if intuitive and usable representations are found. With regard to pluri-platform development, the addition of code generators for further device classes such as smart personal assistants without graphical UI would be valuable future work to assess the potential of a process-oriented notation bridging the gap between heterogeneous devices.

4.6 Contributions to the Field of Research

The dissertation investigates and advances the field of model-driven software development for mobile devices from several perspectives.

Theoretical As clarification and delimitation from the previous terminology, the *pluri-platform* construct is introduced to explicitly relate to app development across device classes. Model-driven theory is then advanced by proposing a multi-step transformation process that successively transforms highly abstract input models to platform-specific representations while avoiding redundant implementations. With regard to the user-centred and modu-

lar design of applications, a data model inference algorithm was devised to consolidate independently created models.

Empirical The dissertation contributes to the empirical record of model-driven approaches concerning the positive impact of abstract and user-oriented notations on usability and comprehensibility. Also, the concept of workflow patterns is empirically supported for the domain of process-oriented applications.

Practical The open-source MAML framework helps developers and non-technical users in creating business apps from a process-oriented perspective. Although conceived as a research prototype, the framework provides the graphical DSL together with an editor component and code generators as practical software artefacts. For developers of business apps, a reference architecture which supports app-internal and inter-app workflows was presented that can be applied to smartphones and similar architectures.

These contributions are described and evaluated in greater detail within the following publications (cf. Part II):

- Christoph Rieger and Herbert Kuchen. "Towards Pluri-Platform Development: Evaluating a Graphical Model-Driven Approach to App Development Across Device Classes". In: *Towards Integrated Web, Mobile, and IoT Technology*. Ed. by Tim A. Majchrzak et al. Vol. 347. Springer International Publishing, 2019, pp. 36–66. DOI: 10.1007/978-3-030-28430-5_3
- Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *Computer Languages, Systems & Structures (COMLAN)* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001
- Christoph Rieger and Herbert Kuchen. "A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes". In: *Hawaii International Conference on System Sciences (HICSS)*. Maui, Hawaii, USA, 2019, pp. 7431–7440
- Christoph Rieger. "Interoperability of BPMN and MAML for Model-Driven Development of Business Apps". In: *Business Modeling and Software Design (BMSD)*. ed. by Boris Shishkov. Springer International Publishing, 2018, pp. 149–166. DOI: 10.1007/978-3-319-94214-8_10
- Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *Hawaii International Conference on System Sciences (HICSS)*. Waikoloa, Hawaii, USA, 2018, pp. 5725–5734
- Christoph Rieger. "Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development". In: *Annual ACM Symposium on Applied Computing (SAC)*. Marrakech, Morocco: ACM, 2017, pp. 1599–1606. DOI: 10.1145/3019612.3019746
- Jan Ernsting, Christoph Rieger, Fabian Wrede, and Tim A. Majchrzak. "Refining a Reference Architecture for Model-Driven Business Apps". In: *12th International Conference on Web Information Systems and Technologies (WEBIST)*. SCITEPRESS, 2016, pp. 307–316. DOI: 10.5220/0005862103070316

5

DSL Design

Besides discussing the technical challenges of cross-platform development (cf. Chapter 3) and suitable model-driven implementations (cf. Chapter 4), the actual design of DSLs can be scrutinised to ease their development and evaluation. The research was conducted in different domains to assess the potential and implications of textual DSLs decisions on maintainability. In particular, considerations of modularisation (Section 5.2), preprocessing (Section 5.3), and cross-cutting concerns (Section 5.4) are described in the following.

5.1 Foundations on DSL Design

Whereas the general benefits and disadvantages of DSLs have already been explained in Section 4.1, the process of creating and evolving a DSL and its surrounding tooling is not straightforward and subject to ongoing research [SV06; CP10]. Introducing formal models as a higher level of abstraction permits domain experts to express their requirements using syntax and semantics close to the notation known within the domain. A general distinction can be made between parser-based and projectional approaches. In parser-based languages, the user interacts with the concrete syntax whose sequence of tokens is parsed to create an abstract syntax tree (AST) representing a valid model according to the specified grammar. Projectional editors differ in that modifications to the models are directly applied to the AST and the concrete representation is derived from this using pre-defined projection rules. Consequently, the projectional approach can be used to build textual or graphical DSLs or combinations of both, whereas parser-based approaches are limited to textual DSLs. On the other hand, the implementation of projections requires additional effort when building a DSL [Völ13].

The concrete syntax of a DSL can be categorised into textual, graphical, tabular, wizard-based, or domain-specific representations as well as combinations of those [LJJ07]. Often, textual DSLs are used in model-driven approaches due to the ease of writing, storing, and parsing models alongside manually programmed source code [Völ13]. In contrast, the advantage of graphical notations lies in the flexibility of representing element relations spatially, which can make the modelled contents easier to grasp. This is perceived as more attractive by inexperienced users [Mye90], for example, in domains relying on sequences or compositions of elements. On the downside, the development effort for developing a DSL and its surrounding tools (e.g., a visual editor) is initially higher. Also, the analysis and modification of models developed by others are cognitively more challenging [Mel+16]. Mixed representations are also possible, e.g., the

projectional framework MPS by JetBrains allows for combined models of text, tables, formulas, and graphical representations to select from adequate representations for each domain concept [Jet19]. Exemplary DSLs for many domains can be found in [vKV00].

In addition, a distinction can be made between *external* and *internal* DSLs. External DSLs are independently developed languages, therefore often providing a custom syntax specifically crafted according to domain experts' requirements [Fow05]. Since they are disjunct from the host-language in which models are executed, appropriate tools such as parsers, compilers, or interpreters need to be provided by the framework developers [Völ13]. In contrast, an internal DSL is embedded into a general-purpose language (GPL) and, consequently, bound to use the same syntax. However, they often utilise only a subset of the host language's features to create domain abstractions [Fow05].

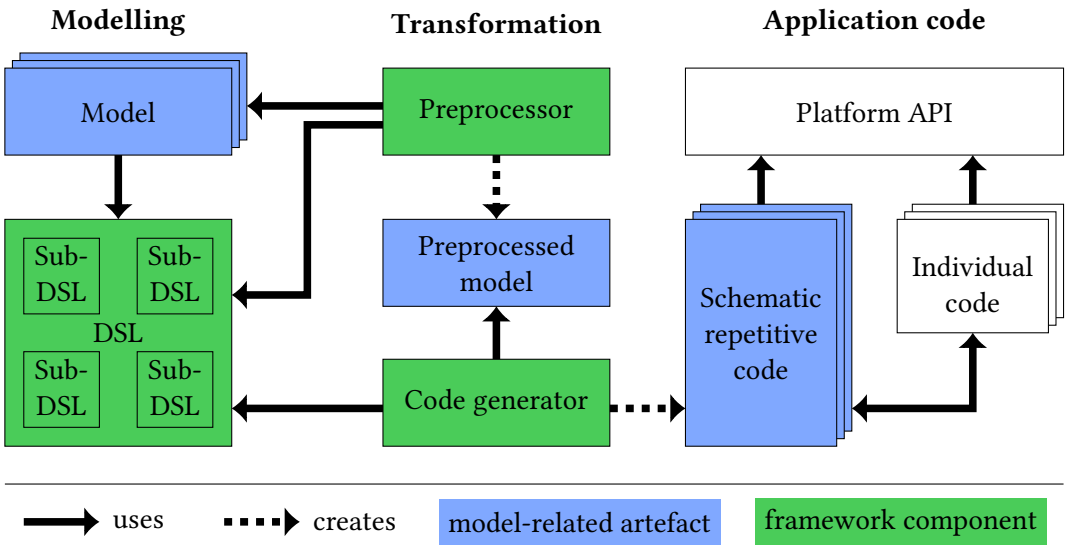


Figure 5.1: Components of a Generative Model-Driven Approach (adapted from [SV06])

Figure 5.1 visualises the components and the overall process of a generative MDSD approach using a DSL as well as possible improvements discussed in this chapter. First, one or multiple models are created by the modeller using a DSL. Modularising application models fosters comprehensibility and usability for the modeller. Conversely, modularising the DSL itself improves maintainability and reusability of the language from a developer perspective as described in the next section. Subsequently, model transformations are applied. In the examples of generative software development demonstrated in this dissertation, the code generation approach can be enhanced using model preprocessing. For example, the input models can be simplified in order to reduce their complexity and, thus, the development effort of multiple generator implementations in cross-platform scenarios such as MD² (cf. Section 4.2). Also, models can be preprocessed for performance purposes as detailed in Section 5.3. Finally, the generated *schematic repetitive code* is complemented with hand-written *individual code* according to the target platform APIs [SV06].

Traceability techniques can be used to keep track of model components across these layers of abstraction as exemplified in Section 5.4.

5.2 Modularisation of Xtext-Based DSLs

Ideally, a DSL could focus on just the domain-specific components and their representation. However, in practice, many fundamental – often technical – concepts such as type systems or arithmetic expressions have to be re-implemented from scratch for multiple DSLs instead of reusing functionality available in other languages.

5.2.1 Foundations of Language Modularisation

Because model-driven approaches aim at increasing productivity through automating software development, modularisation of DSLs has become a topic of increasing interest in academia [Pes+15; KRV10; CV16]. In addition, composing modular languages offers the opportunity to improve software quality and eventually achieve better maintainability and extensibility. When new DSLs are iteratively developed or the domain scope changes, adaptations of language features are applied to a clearly defined and manageable scope [CP10; Vac+14]. Additional complexities arise when DSL evolution requires downward compatibility to previous versions through techniques such as deprecation [Völ13]. Current support reaches from language workbenches specifically targeting modularisation but lacking sophisticated IDE support [Vac+14; EH07] to development frameworks providing an extensive set of features for developers and modellers but only rudimentary modularisation capabilities [Erd+15]. *Neverlang* is an example that explicitly focuses on creating composable features, so-called "slices", which separately define the syntax in Backus-Naur-Form notation and "roles" describing the language semantics [Vac+14]. The *MontiCore* framework allows for including external fragments at runtime such that modellers are capable of utilising arbitrary DSLs when modelling. In addition, Eclipse plug-ins for modelling support such as syntax highlighting are provided [KRV10].

From the variety of design patterns on language modularisation described in the literature (e.g., [Spi01; KRV10]), six modularisation techniques applicable to DSLs are visualised in Figure 5.2 [RWK18]:

Language extension allows new features to be added to an existing language by inheriting from the base language and adopting its semantics and syntax [Spi01]. Mainly single inheritance is used in practice because of possible conflicts but language extension generally allows obtaining features from multiple DSLs [Duc+06].

Mixins are self-contained increments of functionality which specify dependencies to languages or other mixins. As a special form of language extension, a mixin can be used independently by multiple sub-DSLs, thus enabling more flexible composition of functionality than traditional

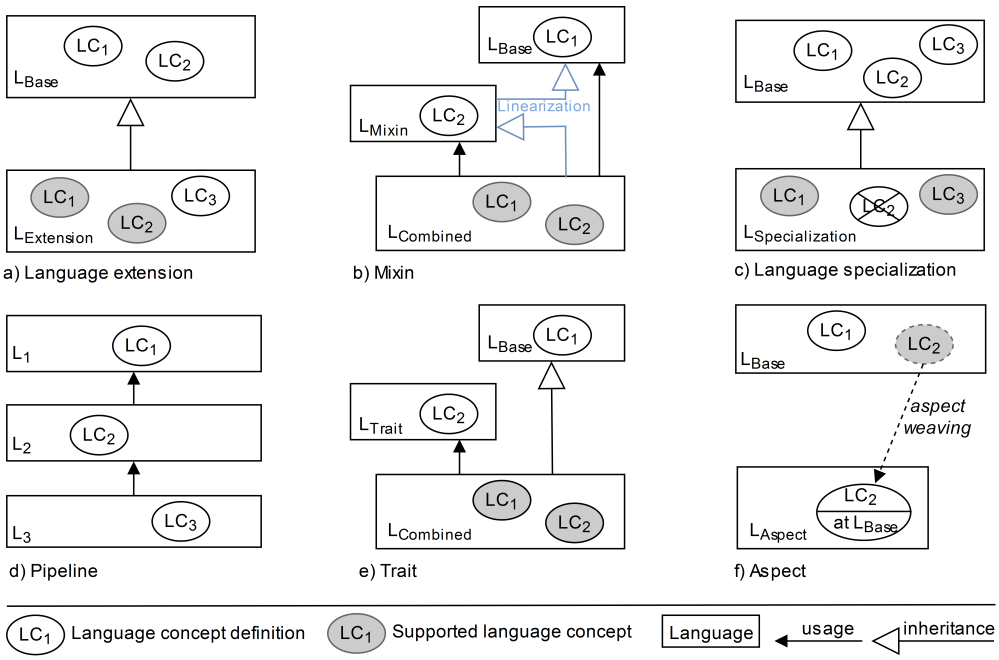


Figure 5.2: Language Modularisation Concepts (cf. [RWK18])

inheritance. A language compilation step linearises the type hierarchy such that all language dependencies are resolved using single inheritance [BC90].

Language specialisation represents the counterpart to language extension by removing unwanted parts of a base language. Consequently, the derived DSL comprises a subset of the original functionality [Spio1].

Pipelining of languages denotes the chained combination of multiple specialised languages on the same hierarchical level. Each language only handles a subset of elements and passes its output to the next language in the pipeline [MHS05].

Traits originate from the object-oriented programming domain and comprise a collection of methods and attributes, similar to Java 8's default method implementations for interfaces. In contrast to abstract classes, a trait is stateless by definition and does not enforce an order of composition compared to traditional inheritance. When applying this concept to language composition, traits provide new or extend existing language constructs while explicitly disambiguating conflicts in the invoking language. In contrast to mixins, reusable functionality is provided without affecting the type hierarchy [Duc+06; CV16].

Aspects also encapsulate composable units of functionality. However, in contrast to traits, the aspect itself declares its *pointcuts*, i.e., the set of events during program execution for which the aspect's action should be executed [WKD04]. With regard to composing language grammars, pointcuts can be declared according to a desired modular structure of

the resulting DSLs but a transformation engine, the so-called *aspect weaver*, is required to resolve the composition of aspects before generating code or executing models [Wu+05].

The Xtext framework only supports language extension through single inheritance. Using the `with` keyword, a grammar can extend another language as depicted in Listing 5.1 (line 1). In addition, a feature called *grammar mixins* – unrelated to the aforementioned modularisation concept – allows *referencing* elements across models using the `import` keyword (line 2). However, Xtext does not actually employ the referenced grammar and its syntax, but its metamodel. Therefore, it is not possible to *define* new elements of the imported class in the including language. For instance, an `OptionInput` element (according to line 4) can reference a list of separately defined values but cannot create an `Enum` object within the same model file. Lastly, the concept of *fragments* allows reusing common parts of the syntax across multiple parser rules [Zar15]. Although similar to multiple inheritance on the level of parser rules, fragments are only available within the same grammar. For example, the fragment `widgetInfo` (line 5) specifies the syntax of two attributes and is included in the `OptionInput` rule but cannot be accessed by other derived languages.

```
1 grammar de.md2.View with de.md2.Md2Basics
2 import "http://md2.de/Model" as model
3
4 OptionInput:  'Option' name=ID widgetInfo 'options' values = [model::Enum]
5 fragment widgetInfo:  'label' labelText = STRING 'tooltip' tooltipText = STRING
```

Listing 5.1: Language Modularisation Features of Xtext

5.2.2 Case Study on MD²

The MD² DSL presented in Section 4.2 consists of 188 parser rules with a total of 2174 lines of code (LOC). In terms of file size and LOC, this makes MD² one of the five largest Xtext-based DSLs publicly available on GitHub. The structure of the DSL follows the MVC pattern with an additional workflow layer to distinguish between high-level functional components and the actual interaction logic within platform-specific views [Ern+16]. This layered architectural design of MD² models is, however, not reflected in the DSL itself which defines all parser rules in a single Xtext grammar. Consequently, the Model, View, Controller, and Workflow rules may exhibit bidirectional dependencies which need to be considered when modularising the language, e.g., the *AutoGenerator* feature which derives a generic representation from the data structure managed by a content provider (within the controller layer). In order to tackle the complexity of MD² and resulting issues of learnability for new developers, software quality challenges of dependent software components, and a lack of reusability, the language was chosen as a case study for modularisation. The new structure aligns with the DSL design and corresponding reference architecture through decomposition into individual DSLs for the MVC+Workflow layers as well as a `Basics` DSL as common infrastructure.

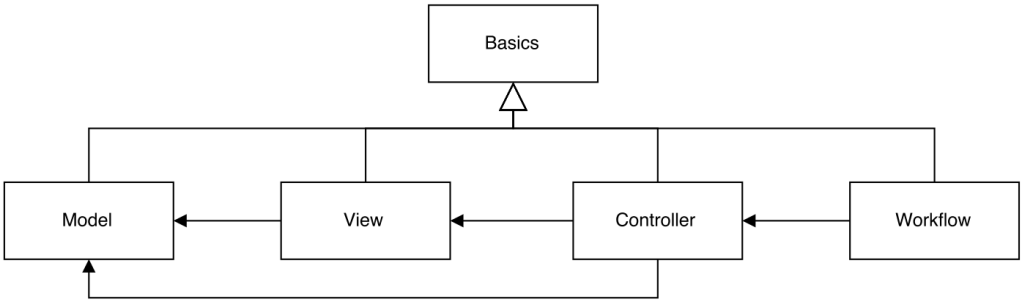


Figure 5.3: Proposed Module Structure for MD² Models [RWK18]

```
1 grammar de.md2.Basics
2
3 MD2Model:
4    package = PackageDefinition & layer = LanguageElement?;
5
6 LanguageElement: {LanguageElement};
7
8 PackageDefinition:
9    'package' pkgName = QUALIFIED_NAME;
```

Listing 5.2: Basics Grammar Defining Interfaces

Within the new module structure depicted in Figure 5.3, the `Model` module only relies on `Basics` and provides the MD² type system, rules for modelling custom entity structures, and typed parameter definitions to be used by other modules. While the `View` module depends on `Model` in order to reference data types, the `Controller` relies on both `Model` and `View` for linking data objects with the desired representation through custom actions. The `Workflow` module only depends on `Controller`, since it builds navigation sequences through the app from low-level process steps.

Because advanced modularisation techniques are not supported by Xtext, achieving the desired flexibility through *inheritance-based modularisation* with single inheritance is not sufficient. In theory, unidirectional dependencies between sub-DSLs could be recompiled into a chain of DSLs only using single inheritance, similar to the aforementioned mixin concept. To avoid recombining languages into temporary inheritance structures and adapting the IDE plug-in each time any of the module dependencies changes, a different approach was chosen to interrelate multiple DSLs.

Instead, *interface-based modularisation* is possible by importing multiple required DSLs in Xtext. Völter and Solomatov suggest the utilisation of interfaces for language composition [VS10]. The modularised MD² DSL combines Xtext's features to create such extensible interfaces as depicted in Listings 17.2 and 17.3. The Xtext concept of *unassigned rule calls* (line 6 in Listing 17.2) forces the instantiation in models even if no further attributes are defined, effectively creating an empty interface. In addition, the `returns` keyword influences the meta model by explicitly merging the inferred class with the given type (commonly used for hiding alternative representations or

```
1 grammar de.md2.Model with de.md2.Basics
2 import "http://md2.de/Basics" as basics
3
4 MD2Model returns basics::MD2Model:
5    super layer = ModelLayer;
6
7 ModelLayer returns basics::LanguageElement:
8    {ModelLayer} modelElements += ModelElement+;
```

Listing 5.3: Model Grammar Implementing Interfaces

subtypes from referencing elements). Taken together, the concept of interfaces is emulated not only within a single grammar but also across DSLs. For example, the `ModelLayer` rule (line 7 in Listing 17.3) "implements" the imported interface by creating valid *Basics::LanguageElement* objects but also contains own attributes. Finally, the `super` keyword (line 5 in Listing 17.3) provides all contents of the inherited rule to the implementing rule, for instance overriding the corresponding *Basics::MD2Model* rule defined in the imported grammar (line 3 in Listing 17.2) with new or overridden.

Table 5.1: DSL Comparison Metrics

(Sub-)DSL	**Rules**	**NCLOC**	**Characters**
Original MD²	188	924	30.208
Basics	26	93	1.893
Model	20	97	3.498
View	62	301	8.319
Controller	85	421	16.198
Workflow	8	31	1.050
AutoGenerator	5	27	955
Modular MD²	206	970	31.913
	+9.6%	+5.0%	+5.6%

With regard to implications for modellers and developers, the scope of the modularised MD² DSL has not changed and all language concepts could be transferred. The main drawback of the extension approach with regard to *modelling experience* is that the flexibility of importing multiple sub-DSLs comes at the cost of modelling components in separate files. Whereas this restriction is acceptable for large modules such as the presented high-level separation of the MVC+Workflow layers and requires minimal changes to the modelling process, it quickly becomes inconvenient for minor feature extensions such as the *AutoGenerator* language that is separated to avoid bidirectional dependencies. As a result, modellers may lose track of the overall specification when spread across multiple files. To quantify the overhead induced by the modularisation, three structural metrics were chosen: the number of grammar rules, the non-comment and non-blank

lines of code (NCLOC), and the amount of (non-comment) characters in each DSL grammar [Čre+10]. Table 5.1 shows a slight overhead of 5% in terms of NCLOC and almost 10% more parser rules resulting from interface definitions and imports but the size of each sub-DSL is greatly reduced. However, the six DSLs represent only a high-level separation of concerns and both positive and negative effects of the modularisation will likely amplify when complex constructs of the large `Controller` and `View` modules are further modularised.

Concerning *development practices*, maintainability of each (sub-)DSL is a major benefit. Refactoring outdated implementations (resulting from the evolving MD²-DSL and the Xtext framework itself) previously caused unknown side-effects. Moreover, the separation of concerns will likely improve development speed and quality in the future because parts of MD² and tooling for validation, formatting, or code completion can evolve separately without comprehensive knowledge of the overall framework. Also, future developments can make use of the modular structure to extend domain concepts or reuse language components in different DSLs. On the downside, the modularisation comes at the cost of fragmentation, in particular creating five Java projects per DSL for grammar definition, editor integration, and unit tests. The current module structure presented in Table 5.1 already results in a set of 30 projects and the configuration of dependencies and overall language bundling requires build tools such as Gradle [Gra19] to remain manageable.

5.2.3 Recommendations on Modularising Xtext-based DSLs

Beyond these summarised MD²-specific implications identified in the case study, four generalised recommendations can be derived for the modularisation of textual DSLs. Obviously, these pieces of advice rely on the current features of the Xtext framework and might be updated when further language modularisation patterns are introduced that allow for embedding language concepts in different DSLs transparently to the modeller.

Recommendation 1 *Inheritance-based modularisation should be limited to closely related language extensions and language specialisation.*

Because inheritance introduces tight coupling between two languages, this approach should be used sparingly. Hierarchical additions or modifications such as company-/project-specific adaptations are well-suited use cases for grammar inheritance (although existing language concepts cannot easily be removed). Still, compared to one large-scale definition, a limited number of sub-languages provides maintainability benefits.

Recommendation 2 *Interface-based modularisation should be used for a flexible combination of independent languages as large-scale layers of the DSL.*

Independent languages can be flexibly combined if the concept of language imports is provided and interfaces can be emulated. Consequently, coexisting DSLs on the same hierarchical level can be represented by cross-referencing elements, even if bidirectional dependencies exist. However, the

presented approach relies on conventions that need to be shared by all involved DSL developers in order to obtain the interface-like structure.

Recommendation 3 *Language reuse works best if a common root language and infrastructure exists.*

Mixing arbitrary DSL specifications in Xtext is not possible. As a replacement, a common infrastructure is required to effectively integrate language concepts such as fundamental interfaces shared by all related languages. However, the dependency on a base language limits wide-spread language reuse because no standard set of generally accepted primitives exists.

Recommendation 4 *The granularity of modules should match the designed DSL's structure.*

Due to the limited flexibility of cross-language references, the module structure ideally matches the structure of the resulting DSL. For example, the requirement of separate model files for language add-ons is acceptable for self-contained units of functionality but otherwise hamper the development process and potentially cause confusion for the modeller.

In conclusion, a lack of modularisation capabilities beyond single-inheritance in the state-of-the-art framework Xtext limits truly domain-specific languages by forcing language developers to repeatedly implement basic language constructs.

5.3 Musket: A DSL for High-Performance Computing

Besides improving modelling experience and development effort, the transformation of DSLs is essential. Depending on the domain and target platform, specific architectures or software quality requirements need to be met. For example, high-performance computing refers to the efficient computation on large-scale data sets using parallel programs which are executed on an infrastructure of multi-core clusters and accelerators such as Graphics Processing Units (GPUs).

5.3.1 Foundations on Algorithmic Skeletons for High-Performance Computing

Developing software in this domain is challenging for developers without experience in several libraries such as MPI, OpenMP, and CUDA to operate the available hardware to its full capacity. Despite the availability of libraries, high-performance computing through large-scale parallelisation introduces pitfalls such as deadlocks and race conditions. At the same time, knowledge of low-level optimisations is required to achieve high performance. Parallel patterns called *Algorithmic Skeletons* have been proposed as a solution to these issues [Col91] and serve as templates and composable building blocks for parallel programs. By decomposing the program into a sequence of generic skeleton calls and providing them with *user functions* for custom calculations, the programmer can abstract from the underlying implementation and distribution of activities across

multiple computation nodes. Thus, application programmers without experience in parallel programming are guided towards correct programs with a focus on the logical order of computation tasks. In addition, cross-platform portability can be achieved by transforming the skeleton calls to different hardware architectures such as CPU or GPU clusters.

Algorithmic skeletons can be classified into different groups, with a common distinction made between *data-parallel* and *task-parallel* skeletons [GL10]. Data-parallel skeletons are, for example, *map* and *fold*. Map applies a function to each data element, e.g., squaring a list of numbers. Fold reduces elements of a data structure to a single element, for example calculating the minimum value of a list of numbers. In contrast, task-parallel skeletons such as *pipeline* and *farm* distribute multiple data manipulation steps. In a pipeline, several tasks are executed after another and the output of the former stage is passed as input to the subsequent stage. A farm typically consists of one master process which launches a number of parallel workers that handle a part of the workload.

There are multiple approaches to introduce algorithmic skeletons in the specification of parallel programs, typically as a library or a DSL[1]. Examples of C++ libraries that implement algorithmic skeletons are FastFlow [Ald+10] which concentrates on stream-parallel skeletons such as for video processing, SkePU [ELK17] which aims for optimised runtime communication through so-called smart containers for caching and memory management, and the Muenster Skeleton Library (Muesli) [Kuc02; CK10; EK12; EK17] which focuses on data-parallel skeletons for distributed data structures and supports the execution on heterogeneous multi-core clusters. DSL-based approaches to algorithmic skeletons are either internal DSLs embedded in a host language or developed as external, stand-alone languages. For example, Danelutto et al. [DTK16] propose an external DSL to compose skeletons and generate FastFlow templates that need to be complemented with custom user functions. Whereas automatic optimisations are possible, e.g., for properties such as power consumption, the user still needs to interact with the library. As an example of an internal DSL, SPar [Gri+17] uses standard C++11 attributes and transforms the source code into a FastFlow-based parallel program. Whereas the user can apply previous knowledge on the C++ programming language, optimisation possibilities are limited due to the low level of abstraction.

In contrast, users of the proposed DSL called *Musket* (Muenster Skeleton Tool for High-Performance Code Generation) can express programs in an easy and concise way, and platform-optimised C++ code for multi-core clusters is generated without further need for libraries [RWK19]. In particular, the use of algorithmic skeletons as first-class language entities allows for the flexible optimisation of execution steps through *rewrite rules*, i.e., formalised transformations for algorithmic skeletons, such as in the work by Steuwer et al. [Ste+15] on GPU programming.

[1] Some general-purpose programming languages such as Chapel [CCZ07] focus on parallel or distributed programming and provide similar concepts

5.3.2 The Musket DSL

Musket represents an external DSL with a textual syntax similar to C++ but which integrates domain concepts of parallel computing such as algorithmic skeletons and distributed data structures. Based on the Xtext framework, Musket addresses programmers who want to write distributed programs for high-performance computation – e.g., simulating physical or biological systems – but have limited specific knowledge of low-level parallel programming. In contrast to working with C++ directly through template metaprogramming [Cza98], the DSL approach offers more flexibility for the design and analysis of programming constructs.

The language extends a subset of the C++ language scope with domain-specific concepts. Nevertheless, external functions can be called from within the model such that the full scope of C++ is indirectly supported – however, this limits potential optimisations due to unforeseeable side effects. An excerpt of a Musket model for the Fish School Search (FSS) metaheuristic, i.e., an algorithm inspired by the movement of a fish school to find good solutions for complex optimisation problems [Men+18], is presented in Listing 5.4. Musket guides users in creating valid programs by structuring the code into four main sections as opposed to arbitrary C++ programs:

Meta Information In order to support the generation process and optimise models, the model headers (lines 1-4 of Listing 5.4) describe the target *platforms* (sequential, multi-core CPU, or different generator implementations), the cluster configuration of *cores* and *processes* for optimisation of distributed data structures, and the level of compiler optimisation (*mode*) for debugging purposes.

Data Declaration Global constants for parametrisation (lines 6-8), custom data structures (lines 10-13) as well as globally available arrays and matrices (line 15) are declared separately from the actual application behaviour. Currently, Musket supports primitive data types (`float`, `double`, `integer`, and `boolean`) for variable declarations and global constants. *Array* and *matrix* collection types are defined using the C++ template style (line 15) which contain the type and dimension of the collection, and also provides a keyword indicating whether the values should be present on all nodes (*copy*), distributed across the nodes in equally sized blocks (*dist*), by row (*rowDist*), column-wise (*columnDist*), or locally instantiated depending on the surrounding context (*loc*). The access to distributed data is simplified by using either the global index (e.g., `table[42]`) or the local index within the current partition (e.g., `table[[42]]`). Moreover, primitive and collection types can be assembled into custom *struct* types to define objects with nested attributes (lines 10-12).

User Function Declaration Algorithmic skeletons decouple the program flow from calculations. Therefore, user functions (lines 17-37) are defined using a variety of common programming constructs such as *arithmetic/boolean expressions*, *type casts*, *assignments*, *if statements*, and *for loops*. Moreover, so-called *CollectionFunction*s serve as helper functions to determine the local/global amount of rows, columns, and values of a collection

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

const int NUMBER_OF_FISH = 4096;
const int ITERATIONS = 5000;
const int DIMENSIONS = 1024;

struct Fish{
 array<double,DIMENSIONS,loc> position, candidate_position, best_position;
 double fitness, candidate_fitness, best_fitness;
};

array<Fish,NUMBER_OF_FISH,dist> population;

Fish movement(double step_size, Fish fi){
 for (int i = 0; i < DIMENSIONS; ++i) {
  double rand_factor= mkt::rand(-1.0, 1.0);
  double direction= rand_factor * step_size * (UPPER_BOUND-LOWER_BOUND);
  double new_value=fi.position[i]+direction;
  [...]
  fi.candidate_position[i] = new_value;
 }

 // fitness function
 double sum = 0.0;
 for (int j = 0; j < DIMENSIONS; ++j) {
  double value= fi.candidate_position[j];
  sum += (std::pow(value, 2) - 10 * std::cos(2 * PI * value));
 }
 fi.candidate_fitness= -(10*DIMENSIONS+sum);

 // update values
 if(fi.candidate_fitness>fi.fitness){[...]}
 return fi;
}

main{
 [...]
 for (int iteration = 0; iteration < ITERATIONS; ++iteration) {
  [...]
  population.map<inPlace>(movement(step_size));
 }
 [...]
}
```

Listing 5.4: Musket Model Excerpt for the FSS Optimisation Algorithm.

structure. *MusketFunction*s additionally provide parallelism-aware functionality such as random number generation (e.g., line 19). Modellers can create local variables and access all globally available data structures for which the system automatically handles the allocation of memory and the transmission of values.

Main Program Declaration The *main* block (lines 39-46) specifies the program flow by using *skeletons* as well as the aforementioned control structures and expressions. In general, a skeleton is applied to a distributed data structure and may take a previously defined user function as an additional argument. For convenience and code readability reasons, the modeller can alternatively specify a lambda abstraction, e.g., `(int a)-> int {return -a;}`. Currently, *map*, *fold*, *zip*, and *shift partition* skeletons are implemented in multiple variants:

- The basic *map* skeleton applies a user function to each element of a data structure. Variants include the memory-efficient *mapInPlace* which directly updates values in the input data structure as well as *mapIndex* and *mapLocalIndex* for context-dependent operations that make use of the current index within the user function.
- The *fold* skeleton joins all elements of a collection according to an associative aggregation function (e.g., `sum, min, max`). Again, a variant includes the current index as additional parameter.
- The *zip* skeleton merges two data structures of equal size by pairs using a user function. Possible variants are the same as for the map skeleton (*inPlace*/*index*/*localIndex*).
- The *shift partitions horizontally/vertically* skeletons simplify inter-process communication by explicitly transferring the content of a local partition between nodes.

For the full language specification, the reader is referred to the open-source code repository [WR19]. Besides the DSL itself, Musket comprises an Xtext-generated parser as well as an editor component for the Eclipse IDE to enable syntax highlighting and auto-completion. Also, custom validation logic was implemented to add semantic modelling support for type checking and conformity between user functions and skeletons [RWK19].

5.3.3 DSL Preprocessing for Performance Optimisation

Whereas a library implementation can only execute method calls in the sequence specified by the programmer, the DSL approach allows for the analysis and preprocessing of models. This does not only enable the support for heterogeneous target platforms as described in Section 3.2 but also provides opportunities for efficient program execution. In the context of high-performance computing, abstract constructs and inefficient patterns within models can be transformed via model-to-model transformations into hardware-specific low-level implementations. It should be noted that compilers already perform many low-level optimisations such as loop unrolling and it is not our aim to replicate these optimisations. However, preprocessing in Musket applies a

series of self-contained transformations to remediate inefficiencies through knowledge of skeleton characteristics and their composition [RWK19]:

Map Fusion Multiple consecutive mapping operations on the same data structure can be combined into a single map skeleton according to the relation $map\ g \circ map\ h \rightarrow map(g \circ h)$ [Ste+15]. Performance gains result from the fact that an intermediate synchronisation between multiple skeleton calls as well as temporary data storage are avoided. Two map skeletons operating on the same data – without accessing other values within the data structure (e.g., pure functions) – can be rewritten to use a combined user function.

Skeleton Fusion Also, skeletons themselves can be recombined such as the fusion of a map and a fold operation to a *mapFold* skeleton. Again, the combined call is more efficient because a temporary data structure can be omitted by folding the result directly.

User Function Transformation Although a user function can be syntactically and semantically correct, its performance depends on the context of where it is called. For example, the naïve implementation of the *map* skeleton call in line 12 of Listing 5.5 can be improved by observing that the second parameter of the user function is never modified, thus safely replacing the call-by-reference parameter by a constant reference (Listing 5.6; top). Static analysis can further identify that the `mapInPlace` skeleton in line 11 of Listing 5.5 returns the (modified) input parameter. Therefore, a transformation is applied during preprocessing which rewrites applicable functions to a call-by-reference scheme and avoid the costly copy operation (Listing 5.6; bottom).

Automated Data Distribution Automatic management of the data when assigning values to differently distributed variables not only unburdens the modeller from correctly specifying data transfers but memory management improvements are also possible using knowledge of the main program control flow. For instance, the map skeleton in line 12 of Listing 5.6 can be replaced by a `mapInPlace` skeleton if there is no further read access to the input array.

Listing 5.5: Excerpt of a Musket Model

```
1  struct Pos { double x, y; }
2  Pos origin;
3  array<Pos,512,dist> positions, result;
4
5  Pos moveTowards(Pos current, Pos target){
6    current.x = (current.x + target.x) / 2;
7    current.y = (current.y + target.y) / 2;
8    return current;
9  }
10 [...]
11 positions.map<inPlace>(moveTowards(origin));
12 result = positions.map(moveTowards(origin));
```

Listing 5.6: Generated User Functions for Listing 5.5

```
Pos moveTowardsMap(Pos current, const Pos& target){
  current.x = (current.x + target.x) / 2;
  current.y = (current.y + target.y) / 2;
  return current;
}

void moveTowardsMapInPlace(Pos& current, const Pos& target){
  current.x = (current.x + target.x) / 2;
  current.y = (current.y + target.y) / 2;
}
```

In order to assess the performance impact of these transformations individually and in larger programs, several stand-alone benchmarks were conducted [WRK19; RWK19]. For example, performing map fusion in a simple scenario of assigning a value and then squaring it leads to a speedup of 1.15 compared to the consecutive – but still parallel – execution for a $100,000 \times 25,000$ matrix using 24 cores. With regard to user function transformations, replacing a naïve `mapInPlace` generation of the user functions for assigning, squaring, and finally summing up values with a call-by-reference strategy achieves a speedup of 2.96 for the same matrix size.

The skeleton fusion technique can be demonstrated with the Frobenius norm which is defined as the square root of the sum of squares $\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}{}^2}$. When applied to a $65,536 \times 65,536$ matrix, the combined `mapFold` skeleton achieves speedups of 2.39, 2.91, and 3.24 on one computation node with 1, 12, and 24, respectively (Figure 5.4). The scenario with 4 computation nodes results in speedups of 2.33, 2.94, and 2.67. However, the smaller speedups can be explained by the increased communication overhead on this simple benchmark.

To evaluate Musket as a whole, further benchmarks were used to calculate an Nbody simulation, matrix multiplication, and the FSS metaheuristic on a multi-core cluster. Detailed performance figures are given in [WRK19] but results show that programs written in Musket can outperform equivalent library implementations such as Muesli, for instance providing speedups up to 1.24 on the complex FSS benchmark [WRK18]. At the same time, Musket models are more concise, even with a syntax intentionally close to C++. For example, the complete FSS model with all operators has 244 lines of code; a reduction by 61% compared to the equivalent Muesli code.

In conclusion, the textual Musket DSL was created to support developers with creating parallel programs for high-performance clusters based on the concept of algorithmic skeletons. In contrast to existing libraries, the design integrates skeletons as first-class language concepts and generates optimised C++ source code with significant performance gains.

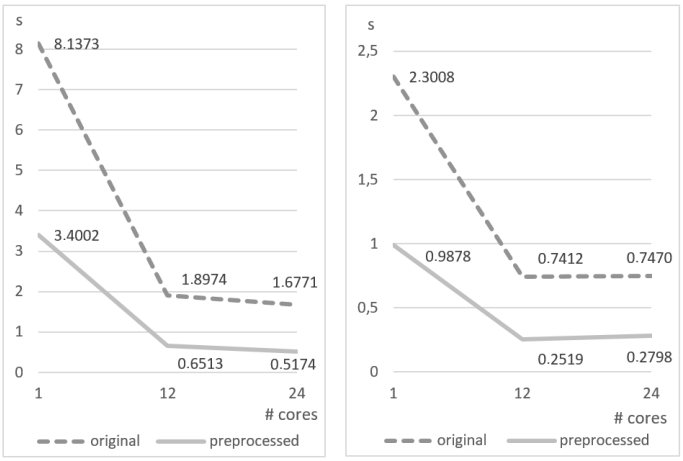


Figure 5.4: Frobenius Benchmark Execution Times for the Original and Preprocessed Models Using 1, 12, and 24 Cores on 1 Node (left) and 4 Nodes (right)

5.4 TAL: A DSL for Configurable Traceability Analysis

DSLs can also help with cross-cutting concerns of software development, i.e., aspects of a program that cannot cleanly be decomposed in a distinct component and result in duplicated code or tangled implementation [Kic+97]. Applied to the software development process itself, traceability is such a concern, denoting the ability to describe and follow an artefact and all its linked artefacts through its whole software development life cycle [GF94].

5.4.1 Foundations on Software Traceability

Companies create traceability information models (TIMs) for their software development activities because of legal regulations [Cle+14] or because it is prescribed by process maturity models, for example, the A-SPICE standard from the automotive industry [Aut15]. A TIM contains *traceable artefacts* such as `requirement`, `unit of code`, or `test case` as well as the *trace links* between them through relationships such as `documents`, `implements`, and `verifies` [MC09; MC13; MGP13]. Relevant metrics, i.e., functions "whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality" [Soco4], exist both for software development in general (e.g., lines of code (LOC) [Rig96]) and for traceability models in particular [RM15]. Traceability metrics include, e.g., the average distance between requirements or linkage statistics concerning the count of related higher/lower level trace artefacts [CL95; RHH98].

However, taking advantage of this information through analysis receives too little attention in research and practice [BMP13; Cle+14]. Traceability tools such as IBM Rational DOORS [IBM16] offer a pre-defined set of metrics and cannot be tailored to company- or project-specific information needs, and much less to answer ad-hoc requests. Frequent questions include the impact of changes on other components in terms of costs or potential bugs (indicated by a high number of trace links) [AB93; RM15] or verifying the progress of software development activities through coverage metrics.

5.4.2 The TAL DSL

To close this gap and integrate artefact retrieval, software metrics, and configurable analyses, we developed the Traceability Analysis Language (TAL), an Xtext-based textual DSL in the domain of software traceability [BRK17a; BRK17b].

Different levels of models need to be considered as depicted in Figure 5.5. From the grammar specification, a meta model of the TAL is created (which is again an instance of the Ecore meta meta model) and concrete models can later be created in the editor component. In addition, the TIM which contains the actual traceability information is an instance of the so-called traceability information configuration model (TICM) which describes the types of traceable artefacts and their link types. A simplified TICM in the A-SPICE context is depicted in Figure 5.6. This model

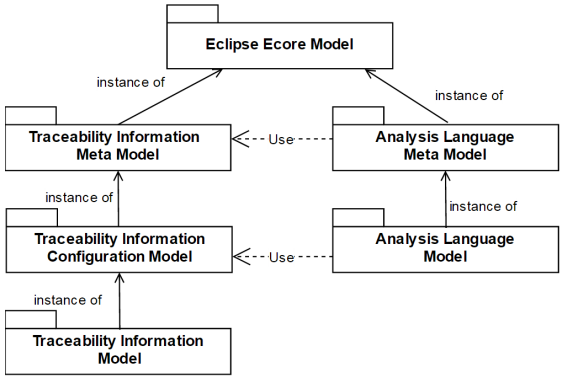


Figure 5.5: Relationships Between TAL Layers

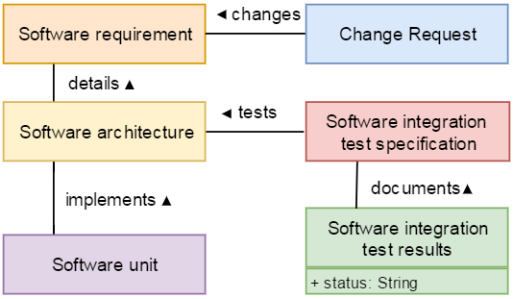


Figure 5.6: Simplified A-SPICE TICM

itself is based on a traceability information meta model (TIMM) that defines the basic traceability constructs such as a traceable artefact types and a trace link types. References between the TAL's meta model and the TICM are the prerequisite for subsequently referencing, e.g., artefact types within analyses. The TAL DSL itself is hierarchically subdivided into three components, namely query, metric, and rule expressions.

```
1 result relatedReqs = tracesFrom Software Requirement to Software Requirement
2   collect(start.name -> srcRequirement, end.name -> trgtRequirement)
3 metric NRR = count(relatedReqs.trgtRequirement) - 1
4 rule NRRwarn = warnIf(NRR>2, "A high number of related requirements may provoke errors.")
```

Listing 5.7: TAL Model of an Impact Analysis

As depicted in Listing 5.7, a typical impact analysis of software requirements on related software requirements can be represented concisely in four lines. The *rule* keyword (line 4) defines an analysis which is characterised by a warning or error if the result of a configured expression exceeds a threshold value. In this example, the analysis relies on a custom *metric* (line 3) which counts the number of related requirements (NRR) excluding itself [RM15]. The underlying query expression (lines 1-2) denoted by the *result* keyword uses the `tracesFrom ... to ...` function in order to search the shortest paths between each element of the specified source and target type in the TIM and `collect` a subset of attributes.

```
1 result swReqToTestResult = tracesFrom Software Requirement to Integration Test Result
2   collect(start.name -> name, count(1) -> numResults)
3   where(end.status = "passed")
4   groupBy(name)
5
6 rule lowTC = warnIf(swReqToTestResult.numResults < 2, "Low number of test results!")
7 rule noTC = errorIf(swReqToTestResult.numResults < 1, "No test results found!")
```

Listing 5.8: TAL Model of a Coverage Analysis

Similar to notations such as SQL, SEMMLE QL [VHM07], or RASCAL [KvV09], functions for aggregation (`groupBy`) and filtering (`where`) can be specified in the query definition itself. For example, the coverage analysis depicted in Listing 5.8 matches software requirements with successful integration test results. By `count`ing elements within the query, distinct metrics can be omitted and the specified rule directly accesses the query result set. This is possible because the result of an executed query, metric, or rule expression is always returned as a tabular structure.

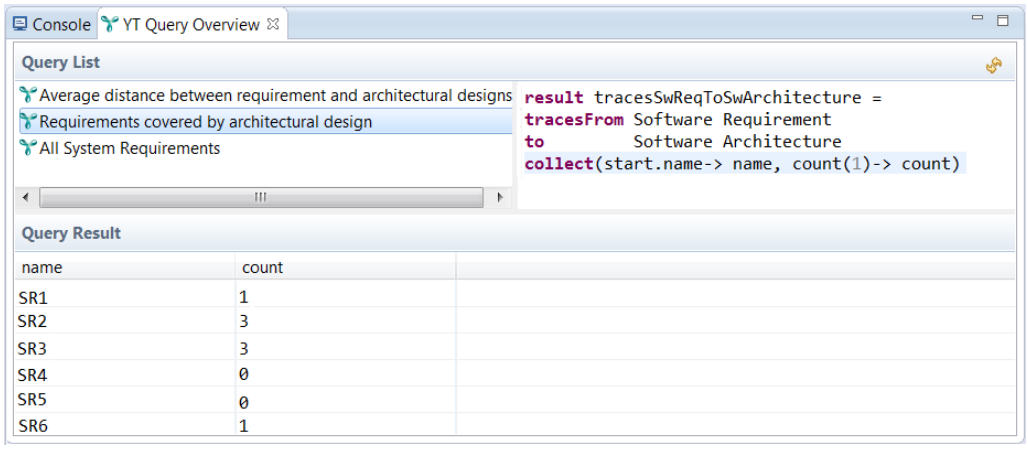


Figure 5.7: Eclipse IDE Integration of the TAL

This concise representation of the modelled content again highlights the user-centred benefit of DSLs. As depicted in Listing 5.9, an equivalent query using SQL is more verbose and requires knowledge of the underlying TIM and possible interconnections of entities. Using the TAL, complex graph traversals are hidden from the modeller and already known terminology can be reused. Furthermore, integration within an IDE enables on-demand analyses (cf. Figure 5.7) and modelling support such as live validation of the models. For instance, staggered analyses that combine both a warning and an error threshold (lines 6-7 of Listing 5.8) help in quickly distinguishing issues with different levels of severity.

```
1  SELECT r.name, count(*) AS numResults
2  FROM SwRequirement r
3  INNER JOIN SwRequirement_SwArchitecture ra ON r.id=ra.r_id
4  INNER JOIN SwArchitecture a ON ra.a_id=a.id
5  INNER JOIN SwArchitecture_SwIntegrationTest ai ON a.id=ai.a_id
6  INNER JOIN SwIntegrationTest i ON ai.i_id=i.id
7  INNER JOIN SwIntegrationTest_SwIntegrationTestResult it ON i.id=it.i_id
8  INNER JOIN SwIntegrationTestResult t ON it.t_id=t.id
9  WHERE t.status='passed'
10 GROUP BY r.name;
```

Listing 5.9: SQL Equivalent to the Analysis in Listing 5.8

The developed prototype (cf. Figure 5.7) is tentatively integrated into a software solution that envisages a commercial application and contains a query interpreter for various file types. With regard to performance, the implemented depth-first algorithm was used to interpret expressions on TIMs ranging from 1,000 to 50,000 artificially created traceable artefacts (based on the A-SPICE TICM depicted in Figure 5.6). Table 5.2 shows the resulting execution times on a developer notebook with Intel Core i7-4700MQ processor at 2.4 GHz and 16 GB RAM. Even for large models, execution is sufficient for real-world applications such as regular reporting purposes or ad-hoc analyses.

Table 5.2: TAL Execution Times

Total Artefacts	Start Artefacts	Duration (in s)
1,000	300	0.012
8,000	1,500	0.1
50,000	8,500	2.2

Because traceability bridges different types of artefacts, relationships to formerly unused documents such as user documentation, test results, or even tickets from collaboration tools [DP13] may be aggregated within a TIM. With the possibilities of configuration, the TAL DSL can additionally be tailored to different company- or even project-specific needs. Software may, for example, be developed using test-driven development, via generative approaches, or simply from a textual requirement. Consequently, relevant metrics for the maintainability of an application's data layer could include the average number of attributes per entity, the number of open bug reports, or the number of classes in relation to the length of the respective requirement definition. Metrics are not an end in themselves and should be purposefully designed to their intended usage context. However, their creation, prototyping, and evolution are simplified by employing the proposed DSL. Though not yet tackled, traceability may also advance model-driven practices with many components that operate on different levels of abstraction. The resulting complexity could be alleviated by tracing the previously unconnected representation of artefacts from abstractly modelled domain concepts down to generated code components.

To sum up, the Traceability Analysis Language (TAL) offers functions and expressions to analyse existing TIMs, structured in query, metric, and rule component. Using a textual DSL, modellers can retrieve traceable artefacts which are cross-cutting all stages of the software development process and perform analyses based on configurable metrics.

5.5 Discussion

In this chapter, three technical perspectives on designing domain-specific languages were demonstrated to answer the third research question of this dissertation. External quality attributes

– often referred to as non-functional aspects – apply not only to traditional software development but also affect the process of creating languages. The development of a DSLs is usually performed iteratively in order to explore adequate representations for domain-specific concepts. After a first stable version, DSLs can further evolve when the domain changes or additional concepts are incorporated [Völ13]. Therefore, maintainability and scalability in language design are important assets for the long-term development of DSLs and language modularisation is an essential prerequisite. Further investigations can consider the complexity of managing the evolution of a DSL in real-world applications, for instance, regarding the migration of generated applications used in production. Also, depending on the control over the user base, multiple versions of the language may co-exist and need to be supported by back-end systems in order to keep up compatibility.

Within a model-driven approach, most of the development effort lies in the creation of transformations that resolve abstractions within the DSL and output executable programs with good performance. To manage the inherent complexity and avoid redundant implementations, optimising the transformation layer (cf. Figure 5.1) through preprocessing is valuable. Depending on the domain requirements, preprocessing can serve the purpose of reducing development efforts as presented in the context of pluri-platform development in Section 4.4, or enhance the performance of generated code (cf. Section 5.3). Both possibilities contribute to research question RQ3.

With regard to cross-cutting concerns, traceability as described in Section 5.4 is only one example of possible improvements that require attention across multiple components of a model-driven framework. If implemented, traceability also improves maintainability and controllability of software projects and, thus, exemplify non-functional aspects of model-driven approaches asked for in research question RQ3. Furthermore, domain-specific cross-cutting concerns exist that can be researched in future work. In the domain of business apps, a particular topic of interest is mobile security [BGG19]. Whereas the implementation of secure apps predominantly refers to the source code and generator component, the design of the DSL also affects potential security measures. For example, MAML's design decision to model all data attributes used in a particular step not only serves the purpose of usability but can be used to identify malicious data changes which are not expected in this step. Another area of cross-cutting concerns is currently under-researched is mobile accessibility [BGG19; KFF16]. Creating apps that can be used by people with disabilities requires changes on all levels of the model-driven process, from DSL specification, modelling support in the editor, adapted transformations, and potentially runtime choices between alternative representations.

5.6 Contributions to the Field of Research

In the field of domain-specific language design, the dissertation provides several contributions.

Theoretical From a theoretical perspective, we review DSL design patterns for techniques applicable to language modularisation as well as the status quo of traceability and algorithmic skeletons.

Empirical This thesis advances the empirical body of knowledge on external quality attributes in DSL development such as maintainability and performance optimisation. Moreover, the juxtaposition of library- and DSL-based approaches with regard to performance is backed by empirical benchmarks.

Practical Results of this research can help in the development of modular Xtext-based DSLs by following the recommendations in Subsection 5.2.3. In addition, the Musket framework can be applied by practitioners to simplify the development of parallel high-performance programs. Furthermore, the thesis provides a configurable DSL to close the gap between query, metrics, and analysis definition in the domain of software traceability.

These contributions are described and evaluated in greater detail within the following publications (cf. Part II):

- Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *The Journal of Supercomputing (SUPE)* (Mar. 2019). DOI: 10.1007/s11227-019-02825-6
- Christoph Rieger, Fabian Wrede, and Herbert Kuchen. "Musket: A Domain-specific Language for High-level Parallel Programming with Algorithmic Skeletons". In: *ACM/SIGAPP Symposium on Applied Computing (SAC)*. Limassol, Cyprus: ACM, 2019, pp. 1534–1543. DOI: 10.1145/3297280.3297434
- Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *High-Level Parallel Programming and Applications (HLPP)*. Orléans, France, 2018
- Christoph Rieger, Martin Westerkamp, and Herbert Kuchen. "Challenges and Opportunities of Modularizing Textual Domain-Specific Languages". In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2018, pp. 387–395. DOI: 10.5220/0006601903870395
- Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Model-Driven Approach for Evaluating Traceability Information". In: *Third International Conference on Advances and Trends in Software Engineering (SOFTENG)*. ed. by Mira Kajko-Mattsson, Pål Ellingsen, and Paolo Maresca. 2017, pp. 59–65
- Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Domain-specific Language for Configurable Traceability Analysis". In: *5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. ed. by Luis Ferreira Pires, Slimane Hammoudi, and Bran Selic. SCITEPRESS, 2017, pp. 374–381

6

Conclusion

This chapter concludes the research overview part of the dissertation by summarising the previous chapters (Section 6.1) and the main contributions (Section 6.2). The limitations of the presented work are discussed in Section 6.3 and lead to topics for future work as presented in Section 6.4.

6.1 Summary

This dissertation set out to advance the state of research on model-driven software development, in particular with regard to the domain of business apps and further applications of DSLs. Three research questions substantiate the initial motivation of reducing the development effort both for domain experts and programmers knowledgeable of the target platform(s). In view of the plethora of novel devices that are capable of running apps, general requirements as well as the applicability of current cross-platform development approaches needed to be investigated. Second, a solution was sought in the domain of business apps to bridge the gap between software developers and non-technical experts and facilitate the cooperation and co-creation of mobile apps for this diverse audience. Third, model-driven approaches which operate on different levels of abstraction and partly decouple the specification of software from its implementation through generative techniques inevitably introduce complexities in the development process. Therefore, improvements for non-functional attributes such as maintainability and usability of language development frameworks needed to be considered.

A structured process according to the design science research paradigm was put forward to pave the way from the identified problems to the design and development of artefacts, their evaluation, and communication in an iterative manner (cf. Chapter 2). Three threads of research provided answers to the aforementioned questions within the overall context of MDSD but with a particular focus on cross-platform app development. In addition, research regarding the third question on external quality attributes also took into account further applications of DSLs for high-performance computing and traceability.

In Chapter 3, current approaches to cross-platform development were reviewed. In particular, scarce previous work on app-enabled devices beyond smartphones and tablets lead to the development of a taxonomy. Results show that current and upcoming devices can be categorised into device classes according to three dimensions: media richness of inputs, media richness of outputs, and degree of mobility. With regard to software development across device classes (so-called pluri-platform development), challenges include the variability of device capabilities

such as sensors as well as the heterogeneity of input and output devices which can have drastically different user interfaces and user interaction patterns. Furthermore, multi-device interoperability is a new challenge resulting from the ownership of more than one device and the sequential or concurrent usage according to personal preferences or contextual factors. In order to ease the selection of app development frameworks, an extensive criteria catalogue is provided that can be adapted through weight profiles according to developer requirements.

Subsequently, two model-driven frameworks were presented in Chapter 4 that address the domain of business apps. MD² provides a textual DSL for specifying apps whereas the MAML framework contains a graphical DSL to specify apps on a high level of abstraction in a process-oriented fashion. In contrast to process modelling notations, MAML models comprise all technical details required for app development, i.e., data layer, business logic, UI, and user interactions. The common BPMN notation largely omits the data required in processing steps but many commonalities with regard to workflow-related concepts exist which allows the import and reuse of BPMN models. A usability evaluation highlights that MAML is more comprehensible and usable than the technical IFML notation, especially considering a diverse audience of software developers, process modellers, and domain experts without technical knowledge. Models are then transformed into the MD² representation and reuse its code generators to create native source code for the Android and iOS platform together with a back-end server component for inter-app communication and remote data storage. Although it is possible to modify the app outcome both on the intermediate level of the textual DSL and the resulting source code, fully functional apps are generated by default without the need for manual programming. In addition, the applicability of model-driven approaches to pluri-platform development is demonstrated using a code generator for the Wear OS smartwatch platform using the same input models.

Thirdly, Chapter 5 presents research on non-functional aspects of MDSD frameworks. With regard to language modularisation, several approaches can be found in the literature but the state-of-the-art framework Xtext which is widely used in practice provides only limited support for language modularisation. An interface-based solution is presented to bypass the inflexibility of single inheritance but native support for more advanced techniques is desirable. Subsequently, the benefits of model preprocessing are highlighted in the context of the Musket framework created to simplify the creation of high-performance programs. To prepare models for the code generators, domain-specific abstractions can be resolved and performance optimisations can be applied. Empirical benchmarks underline that model-driven frameworks can outperform library-based approaches. Finally, the TAL DSL considers traceability as an example of cross-cutting concerns. The inherent complexity of multiple artefacts employed within the software development process can be alleviated with the help of configurable metrics and analyses that support developers and managers in controlling software projects, e.g., concerning the development progress, impact of changes, or test coverage.

6.2 Contributions

This dissertation makes several theoretical, empirical, and practical contributions. Regarding theories, contributions include a review of current cross-platform development approaches in the extended scope of app-enabled devices that comprises not only the narrow field of smartphone apps but mobile computing in a wider sense as well as stationary devices extensible by apps. The concept of pluri-platform development across device classes is further exemplified by proposing a model-driven process that successively transforms input models into platform-specific representations via multiple steps of refinements in order to avoid redundant implementations. Also, a data model inference algorithm is presented to aggregate distinct models and, thus, enable the creation of modular and user-centred DSLs.

These theoretical contributions are substantiated by empirical artefacts and evaluations. For instance, a taxonomy of app-enabled devices provides a simple – yet actionable – classification according to three dimensions that can serve for future research on characteristics and development approaches for specific device classes. In addition, usability evaluations and prior research on workflow patterns are applied to empirically assess the developed MAML DSL in the domain of business apps and compare it to related notations. Also, empirical research on external quality attributes contributes to the body of knowledge on MDSD approaches regarding maintainability, cross-cutting concerns, and model preprocessing.

For practical applications in the domain of mobile apps, this dissertation provides a criteria catalogue to guide developers in assessing and selecting a development framework according to company- or project-specific needs. In addition, the MD² and MAML frameworks are two open-source frameworks to develop business apps using textual or graphical notations and generate native source code for the Android, iOS, and Wear OS platforms. Their continued development can support practitioners with the actual creation of apps and the accompanying research, e.g., on reference architectures, highlights best practices of app development beyond individual platforms. Furthermore, the Musket DSL simplifies the development of high-performance programs using algorithmic skeletons and the TAL language provides configurable analyses in the domain of software traceability.

6.3 Discussion and Limitations

Albeit answering the research questions, this work is not without limitations that threaten the validity of results and call for future investigations.

Concerning research question RQ1, model-driven software development turned out to be a viable method for extending cross-platform app development beyond the current scope of smartphones and tablets towards novel app-enabled devices such as smartwatches. Specific requirements may exist within each device class. However, heterogeneity of inputs, outputs, and device capabilities remain overarching challenges. In addition, the interaction between multiple

devices mandates a more detailed investigation. Of course, developing cross-platform apps within each of the emerging device classes is already no trivial task for lack of generally accepted user interactions. As regards the MD² and MAML frameworks, only three target platforms are available for smartphones, tablets, and smartwatches so far. The applicability to novel forms of user interfaces such as voice-based smart personal assistants needs to be backed by developing more code generators. Real-world adoption is pending in order to assess the overall reduction of development effort and time-to-market.

Research question RQ2 is also answered by the design of MAML as a high-level graphical notation that can be used to communicate and co-create the content of business apps. The research prototype demonstrates its applicability to codeless app development and receives favourable feedback from software developers, process modellers, and non-technical users alike. It should be noted that the usability study was conducted mainly with students which potentially reduces the generalisability of results. More in-depth studies are required during the ongoing development but the participants' range of technical skills already provided valuable insights from a heterogeneous (albeit not representative) sample of potential users. The framework can, however, be improved through further iterations of the design science research process towards better platform conformity, improved performance of the development environment, and modelling support as identified in the usability studies and exemplary use cases. The prototypical nature of the Musket and TAL implementations also leaves room for improvement on feature completeness and the extent of transformations.

Finally, model-driven software development is a broad area of research and RQ3 only touches selected aspects of external quality attributes. Whereas modularisation techniques are well studied in literature and modern general-purpose programming languages exhibit various capabilities for modularisation language workbenches for DSL creation are often complex to learn or provide insufficient tool support. For example, the core components of the Xtext framework would need to be adapted in order to integrate advanced modularisation concepts. This is, however, necessary to enable large-scale reuse of language components and eventually foster wide-spread interoperability of DSL modules for true domain specificity. Similarly, cross-cutting concerns such as traceability facilitate the development of DSLs but are not yet widely applied.

6.4 Future Work

These limitations and ongoing technological developments offer opportunities for further research with regard to the presented frameworks, app development, and the field of MDSD in general.

This thesis has shown that model-driven software development on different levels of abstraction can increase the usability and efficiency of development activities – especially for domain experts without technical knowledge. Although it is possible to provide advanced modelling support to future users using state-of-the-art language development frameworks such as Xtext, tool support is required to manage the complexity. Chapter 5 has shown that a modular composition of DSLs

can reduce the effort to build model-driven solutions. However, current capabilities are not yet fully mature and further research on the creation, testing, and integration of all components in a model-driven ecosystem is required to increase the productivity of developing, maintaining, and evolving a model-driven framework.

For the business app frameworks MD² and MAML, further improvements are possible with regard to extending the set of target platforms, providing additional features such as other layout types, or technical issues such as cloud integration. Also, cross-cutting concerns in the field of mobile apps can be further investigated. For example, accessibility and security are two under-researched aspects whose incorporation in model-driven transformations could automatically enhance all future apps built with the framework [BGG19] but amplify the negative impacts if not considered in generated apps [Olt+18].

Although in the domain of smartphone platforms a tentative duopoly has formed between Android and iOS, the current trend towards app-enablement is just at the beginning. Many new devices are emerging, thus evolving the landscape of app-enabled devices mapped out in our taxonomy (cf. Subsection 3.2.2) by creating new device classes, converging existing ones, or shaping characteristic features. For example, cars with built-in support for apps are just starting to reach the consumer market and approaches to develop apps for them reach from native apps on the car's head unit, screen mirroring of external devices, remote APIs for accessing data, to interactions via the on-board diagnostics port [SV14]. At the same time, car manufacturers are experimenting with embedded infotainment platforms such as Blackberry QNX [QNX18] or form alliances with software vendors offering platforms such as Android Auto [Ren18; OKa19]. A dominant infrastructure towards in-vehicle apps is not yet foreseeable, which demonstrates the complexity of creating cross-platform approaches for emerging device classes.

Regarding the convergence of device classes, so-called *phablets* as a potential compromise between the portability of smartphones and the screen dimensions of tablets proved to be not successful in practice. However, technological advancements of flexible displays currently fuel the hype about *foldable* devices which may or may not lead to a convergence of smartphones and tablets. OS vendors, therefore, also need to evolve and utilise new capabilities of mobile devices through features such as multi-window modes to display apps simultaneously side-by-side [Goo19h]. Moreover, new operating systems are under development. For example, ChromeOS will compete with Android for tablets [Goo19d; Sum18] and the new architecture of Fuchsia OS may supersede Android [Bra18]. Consequently, cross-platform development approaches also have to adopt new interaction possibilities.

Finally, app developers need to rethink the current focus on individual device classes because the plethora of new app-enabled devices will most likely increase the demand for apps on multiple owned devices and separate development efforts for each platform or per device class are probably unsustainable. Pluri-platform development approaches as proposed in this thesis are, therefore, a field of research with many open questions concerning, e.g., user interface design, adequate multi-device interactions, and development methodologies.

References

[AB93] Robert S. Arnold and Shawn A. Bohner. "Impact Analysis-Towards a Framework for Comparison". In: *ICSM*. Vol. 93. 1993, pp. 292–301.

[ACC19] Gopala K. Anumanchipalli, Josh Chartier, and Edward F. Chang. "Speech synthesis from neural decoding of spoken sentences". In: *Nature* 568.7753 (2019), pp. 493–498. DOI: 10.1038/s41586-019-1119-1.

[Ace+15] Roberto Acerbis et al. "Model-driven Development of Cross-platform Mobile Applications with WebRatio and IFML". In: *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. MOBILESoft '15. IEEE Press, 2015, pp. 170–171.

[Ado18] Adobe Systems Inc. *PhoneGap Documentation*. 2018. URL: http://docs.phonegap.com (visited on 07/25/2019).

[AK14] Suyesh Amatya and Arianit Kurti. "Cross-Platform Mobile Development: Challenges and Opportunities". In: *ICT Innovations 2013*. Vol. 231. Springer, 2014, pp. 219–229.

[Ald+10] Marco Aldinucci et al. "Efficient Streaming Applications on Multi-core with FastFlow: The Biosequence Alignment Test-bed". In: *Advances in Parallel Computing* 19 (2010). Ed. by B. Chapman et al. DOI: 10.3233/978-1-60750-530-3-273.

[ANP12] Oren Antebi, Markus Neubrand, and Arno Puder. "Cross-Compiling Android Applications to Windows Phone 7". In: *Mobile Computing, Applications, and Services*. Ed. by Joy Ying Zhang, Jarek Wilkiewicz, and Ani Nahapetian. Springer Berlin Heidelberg, 2012, pp. 283–302.

[Apa19] Apache Software Foundation. *Apache Cordova Documentation*. 2019. URL: https://cordova.apache.org/docs/en/latest/guide/overview/ (visited on 07/25/2019).

[App07] Apple Inc. *Apple Reinvents the Phone with iPhone*. 2007. URL: http://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html (visited on 07/25/2019).

[App08] Apple Inc. *iPhone App Store Downloads Top 10 Million in First Weekend*. 2008. URL: https://www.apple.com/newsroom/2008/07/14iPhone-App-Store-Downloads-Top-10-Million-in-First-Weekend/ (visited on 07/25/2019).

[App19a] Apple Inc. *iOS Human Interface Guidelines*. 2019. URL: https://developer.apple.com/design/human-interface-guidelines/ios/ (visited on 07/25/2019).

[App19b] Apple Inc. *watchOS 5*. 2019. URL: www.apple.com/watchos/ (visited on 07/25/2019).

[Arc01] Architecture Board ORMSC. *Model Driven Architecture (MDA): Document number ormsc/01-07-01.* 2001. URL: http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01 (visited on 07/25/2019).

[Arc14] Architecture Board ORMSC. *MDA Guide revision 2.0: Document number ormsc/14-06-01.* 2014. URL: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01 (visited on 07/25/2019).

[Aut15] Automotive Special Interest Group. *Automotive SPICE Process Reference Model.* 2015. URL: http://automotivespice.com/fileadmin/software-download/Automotive_SPICE_PAM_30.pdf (visited on 07/25/2019).

[Axw19] Axway. *Appcelerator - The mobile first platform.* 2019. URL: https://www.appcelerator.com/ (visited on 07/25/2019).

[Bai+13] Engineer Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (2013), 52:1–52:34. DOI: 10.1145/2501654.2501666.

[Bar+15] Scott Barnett et al. "A Multi-view Framework for Generating Mobile Apps". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 305–306. DOI: 10.1109/VLHCC.2015.7357239.

[Bar19] Adam Bar. *What Web Can Do Today.* 2019. URL: https://whatwebcando.today/ (visited on 07/25/2019).

[BB96] F Hutton Barron and Bruce E Barrett. "Decision quality using ranked attribute weights". In: *Management science* 42.11 (1996), pp. 1515–1523.

[BC00] John A. Bargh and Tanya L. Chartrand. "Studying the mind in the middle: A practical guide to priming and automaticity research. Handbook of research methods in social psychology". In: *Handbook of research methods in social and personality psychology.* Ed. by C. M. Judd and H. T. Reis. Cambridge University Press, 2000, pp. 253–285.

[BC90] Gilad Bracha and William Cook. "Mixin-based inheritance". In: *ACM SIGPLAN Notices* 25.10 (Oct. 1990), pp. 303–311. DOI: 10.1145/97946.97982.

[BDF09] Marco Brambilla, Matteo Dosmi, and Piero Fraternali. "Model-driven engineering of service orchestrations". In: *5th World Congress on Services* (2009). DOI: 10.1109/SERVICES-I.2009.94.

[Ben+16] Hanane Benouda et al. "MDA Approach to Automate Code Generation for Mobile Applications". In: *Mobile and Wireless Technologies 2016.* Ed. by Kuinam J. Kim, Naruemon Wattanapongsakorn, and Nikolai Joukov. Vol. 391. Lecture Notes in Electrical Engineering. Springer Singapore, 2016, pp. 241–250. DOI: 10.1007/978-981-10-1409-3_27.

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing, 2013.

[BGG19] Andreas Biørn-Hansen, Tor-Morten Grønli, and George Ghinea. "A survey and taxonomy of core concepts and research challenges in cross-platform mobile development". In: *ACM Computing Surveys* 51.5 (2019). DOI: 10.1145/3241739.

[Bho13] Achintya K. Bhowmik. "39.2: Invited Paper: Natural and Intuitive User Interfaces: Technologies and Applications". In: *SID Symposium Digest of Technical Papers* 44.1 (2013), pp. 544–546. DOI: 10.1002/j.2168-0159.2013.tb06266.x.

[Biø+19] Andreas Biørn-Hansen et al. "An Empirical Study of Cross-Platform Mobile Development in Industry". In: *Wireless Communications and Mobile Computing* 2019.2 (2019), pp. 1–12. DOI: 10.1155/2019/5743892.

[Biz16] Bizness Apps. *Mobile App Maker | Bizness Apps*. 2016. URL: http://biznessapps.com/ (visited on 07/25/2019).

[BKF14] Ruth Breu, Annie Kuntzmann-Combelles, and Michael Felderer. "New Perspectives on Software Quality [Guest editors' introduction]". In: *IEEE Software* 31.1 (2014), pp. 32–38. DOI: 10.1109/MS.2014.9.

[BKM09] Aaron Bangor, Philip Kortum, and James Miller. "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale". In: *J. Usability Studies* 4.3 (2009), pp. 114–123.

[BMG18] Andreas Biørn-Hansen, Tim A. Majchrzak, and Tor-Morten Grønli. "Progressive Web Apps for the Unified Development of Mobile Applications". In: *Revised Selected Papers WEBIST 2017*. Ed. by Tim A. Majchrzak et al. Vol. 322. Lecture Notes in Business Information Processing (LNBIP). Springer, 2018, pp. 64–86.

[BMP13] Elke Bouillon, Patrick Mäder, and Ilka Philippow. "A Survey on Usage Scenarios for Requirements Traceability in Practice". In: *Requirements Engineering: Foundation for Software Quality*. Springer, 2013, pp. 158–173. DOI: 10.1007/978-3-642-37422-7_12.

[BMU14] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. "Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End". In: *Lecture Notes in Computer Science* 8640 (2014), pp. 176–191. DOI: 10.1007/978-3-319-10359-4_15.

[Bra18] Kyle Bradshaw. *Huawei is testing Google's Fuchsia OS on the Honor Play*. 2018. URL: https://9to5google.com/2018/11/22/google-fuchsia-huawei-honor-play/ (visited on 07/25/2019).

[BRK17a] Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Domain-specific Language for Configurable Traceability Analysis". In: *5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. Ed. by Luis Ferreira Pires, Slimane Hammoudi, and Bran Selic. SCITEPRESS, 2017, pp. 374–381.

[BRK17b] Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Model-Driven Approach for Evaluating Traceability Information". In: *Third International Conference on Advances and Trends in Software Engineering (SOFTENG)*. Ed. by Mira Kajko-Mattsson, Pål Ellingsen, and Paolo Maresca. 2017, pp. 59–65.

[Bro19] Jas Brooks. "Promises of the Virtual Museum". In: *XRDS* 25.2 (Jan. 2019), pp. 46–50. DOI: 10.1145/3301483.

[Bro96] John Brooke. "SUS-A quick and dirty usability scale". In: *Usability evaluation in industry*. Ed. by P. W. Jordan et al. Taylor and Francis, 1996, pp. 189–194.

[Bub19] Bubble Group. *Bubble - Visual Programming*. 2019. URL: https://bubble.is/ (visited on 07/25/2019).

[Buc18] Thomas Buchmann. "BXtend - A Framework for (Bidirectional) Incremental Model Transformations". In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2018. DOI: 10.5220/0006563503360345.

[Cam19] Camunda Services GmbH. *Workflow And Decision Automation Platform - Camunda BPMN*. 2019. URL: https://camunda.org/ (visited on 07/25/2019).

[CCZ07] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. "Parallel Programmability and the Chapel Language". In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[CFS16] Alessandro Carcangiu, Gianni Fenu, and Lucio Davide Spano. "A design pattern for multimodal and multidevice user interfaces". In: *8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM. 2016, pp. 177–182.

[CG15] Matteo Ciman and Ombretta Gaggi. "Measuring energy consumption of cross-platform frameworks for mobile applications". In: *LNBIP* 226 (2015), pp. 331–346. DOI: 10.1007/978-3-319-27030-2_21.

[CGG14] Matteo Ciman, Ombretta Gaggi, and Nicola Gonzo. "Cross-platform mobile development: A study on apps with animations". In: *Proc. ACM Symposium on Applied Computing*. 2014. DOI: 10.1145/2554850.2555104.

[Cho14] Looi Theam Choy. "The strengths and weaknesses of research methodology: Comparison and complimentary between qualitative and quantitative approaches". In: *IOSR Journal of Humanities and Social Science* 19.4 (2014), pp. 99–104.

[Chu+12] Byung-Gon Chun et al. "Mobius: Unified messaging and data serving for mobile apps". In: *10th International Conference on Mobile Systems, Applications, and Services - MobiSys '12*. Ed. by Nigel Davies, Srinivasan Seshan, and Lin Zhong. ACM Press, 2012, pp. 141–154. DOI: 10.1145/2307636.2307650.

[CK10] Philipp Ciechanowicz and Herbert Kuchen. "Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures". In: *IEEE International Conference on High Performance Computing and Communications (HPPC '10)*. IEEE, 2010, pp. 108–113. DOI: 10.1109/HPCC.2010.23.

[CL95] Rita J. Costello and Dar-Biau Liu. "Metrics for requirements engineering". In: *Journal of Systems and Software* 29.1 (1995), pp. 39–63. DOI: 10.1016/0164-1212(94)00127-9.

[Cle+14] Jane Cleland-Huang et al. "Software Traceability: Trends and Future Directions". In: *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 55–69. DOI: 10.1145/2593882.2593891.

[Col91] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[Cor+18] Leonardo Corbalan et al. "Development Frameworks for Mobile Devices: A Comparative Study About Energy Consumption". In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '18. ACM, 2018, pp. 191–201. DOI: 10.1145/3197231.3197242.

[CP10] Walter Cazzola and Davide Poletti. "DSL Evolution through Composition". In: *7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*. 2010, pp. 1–6. DOI: 10.1145/1890683.1890689.

[Čre+10] Matej Črepinšek et al. "On automata and language based grammar metrics". In: *Computer Science and Information Systems* 7.2 (2010), pp. 309–330. DOI: 10.2298/CSIS1002309C.

[CV16] Walter Cazzola and Edoardo Vacchi. "Language components for modular DSLs using traits". In: *Computer Languages, Systems & Structures* 45 (Apr. 2016), pp. 16–34. DOI: 10.1016/j.cl.2015.12.001.

[Cza98] Krzysztof Czarnecki. "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models". Dissertation. Ilmenau: Technical University of Ilmenau, 1998.

[Dag+16] Jan C. Dageförde et al. "Generating App Product Lines in a Model-Driven Cross-Platform Development Approach". In: *Hawaii International Conference on System Sciences (HICSS)*. 2016.

[Dal+13] Isabelle Dalmasso et al. "Survey, comparison and evaluation of cross platform mobile application development tools". In: *International Wireless Communications and Mobile Computing Conference (IWCMC)*. 2013. DOI: 10.1109/IWCMC.2013.6583580.

[dB14] Luis Pires da Silva and Fernando Brito e Abreu. "Model-driven GUI generation and navigation for Android BIS apps". In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2014, pp. 400–407.

[Die+15] Diego Albuquerque et al. "Quantifying usability of domain-specific languages: An empirical study on software maintenance". In: *Journal of Systems and Software* 101 (2015), pp. 245–259. DOI: 10.1016/j.jss.2014.11.051.

[DLT87] Richard L. Daft, Robert H. Lengel, and Linda Klebe Trevino. "Message Equivocality, Media Selection, and Manager Performance: Implications for Information Systems". In: *MIS quarterly* 11.3 (1987), p. 355. DOI: 10.2307/248682.

[DM15] Sunny Dhillon and Qusay H. Mahmoud. "An evaluation framework for cross-platform mobile application development tools". In: *Software – Prac. and Exp.* 45.10 (2015), pp. 1331–1357. DOI: 10.1002/spe.2286.

[Dob12] Alex Dobie. *Why you'll never have the latest version of Android*. 2012. URL: http://www.androidcentral.com/why-you-ll-never-have-latest-version-android (visited on 07/25/2019).

[DP12] Jose Danado and Fabio Paternò. "Puzzle: A Visual-Based Environment for End User Development in Touch-Based Mobile Phones". In: *Human-Centered Software Engineering (HCSE)*. Ed. by Marco Winckler, Peter Forbrig, and Regina Bernhaupt. Springer, 2012, pp. 199–216. DOI: 10.1007/978-3-642-34347-6_12.

[DP13] Alexander Delater and Barbara Paech. "Analyzing the Tracing of Requirements and Source Code during Software Development". In: *Requirements Engineering: Foundation for Software Quality*. Springer, 2013, pp. 308–314. DOI: 10.1007/978-3-642-37422-7_22.

[Dri19] Drifty Co. *Ionic - Cross-Platform Mobile App Development*. 2019. URL: https://ionicframework.com/ (visited on 07/25/2019).

[DTK16] Marco Danelutto, Massimo Torquati, and Peter Kilpatrick. "A DSL Based Toolchain for Design Space Exploration in Structured Parallel Programming". In: *Procedia Computer Science* 80 (2016), pp. 1519–1530. DOI: 10.1016/j.procs.2016.05.477.

[Duc+06] Stéphane Ducasse et al. "Traits: A Mechanism for Fine-Grained Reuse". In: *ACM Transactions on Programming Languages and Systems* 28.2 (2006), pp. 331–388. DOI: 10.1145/1119479.1119483.

[Ecl19a] Eclipse Foundation Inc. *Eclipse Modeling Project*. 2019. URL: https://www.eclipse.org/modeling/emf/ (visited on 07/25/2019).

[Ecl19b] Eclipse Foundation Inc. *Eclipse QVT Operational*. 2019. URL: http://www.eclipse.org/mmt/qvto (visited on 03/08/2019).

[Ecl19c] Eclipse Foundation Inc. *GEF*. 2019. URL: https://www.eclipse.org/gef/ (visited on 07/25/2019).

[Ecl19d] Eclipse Foundation Inc. *Graphical Modeling Framework*. 2019. URL: https://www.eclipse.org/modeling/gmp/ (visited on 07/25/2019).

[Ecl19e] Eclipse Foundation Inc. *Sirius*. 2019. URL: https://eclipse.org/sirius/ (visited on 02/26/2019).

[Ecl19f] Eclipse Foundation Inc. *Xtend - Modernized Java*. 2019. URL: http://www.eclipse.org/xtend/ (visited on 02/26/2019).

[Ecl19g] Eclipse Foundation Inc. *Xtext Documentation*. 2019. URL: https://eclipse.org/Xtext/documentation/ (visited on 02/26/2019).

[EEM16] Sören Evers, Jan Ernsting, and Tim A. Majchrzak. "Towards a Reference Architecture for Model-Driven Business Apps". In: *Hawaii International Conference on System Sciences (HICSS)*. 2016.

[EH07] Torbjörn Ekman and Görel Hedin. "The JastAdd system — modular extensible compiler construction". In: *Science of Computer Programming* 69.1–3 (2007), pp. 14–26. DOI: 10.1016/j.scico.2007.02.003.

[EI10] Hidekazu Enjo and Junichi Iijima. "Towards Class Diagram Algebra for Composing Data Models". In: *Proceedings of the 2010 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the 9th SoMeT_10*. IOS Press, 2010, pp. 112–133.

[EK12] Steffen Ernsting and Herbert Kuchen. "Algorithmic Skeletons for Multi-core, Multi-GPU Systems and Clusters". In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138. DOI: 10.1504/IJHPCN.2012.046370.

[EK17] Steffen Ernsting and Herbert Kuchen. "Data Parallel Algorithmic Skeletons with Accelerator Support". In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299. DOI: 10.1007/s10766-016-0416-7.

[El-+17] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* 8.2 (2017), pp. 163–190. DOI: 10.1016/j.asej.2015.08.004.

[ELK17] August Ernstsson, Lu Li, and Christoph Kessler. "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems". In: *International Journal of Parallel Programming* (2017). DOI: 10.1007/s10766-017-0490-5.

[eMa17] eMarketer Inc. *Average Time Spent per Day with Major Media by US Adults, 2017 (hrs:mins)*. 2017. URL: https://www.emarketer.com/Chart/Average-Time-Spent-per-Day-with-Major-Media-by-US-Adults-2017-hrsmins/206481 (visited on 07/25/2019).

[Erd+13] Sebastian Erdweg et al. "The State of the Art in Language Workbenches". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8225 LNCS. Springer, 2013, pp. 197–217. DOI: 10.1007/978-3-319-02654-1_11.

[Erd+15] Sebastian Erdweg et al. "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer Languages, Systems & Structures* 44, Part A (2015), pp. 24–47. DOI: 10.1016/j.cl.2015.08.007.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *12th International Conference on Web Information Systems and Technologies (WEBIST)*. SCITEPRESS, 2016, pp. 307–316. DOI: 10.5220/0005862103070316.

[EVP01] Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. "Applying model-based techniques to the development of UIs for mobile computers". In: *IUI* (2001).

[Fac19] Facebook Inc. *React Native - A framework for building native apps using React*. 2019. URL: https://facebook.github.io/react-native/ (visited on 07/25/2019).

[FMO11] Fazal-e-Amin, Ahmad Kamil Mahmood, and Alan Oxley. "An analysis of object oriented variability implementation mechanisms". In: *ACM SIGSOFT Software Engineering Notes* 36.1 (2011), p. 1.

[Fow05] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: http://martinfowler.com/articles/languageWorkbench.html (visited on 03/25/2019).

[FR07] Robert France and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap". In: *Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54. (Visited on 06/09/2014).

[Fra+06] Robert B. France et al. "Model-Driven Development Using UML 2.0: Promises and Pitfalls". In: *Computer* 39.2 (2006), pp. 59–66. DOI: 10.1109/MC.2006.65.

[Fra+15] Rita Francese et al. "Model-Driven Development for Multi-platform Mobile Applications". In: *International Conference for Product-Focused Software Process Improvement (PROFES)*. Ed. by Pekka Abrahamsson et al. Springer International Publishing, 2015, pp. 61–67. DOI: 10.1007/978-3-319-26844-6_5.

[Fra+17] R. Francese et al. "Mobile App Development and Management: Results from a Qualitative Investigation". In: *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2017, pp. 133–143. DOI: 10.1109/MOBILESoft.2017.33.

[Ful+06] J. B. Fuller et al. "Perceived external prestige and internal respect: New insights into the organizational identification process". In: *Human Relations* 59.6 (2006), pp. 815–846. DOI: 10.1177/0018726706067148.

[Gam+95] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995.

[GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. "An analysis of the requirements traceability problem". In: *IEEE International Conference on Requirements Engineering*. 1994, pp. 94–101. DOI: 10.1109/ICRE.1994.292398.

[GH99] Günther Gediga and Kai-Christoph Hamborg. "IsoMetrics: Ein Verfahren zur Evaluation von Software nach ISO 9241-10". In: *Evaluationsforschung. Göttingen: Hogrefe* (1999), pp. 195–234.

[GK91] Geoffrey K. Gill and Chris F. Kemerer. "Cyclomatic complexity density and software maintenance productivity". In: *IEEE Transactions on Software Engineering* 17.12 (Dec. 1991), pp. 1284–1288. DOI: 10.1109/32.106988.

[GL10] Horacio González-Vélez and Mario Leyton. "A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers". In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160. DOI: 10.1002/spe.1026.

[Goo19a] GoodBarber. *Create an app with GoodBarber app builder*. 2019. URL: https://www.goodbarber.com/ (visited on 07/25/2019).

[Goo19b] Google LLC. *AMP - a web component framework to easily create uder-first web experiences*. 2019. URL: https://amp.dev/ (visited on 07/25/2019).

[Goo19c] Google LLC. *Android Developers - design for Android*. 2019. URL: https://developer.android.com/design/ (visited on 03/03/2019).

[Goo19d] Google LLC. *Chrome OS devices*. 2019. URL: https://developer.android.com/chrome-os (visited on 07/25/2019).

[Goo19e] Google LLC. *Flutter - beautiful native apps in record time*. 2019. URL: https://flutter.dev/ (visited on 07/25/2019).

[Goo19f] Google LLC. *Instant Apps*. 2019. URL: https://developer.android.com/topic/google-play-instant (visited on 07/25/2019).

[Goo19g] Google LLC. *J2ObjC*. 2019. URL: http://j2objc.org/ (visited on 07/25/2019).

[Goo19h] Google LLC. *Multi-window support*. 2019. URL: https://developer.android.com/guide/topics/ui/multi-window (visited on 07/25/2019).

[Goo19i] Google LLC. *Wear OS by Google Smartwatches*. 2019. URL: https://wearos.google.com/ (visited on 07/25/2019).

[Gra+15] David Granada et al. "Analysing the cognitive effectiveness of the WebML visual notation". In: *Software & Systems Modeling* (2015). DOI: 10.1007/s10270-014-0447-8.

[Gra19] Gradle Inc. *Gradle Build Tool*. 2019. URL: https://gradle.org/ (visited on 07/25/2019).

[Gri+17] D. Griebler et al. "SPar: A DSL for high-level and productive stream parallelism". In: *PPL* 27.01 (2017), p. 1740005.

[GS03] Jack Greenfield and Keith Short. "Software factories". In: *18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*. Ed. by Ron Crocker and Guy L. Steele. 2003, pp. 16–27. DOI: 10.1145/949344.949348.

[Gub+13] Jayavardhana Gubbi et al. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660. DOI: 10.1016/j.future.2013.01.010.

[HBB01] Marc Hassenzahl, Andreas Beu, and Michael Burmester. "Engineering joy". In: *IEEE Software* 18.1 (2001), pp. 70–76. DOI: 10.1109/52.903170.

[HEE13] Shah Rukh Humayoun, Stefan Ehrhart, and Achim Ebert. "Developing Mobile Apps Using Cross-Platform Frameworks: A Case Study". In: *International Conference on Human-Computer Interaction. Human-Centred Design Approaches, Methods, Tools, and Environments (HCI International), Part I*. Ed. by Masaaki Kurosu. Springer Berlin Heidelberg, 2013, pp. 371–380. DOI: 10.1007/978-3-642-39232-0_41.

[Hei+14] Henning Heitkötter et al. "Comparison of Mobile Web Frameworks". In: *LNBIP*. Vol. 189. Springer, 2014, pp. 119–137.

[Hev+04] Alan R. Hevner et al. "Design Science in Information Systems Research". In: *MIS Q* 28.1 (2004), pp. 75–105.

[HHH15] A. Hudli, S. Hudli, and R. Hudli. "An evaluation framework for selection of mobile app development platform". In: *Proc. 3rd MobileDeLi*. 2015. DOI: 10.1145/2846661.2846678.

[HHM13] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. "Evaluating Cross-Platform Development Approaches for Mobile Applications". In: *Proceedings 8th WEBIST, Revised Selected Papers*. Vol. 140. LNBIP. Springer, 2013, pp. 120–138.

[Hit19] Hitachi Vantara Corp. *Pentaho Data Integration*. 2019. URL: https://www.hitachivantara.com/en-us/products/big-data-integration-analytics/pentaho-data-integration.html (visited on 07/25/2019).

[HMK13] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. "Cross-platform model-driven development of mobile applications with MD2". In: *Annual ACM Symposium on Applied Computing*. Ed. by Sung Y. Shin and José Carlos Maldonado. SAC '13. ACM, 2013, pp. 526–533. DOI: 10.1145/2480362.2480464.

[Hob80] Benjamin F Hobbs. "A comparison of weighting methods in power plant siting". In: *Decision Sciences* 11.4 (1980), pp. 725–737.

[HV11] Zef Hemel and Eelco Visser. "Declaratively Programming the Mobile Web with Mobl". In: *ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. ACM, 2011, pp. 695–712. DOI: 10.1145/2048066.2048121.

[Iba+17] Jorge E. Ibarra-Esquer et al. “Tracking the Evolution of the Internet of Things Concept Across Different Application Domains”. In: *Sensors (Basel, Switzerland)* 17.6 (2017). DOI: 10.3390/s17061379.

[IBM16] IBM. *Rational DOORS Family*. 2016. URL: https://www.ibm.com/us-en/marketplace/rational-doors (visited on 07/25/2019).

[Into6] International Organization for Standardization. *ISO 9241-110:2006*. 2006.

[Int11] International Organization for Standardization. *ISO 9241-210:2011*. 2011.

[Int85] International Organization for Standardization. *ISO 5807:1985*. 1985.

[Jäg+16] Sven Jäger et al. “Creation of domain-specific languages for executable system models with the Eclipse Modeling Project”. In: *Annual IEEE Systems Conference (SysCon)*. 2016, pp. 1–8. DOI: 10.1109/SYSCON.2016.7490558.

[Jav+08] Faizan Javed et al. “MARS: A metamodel recovery system using grammar inference”. In: *Information and Software Technology* 50.9-10 (2008), pp. 948–968. DOI: 10.1016/j.infsof.2007.08.003.

[JB13] Slinger Jansen and Ewoud Bloemendal. “Defining App Stores: The Role of Curated Marketplaces in Software Ecosystems”. In: *Software Business. From Physical Products to Software Services and Solutions. ICSOB 2013*. Ed. by Georg Herzwurm and Tiziana Margaria. Vol. 150. Lecture Notes in Business Information Processing. Springer, 2013, pp. 195–206. DOI: 10.1007/978-3-642-39336-5_19.

[Jet19] JetBrains s.r.o. *MPS: The Domain-Specific Language Creator by JetBrains*. 2019. URL: https://www.jetbrains.com/mps (visited on 07/25/2019).

[JJ14] Chris Jones and Xiaoping Jia. “The AXIOM Model Framework: Transforming Requirements to Native Code for Cross-platform Mobile Applications”. In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, 2014.

[JM15] Chakajkla Jesdabodi and Walid Maalej. “Understanding Usage States on Mobile Devices”. In: *2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. UbiComp ’15. ACM, 2015, pp. 1221–1225. DOI: 10.1145/2750858.2805837.

[KB17] Jan Köhnlein and Cedric Brun. *Xtext / Sirius - Integration: The Main Use-Cases. White Paper*. 2017.

[Kej09] Ankita Arvind Kejriwal. “MobiDSL - a Domain Specific Language for Mobile Web Applications: Developing applications for mobile platform without web programming”. In: *OOPSLA Workshop on Domain-Specific Modeling* (2009).

[Kel+12] Patrick Gage Kelley et al. "A Conundrum of Permissions: Installing Applications on an Android Smartphone". In: *Financial Cryptography and Data Security: FC 2012 Workshops, USEC and WECSR*. Ed. by Jim Blyth, Sven Dietrich, and L. Jean Camp. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 68–79. DOI: 10.1007/978-3-642-34638-5_6.

[KFF16] Elmar Krainz, Johannes Feiner, and Martin Fruhmann. "Accelerated Development for Accessible Apps – Model Driven Development of Transportation Apps for Visually Impaired People". In: *Human-Centered and Error-Resilient Systems Development*. Ed. by Cristian Bogdan et al. Springer International Publishing, 2016, pp. 374–381.

[Kha19] Vladimir Kharlampidi. *Framework7 - Full featured framework for building iOS, Android & Desktop apps*. 2019. URL: https://framework7.io/ (visited on 07/25/2019).

[Kic+97] Gregor Kiczales et al. "Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming*. Ed. by Gerhard Goos et al. Vol. 1241. Lecture Notes in Computer Science. 1997, pp. 220–242. DOI: 10.1007/BFb0053381.

[KK16] István Koren and Ralf Klamma. "The Direwolf Inside You: End User Development for Heterogeneous Web of Things Appliances". In: *International Conference on Web Engineering (ICWE)*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Springer, 2016, pp. 484–491. DOI: 10.1007/978-3-319-38791-8_35.

[Knu+13] David Knuplesch et al. "Visual Modeling of Business Process Compliance Rules with the Support of Multiple Perspectives". In: *Conceptual Modeling (ER)*. Ed. by Wilfred Ng, Veda C. Storey, and Juan C. Trujillo. Springer, 2013, pp. 106–120. DOI: 10.1007/978-3-642-41924-9_10.

[Kon19] Kony. *Kony Quantum - low-code without limits*. 2019. URL: https://www.kony.com/quantum/ (visited on 07/25/2019).

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "MontiCore: a framework for compositional development of domain specific languages". In: *International Journal on Software Tools for Technology Transfer* 12.5 (Sept. 2010), pp. 353–372. DOI: 10.1007/s10009-010-0142-1.

[Kuc02] Herbert Kuchen. "A Skeleton Library". In: *International Euro-Par Conference on Parallel Processing*. Ed. by B. Monien and R. Feldmann. Vol. 2400. LNCS. Springer, 2002, pp. 620–629.

[Kun+14] Michael Kunz et al. "Analyzing Recent Trends in Enterprise Identity Management". In: *25th International Workshop on Database and Expert Systems Applications*. 2014, pp. 273–277. DOI: 10.1109/DEXA.2014.62.

[KvV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2009, pp. 168–177. DOI: 10.1109/SCAM.2009.28.

[Lel17] Wm Leler. *What's Revolutionary about Flutter*. 2017. URL: https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514 (visited on 07/25/2019).

[LG 19] LG Electronics. *Smart TVs: WebOS Ready LG Smart TVs*. 2019. URL: https://www.lg.com/uk/smart-tvs (visited on 07/25/2019).

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. "DSMDiff: a differentiation tool for domain-specific models". In: *European Journal of Information Systems* 16.4 (2007), pp. 349–361. DOI: 10.1057/palgrave.ejis.3000685.

[Liu+12] Qichao Liu et al. "Application of Metamodel Inference with Large-Scale Metamodels". In: *International Journal of Software and Informatics* 6.2 (2012), pp. 201–231.

[LJJ07] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. "DSL classification". In: *OOPSLA Workshop on Domain-Specific Modeling*. 2007.

[LKN17] Houssem Lahiani, Monji Kherallah, and Mahmoud Neji. "Vision Based Hand Gesture Recognition for Mobile Devices: A Review". In: *Proceedings of the 16th International Conference on Hybrid Intelligent Systems (HIS 2016)*. Ed. by Ajith Abraham et al. Springer International Publishing, 2017, pp. 308–318. DOI: 10.1007/978-3-319-52941-7_31.

[LL15] A. De Luca and J. Lindqvist. "Is secure and usable smartphone authentication asking too much?" In: *Computer* 48.5 (May 2015), pp. 64–68. DOI: 10.1109/MC.2015.134.

[Lóp+15] Jesús J. López-Fernández et al. "Example-driven meta-model development". In: *Software & Systems Modeling* 14.4 (2015), pp. 1323–1347. DOI: 10.1007/s10270-013-0392-y.

[LSH06] Bettina Laugwitz, Martin Schrepp, and Theo Held. "Konstruktion eines Fragebogens zur Messung der User Experience von Softwareprodukten". In: *Mensch und Computer 2006: Mensch und Computer im Strukturwandel*. Ed. by Andreas M. Heinecke and Hansjürgen Paul. München: Oldenbourg Verlag, 2006, pp. 125–134.

[Lud17] Ludei Inc. *Canvas+ Cocoon Documentation*. 2017. URL: https://docs.cocoon.io/article/canvas-engine/ (visited on 07/25/2019).

[LvV14] Assistant N. Luciano Morganti, Shenja van der Graaf, and Carina Veeckman. "Designing for participatory governance: Assessing capabilities and toolkits in public service delivery". In: *info* 16.6 (2014), pp. 74–88. DOI: 10.1108/info-07-2014-0028.

[LW13] Olivier Le Goaer and Sacha Waltham. "Yet another DSL for cross-platforms mobile development". In: *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*. Ed. by Benoit Combemale, Walter Cazzola, and Robert B. France. 2013, pp. 28–33. DOI: 10.1145/2489812.2489819.

[Maj+17] Tim A. Majchrzak et al. "How Cross-Platform Technology Can Facilitate Easier Creation of Business Apps". In: *Apps Management and E-Commerce Transactions in Real-Time*. Ed. by In Lee and Sajad Rezaei. Advances in E-Business Research. IGI Global, 2017, pp. 104–140. DOI: 10.4018/978-1-5225-2449-6.ch005.

[Man+18] Amit Kr Mandal et al. "Vulnerability Analysis of Android Auto Infotainment Apps". In: *ACM International Conference on Computing Frontiers*. CF '18. ACM, 2018, pp. 183–190. DOI: 10.1145/3203217.3203278.

[Man18] Amit Manchanda. *12 Mobile App Development Trends to Watch Out for in 2019*. 2018. URL: https://www.netsolutions.com/insights/12-mobile-app-development-trends-to-watch-out-for-in-2019/ (visited on 07/25/2019).

[MBG18] Tim A. Majchrzak, Andreas Biørn-Hansen, and Tor-Morten Grønli. "Progressive Web Apps: the Definite Approach to Cross-Platform Development?" In: *Proceedings 51th Hawaii International Conference on Systems Science (HICSS-51)*. AIS Electronic Library (AISeL), 2018.

[MC09] Jonathan I. Maletic and Michael L. Collard. "TQL: A query language to support traceability". In: *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*. 2009, pp. 16–20. DOI: 10.1109/TEFSE.2009.5069577.

[MC13] Patrick Mäder and Jane Cleland-Huang. "A visual language for modeling and executing traceability queries". In: *Software and Systems Modeling* 12.3 (2013), pp. 537–553. DOI: 10.1007/s10270-012-0237-0.

[ME15] Tim A. Majchrzak and Jan Ernsting. "Reengineering an Approach to Model-Driven Development of Business Apps". In: *SIGSAND/PLAIS EuroSymposium*. Gdansk, Poland, 2015, pp. 15–31. DOI: 10.1007/978-3-319-24366-5_2.

[MEK15] Tim A. Majchrzak, Jan Ernsting, and Herbert Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14. DOI: 10.19210/OJIS_2015v2i2n02_Majchrzak.

[Mel+16] Santiago Meliá et al. "Comparison of a textual versus a graphical notation for the maintainability of MDE domain models: An empirical pilot study". In: *Software Quality Journal* 24.3 (2016), pp. 709–735. DOI: 10.1007/s11219-015-9299-x.

[Men+18] Breno A.M. Menezes et al. "Parameter selection for swarm intelligence algorithms - Case study on parallel implementation of FSS". In: vol. 2017-November. 2018, pp. 1–6. DOI: 10.1109/LA-CCI.2017.8285694.

[Met19] MetaBorg. *The Spoofax Language Workbench*. 2019. URL: https://www.metaborg.org (visited on 07/25/2019).

[Meu+00] Matthew L Meuter et al. "Self-service technologies: understanding customer satisfaction with technology-based service encounters". In: *Journal of marketing* 64.3 (2000), pp. 50–64.

[MGP13] Patrick Mäder, Orlena Gotel, and Ilka Philippow. "Getting back to basics: Promoting the use of a traceability information model in practice". In: *7th Intl. Workshop on Traceability in Emerging Forms of Software Engineering* (2013), pp. 21–25. DOI: 10.1109/TEFSE.2009.5069578.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[Mic19] Microsoft Corp. *Xamarin Documentation*. 2019. URL: https://developer.xamarin.com (visited on 07/25/2019).

[Mon19] Monaca Inc. *Onsen UI 2: Beautiful HTML5 Hybrid Mobile App Framework and Tools*. 2019. URL: https://onsen.io/ (visited on 07/25/2019).

[Moo09] Daniel Moody. "The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 756–778.

[MS15] Tim A. Majchrzak and Matthias Schulte. "Context-Dependent Testing of Applications for Mobile Devices". In: *Open Journal of Web Technologies (OJWT)* 2.1 (2015), pp. 27–39.

[MS95] Salvatore T. March and Gerald F. Smith. "Design and natural science research on information technology". In: *Decision Support Systems* 15.4 (1995), pp. 251–266. DOI: 10.1016/0167-9236(94)00041-2.

[Mye90] Brad A. Myers. "Taxonomies of visual programming and program visualization". In: *Journal of Visual Languages & Computing* 1.1 (1990), pp. 97–123. DOI: 10.1016/S1045-926X(05)80036-9.

[Nak+14] Elisa Y. Nakagawa et al. "Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures". In: *IEEE/IFIP Conference on Software Architecture*. 2014, pp. 143–152. DOI: 10.1109/WICSA.2014.25.

[Nam+16] Abdallah Namoun et al. "Exploring Mobile End User Development: Existing Use and Design Factors". In: *IEEE Transactions on Software Engineering* 42.10 (2016), pp. 960–976. DOI: 10.1109/TSE.2016.2532873.

[Nie93] Jakob Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 1993.

[NJE17] Timothy Neate, Matt Jones, and Michael Evans. "Cross-device Media: A Review of Second Screening and Multi-device Television". In: *Personal Ubiquitous Comput* 21.2 (2017), pp. 391–405. DOI: 10.1007/s00779-017-1016-2.

[Obj11] Object Management Group. *Business Process Model and Notation 2.0*. 2011.

[Obj15a] Object Management Group. *Interaction Flow Modeling Language 1.0*. 2015.

[Obj15b] Object Management Group. *Unified Modeling Language 2.5*. 2015.

[OKa19] Sean O'Kane. *The all-electric Polestar 2 will be the first car with Google's native Android Auto*. 2019. URL: https://www.theverge.com/2019/1/4/18168964/volvo-polestar-2-ev-android-auto-google-assistant (visited on 07/25/2019).

[Olt+18] Marten Oltrogge et al. "The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators". In: *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 634–647. DOI: 10.1109/SP.2018.00005.

[Out19] OutSystems. *The #1 Low-Code Platform for digital transformation - OutSystems*. 2019. URL: https://www.outsystems.com/ (visited on 07/25/2019).

[Par98] Donn B. Parker. *Fighting Computer Crime: A New Framework for Protecting Information*. New York, NY, USA: John Wiley & Sons, Inc, 1998.

[Pat00] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2000. DOI: 10.1007/978-1-4471-0445-2.

[Pef+07] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77. DOI: 10.2753/MIS0742-1222240302.

[Pes+15] Ana Pescador et al. "Pattern-based development of Domain-Specific Modelling Languages". In: *International Conference on Model Driven Engineering Languages and Systems*. Sept. 2015, pp. 166–175. DOI: 10.1109/MODELS.2015.7338247.

[Pro16] Product Hunt. *7 Tools to Help You Build an App Without Writing Code*. 2016. URL: https://medium.com/product-hunt/7-tools-to-help-you-build-an-app-without-writing-code-cb4eb8cfe394 (visited on 07/25/2019).

[Pro19] Progress Software Corp. *Native mobile apps with Angular, Vue.js, TypeScript, JavaScript - NativeScript*. 2019. URL: https://www.nativescript.org/ (visited on 07/25/2019).

[PSC12] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. "Comparison of cross-platform mobile development tools". In: *Proc. 16th ICIN*. IEEE, 2012, pp. 179–186. DOI: 10.1109/ICIN.2012.6376023.

[QGZ17] Peixin Que, Xiao Guo, and Maokun Zhu. "A Comprehensive Comparison between Hybrid and Native App Paradigms". In: *International Conference on Computational Intelligence and Communication Networks (CICN)*. Ed. by Tomar G.S. IEEE, 2017, pp. 611–614. DOI: 10.1109/CICN.2016.125.

[QNX18] QNX Software Systems Limited. *QNX operating systems, development tools, and professional services for connected embedded systems*. 2018. URL: http://blackberry.qnx.com/de (visited on 07/25/2019).

[QPM17] Ricardo Queirós, Filipe Portela, and José Machado. "Magni - A Framework for Developing Context-Aware Mobile Applications". In: *Recent Advances in Information Systems and Technologies: Volume 3*. Ed. by Álvaro Rocha et al. Springer International Publishing, 2017, pp. 417–426. DOI: 10.1007/978-3-319-56541-5_43.

[RB01] Erhard Rahm and Philip A. Bernstein. "A survey of approaches to automatic schema matching". In: *VLDB Journal* 10.4 (2001), pp. 334–350. DOI: 10.1007/s007780100057.

[RD90] Detlef Rhenius and Gerhard Deffner. "Evaluation of Concurrent Thinking Aloud Using Eye-tracking Data". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 34.17 (1990), pp. 1265–1269. DOI: 10.1177/154193129003401719.

[Rec10] Jan Recker. "Opportunities and constraints: The current struggle with BPMN". In: *Business Process Management Journal* 16.1 (2010), pp. 181–201. DOI: 10.1108/14637151011018001.

[Ren18] Renault-Nissan-Mitsubishi. *Renault-Nissan-Mitsubishi and Google join forces on next-generation infotainment*. 2018. URL: https://www.alliance-2022.com/news/renault-nissan-mitsubishi-and-google-join-forces-on-next-generation-infotainment/ (visited on 07/25/2019).

[Res+18] Marty Resnick et al. *Magic Quadrant for Mobile App Development Platforms*. 2018. URL: https://www.gartner.com/en/documents/3882864 (visited on 07/25/2019).

[Res14] Research2guidance. *Cross-Platform Tool Benchmarking 2014*. 2014. URL: http://research2guidance.com/product/cross-platform-tool-benchmarking-2014/ (visited on 07/25/2019).

[RHH98] Linda Rosenberg, Theodore F. Hammer, and Lenore L. Huffman. "Requirements, testing and metrics". In: *Annual Pacific Northwest Software Quality Conference*. 1998.

[Rie16] Christoph Rieger. "A Data Model Inference Algorithm for Schemaless Process Modelling". In: *Working Papers, European Research Center for Information Systems No. 29*. Ed. by Jörg Becker et al. ERCIS, University of Münster, 2016, pp. 1–17.

[Rie17] Christoph Rieger. "Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development". In: *Annual ACM Symposium on Applied Computing (SAC)*. Marrakech, Morocco: ACM, 2017, pp. 1599–1606. DOI: 10.1145/3019612.3019746.

[Rie18a] Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *Hawaii International Conference on System Sciences (HICSS)*. Waikoloa, Hawaii, USA, 2018, pp. 5725–5734.

[Rie18b] Christoph Rieger. "Interoperability of BPMN and MAML for Model-Driven Development of Business Apps". In: *Business Modeling and Software Design (BMSD)*. Ed. by Boris Shishkov. Springer International Publishing, 2018, pp. 149–166. DOI: 10.1007/978-3-319-94214-8_10.

[Rig96] Fabrizio Riguzzi. *A Survey of Software Metrics: DEIS Technical Report no. DEIS-LIA-96010*. 1996.

[RK18a] Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *Computer Languages, Systems & Structures (COMLAN)* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001.

[RK18b] Christoph Rieger and Herbert Kuchen. "Towards Model-Driven Business Apps for Wearables". In: *Mobile Web and Intelligent Information Systems (MobiWIS)*. Ed. by Muhammad Younas et al. Springer International Publishing, 2018, pp. 3–17. DOI: 10.1007/978-3-319-97163-6_1.

[RK19a] Christoph Rieger and Herbert Kuchen. "A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes". In: *Hawaii International Conference on System Sciences (HICSS)*. Maui, Hawaii, USA, 2019, pp. 7431–7440.

[RK19b] Christoph Rieger and Herbert Kuchen. "Towards Pluri-Platform Development: Evaluating a Graphical Model-Driven Approach to App Development Across Device Classes". In: *Towards Integrated Web, Mobile, and IoT Technology*. Ed. by Tim A. Majchrzak et al. Vol. 347. Springer International Publishing, 2019, pp. 36–66. DOI: 10.1007/978-3-030-28430-5_3.

[RM15] Patrick Rempel and Patrick Mäder. "Estimating the Implementation Risk of Requirements in Agile Software Development Projects with Traceability Metrics". In: *Requirements Engineering: Foundation for Software Quality*. Springer International Publishing, 2015, pp. 81–97. DOI: 10.1007/978-3-319-16101-3_6.

[RM16] Christoph Rieger and Tim A. Majchrzak. "Weighted Evaluation Framework for Cross-Platform App Development Approaches". In: *Information Systems: Development, Research, Applications, Education: SIGSAND/PLAIS EuroSymposium*. Ed. by Stanislaw Wrycza. Springer International Publishing, 2016, pp. 18–39. DOI: 10.1007/978-3-319-46642-2_2.

[RM17] Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-enabled Devices". In: *13th International Conference on Web Information Systems and Technologies (WEBIST)*. Porto, Portugal, 2017, pp. 332–339. DOI: 10.5220/0006353003320339.

[RM18] Christoph Rieger and Tim A. Majchrzak. "A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle". In: *Web Information Systems and Technologies*. Ed. by Tim A. Majchrzak et al. Springer International Publishing, 2018, pp. 202–220. DOI: 10.1007/978-3-319-93527-0_10.

[RM19] Christoph Rieger and Tim A. Majchrzak. "Towards the Definitive Evaluation Framework for Cross-Platform App Development Approaches". In: *Journal of Systems and Software (JSS)* (2019). DOI: 10.1016/j.jss.2019.04.001.

[ROS14] Jean Michel Rouly, Jonathan D. Orbeck, and Eugene Syriani. "Usability and Suitability Survey of Features in Visual IDEs for Non-Programmers". In: *Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU. ACM, 2014, pp. 31–42. DOI: 10.1145/2688204.2688207.

[Rtv04] Nick Russell, Arthur H.M. ter Hofstede, and Wil M.P. van der Aalst. *Workflow Resource Patterns. BETA Working Paper Series, WP 127*. Ed. by Eindhoven University of Technology. Eindhoven, 2004.

[Rus+05] Nick Russell et al. "Workflow Data Patterns: Identification, Representation and Tool Support". In: *Conceptual Modeling – ER*. Springer, 2005, pp. 353–368. DOI: 10.1007/11568322_23.

[Rus+06] Nick Russell et al. *Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22*. 2006.

[Rus15] Alex Russell. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. 2015. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/ (visited on 07/25/2019).

[Rv14] Janessa Rivera and Rob van der Meulen. *Gartner Says By 2018, More Than 50 Percent of Users Will Use a Tablet or Smartphone First for All Online Activities*. 2014. URL: https://www.gartner.com/en/newsroom/press-releases/2014-12-08-gartner-says-by-2018-more-than-50-percent-of-users-will-use-a-tablet-or-smartphone-first-for-all-online-activities (visited on 07/25/2019).

[RWK18] Christoph Rieger, Martin Westerkamp, and Herbert Kuchen. "Challenges and Opportunities of Modularizing Textual Domain-Specific Languages". In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2018, pp. 387–395. DOI: 10.5220/0006601903870395.

[RWK19] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. "Musket: A Domain-specific Language for High-level Parallel Programming with Algorithmic Skeletons". In: *ACM/SIGAPP Symposium on Applied Computing (SAC)*. Limassol, Cyprus: ACM, 2019, pp. 1534–1543. DOI: 10.1145/3297280.3297434.

[RZ15] Andreas Reiter and Thomas Zefferer. “POWER: A cloud-based mobile augmentation approach for web- and cross-platform applications”. In: *CloudNet*. IEEE, 2015, pp. 226–231. DOI: 10.1109/CloudNet.2015.7335313.

[San+14] Zohreh Sanaei et al. “Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges”. In: *IEEE Communications Surveys & Tutorials* 16.1 (2014), pp. 369–392. DOI: 10.1109/SURV.2013.050113.00090.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. “Context-Aware Computing Applications”. In: *First Workshop on Mobile Computing Systems and Applications (WMCSA)*. IEEE CS, 1994, pp. 85–90.

[SB16] Kundan Singh and John Buford. “Developing WebRTC-based team apps with a cross-platform mobile framework”. In: *IEEE Annual Consumer Communications and Networking Conference (CCNC)* (2016). DOI: 10.1109/CCNC.2016.7444762.

[Sch13] Christian Schalles. *Usability evaluation of modeling languages*. Springer Fachmedien Wiesbaden, 2013. DOI: 10.1007/978-3-658-00051-6.

[SCK07] Daniel Sinnig, Patrice Chalin, and Ferhat Khendek. “Common Semantics for Use Cases and Task Models”. In: *Integrated Formal Methods*. Ed. by Jim Davies and Jeremy Gibbons. Vol. 4591. LNCS. Springer, 2007, pp. 579–598. DOI: 10.1007/978-3-540-73210-5_30.

[Sey+15] Teddy Seyed et al. “SoD-toolkit: A toolkit for interactively prototyping and developing multi-sensor, multi-device environments”. In: *ACM International Conference on Interactive Tabletops and Surfaces (ITS)* (2015). DOI: 10.1145/2817721.2817750.

[SH09] Dominik Stein and Stefan Hanenberg. “Assessing the Power of a Visual Modeling Notation – Preliminary Contemplations on Designing a Test –”. In: *Models in Software Engineering: Workshops and Symposia at MODELS 2008. Reports and Revised Selected Papers*. Ed. by Michel R. V. Chaudron. Springer Berlin Heidelberg, 2009, pp. 78–89. DOI: 10.1007/978-3-642-01648-6_9.

[SK13] Andreas Sommer and Stephan Krusche. “Evaluation of cross-platform frameworks for mobile applications”. In: *LNI* P-215 (2013).

[SLT09] M. Saunders, P. Lewis, and A. Thornhill. *Research Methods for Business Students*. Always learning. Prentice Hall, 2009.

[SMB95] Mark S. Silver, M. Lynne Markus, and Cynthia Mathis Beath. “The Information Technology Interaction Model: A Foundation for the MBA Core Course”. In: *MIS Quarterly* 19.3 (1995), pp. 361–390.

[Soc04] IEEE Computer Society. *IEEE Standard for a Software Quality Metrics Methodology: IEEE std 1061-1998 (R2004)*. Tech. rep. IEEE Computer Society, 2004.

[Spi01] Diomidis Spinellis. "Notable design patterns for domain-specific languages". In: *Journal of Systems and Software* 56.1 (2001), pp. 91–99. DOI: 10.1016/S0164-1212(00)00089-3.

[SS01] Jared Spool and Will Schroeder. "Testing Web Sites: Five Users is Nowhere Near Enough". In: *CHI '01 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2001, pp. 285–286. DOI: 10.1145/634067.634236.

[Sta16] Statista Inc. *Mobile app revenues 2015-2020*. 2016. URL: https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/ (visited on 07/25/2019).

[Sta17a] Statista Inc. *Annual number of mobile app downloads worldwide 2022*. 2017. URL: https://www.statista.com/statistics/692752/app-share-of-mobile-minutes-countries/ (visited on 07/25/2019).

[Sta17b] Statista Inc. *App share of mobile minutes in selected markets 2017*. 2017. URL: https://www.statista.com/statistics/692752/app-share-of-mobile-minutes-countries/ (visited on 07/25/2019).

[Sta18] Statista Inc. *Smartwatches*. 2018. URL: https://www.statista.com/study/36038/smartwatches-statista-dossier/ (visited on 07/25/2019).

[Sta19a] Statista Inc. *Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018*. 2019. URL: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/ (visited on 07/25/2019).

[Sta19b] Statista Inc. *Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 1st quarter 2019*. 2019. URL: https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/ (visited on 07/25/2019).

[Ste+15] Michel Steuwer et al. "Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code". In: *ICFP '15*. Vancouver, BC, Canada: ACM, 2015, pp. 205–217. DOI: 10.1145/2784731.2784754.

[Sum18] Cameron Summerson. *How Google is Turning Chrome OS into a Powerful Tablet OS*. 2018. URL: https://www.howtogeek.com/368199/how-google-is-turning-chrome-os-into-a-powerful-tablet-os/ (visited on 07/25/2019).

[Sut+16] Edhy Sutanta et al. "Survey: Models and prototypes of schema matching". In: *International Journal of Electrical and Computer Engineering* 6.3 (2016), pp. 1011–1022. DOI: 10.11591/ijece.v6i3.9789.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Chichester: John Wiley & Sons, 2006.

[SV14] Stijn Schuermans and Michael Vakulenko. *Apps for connected cars? Your mileage may vary*. 2014. URL: https://app.box.com/s/r5bvro03mb4ptlnd4fnd (visited on 07/25/2019).

[SW07] Christof Simons and Guido Wirtz. "Modeling context in mobile distributed systems with the UML". In: *Journal of Visual Languages and Computing* 18.4 (2007), pp. 420–439. DOI: 10.1016/j.jvlc.2007.07.001.

[SW14] Artem Syromiatnikov and Danny Weyns. "A Journey through the Land of Model-View-Design Patterns". In: *WICSA*. 2014, pp. 21–30. DOI: 10.1109/WICSA.2014.13.

[Tan16] Ailie K. Y. Tang. "Mobile app monetization: App business models in the digital era". In: *International Journal of Innovation, Management and Technology* 7.5 (2016), p. 224.

[The19a] The Linux Foundation. *Tizen*. 2019. URL: https://www.tizen.org (visited on 07/25/2019).

[The19b] The Qt Company. *QT - Cross-platform software development for embedded & desktop*. 2019. URL: https://www.qt.io/ (visited on 07/25/2019).

[Tho12] Gordon Thomson. "BYOD: enabling the chaos". In: *Network Security* 2012.2 (2012), pp. 5–8. DOI: 10.1016/S1353-4858(12)70013-2.

[TK08] Hallvard Trætteberg and John Krogstie. "Enhancing the Usability of BPM-Solutions by Combining Process and User-Interface Modelling". In: *The Practice of Enterprise Modeling (PoEM)*. Springer, 2008, pp. 86–97. DOI: 10.1007/978-3-540-89218-2_7.

[Tuo+07] Mika Tuomainen et al. "Model-centric approaches for the development of health information systems". In: *Studies in Health Technology and Informatics* 129 (2007).

[UB16] Eric Umuhoza and Marco Brambilla. "Model Driven Development Approaches for Mobile Applications: A Survey". In: *Mobile Web and Intelligent Information Systems: 13th International Conference*. Springer International Publishing, 2016, pp. 93–107. DOI: 10.1007/978-3-319-44215-0_8.

[Vac+14] Edoardo Vacchi et al. "Neverlang 2: A Framework for Modular Language Implementation". In: *International Conference on Modularity*. ACM, 2014, pp. 29–32. DOI: 10.1145/2584469.2584478.

[van09] Wil M. P. van der Aalst. "Process-Aware Information Systems: Lessons to Be Learned from Process Mining". In: *Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems*. Ed. by Kurt Jensen and Wil M. P. van der Aalst. Springer Berlin Heidelberg, 2009, pp. 1–26. DOI: 10.1007/978-3-642-00899-3_1.

[van99] Wil M.P. van der Aalst. "Formalization and verification of event-driven process chains". In: *Information and Software Technology* 41.10 (1999), pp. 639–650. DOI: 10.1016/S0950-5849(99)00016-6.

[Vau+14] Steffen Vaupel et al. "Model-driven development of mobile applications allowing role-driven variants". In: *Lecture Notes in Computer Science* 8767 (2014), pp. 1–17.

[VHM07] Mathieu Verbaere, Elnar Hajiyev, and Oege de Moor. "Improve Software Quality with SemmleCode: An Eclipse Plugin for Semantic Code Search". In: *22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. ACM, 2007, pp. 880–881. DOI: 10.1145/1297846.1297936.

[Vir92] Robert A. Virzi. "Refining the Test Phase of Usability Evaluation: How Many Subjects is Enough?" In: *Hum. Factors* 34.4 (1992), pp. 457–468.

[vKV00] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *SIGPLAN Notices* 35.6 (2000), pp. 26–36. DOI: 10.1145/352029.352035.

[VMP14] Vladimir Vujović, Mirjam Maksimović, and Branko Perisic. "Sirius: A rapid development of DSM graphical editor". In: *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, 2014, pp. 233–238. DOI: 10.1109/INES.2014.6909375.

[Völ13] Markus Völter. *DSL engineering: Designing, implementing and using domain-specific languages*. Lexington, KY: CreateSpace Independent Publishing Platform, 2013.

[VS10] Markus Völter and Konstantin Solomatov. "Language modularization and composition with projectional language workbenches illustrated with MPS". In: *3rd Intl. Conference on Software Language Engineering, LNCS* (2010).

[vt05] Wil M.P. van der Aalst and Arthur H.M. ter Hofstede. "YAWL: Yet another workflow language". In: *Information Systems* 30.4 (2005), pp. 245–275. DOI: 10.1016/j.is.2004.02.002.

[Was10] Anthony I. Wasserman. "Software engineering issues for mobile application development". In: *Proc. FoSER '10*. Ed. by Gruia-Catalin Roman and Kevin Sullivan. 2010, p. 397. DOI: 10.1145/1882362.1882443.

[Wat+17] Takuya Watanabe et al. "Understanding the Origins of Mobile App Vulnerabilities: A Large-scale Measurement Study of Free and Paid Apps". In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 14–24. DOI: 10.1109/MSR.2017.23.

[WB13] Karl Wiegers and Joy Beatty. *Software Requirements (Developer Best Practices)*. Microsoft Press, 2013.

[Web19] WebRatio s.r.l. *Rapid mobile app and web application development platform - WebRatio*. 2019. URL: http://www.webratio.com (visited on 07/25/2019).

[Wei91] Mark Weiser. "The Computer for the 21st Century". In: *Scientific American* 265.3 (1991), pp. 94–104. DOI: 10.1038/scientificamerican0991-94.

[Wes12] Mathias Weske. *Business Process Management*. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-28616-2.

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. "A Semantics for Advice and Dynamic Join Points in Aspect-oriented Programming". In: *TOPLAS* 26.5 (Sept. 2004), pp. 890–910. DOI: 10.1145/1018203.1018208.

[Wol11] David Wolber. "App inventor and real-world motivation". In: *ACM Technical Symposium on Computer Science Education (SIGCSE)* (2011). DOI: 10.1145/1953163.1953329.

[Wor11] Workflow Patterns Initiative. *Workflow Patterns - Evaluations - Standards*. 2011. URL: http://www.workflowpatterns.com/evaluations/standard/ (visited on 07/25/2019).

[WR19] Fabian Wrede and Christoph Rieger. *Musket Material Respository*. 2019. URL: https://github.com/wwu-pi/musket-material (visited on 07/25/2019).

[WRK18] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *High-Level Parallel Programming and Applications (HLPP)*. Orléans, France, 2018.

[WRK19] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *The Journal of Supercomputing (SUPE)* (Mar. 2019). DOI: 10.1007/s11227-019-02825-6.

[Wu+05] Hui Wu et al. "Weaving a debugging aspect into domain-specific language grammars". In: *ACM Symposium on Applied Computing (SAC)*. 2005, pp. 1370–1374. DOI: 10.1145/1066677.1066986.

[XX13] Spyros Xanthopoulos and Stelios Xinogalos. "A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications". In: *Proc. 6th BCI*. ACM, 2013, pp. 213–220. DOI: 10.1145/2490257.2490292.

[Zar15] Sebastian Zarnekow. *The Xtext Grammar Learned New Tricks*. 2015. URL: http://zarnekow.blogspot.de/2015/10/the-xtext-grammar-learned-new-tricks.html (visited on 07/25/2019).

[Żył15] Kamil Żyła. "Perspectives of Simplified Graphical Domain-Specific Languages as Communication Tools in Developing Mobile Systems for Reporting Life-Threatening Situations". In: *Studies in Logic, Grammar and Rhetoric* 43.1 (2015). DOI: 10.1515/slgr-2015-0048.

[ZZ12] Kevin Xiaoguo Zhu and Zach Zhizhong Zhou. "Research note—Lock-in strategy in software competition: Open-source software vs. proprietary software". In: *Information Systems Research* 23.2 (2012), pp. 536–545.

Part II

Included Publications

The second part of the cumulative dissertation contains the 18 publications (P1-P18) that were of relevance to the research presented in Part I. The research contributions of each publication were already discussed at the end of each chapter pertinent to the respective field of research (cf. Sections 3.6, 4.6 and 5.6). The following chapters present all publications that are included in the cumulative dissertation. All publications underwent a rigorous peer-view process to be included in conference proceedings or journals. Furthermore, all work that was published in proceedings also required to be presented and discussed at the respective conference.

Table 6.1 presents an overview of the publications indicating each publication's title and their respective points according to three common rankings. The first ranking entry refers to Australian *Core* ranking which covers many journals and conference venues in the field of software engineering[1]. The second ranking is the VHB-JOURQUAL3 (JQ3), which was released by the German Academic Association for Business Research[2]. Their focus on IS and business administration research frequently leaves software engineering related publications unranked. As a third ranking, the ranking scheme proposed by the Wissenschaftliche Kommission Wirtschaftsinformatik (WKWI)[3]. is commonly employed at the Department of Information Systems at the University of Münster but similarly has limited coverage of the software engineering field.

The implementation rules for a cumulative dissertation require at least three single-author publications or an equivalent in co-authored publications. The WKWI assigns the A level to the Symposium on Applied Computing (SAC) in case the acceptance rate for a specific year is below 30%. This has been the case for several years[4], including the publications P5 and P16 from this dissertation. In addition, Table 6.2 shows further related work that was published as a book chapter or report, without following a strict peer-review process.

Regarding the formatting of the publications in the following chapters, published work can either be included in its original form or reformatted to match the overall typeface of the thesis. The latter approach was taken in order to improve readability and achieve a harmonised appearance throughout all publications. Adaptations include fonts and font sizes, the positions and sizes of tables and figures as well as a consistent citation style. However, the contents were not modified. Every work is preceded by a short table that states the main information on authors, publication date, type of outlet, and the full quotations.

[1] http://www.core.edu.au/conference-portal
[2] https://vhbonline.org/en/service/jourqual/vhb-jourqual-3/
[3] http://wi.vhbonline.org/zeitschriftenrankings/
[4] https://www.sigapp.org/sac/SAC_STATS.pdf

Table 6.1: Overview of Published Work

No.	Publication	CORE	JQ3	WKWI
	Journal publications			
P1	[RM19]: Christoph Rieger and Tim A. Majchrzak. "Towards the Definitive Evaluation Framework for Cross-Platform App Development Approaches". In: *Journal of Systems and Software (JSS)* (2019). DOI: 10.1016/j.jss.2019.04.001	A	-	-
P2	[WRK19]: Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *The Journal of Supercomputing (SUPE)* (Mar. 2019). DOI: 10.1007/s11227-019-02825-6	B	-	-
P3	[RK18a]: Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *Computer Languages, Systems & Structures (COMLAN)* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001	C	-	-
	Conference (post-)proceedings			
P4	[RK19b]: Christoph Rieger and Herbert Kuchen. "Towards Pluri-Platform Development: Evaluating a Graphical Model-Driven Approach to App Development Across Device Classes". In: *Towards Integrated Web, Mobile, and IoT Technology*. Ed. by Tim A. Majchrzak, Cristian Mateos, Francesco Poggi, and Tor-Morten Grønli. Vol. 347. Springer International Publishing, 2019, pp. 36–66. DOI: 10.1007/978-3-030-28430-5_3	-	C	B [6]
P5	[RWK19]: Christoph Rieger, Fabian Wrede, and Herbert Kuchen. "Musket: A Domain-specific Language for High-level Parallel Programming with Algorithmic Skeletons". In: *ACM/SIGAPP Symposium on Applied Computing (SAC)*. Limassol, Cyprus: ACM, 2019, pp. 1534–1543. DOI: 10.1145/3297280.3297434	B	-	A [7]
P6	[RK19a]: Christoph Rieger and Herbert Kuchen. "A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes". In: *Hawaii International Conference on System Sciences (HICSS)*. Maui, Hawaii, USA, 2019, pp. 7431–7440	A	C	B
P7	[WRK18]: Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *High-Level Parallel Programming and Applications (HLPP)*. Orléans, France, 2018	C	-	-
P8	[RK18b]: Christoph Rieger and Herbert Kuchen. "Towards Model-Driven Business Apps for Wearables". In: *Mobile Web and Intelligent Information Systems (MobiWIS)*. ed. by Muhammad Younas, Irfan Awan, George Ghinea, and Marisa Catalan Cid. Springer International Publishing, 2018, pp. 3–17. DOI: 10.1007/978-3-319-97163-6_1	-	C	C
P9	[RM18]: Christoph Rieger and Tim A. Majchrzak. "A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle". In: *Web Information Systems and Technologies*. Ed. by Tim A. Majchrzak, Paolo Traverso, Karl-Heinz Krempels, and Valérie Monfort. Springer International Publishing, 2018, pp. 202–220. DOI: 10.1007/978-3-319-93527-0_10	-	C	B [6]
P10	[Rie18b]: Christoph Rieger. "Interoperability of BPMN and MAML for Model-Driven Development of Business Apps". In: *Business Modeling and Software Design (BMSD)*. ed. by Boris Shishkov. Springer International Publishing, 2018, pp. 149–166. DOI: 10.1007/978-3-319-94214-8_10	-	C	B [6]

No.	Publication	CORE	JQ3	WKWI
P11	[RWK18]: Christoph Rieger, Martin Westerkamp, and Herbert Kuchen. "Challenges and Opportunities of Modularizing Textual Domain-Specific Languages". In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2018, pp. 387–395. DOI: 10.5220/0006601903870395	-	-	-
P12	[Rie18a]: Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *Hawaii International Conference on System Sciences (HICSS)*. Waikoloa, Hawaii, USA, 2018, pp. 5725–5734	A	C	B
P13	[BRK17b]: Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Model-Driven Approach for Evaluating Traceability Information". In: *Third International Conference on Advances and Trends in Software Engineering (SOFTENG)*. ed. by Mira Kajko-Mattsson, Pål Ellingsen, and Paolo Maresca. 2017, pp. 59–65	-	-	-
P14	[BRK17a]: Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Domain-specific Language for Configurable Traceability Analysis". In: *5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. ed. by Luis Ferreira Pires, Slimane Hammoudi, and Bran Selic. SCITEPRESS, 2017, pp. 374–381	-	-	-
P15	[RM17]: Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-enabled Devices". In: *13th International Conference on Web Information Systems and Technologies (WEBIST)*. Porto, Portugal, 2017, pp. 332–339. DOI: 10.5220/0006353003320339	C	-	-
P16	[Rie17]: Christoph Rieger. "Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development". In: *Annual ACM Symposium on Applied Computing (SAC)*. Marrakech, Morocco: ACM, 2017, pp. 1599–1606. DOI: 10.1145/3019612.3019746	B	-	A [7]
P17	[RM16]: Christoph Rieger and Tim A. Majchrzak. "Weighted Evaluation Framework for Cross-Platform App Development Approaches". In: *Information Systems: Development, Research, Applications, Education: SIGSAND/PLAIS EuroSymposium*. Ed. by Stanislaw Wrycza. Springer International Publishing, 2016, pp. 18–39. DOI: 10.1007/978-3-319-46642-2_2	-	C	B [6]
P18	[Ern+16]: Jan Ernsting, Christoph Rieger, Fabian Wrede, and Tim A. Majchrzak. "Refining a Reference Architecture for Model-Driven Business Apps". In: *12th International Conference on Web Information Systems and Technologies (WEBIST)*. SCITEPRESS, 2016, pp. 307–316. DOI: 10.5220/0005862103070316	C	-	-

[6] considering LNBIP as LNCS Satellite Series

[7] ranked A because of acceptance rate <30%

Table 6.2: Overview of Further Work Related to This Dissertation

Publication	Year	Outlet
[Maj+17]: Tim A. Majchrzak, Jan C. Dageförde, Jan Ernsting, Christoph Rieger, and Tobias Reischmann. "How Cross-Platform Technology Can Facilitate Easier Creation of Business Apps". In: *Apps Management and E-Commerce Transactions in Real-Time*. Ed. by In Lee and Sajad Rezaei. Advances in E-Business Research. IGI Global, 2017, pp. 104–140. DOI: 10.4018/978-1-5225-2449-6.ch005	2017	IGI book chapter
[Rie16]: Christoph Rieger. "A Data Model Inference Algorithm for Schemaless Process Modelling". In: *Working Papers, European Research Center for Information Systems No. 29*. Ed. by Jörg Becker, Klaus Backhaus, Martin Dugas, Bernd Hellingrath, Thomas Hoeren, Stefan Klein, Herbert Kuchen, Ulrich Müller-Funk, Heike Trautmann, and Gottfried Vossen. ERCIS, University of Münster, 2016, pp. 1–17	2016	ERCIS Working Papers

7

Towards the Definitive Evaluation Framework for Cross-Platform App Development Approaches

Table 7.1: Fact sheet for publication P1

Title	Towards the Definitive Evaluation Framework for Cross-Platform App Development Approaches
Authors	Christoph Rieger[1] Tim A. Majchrzak[2] [1] *ERCIS, University of Münster, Münster, Germany* [2] *ERCIS, University of Agder, Kristiansand, Norway*
Publication Date	2019
Publication Outlet	Journal of Systems and Software
Copyright	Elsevier open access (CC BY 4.0)
Full Citation	Christoph Rieger and Tim A. Majchrzak. "Towards the Definitive Evaluation Framework for Cross-Platform App Development Approaches". In: *Journal of Systems and Software (JSS)* (2019). DOI: 10.1016/j.jss.2019.04.001

Towards the Definitive Evaluation Framework for Cross-Platform App Development Approaches

Christoph Rieger Tim A. Majchrzak

Keywords: Mobile App, Mobile Computing, Cross-Platform, Multi-Platform, Development Framework

Abstract: Mobile app development is hindered by device fragmentation and vendor-specific modifications. Boundaries between devices blur with PC-tablet hybrids on the one side and wearables on the other. Future apps need to support a host of app-enabled devices with differing capabilities, along with their software ecosystems. Prior work on cross-platform app development concerned concepts and prototypes, and compared approaches that target smartphones. To aid choosing an appropriate framework and to support the scientific assessment of approaches, an up-to-date comparison framework is needed. Extending work on a holistic, weighted set of assessment criteria, we propose what could become the definitive framework for evaluating cross-platform approaches. We have based it on sound abstract concepts that allow extensions. The weighting capabilities offer customisation to avoid the proverbial comparison of apples and oranges lurking in the variety of available frameworks. Moreover, it advises on multiple development situations based on a single assessment. In this article, we motivate and describe our evaluation criteria. We then present a study that assesses several frameworks and compares them to Web Apps and native development. Our findings suggest that cross-platform development has seen much progress but the challenges are ever growing. Therefore, additional support for app developers is warranted.

7.1 Introduction

With iOS and Android, only two platforms with significant market share remain for the development of mobile apps [Fv17]. Nevertheless, cross-platform technology continues to be very important and many app development frameworks exist [HHM13; El-+17]. When creating apps, there is still no uniform recommendation whether – or in which case – to employ web technology, a cross-platform approach, or a native Software Development Kit (SDK) [RM16]. The emergence of Progressive Web Apps (PWA) has on the one hand brought a contestant for unification [MBG18; BMG17]; on the other hand, it underlines that professional developers still seek for *easier* ways of developing once but having their app run on multiple platforms. There seems to be profound interest in straightforward yet customisable solutions, for instance demonstrated in trivial patents (such as for "customizing a mobile application using a web-based interface" [Bri+17]).

The complexity of app development does not merely come from the need to cover two more or less incompatible platforms. Device fragmentation and vendor-specific modifications incur that particularly developing for Android is not uniform [Dob12]. Additionally, the boundaries between devices are blurring with PC-tablet hybrids or wearables which extend computing into domains of watches, formerly unconnected electronic helpers, and even clothing [Nan+17]. There is a

jungle of *app-enabled* devices [RM18], each posing specific capabilities and idiosyncrasies. The different categories of devices also bring their own ecosystems and contexts of usage. It is easy to imagine that developing an app supposed to run on a smartphone as well as within the system of a car [Wol13] and additionally on a screenless smart home/Internet-of-Things (IoT) device [Ala+17] poses a tremendous challenge. This convergence of smart, user-targeted gadgets and formerly hidden small-scale information technology will need to be reflected in the development approaches used in the future.

Prior work on cross-platform app development has mainly been concerned with two topics. First, concepts and prototypes were proposed, such as by [HM13] and like applause [Fri14]. Second, frameworks that target smartphones and tablets were compared, for example by [HHM13], [OT12], and [LW13]. To aid developers in choosing an appropriate framework and to support the scientific assessment of approaches, an update to the comparison frameworks is needed. It ought to consider the recent technological developments both regarding the capabilities of cross-platform frameworks and novel ideas such as PWAs. At the same time, it needs to reflect the demands from developers while not compromising academic rigour. Moreover, it needs to take into account that the consequences from using cross-platform technology impact the user experience (UX). Therefore, wisely using such approaches is indicated [MMM16], and this line of thinking should be reflected in any attempt to evaluate approaches.

Extending the first attempt to provide a holistic, weighted set of assessment criteria by [RM16], we propose steps towards what we believe could become the *definite* framework for evaluating cross-platform development approaches. We have based it on sound abstract concepts that allow adaptability to future technological developments. It seeks to be versatile and handy for practitioners yet fulfil what is needed to satisfy a critical scientific assessment. The weighting capabilities offer individualisation and customisation. We, thus, ensure that the proverbial comparison of apples to oranges that lurks in the variety of available frameworks and goals of development is avoided. Moreover, our evaluation framework offers the opportunity to get advice for multiple development endeavours based on a single assessment.

Please note the usage of terms in the following. Framework is *overloaded* as it refers to our evaluation framework as well as to software frameworks for cross-platform app development. Unless we refer to the evaluation, framework denotes the latter. If the context poses the chance of confusion, we qualify framework with "evaluation". We speak of a cross-platform development approach when we do not consider a concrete software implementation but rather the general way of solving the challenge of developing one but running apps on several platforms.

This article makes a number of contributions. First, it provides an evaluation framework for cross-platform development approaches for app-enabled devices. It can not only be used as provided but the extensive criteria catalogue serves as a reference which may also be employed for purposes beyond our framework. Second, we provide weight profiles to be used in conjunction with the framework. These profiles enable a non-generic usage of the framework, which allows users to adapt it to their company- or project-specific needs. Third, we present the results from

an exemplary study with several development approaches, including (Progressive) Web Apps, hybrid apps, runtime approaches, and native development for comparison. The study not only seeks to underline the feasibility of our framework but to provide concrete advice.

The remainder of this article is structured as follows. In Section 7.2 we give an overview of works that provide a precondition to ours, are complementary, or are otherwise related in content or approach. Section 7.3 comprehensively presents our criteria catalogue and the rationale behind each of the criteria. It thereby serves both as a core scientific contribution and as a reference. The evaluation criteria are motivated and explained in detail; where applicable, examples are given for better illustration. Section 7.4 proposes weight profiles, which can be used to evaluate development approaches in a customized fashion. To demonstrate the feasibility of our work and to give practical recommendations, we present an evaluation study in Section 7.5. Our findings are then discussed in Section 7.6, which includes results from expert feedback, a proposal for a research agenda, limitations, and indication of our future work. Finally, in Section 7.7 we draw a conclusion.

7.2 Related Work

The work presented in this article draws from the field of cross-platform development frameworks, which has emerged since the advent of smartphones. In addition, it takes account of the recent developments in the domain of mobile devices and the implications on future mobile app development approaches. Therefore, related work on both fields is presented in the following.

7.2.1 Cross-Platform Frameworks

Resulting from the increasing popularity of cross-platform development frameworks, a multitude of scientific works has been prepared for this topic. However, most papers are of experimental nature and restricted to single frameworks, or limited by the choice of considered development approaches. Only few provide a thorough evaluation based on a diverse set of criteria. A comprehensive summary of related literature regarding covered tools, criteria[1], and focal areas of comparison is given in Table 7.2. It combines a literature search within the scientific database Scopus on evaluations of mobile cross-platform frameworks, using the query

```
TITLE-ABS-KEY((comparison OR evaluation OR review OR survey) AND
(mobile OR app OR wearable OR application OR vehicular OR "in-vehicle") AND
("cross-platform" OR "multi-platform" OR "cross platform") AND
(framework OR approach))
```

combined with a forward search on the papers by Heitkötter et al. (2012, 2013). The latter represent early work on systematic assessment of app development frameworks for smartphones

[1] If multiple related criteria are used, similar subcriteria are grouped for brevity reasons, e.g., energy, CPU load, and duration measurements in [Cor+18] are aggregated to "performance (3)".

and have been used by many authors as basis for further research on apps. Examples include the definition of quality criteria for HTML5 frameworks [Soh+15], quantitative performance evaluations [WVN15], and the creation of cross-platform development frameworks such as ICPMD [El-+14] and MD² [HKM15]. To put the identified literature into context, we highlight notable details in the following.

Table 7.2: Literature on Cross-Platform App Development Tool Evaluations

Paper	Evaluated tools	Evaluation criteria (number of subcriteria)	Focal areas
[BG18]	Ionic, React Native	file system access performance	quantitative comparison
[Cor+18]	Cordova, Corona, Native app, NativeScript, Titanium, Xamarin	performance (3)	resource usage and execution time of calculations and audio/video playback
[Del+18]	Cordova, Corona, Native app, NativeScript, Titanium, Web Apps, Xamarin	performance	Execution time of calculations
[Fer+18]	Native app, PhoneGap, Sensa Touch, Titanium	performance of device features (2)	app scenarios with calculations as well as camera and GPS access
[JET18]	Cordova, Native app, Titanium, Xamarin	performance of build, rendering, and UI response	specific combinations of platforms and cross-platform technologies
[BMG17]	Ionic, Progressive Web Apps, ReactNative	performance (3)	quantitative study using app scenario
[CG17]	MoSync, PhoneGap, Titanium, Web Apps	battery usage, device sensors (7)	evaluation of energy consumption in combination with sensor usage
[LA17]	*none* (native vs. web vs cross-platform in general)	14 rather simple questions to be answered before developing; six criteria for the tool selection step	two-step process: *tools* to be selected after the main approach is chosen
[El-+17]	ICPMD, J2ObjC, MD², MoSync, PhoneGap, Titanium, xFace, XMLVM	tool architecture, platform support, app type, license	variety of development approaches
[QGZ17]	Cordova, Native app	development support (6), device features (5), performance (6)	quantitative tool comparison
[VJ17]	Ionic, PhoneGap, NativeScript, Native app	platform support (3), development support (3)	qualitative comparison
[AHN16]	PhoneGap	starting duration, memory usage, app size, user experience, appearance, development support	quantitative and qualitative evaluation criteria
[BEP16]	Ionic, Sencha	user (functionality, UI, platform support), developer (developing time, reuse, native access)	qualitative tool comparison
[Lat+16]	*none* (cross-platform approaches in general)	scalability and maintainability, device features, resource consumption, security, IDE	criteria definition and variety of development approaches
[RM16]	PhoneGap, Web Apps	infrastructure (7), development (11), app (9), usage (4)	criteria weighting
[UB16]	13 research frameworks, 4 commercial solutions	development process, app layer, development technique, platform support	model-driven approaches

Paper	Evaluated tools	Evaluation criteria (number of subcriteria)	Focal areas
[CAB15]	*none* (cross-platform approaches in general)	targeted public, programming language, app type	qualitative comparison of cross-platform approaches
[CG15]	PhoneGap, Titanium	battery usage, device resource usage	evaluation of battery usage
[DM15]	Adobe Air, MoSync, PhoneGap, Titanium	platform support, license (2), development environment (8), user experience (6), functionality (29), monetization (4), security (2)	performance benchmarks and development experience discussion
[HHH15]	AngularJS, HTML5/JS, jQuery Mobile, PhoneGap, RhoMobile, Sencha Touch	platform support (4), development support (7), deployment factors (6)	criteria definition and qualitative tool comparison
[CGG14]	jQuery Mobile, MoSync, PhoneGap, Titanium	license, community, API, tutorials, complexity, IDE, devices, GUI, knowledge	qualitative tool comparison for apps with animations
[Dal+13]	jQuery Mobile, PhoneGap, Sencha Touch, Titanium	platform support, rich user interface, backend communication, security, app extensions, energy consumption, device features, license	performance evaluation (memory, CPU, energy consumption)
[HEE13]	MoSync, Native app, Titanium	responsiveness	qualitative user evaluation
[SK13]	PhoneGap, Rhodes, Sencha Touch, Titanium	functionality (8), usability (6), developer support (4), reliability/performance (4), deployment (8)	criteria definition and qualitative tool comparison
[VSB13]	Cordova, PhoneGap, RhoMobile, Titanium	platform support, framework development activity/maturity (3), license, device features (11)	quantitative comparison
[XX13]	*none* (cross-platform approaches in general)	distribution, programming languages, hardware & data access, user interface, perceived performance	criteria definition
[HHM12]	PhoneGap, Titanium, Web Apps	Infrastructure (7), development (6)	foundational cross-platform criteria catalogue for this work
[OT12]	9 commercial frameworks	developer support (8), user expectations (6)	criteria definition and challenges of cross-platform development
[PSC12]	DragonRad, MoSync, RhoMobile, PhoneGap	platform compatibility (2), development features (4), general features (4), device APIs (17)	qualitative tool comparison
[Rd12]	Canappi mdsl, DragonRAD, mobl, PhoneGap, Rhodes, Titanium	technology approach, platform support, development environment, app type, device features (5)	criteria definition

Early papers have typically only considered few criteria – if at all [RT12; SKS14]. It can be noticed that few works perform a rather comprehensive evaluation, often neglecting a user's perspective on cross-platform app development (cf., e.g., [El-+17]). For example, [OT12] have analysed nine tools with regard to developers' needs and user expectations. Many papers focus on particular aspects of apps such as animations [CGG14], performance [Dal+13], or energy consumption [CG15].

We can also observe that the set of considered criteria does not appear to be coherent over time. Criteria are often grouped into common categories [OT12; XX13; HHH15; SK13] but no clear categorisation scheme has emerged. One additional problem typically found is a shortage of criteria explanations (e.g., [CAB15; HHH15]). Furthermore, these inconsistencies are also reflected in the lack of measurable metrics for the respective criteria.

It can be summed up that many authors set out to conquer the field of cross-platform app development. Without doubt, the papers shown in Table 7.2 provide substantial contributions. However, both the rapid evolution on the mobile device market and the proliferation of individual frameworks in the field of cross-platform development thwart the process of theory-building and mandate further work. This is also illustrated by many recently published papers that – more or less isolated – address distinct issues also discussed in this article. To conclude the study of related work, we highlight such works that address novel mobile devices.

7.2.2 Novel App-Enabled Devices

Only few years ago, mobile app development exclusively focused on frameworks and applications for smartphones and – sporadically – tablets. Nowadays, many more devices have become app-enabled and to some extent mobile, ranging from Internet-of-Things (IoT) functionalities in tiny units to self-driving vehicles.

In previous work, we have presented an initial taxonomy for the variety of consumer devices whose functionality is extensible by third-party apps already today or will be in foreseeable future [RM18]. While formerly it was possible to categorise devices by operation system or hardware features, this approach is not feasible anymore: for example, the Android platform spans multiple device classes. Instead, the taxonomy distinguishes device classes according to the dimensions of *media richness of inputs*, *media richness of outputs*, and *degree of mobility* [RM18]. Within each of these device classes, various devices and platforms have emerged. Whereas Android and iOS have divided most of the smartphone market amongst themselves [Fv17], competition among novel mobile device platforms is high and no clear winners are foreseeable. Therefore, these devices pose similar challenges for app developers compared to the beginning of smartphones several years ago [HHM13]. An overview of scientific work on apps for several novel device classes according to this classification is presented next, together with existing literature on cross-platform development approaches.

Smart TVs are on the rise worldwide, with all major manufacturers offering such devices. As a consequence, more than 90% of connected TVs sold in Germany support the HbbTV standard that has evolved from previous approaches such as CE-HTML and Open IPTV [Sta18b; Hbb18]. In the U.S., app-enabled smart TVs are already present in 35% of households [Sta18b]. Many platforms have emerged in practice, for example the open-source media centre Kodi/XBMC with various forks, Android TV, Tizen OS for TV, and webOS [XBM18; Goo18b; Lin18; LG 18]. However, app development is often tied to a specific TV manufacturer and reflects the fragmentation in the

market. Interestingly from a cross-platform perspective, many smart TV frameworks natively support app development using web technologies such as HTML5 and JavaScript, thus being well-suited for cross-platform approaches. So far, scientific research often concentrates on specific sub-topics such as interactive ads [PG15b], serious games [Ryu+14], and 3D content [PG15a] across multiple smart TV platforms.

Regarding *smartwatches*, which now have found more than just a niche in the market [RPP14], Google and Apple again compete for dominance with their respective Android Wear (now Wear OS) and watchOS platforms. Further players in this field are Tizen OS and webOS [Bou15]. Some vendors have open-sourced their operating system (e.g., Android, Tizen, or webOS); however, few truly vendor-independent platforms such as AsteroidOS exist [Rev18]. To complicate matters, smartwatches are usually paired with a smartphone [Dou15], e.g., for performance reasons [LL16] and to benefit from Internet connectivity (the latter being a specific challenge, cf. [Aho15]). Smartwatch apps often rely on the respective smartphone companion app; thus, cross-platform development approaches must support each combination of host and watch platform. However, some smartwatch platforms recently added stand-alone capabilities on supported devices, for instance since the launch of Android Wear 2.0 in early 2017 [Goo18j].

In a wider sense, *wearables* such as fitness trackers are often tied to proprietary platforms, e.g., Microsoft Band [Mic18]. Whereas those devices usually support pairing with different smartphone platforms, third-party app development is still limited. Vendors such as Fitbit and Garmin do not even produce devices with modifiable operating system [Bou15]. Scientific work on wearables is therefore scarce [Kim+16]. Some authors have proposed middleware approaches to span a broad range of devices [Chm13], in one case even on the hardware layer [Zha+11].

Despite the vagueness of the terms, *smart home* and *IoT* devices could be a future domain for cross-platform research [Jie+15]. Several open-source and closed-source platforms exist that try to attract app developers and claim to integrate a plethora of devices. Qualcomm's AllJoyn, Intel's IoTivity, Apple HomeKit, and Google Brillo are the most important players that try to establish their middleware as comprehensive solutions [Car15]. For *home automation*, a host of proprietary solutions exist with a variety of application targets [SMP12]. Whether existing industry standards such as KNX can form the backbone of IoT-enabled smart homes remains to be seen. Transitions towards hybrid systems [Lil+17] and gateway usage [Fan+14] will possibly solve the challenges regarding hardware but may complicate app development further.

Concerning the upcoming generation of connected cars, four main approaches for developing *in-vehicle apps* exist [SV14]. First, Android Auto, Blackberry QNX, and Windows Embedded are technologies that are rebranded by car manufacturers and run native apps on the car's head unit. Second, some cars provide a remote application programming interface (API) to allow access and control of features such as door locks. For instance, General Motors, Airbiquity, and an unofficial API for Tesla cars make use of this approach [Dor18]. Third, platforms including Apple CarPlay and the MirrorLink alliance use screen mirroring, i.e., the app runs externally on the smartphone and is displayed on the car's screen [DHH13]. This approach was established due

to security concerns in order to avoid executing app code on the car's main hardware. Fourth, Dash Labs, Mojio, and Automatic connect to the on-board diagnostics port to interact with the car [Das18; Moj18; Aut18]. Although this approach requires a Bluetooth dongle as additional hardware, support is given for many cars that were not designed to be app-enabled in the first place.

Besides the underlying development approach, several papers focus on usability issues [QG14] and "remote" human machine interfaces (HMI) [DHH13] for the specific challenges of in-vehicle apps. For example, experimental implementations of novel concepts such as a route planning app for head-up displays (HUD) [NBN14] and other potential in-vehicle apps [Wol13] are explored. To the best of our knowledge, no cross-platform framework currently exists due to the novelty of the field as well as a lack of platform accessibility from the fight for dominance between car manufacturers "owning" the platform [SV14]. Current works deal with an *Open Service Cloud* for cars [DRB15] and the integration of non-automotive applications into the automotive environment [RP12]. Potentially, also a middleware approach [DRB15] might be an option to bridge different approaches and at the same time mitigate security risks.

This overview of novel mobile device platforms shows similar characteristics of fragmentation as the smartphone market experienced several years ago. Moreover, interactions between different device classes result in an exponentially growing amount of combinations, causing additional complexities to consider for app development. Some platforms such as Android and Tizen have branches that run on multiple devices from smart TVs to wearables, potentially simplifying the future development across device class borders. Samsung TOAST is an early initiative to simultaneously develop for Samsung Smart TV, the new Tizen platform and browsers, based on the established Apache Cordova framework [Sam18]. However, barely any approach covering multiple device classes currently exists in literature or practice, with the notable exception of the gaming domain. Unity3D [Uni18], one of the best-known game engines for 2D/3D games (and one even targeted by scientific research, as exemplarily illustrated by the works of [Xie12] and [MSK15]), supports 29 platforms including smartphones, smart TVs, consoles, and augmented reality devices.

7.3 Criteria Catalogue

In the following, we describe our catalogue of criteria, which marks the foundation of our evaluation framework. We start by illustrating fundamental considerations. Then, the four perspectives of the framework – *infrastructure*, *development*, *app*, and *user* – are explained in detail.

7.3.1 Fundamental Considerations and Structure

Our aim for this paper is to propose a future-proof, long-lived, adaptive evaluation framework for cross-platform app technology. It would be a presumptuous attempt to create such a framework from scratch. Thus, the structure and the selection of criteria is based on extensive prior work, as illustrated in Section 7.2. Moreover, we give a rationale for criteria that we added or that we use in an extended way in comparison to existing evaluation frameworks. This follows specific literature, as far as such works are available. Alternatively, we argue for such criteria based on our experience in working on cross-platform app development frameworks. We will revisit literature gaps as part of the discussion later in this article.

Consequently, the proposed criteria are the result of a process. First, we created a synopsis of existing approaches. Then, this synopsis was extended and revised. Thereby, our criteria catalogue not only caters for the latest developments in the field but also benefits from increased flexibility and versatility. Combined with the weight profiles as explained in Section 7.4, we are confident in being able to set the standard for future evaluation activities.

Most notably, we categorize our criteria. Instead of presenting one large catalogue, we summarize criteria by the *perspective* on development they take. Perspectives mark a specific view on the aspects being evaluated. They thereby provide coherence: although *all* criteria are important when evaluating a framework, those that have been grouped into the same perspective are stronger related to each other than those that we put into different perspectives. Not only does this foster the comprehensibility of the criteria, but also the weighting is much easier (as will be seen in Section 7.4).

The consideration of different perspectives is already found in the often-cited paper by [HHM13]. We have extended the original two perspectives (infrastructure and development) to four:

- *Infrastructure*: Using a cross-platform app development framework is inherently bound to preconditions. Typically, ramifications arise regarding the life cycle of developed apps. This can be summarized as the infrastructure a framework provides. Most fundamentally, this concerns the supported target platforms. Moreover, aspects of licensing, usage, and long-term prospects are considered.
- *Development*: A cross-platform framework is only as good as you can use it for developing apps. Frameworks may offer further built-in development support that can make development more rapid, support inexperienced developers, or both. Being proper for development is bound to a host of criteria that all have a technical appeal. Development criteria are those that programmers and software engineers will ask for, other aspects notwithstanding.
- *App*: Naturally, the actual app denotes whether development was successful. If an app is developed using a platform's native framework, it has access to all device features regarding sensors as well as user input and device output. A development framework should ideally provide a near-native range of support for device features such that access is versatile and easy to employ. Also, the integration of business concerns with regard to an app as a

product can be subsumed by this perspective. A good example for this is security, which is considered to be very important while becoming increasingly harder to overview for developers due to its multi-faceted nature [Wat+17].

- *Usage*: An app is more than the sum of its functionality. Therefore, the usage perspective comprises many aspects that in systems' design would fall under the non-functional (or: quality) requirements. Besides management aspects, this perspective embodies performance characteristics and how user-friendly an app is, including considerations of aesthetics, ergonomics, and efficiency.

We deem this distinction into categories not only helpful for assessing a framework with different aspects and stakeholders (such as developers, managers, and users) in mind but also to support different devices. As already argued, the devices found in modern mobile computing are no more limited to smartphones and – technologically rather similar – tablets [RM18]. The distinct app perspective (compared to [HHM13]) leads to more clarity with regard to the development outcome which might differ significantly across different device classes, whereas the development itself might be similar. Thus, perspectives offer an easier way to tailor an assessment to the desired device category: In some cases, assessments might be very broad, in some very narrow. And, as we will argue later, also cases such as "good smartphone support is mandatory, but compatibility with smartwatches would be nice" can be designed. Also, the additional usage perspective focuses on cross-cutting concerns such as usability and performance which largely affect user acceptance and joy of use.

Which devices are to be targeted – or, in other words, which role multi-device support plays – is merely one aspect when thinking about case-based assessment of development frameworks. The underlying development paradigms might be to some degree tailored to more or less specific use cases. For example, cross-platform development for business apps has been discussed [MWA15]. Likewise, a focus on consumers or mobile gaming is imaginable. These cases would even combine aspects of intended usage with those deriving from multi-device support. We will further elaborate on cases when explaining the weight profiles and in the discussion.

The following four subsections explain the criteria of the perspectives in detail. Besides explaining what should actually be measured respectively expressed by a category, we also give its rationale. Whenever possible, this is done based on the literature. We refer both to evaluation papers mentioned in Section 7.2 and to additional work specific to the very criterion. The only work not explicitly cited is that by [HHM13]. As the trailblazer for cross-platform evaluation frameworks, it contained 14 criteria of our framework already, even though in a premature form from today's perspective. Extending the already thorough work of a previous conference paper [RM16], the criteria catalogue has been refined by consulting experts in this field (cf. Section 7.6). Resulting from an iterative process, we have reworked the criteria descriptions to sharpen their scope (e.g., covering multiple aspects of robustness (A9) instead of a limited focus on degrading functionality), apply precise terminology (e.g., "user authentication" (U4) instead of the too

general term "user management"), and eliminate potential overlaps (e.g., discerning the fields of internationalisation (I6) from the subsequent app distribution (I4)). Also, two new criteria have been added in order to better suit the needs for large-scale app development through configuration management (D8) and incorporate app development for the multitude of new mobile devices (A10). For addressability, all criteria are numbered in the form Xy where X is a character denoting the perspective (I, D, A, U) and y a continuous number for the specific criterion.

The last subsection is followed by Table 7.3 (p. 150 onwards). While we reference related literature – particularly the related evaluation articles compiled in Table 7.2 – directly for each introduced criterion, this table provides a compilation of similarities of terminology. For each criterion, we state the related work that proposed a criterion by the same term. Moreover, we name terms used with a similar meaning to our criterion. The table not only means to better relate our contribution to the existing literature but also to identify ambiguities – not all criteria must always be referred to with the same term. In addition, some authors proposed criteria that are subsumed by ours, with the term thus only appearing in the detailed description of the criterion. The table can also help to identify weakly delimited terms that are used for multiple criteria (typically *overloaded* terms such as *operating system*) as well as super terms (e.g., *features*, which can mean hardware feature, system feature, or both).

7.3.2 Infrastructure Perspective

(I1) License: Particularly for commercial development, a framework's license is important. This question is often raised but not only relevant for open-source software [Dal+13; CGG14; PSC12]. Moreover, a framework might be restrictive with regard to the usage of developed apps. While it typically is most important to consider the terms for apps developed by using the framework, license particularities regarding the framework itself can also play a role. Consider, e.g., that the long-term feasibility (I7) of a framework is limited. If the license is liberal concerning modifications of the framework and an adopting company has the resources and willingness to put effort into maintenance of it, the impact of a questionable long-term feasibility might be reduced. As part of the licensing, the pricing model needs to be considered [HHH15; SK13]. Open-source frameworks are typically distributed freely under varyingly permissive regulations; a premium might be charged for maintenance and consultancy [Fit06]. For-payment frameworks might have a flat fee or either one-time or regular payments bound to certain conditions such as the number of developers, developed apps, and so on.

(I2) Supported Target Platforms: The reason for using a cross-platform framework is to provide apps for several platforms while developing only once. Consequently, the supported platforms are a major concern [CGG14; PSC12]. This remains true with Android and iOS essentially dividing the market for smartphones and tablets among themselves [Fv17]. Widening cross-platform app development to further device categories might in fact increase the number of attractive platforms again [RM18]. In addition, two versions of a platform might be different

enough to consider developing for them to be similar to developing for two distinct platforms (major versions often introduce breaking changes to internal APIs as well as interface and design guidelines, e.g., [Goo18c]). This, again, is particularly relevant when considering different device categories. Typically, recent versions of platforms provide novel features exploited (only) by flagship devices. These might be heavily marketed – consider, e.g., Samsung's Edge displays [Sam14] –, wherefore support is important to reach early adopters [BB57]. At the same time, many users will not adopt new devices, thereby not frequently getting platform upgrades – or none but for a few security upgrades. This problem is worsened by the update behaviour of device vendors who, particularly for Android, maintain *forks* widely compatible to the official release but augmented with custom user interfaces and apps [Dob12]. The situation is likely to become direr in the near future, with markets in developing countries being entered. Inexpensive low-end devices might quickly scale up in those markets but note that *current* devices in several markets *cannot* run the same version of a platform for reason of capabilities (see for example work by [Dono8], [Pén+12], and [MD13]). A final consideration are combined apps that bridge more than one device class. Such an app could, e.g., be designed for a *second screen* and support both smart TV and tablet [NJE17], or serve as companion app such as for smartphones and smartwatches in order to offload computation, use alternative input and output capabilities, or simply cater for different user preferences.

(I3) Supported Development Platforms: Even though apps are not normally developed on the platform they are designed for, multiple possibilities can be encountered. Custom business logic and advanced configuration of the apps can be expressed to different extents, possibly differing from the actual app specification (e.g., domain-specific notations) using one or multiple interoperable programming languages. In addition, some degrees of flexibility play a particular role if teams are heterogeneous, i.e., developers use specific hardware and software [PSC12]. Software in this sense does not only comprise the operation system (with Microsoft Windows and Apple MacOS being the typical choices) but also development tools including the development environment. I3 thereby is related to D1 (Development Environment), although the latter concerns the Integrated Development Environment (IDE) typically used (and often enhanced) for a framework. A good development platform support is furthermore beneficial for integration with additional app development tasks. For example, user interface (UI) and UX design might benefit from multi-platform support [Biso6].

(I4) Distribution Channels: For the majority of users, there are only few ways to acquire news apps for their devices. Typically, platform- or vendor-specific app stores provide large repositories of apps, such as the Apple App Store and Google Play [JB13]. As users are accustomed to searching for apps on these platforms, it is essential to support the proprietary stores to reach a high number of users. While cross-loading of apps technically is easy, vendors might hide this functionality for strategical reasons, and to make sure users do not compromise their own safety. Therefore, broad support for the relevant stores is desirable. While this might seem as naturally given, not all kinds of apps can necessarily be uploaded to all stores. One example are PWAs,

progressively-enhanced responsive web sites that are discoverable via search engines and provide app-like interactions using modern web technologies for offline capabilities, content updates, and notifications [Rus15; MBG18; BMG18]). While explicitly designed for mobile devices, PWAs cannot be installed via traditional app stores. Also, cross-platform frameworks differ in the degree of compatibility with app store restrictions and submission regulations [SK13; DM15]. Particularly well integrated frameworks may support features such as the rating of apps to improve app store ranking as well as roll-out support for updates [HHH15].

(I5) Monetisation: Most apps have neither been created with purely philanthropic purposes nor merely for the joy of programming – although such apps surely exist [Jak13]. Therefore, the monetisation possibilities of apps created with a certain framework need to be assessed. There are several possibilities, which might also be used in combination [DM15]. [Tan16] distinguishes four major monetization models, and we add free apps as a fifth category of apps with specific business purpose:

- *Paid*: Apps can be sold for a one-time fee before downloading or after a limited test period. This is typically done using the proprietary stores (see I4) by using their integrated payment options.
- *Freemium*: If apps follow a freemium model, they can be downloaded and used for free, but users need to pay if they want to have access to advanced features or full content and services after reaching a predefined usage threshold. This model is often employed in games (deprecatorily coined *pay-to-win* [Alh+14] if used excessively), where players for example will progress quicker when buying items for actual money [HC16]. The payment is usually performed via *in-app-purchases*, in case of games often consisting of very small payments (*micropayment*) per feature or upgrade.
- *Paidmium*: This model combines paid downloads and in-app purchases, usually found in complex apps such as navigation. Although not always paid for the initial download, subscription-based models are a form of paidmium as they are not usable without login to a paid account and necessitate a regular payment in order to retain access to the app's full functionality.
- *In-app advertising*: Advertisements can be shown as part of the usage of the app. There are ample possibilities how this can be done (including banner ads, sponsored content, and white-labelling of the app itself) and to which degree the advertisements interfere with the usage of the app [LZI18].
- *Free*: Especially business apps [HKM15] may be offered for free to potential users while simultaneously serving a specific business purpose. This includes information portals to increase customer satisfaction as well as additional services (e.g., apps for mobile banking or service booking). They provide value to mobile users and improve customer loyalty, besides fostering process automation (as even studied before the emergence of the widened possibilities through mobile computing [Meu+00]).

Strictly speaking, apps might also provide features that are undesirable for users. For example, recent studies have revealed that *some* apps contain software components that are able to track ultrasonic sounds used for perfidiously tracking users [Arp+16]. While such means might offer a source of data monetisation, we exclude it from further considerations since we deem it ethically indefensible.

Development frameworks may or may not support means of monetisation and they might offer particular good support for some of them. Such features need to be judged in the light of direct costs and omissions to the app store operator (see I4 (Distribution Channels)). Good support includes interfaces to payment providers, pre-designed functionality for in-app payments, support for various types of advertisements, and access to advertising networks [DC14; Goo18e].

(I6) Internationalisation: Apps are typically distributed globally. Even if only one language version is available, there are normally no restrictions regarding who can install an app. There might be specific reasons to restrict users to local versions or to even prevent the distribution in certain geographic regions. For example, legal conflicts and national legislation may prohibit the distribution in parts of the world (as reported by [Ng+14] for China). From a positive point of view, internationalisation and localisation can offer added value by broadening the base of potential users and by providing better targeted functionality. Localisation can be supported by the development framework. It can even go as far as built-in translation capabilities as well as an easy support of a multi-language operation mode. This is further aided if features such as conversion tools (e.g., for dates, currencies, and units) are provided [SK13]. Additionally, frameworks might bring in support for national idiosyncrasies, e.g., API support for state-specific services, for example regarding authentication or personal data records.

(I7) Long-term Feasibility: The choice of an app development approach can be a strategic decision for a commitment over multiple years. Depending on the framework, the kind of apps developed as well as their intended lifetime, and the situation in the developing company, significant initial investment might be necessary. Moreover, there can be the risk of a technological lock-in (as particularly discussed in the context of proprietary software by [ZZ12]). Initial investment includes market studies, assessments (which, as we hope, are much easier using our framework), fees, training materials and training courses, and recruiting. The risk of lock-in can only be partly mitigated by looking for good compatibility, adherence to standards, and the usage of well-known technologies. It is particularly high for small companies, which might lack the resources to quickly correct an ill choice and which typically will invest in just one cross-platform development framework at a time. Whether a framework is suitable for an extended period cannot be assessed in a completely objective way (you may forecast but you cannot prophesy), but maturity, stability, and activity are indicators that help with an educated judgement:

- The *maturity* of a framework can be judged according to a long-lasting existence, a large community of developers and resulting apps, as well as historic events. The latter may for example mean that it can be analysed how emergent security flaws have been handled.

- The *stability* can be particularly seen when looking at the history and future schedule of releases. At which rate have new features been introduced? Have major releases been backward compatible, and if so, how far? Are update cycles (for minor releases) sufficiently short? Are bug-fixes and security updates provided regularly and timely in case of major flaws?
- Besides release cycles, the *activity* of a framework relates to the general contributions of developers and users: Does an active community exist that reports bugs and discusses solutions to these issues? Is this community likely to provide support where official documentation falls short? Does this community probably even support the future development of the framework when a major backer withdraws? Particularly in case of open-source products not backed by a large corporation, a healthy community might even blend with the development team.

Moreover, if a framework is supported, led, or even owned by a company or a consortium, the reputation of the key stakeholders should be scrutinized. Typically, financial or even technological support (such as code contributions) by commercial entities is particularly valuable for open-source frameworks (cf. the work of [AGB12]). Additionally, news, plans, and rumours can be checked. For example, the announcement of a new framework by a company might eventually mean the demise of its predecessor. Likewise, changes to fundamental technology (e.g., a JavaScript engine) could mean that a framework is strengthened or becomes obsolete. Technology breakthroughs may have the same effect – as could happen with WebAssembly [Wag17]. Finally, it should be considered whether support inquiries require a premium. Such costs might not necessarily be considered negative; in fact, they may hint to a good outlook particularly in the case of open-source software for which commercial "premium support" exists to help with development issues [HHH15; SK13].

7.3.3 Development Perspective

(D1) Development Environment: Rapid development is typically supported by the use of an IDE. The maturity and feature-richness of IDEs can greatly influence development productivity – sometimes also negatively when usability challenges of managing too much functionality overburdens users [KS05]. Features such as auto-completion and the integration of library documentation help with the actual coding. Built-in debuggers and emulators support a rapid app development cycle [HHH15; SK13; CGG14; PSC12; DM15]. If a certain IDE is not enforced by the cross-platform framework, and in particular if there is freedom with regard to accustomed workflows, the initial effort of starting to work with a framework can be significantly lowered. This can lower the set-up effort of dependencies such as runtime environments or SDK [SK13].

(D2) Preparation Time: Apps are typically developed rapidly. Thus, the realized learning curve should be favourable, reflecting rapid subjective progress of a developer in getting acquainted with the capabilities of a framework. The entry barrier is also influenced by the required

technology stack and the number and kind of supported programming languages [XX13; CGG14; SK13; PSC12]. Being able to rely on well-known programming paradigms can further reduce the learning efforts needed before being able to work productively [CGG14]. Moreover, the documentation of the API is important – particularly, if a framework poses unique characteristics or novel ways of providing common functionality. Additionally, "Getting started" guides, tutorials, screencasts, and code examples make a framework more accessible and help to clarify features and idiosyncrasies; a corpus of best practices, user-comments, and technical specifications helps with staying productive once an approach is initially conquered [SK13; DM15].

(D3) Scalability: Particularly in large-scale or distributed development projects, apps need to scale. For this purpose, proper modularisation is needed. The app structure is heavily influenced by the general possibilities for partitioning into subcomponents and by architectural conditions. For example, using the widely applied Model-View-Controller pattern has profound ramifications for other design decisions. Ideally, more developers can be added to a project while the app's functionality grows [HHH15; PSC12]. A framework that supports modular or even component-based development can support this division of labour – or even guide it. Moreover, when layering is supported and components can be given specified interfaces and interaction, a higher level of specialisation is possible for developers. Besides adding to the scalability, this might have a positive impact on software quality.

(D4) Development Process Fit: From the traditional waterfall approach [Roy70] over integrated methodologies such as the Rational Unified Process [Jac99] till the variety of agile methods, many ways of developing software are employed. Although all methodologies have common characteristics [DM12], actual development differs widely. Compare, for example, the design-heavy waterfall approach to Extreme Programming [Bec99]. Consequently, a framework should be compatible with custom ways of developing software. As the first step, it can be scrutinized how much effort is required to create the *minimum viable product*. Frameworks differ with regard to the initial configuration that has to be made, so-called boilerplate code, and the following effort for incrementing the scope. Thereby, D4 is also related to D3 (Scalability), as the organisational aspect of specialisation, which might be fostered by methodology-fit, influences the scalability in terms of functionality. Tailored views and specialised tool can support modularising development with a profound role concept, contrasting the work of full-stack developers typically encountered in small projects [Was10].

(D5) User Interface Design: The UI design is essential when developing user-centred application, which most apps are. At the same time, the input and output heterogeneity of mobile device hardware (A4, A5) poses challenges to the development approach of mobile UIs using either flexible descriptions such as responsive designs or multiple layouts for specific ranges of screen sizes [EVP01; RK18b]. Commonly, not all cross-platform frameworks put weight on platform-agnostic UI aspects, partially leaving it at individual implementations per target platform [Goo18f]. Graphical user interfaces are usually specific to a platform and in many cases only covered by a default appearance defined by the framework [HMK13]. Depending on development

requirements, a separate *What You See Is What You Get* (WYSIWYG) editor can be very helpful. Such editors can be used to design appealing, ergonomic interfaces for multiple devices. They can also increase the pace of development compared to repeatedly deploying the full app to a device or an emulator. At the same time, reasonable support for platform-adequate designs without too much effort from developers is preferable, for instance considering round and rectangular layout types for smartwatches.

(D6) Testing: User interface, business logic, and possible additional components of apps need to be thoroughly tested [HHH15; SK13]. In addition to the well-known techniques and approved strategies of testing desktop and server applications such as unit tests, the context-sensitivity of mobile devices should be honoured. For this purpose, mobile scenarios (such as moving around, or getting a phone call while using an app) need to be considered and external influences (such as varying connectivity) could be simulated [MS15]. In addition to this, monitoring the app at runtime can further improve its testability. This includes, e.g., providing a developer console, meaningful error reporting, and logging functionalities for app-specific and system events. Also, remote debugging on a connected device rather than using emulator environments allows for more realistic test results. Additional tool support can aid testing further and provide test coverage visualisation and metrics to support test controlling [HHH15].

(D7) Continuous Delivery: Life cycle support does not end with testing but should also include deployment. Being able to rely on a solid toolchain greatly simplifies the deployment. For example, a framework may leave you with source code generated for the target platforms, may support the generation of native apps, or may go all the way by providing signed packages, possibly even supporting developers in deploying these to devices or app stores. Frameworks vary greatly, from requiring each target platform's native SDKs to using external build services and cloud-based approaches [HHH15; SK13]. Particularly if following an agile method, *continuous delivery* platforms to automate building, testing, and deploying an app may be used. Frameworks can be explicitly designed to integrate with such toolchains, for example by generating project-specific scripts. Additionally, a framework might offer advanced build options (such as code *minification*) and continuous app store integration (e.g., for automatically publishing updated versions) [HHH15].

(D8) Configuration Management: Often, apps do not exist in isolation but have multiple versions when considering multiple roles (such as user and administrator with different capabilities), theming (or branding for white-label apps), and significant regional peculiarities (e.g., right-to-left script). Depending on available app store features, developers in addition might need to supply different app packages for free and paid versions with varying functionality. Cross-platform frameworks can also support the development of such feature variations similar to product lines, either by providing different app packages or by allowing a dynamic transition between versions without re-installing the app.

(D9) Maintainability: The application life cycle does not typically end with one-time deployment (D7). Rather, software is maintained for a shorter or longer period, over which the code

base evolves [SK13]. Maintainability generally is hard to quantify. Although simple metrics such as Lines of Code (LOC) can give a basic idea (the more source code the harder to maintain), more complex metrics might reveal a different picture, for instance when considering code complexity (cf. with the work of [GK91], but also with the critical assessment of [She88]). While such metrics can be used to compare an app against reference apps, qualitative aspects need to be taken into account. This concerns readability of code, use of design patterns, the kind of in-code documentation and similar aspects; possibly in conjunction with the amount of training, familiarisation, and other preparatory efforts (see also D2). These considerations are similar to the discussion about programming languages, where so-called *gearing factors* are used to compare the amount of code per unit of functionality [QSM09]. It is problematic to apply advanced maintainability metrics due to the heterogeneity and varying complexity of frameworks. This counts even stronger for generative approaches and the resulting diversity of platform-specific programming languages. As an additional aspect, the reusability of source code across development projects can be evaluated as well as the portability to other software projects [SK13].

(D10) Extensibility: Although a framework should cover a certain scope and enable the development of *typical* apps within that scope, project-specific requirements may go beyond the provided functionality. However, if they are unlikely to be added as features due to a low general priority, a framework's extensibility becomes important. It can be more flexibly used if custom components can be added and third-party libraries can be included. Typically covered areas include extensions for the UI (such as alternative or additional widgets), access to device features, and libraries for common tasks such as networking and data transfer [HHH15; PSC12].

(D11) Integrating Custom Code: Some applications require native code or third-party libraries to be run. While this seemingly contradicts the principle of cross-platform development, it can be necessary in some cases. This applies in particular when the desired functionality cannot be realized using extensions (D10). Using native platform APIs might enable access to platform functionalities and device features that are not currently supported by a framework or unique to a platform [SK13; PSC12]. Moreover, companies might want to integrate native code to reuse functionalities, for example when successively migrating apps that were developed natively to a cross-platform approach.

(D12) Pace of Development: While many of the above criteria have an influence on how rapid development will be, there are some particularly influencing factors. Especially the amount of boilerplate code necessary for functional app skeletons (cf. [Hei+14]) and the availability of pre-defined functionality for typical requirements (such as user authentication) facilitate swiftness. Ignoring possible salary differences based on programming language proficiency, the overall development speed has direct influence on the variable costs and, ultimately, the return-on-investment.

7.3.4 App Perspective

(A1) Access to Device-specific Hardware: While today's devices possess high processing power, their hardware features – especially sensors – account for the versatility and ubiquitous use. In consequence, access to platform- and device-specific hardware is vital for cross-platform frameworks [HHH15; Dal+13; CGG14; SK13; DM15]. Frameworks with poor coverage incur the risk of feature-poor apps to be developed. A plethora of device hardware is present today, including sensors such as camera, microphone, GPS, accelerometer, gyroscope, magnetometer, and temperature scale as well as novel additions such as a heart rate monitor. Moreover, devices (more precisely: cyberphysical systems) may also offer bidirectional interaction through actuators, enabling them to modify their environment (e.g., in smart home apps).

(A2) Access to Platform-specific Functionality: Similar to A1, apps can only make use of the versatility of modern mobile devices if frameworks provide access to the possibilities they offer. Such platform-specific functionalities include a persistence layer, providing file system access and storage to a database, contact lists, information on the network connection, and battery status [HHH15; DM15]. Furthermore, support for extending the app with complex business logic using general-purpose programming languages might be required in specific app projects but may be inherently restricted by the framework's paradigm of development. In-app browser support can make development much easier when web-based content is accessed [HHH15]. Background services can serve for the realisation of continuous feature execution such as push notifications and monitoring [SK13].

(A3) Support for Connected Devices: In addition to accessing hardware and platform functionality (A1, A2), the support for connected devices can be scrutinized. Wearables and other small mobile devices as well as sensor/actuator networks of cyberphysical systems often rely on coupling with a master device, typically a smartphone. This enhances their capabilities or might be used for occasional synchronisation. The interaction of devices respectively the extension of capabilities through other gadgets has become more important; thus, the support of viable device combinations by frameworks should be assessed [Sey+15]. Specifically, this concerns the kind and richness of access to coupled devices, their data, and their sensors. Moreover, the provision of additional UI components should be evaluated, if applicable. The latter for example applies to smartwatches, which on a smaller screen provide a selection of a host device's functionality. Realising the support can be trivial if a platform provides a layer of abstraction exposing the coupled device as if it was a regular device component. However, particularly due to the multitude of possible combinations and the specificity of these, cross-platform frameworks will need to provide explicit support in many cases, which brings additional complexity.

(A4) Input Device Heterogeneity: Mobile devices allow for a multitude of inputs. This includes traditional means such as keyboard and mouse, (multi-) touch screens, remote controls, and hardware buttons, as well as modern means such as voice recognition and futuristic input technologies using gestures or neural interfaces [RM18]. However, not all devices allow all

possible means of input; this is even true within one device class (consider, e.g., smartphones that understand ultrasound gestures [Hor16]). In addition, devices typically support complex inputs via several alternatives. Think, e.g., of a smartphone screen, which can be manipulated via multi-touch gestures such as taps, swipes, pinches, and pressure, but also reacts to orientation changes and hardware buttons. Cross-platform frameworks need to make these possibilities available to developers, consider the lack of input actions on individual devices, and respect platform- or device-specific patterns (e.g., scrolling on smartwatches may be achieved by rotating bezels, digital crowns, or screen swiping). Ideally, they should also provide support for simple usage – for example by providing means to register multi-touch events or more abstract user actions instead of the need to observe single touches and make sense of their combination.

(A5) Output Device Heterogeneity: Heterogeneity is also given for the output possibilities a device offers. Most have screens for visual output, which differ in size, resolution, format (quadratic vs. rectangular vs. round), colour palette, frame rate (e.g., very slowly updated E-ink screens), and opacity (e.g., augmented reality projections). Moreover, many other possibilities for output exist, such as projection and sounds [RM18]. Adaptability is challenging for traditional devices already [AK14] and becomes very complicated with novel gadgets. Moreover, apps need to cope with device class specific context changes to realize well-understood design ideals [SAW94] such as a day/night screen mode for in-vehicle apps.

(A6) Application Life Cycle: The life cycle inherent to an app should be supported by a framework. This must not be confused with the development life cycle addressed in D4 (Development Process Fit), D7 (Continuous Delivery), and D9 (Maintainability). The app life cycle comprises of starting, pausing, continuing, and exiting an app [SK13], as well as possible others states in dependence on the platform. Multithreading, continuously running background services, and notifications further extend the states in which an app is executed without necessarily providing a graphical UI. In addition, individual views and view elements might have divergent states or even life cycles, e.g., enforced teardown of inactive widgets on Android to reduce memory usage [Goo18i].

(A7) System Integration: Many apps rely on (business) backend systems, which is typically in the interest of the app vendors [MH13]. Frameworks preferably offer several options for integration in existing ecosystems and workflows, including support for data exchange protocols and serialisation as well as multiple data formats [Dal+13]. Apps need to be able to consume web services for data storage and processing. Ideally, inter-app communication should be possible (consider a banking app requesting transaction authorisation from an identity verification app). Additionally, workflow-oriented use cases typically rely on collaboration from several user roles, which may be supported by an app framework. Finally, system integration also means that apps need to be customisable, e.g., to follow the overall design endorsed by a *corporate identity* [SK13].

(A8) Security: App security is an increasingly discussed topic with many facets [Wat+17]. Frameworks can support the development of secure apps with regard to several security attributes [Par98]:

- As regards *confidentiality*, mobile platforms provide means for managing access permissions regarding platform and device features. In general, such permissions should be handled restrictively. Apps should only request permissions on demand (e.g., access to contacts only if contacts are to be imported) [Goo18h]. Concerning the generally low understanding of app permissions [Kel+12], this might raise the user awareness and acceptance of apps respecting security best practices.
- From an *integrity* point of view, sensitive data should be secured using encryption on the device file system or database. Moreover, secure data transfer protocols impede eavesdropping when communicating with backend systems and web services [Dal+13; HHH15].
- Regarding *control*, support for user-input validation and prevention of code injections, cross-site request forgery, and similar attack patterns are preferable [HHH15].

A framework might provide basic or advanced support for security, ideally freeing inexperienced developers from explicitly implementing security-relevant functionality.

(A9) Robustness: Criteria A1–A5 leave much freedom to the app developer. Apps should include intelligent fallback mechanisms in case specific features are unsupported or restricted. The naïve option is to redirect a user to a web page. More sophisticated handling includes *graceful degradation* techniques such as simpler representations [Ern+16] and the employment of alternative functions that make up for the unavailable ones. In addition, robustness also refers to fault-tolerant and resilient mechanisms, for example by acting gracefully if permissions are denied by the user or sensors are deactivated. Fault-tolerance particularly applies to common situations with poor or unavailable Internet access. A framework should enable offline capabilities to keep apps operational in low connectivity situations, for example by storing assets and content locally on the device and caching data that needs to be sent to the backend server as soon as connectivity is re-established (e.g., [Chu+12]).

(A10) Degree of Mobility: Mobility considerations for the desired apps also influence the framework choice. In contrast to the related criteria on available target platforms (I2) and hardware access (A1), different degrees of mobility strongly affect app mechanics and emphasize features beyond infrastructure considerations. On a high level, four categories can be distinguished [RM18]:

- *Stationary*: App-enabled devices do not need to be mobile (e.g., smart home devices). They differ from traditional desktop applications regarding input/output characteristics but barely need to consider contextual information.
- *Mobile*: Typical mobile applications must process various types of context information such as location, time, and further sensor values (e.g., ambient light for in-vehicle UIs).
- *Wearable*: In addition to the usage context, wearables need to adapt to personal preferences and unobtrusively blend with the user's life (e.g., when to propose recommendations or avoid distraction through notifications). Also, other types of sensors might require continuous event processing for applications such as health monitoring.

- *Autonomous*: The highest degree of mobility requires advanced self-adaptation capabilities for the cyber-physical system to react to expected and unexpected situations. The app might therefore exhibit agent characteristics or apply business rules for automated decision making.

7.3.5 Usage Perspective

(U1) Look and Feel: The UI elements provided by a framework should have a native look and feel rather than resembling a web site [SK13]. If generated apps use a truly native interface, it should be created in the way typical for a platform. Elements, views, and interaction possibilities can be evaluated according to the respective human interface guidelines provided by platform vendors [Goo18a; App18a]. Apps should also support the platform-specific usage philosophy, e.g., regarding the position of navigation bars, scrolling, and gestures [SK13]. While form-based interfaces suffice in many cases [HMK13] and are relatively simple to realize, richer user interfaces with 2D animations are harder to realize. 3D environments and multimedia features are particularly challenging for cross-platform frameworks [Dal+13].

(U2) Performance: Poorly performing apps likely face low user acceptance. Performance comprises of aspects such as app load time, app speed for changing views and computations resulting from user interaction (*responsiveness*), perceived speed of network access, and stability. While the subjective impression is important and might differ according to individual projects' requirements, some performance aspects can be measured. This includes the start-up time, the time to awake after interruptions, and the time to shut down [DM15]. Additionally, resource utilisation can be scrutinized. This includes CPU load, memory usage, battery drain during runtime (and possibly while background services remain active), and download size [SK13; CGG14; Dal+13; CG15]. Performance aspects need to be carefully balanced, as a pure focus on performance can negatively impact many other criteria and optimisation for other criteria might negatively affect performance (for instance by bloating an app).[2]

(U3) Usage Patterns: Apps are used in typical patterns. This includes many apps that are used infrequently and some apps that are often used, although normally only for a short amount of time and often with interruptions. Users desire an "instant on" experience and continue where they left the app. Unsaved data ideally should be available even after closing the app or even after rebooting a device. Data retrieved from the Internet should stay available when temporarily loosing connectivity. Apps should align with personal workflows for information processing such as sharing with other apps or saving to persistent storage. Moreover, they should integrate with common apps for interaction with other users, such as messaging, email and social media services. Data-intensive apps, especially if they have desktop counterparts, should support synchronisation of app data across multiple devices. In particular, background synchronisation for seamless, transparent context switching is desirable. Additionally, apps should make use of platform-wide

[2] The interrelation of criteria is illustrated in Figure 7.1 and discussed in Subsection 7.6.1 (p. 165 onwards).

services such as a notification centre or means to store certain types of documents (such as *Apple Wallet* for boarding passes and similar documents [App18b]).

(U4) User Authentication: User management becomes increasingly important: apps may have a purely local, single user, cloud-based accounts (e.g., enabling services such as synchronisation), or employ centralised user management with multi-device account management or role-based access rights [Kun+14]. Similarly, authentication is possible on an app-level or server-based. Apps might include session management, and they might cache login information (e.g., for a limited-functionality offline mode). Ideally, frameworks should offer several ways for user authentication, including traditional pins and passwords, gestures, based on biometric information, and voice recognition [LL15]. While this criterion has many implications for the app perspective (and clever features of a framework can considerably simplify the developers' job), we have put it under the user perspective to underline the importance of it for working with an app.

Table 7.3: Literature references to criteria and related terms

Criterion		Literature referencing the criterion or related/subordinate terms
I1	License	[PSC12; HHM13; DM15], (direct) costs [DM15; HHH15; HHM13; SK13], open-source [HHH15; VSB13; PSC12], availability [El-+17]
I2	Target Platforms	Supported platforms [El-+17; DM15; Dal+13; SK13; VSB13; HHM13], mobile platforms [VJ17], versions [BEP16], portability [SK13], mobile operating systems [PSC12]
I3	Development Platforms	(Programming/development) languages [QGZ17; BEP16; CAB15; OT12; PSC12; VJ17; Rd12; DM15], computer operating systems [DM15], technologies [XX13], OS support [PSC12]
I4	Distribution Channels	App store [CAB15; DM15; SK13] publishing [LA17], distribution [QGZ17; XX13; HHM13], market [CAB15] market place deployment [XX13], analytics platform [DM15]
I5	Monetisation	Sales [LA17], in-app purchases [DM15], mobile ad platform support [DM15]
I6	Internationalisation	[SK13]
I7	Long-term Feasibility	[HHM13], popularity [LA17], count of updates [VSB13], community [VSB13]
D1	Development Environment	[Lat+16; HHM13; Rd12], IDE [QGZ17; DM15; HHH15; PSC12], tool restrictions [SK13], dependencies [SK13]
D2	Preparation Time	Documentation/documents [LA17; QGZ17; VJ17; BEP16], documentation completeness and quality [SK13], learning curve [LA17], learning effort [SK13], community [VJ17], speed and complexity of installation [VJ17], developer and user support groups [HHH15], ease of development [HHM13]
D3	Scalability	[Lat+16; HHM13], complexity [LA17], architecture [El-+17], architectural implication [HHH15], MVC support [DM15; PSC12]
D4	Development Process Fit	Development process [UB16], architecture [PSC12]
D5	UI Design	GUI design(er) [HHM13; OT12], graphical tool for GUI [LA17], UI design assistant [BEP16], no-code/low-code support [HHH15], customizability [SK13]
D6	Testing	[UB16; SK13], debugging [QGZ17; BEP16; DM15; SK13; OT12], simulator [Lat+16], emulator [HHH15; OT12], test framework [HHH15]
D7	Continuous Delivery	building time [JET18], build service availability [DM15], build support [HHH15], simplified/automatic builds [SK13], compile without SDK [OT12], instant update [CAB15], upgrade [QGZ17], updates [HHH15]
D8	Configuration Management	Production support [HHH15]

Criterion		Literature referencing the criterion or related/subordinate terms
D9	Maintainability	[Lat+16; HHM13], supportability [SK13]
D10	Extensibility	[SK13], libraries [HHH15], app extensions [Dal+13], plug-in repository [VSB13], plug-in extendibility [PSC12]
D11	Custom Code Integration	Native access [BEP16], extensibility with native code [OT12], native APIs [PSC12], reuse [BEP16], app layer [UB16], access to native UI [DM15]
D12	Pace of Development	Development rate [LA17], speed of development [HHM13], developing time [BEP16], time to market [LA17], budget [LA17], complexity of development [VJ17], easiness of development [AHN16]
A1	Hardware Access	[XX13], device features [Lat+16], device API [CAB15], device resource support [HHH15], sensor data capture [DM15], built-in features [Dal+13], hardware sensors [SK13], mobile device functions [VSB13], platform-specific features [HHM13], APIs [PSC12], accelerometer [CG17; CG15; CGG14; DM15; VSB13; PSC12; Rd12], compass [CG17; CG15; CGG14; DM15; PSC12], proximity [CG17; DM15], GPS [CG17; QGZ17; CG15; CGG14; DM15; Rd12], geolocation [SK13; VSB13; PSC12] camera [CG17; QGZ17; CG15; CGG14; DM15; VSB13; PSC12; Rd12], audio record [CG17], microphone [CG15; CGG14; DM15], Bluetooth [DM15; OT12; PSC12], accelerator [QGZ17], GPU acceleration [DM15], light [CG17], notification light activation [DM15], noise cancelation microphone [DM15], NFC [DM15; PSC12], gyroscope [DM15], barometer [DM15], Wi-Fi positioning [DM15], cellular positioning [DM15], network [SK13; VSB13], low-level networking [DM15], connection [PSC12], (hardware) buttons [SK13], device (information) [PSC12]
A2	Platform Functionality	Native features [LA17], device resource support [HHH15], functionality [BEP16], platform-specific features [HHM13], contacts [QGZ17; DM15; VSB13; PSC12; Rd12], media [QGZ17; VSB13; Rd12], files/file system access [DM15; VSB13; PSC12], (user/system/push/alert/sound) notifications [DM15; VSB13; PSC12], calendar [DM15; PSC12], SMS [DM15], call log [DM15], voice activation [DM15], native map support [DM15], background processes [DM15], in-app browser [HHH15], storage [VSB13; PSC12], data access [XX13], local database [HHH15], database access [SK13], barcode (scanner) [VSB13; PSC12], menu [PSC12]
A4	Input Heterogeneity	touch support [HHH15], gestures [SK13], swipe, pinch [DM15]
A6	App Life Cycle	[SK13]
A7	System Integration	Social APIs, Cloud APIs [DM15], backend communication [Dal+13], corporate identity [SK13]
A8	Security	[LA17; Lat+16; Dal+13], secure storage access, code obfuscation [DM15], security vulnerabilities, encrypted local storage [HHH15]
A9	Robustness	stability, reliability [SK13]
U1	Look and Feel	[XX13; HHM13; LA17], user experience [LA17; AHN16; HHM13], appearance [AHN16], (rich) UI [BEP16; Dal+13], UI response time [JET18], interaction-response [HEE13], user-perceived performance [XX13], intuitiveness [OT12], UI functionality, native UI components [SK13], fluidity, animations [LA17]
U2	Performance	[SK13], execution time [BG18], duration [Cor+18; Del+18], energy/power consumption [Cor+18; CG17; CG15; CGG14; Lat+16; Dal+13], app size [JET18; AHN16; OT12], size of installation [BMG17; SK13], CPU (load) [Cor+18; Lat+16], CPU occupancy ratio [QGZ17], RAM/memory usage [JET18; OT12; AHN16], memory occupancy [QGZ17], application/activity launch time [BMG17; OT12], rendering time [JET18; BMG17], start-up consuming time [QGZ17], app starting time [AHN16], installation consuming time [QGZ17], battery temperature [QGZ17], network flow [QGZ17], resources consumption [Lat+16], application speed [HHM13]
U3	Usage Patterns	user experience conventions [OT12], screen rotation [DM15; PSC12], device orientation [CG17], accessibility features [OT12], frequency of use [LA17], offline mode [LA17; CAB15]

7.4 Weight Profiles

In the following, we first give the rationale of weight profiles before explaining the application. We then provide notable examples.

7.4.1 Rationale

There are two sides to the evaluation of technological and technical criteria. First, there is the actual assessment of a phenomenon (in this case of a cross-platform development framework) under that criterion. It should be based on facts and made as little subjective as possible. Thereby, it should also be replicable. In general, this assessment is not individual, i.e., it should be the same independent of situation, context, and assessor. Second, there is the importance of the criterion for the individual situation. This is specific to a setting and dependent on context, personal preferences, and applicability of the criterion.

To cater for this observation, criteria ought to be weighted instead of simply using their average assessment score to denote the overall assessment of a framework. This also serves the purpose of balancing different levels of technical depths of criteria, which is unavoidable given the heterogeneity of the considerations reflected in our criteria catalogue.

Despite the individuality of the weighting, developers and companies in need of a decision typically face one of a number of comparable settings. Therefore, along with our evaluation framework we propose *weight profiles*. These correspond to typical development settings; they are a kind of patterns or templates. Profiles can be used *as-is* for a quick assessment and to gain an overview. Alternatively (or successively), they can be used as the foundation for an individual assessment. Instead of needing to start from scratch, a profile provides a reasonable weighting for a typical situation, which commonly will need only some tweaking. Weight profiles are not meant to be static but to evolve with the general evolution of the mobile computing ecosystems. Thus, they also serve an important part of keeping our framework timely.

One of the experts we asked to assess our criteria catalogue (see Section 7.5) noted that such an "approach could help also to document thoroughly the rationale behind specific tool selections". This might be particularly important in corporate decision making, where ultimately "supervisors with or without technical experience" will decide. Additionally, weight profiles also honour a divide-and-conquer proceeding in which individual criteria are assessed by narrow-area experts, criteria categories are evaluated by experts with more general focus (compare, e.g., a backend programmer to a senior software engineer), IT managers set the weights, and finally general management makes the decisions.

7.4.2 Application

Each of the 33 criteria receive a weight between 1 and 7[3]. The total is 100 points, i.e., each point denotes a 1% weight. In case a criterion should be omitted in the assessment, assigning a weight of 0 is also possible.

Each evaluated criterion is assigned a score from 0 (criterion unsatisfied) to 5 (criterion fully satisfied). The overall score S of an assessment is then calculated as the weighted arithmetic mean of the criteria, i.e., as

$$S = \frac{\sum_{j=1}^{7} w_{i,j} * i_j + \sum_{j=1}^{12} w_{d,j} * d_j + \sum_{j=1}^{10} w_{a,j} * a_j + \sum_{j=1}^{4} w_{u,j} * u_j}{100}, \qquad \sum_k w_k = 100$$

with i_j, d_j, a_j, and u_j being the criteria score from the infrastructure, development, app, and user perspective, respectively, and $w_{i,j}$, $w_{d,j}$, $w_{a,j}$, and $w_{u,j}$ the corresponding weights for each criterion. We suggest rounding to two decimal digits; finer-grained scores hardly have a practical relevance. The nearer the score is to 5, the more wholesome is a framework; a score of 0 would be given to one that is entirely dysfunctional. Which ranges of scores are to be expected, sufficient, and realistic will be revisited in the discussion (Section 7.6) of this article.

This weighted summation is deliberately chosen to ensure simplicity and understandability for all stakeholders in the assessment process. Essential requirements such as weights being proportional to the relative value of criterion score changes are fulfilled by using equal intervals for all scores across heterogeneous criteria [Hob80]. In contrast to more advanced techniques for multi-criteria analysis such as rank-based criteria assessment or pair-wise comparisons [BB96], the chosen approach is modular in nature such that weights and scores can be defined by different experts in the respective domain. In addition, new frameworks can easily be added to the decision process without re-evaluating all combinations. This capability is particularly suited for today's fast-changing world in which mobile development frameworks constantly appear and disappear.

7.4.3 Example Profiles

Table 7.4 (following in the next section, p. 158) provides a weighting with the weights of the *smartphone* weight profile, along with the weights of five additional profiles. The smartphone profile is the most standardized profile; its weights are well backed by empirical and theoretical work as well as experience from practice.

Prior work has shown that cross-platform approaches often focus developers' need [Res14]. Typically, open-source approaches will be appreciated (I1 (License)). Undoubtedly, good support

[3]Higher weights are not only impractical but also questionable given the granularity of our criteria. With high weights for single criteria, the weight for the majority of criteria would be very low, possibly leading to a point where assessing them would make little sense since they would have no significant effect on the overall score. Our criteria are meant to offer a balanced assessment where no single criterion inherently is more important than the other.

of the desired target platforms has a high value (I2), and a long-term feasibility will be sought (I7). In alignment with these infrastructure considerations, developers find a proper development environment (D1), an adequate preparation time (D2) and swift development progress (D12 (Pace of Development)) important. For smartphone usage, a good user interface (D5 (UI Design)) is essential. Regarding the actual app, particularly proper access to device hardware (A1) and platform functionality is needed (A2). Finally, from the user perspective smartphone apps needs to provide a near-native appearance (U1 (Look and Feel)) and a good runtime performance (U2). Consequently, in the smartphone weight profile $\frac{1}{3}$ of the criteria (i.e., 11) have 56% of the total weight.

This generic smartphone profile can be amended to fit with particular needs. For example, if the choice of a framework has specific strategical significance, full weight (7) could be given to I1 (License) and I7 (Long-term Feasibility), and possibly a higher weight to D9 (Maintainability) and A7 (System Integration). At the same time, D2 (Preparation Time) and D3 (Scalability) could be assigned very low weights.

Similarly to the smartphone profile, it makes sense to provide a *tablet* profile. It could be used for apps specifically targeting tablets. Depending on the target of development, it might be used in conjunction with the smartphone profile, e.g., when apps are supposed to provide specific support to different screen sizes and orientations. The tablet profile is very similar with smaller deviation of typically only 1. However, such small deviations might tip the scales if several matured frameworks are almost equipollent with the smartphone weights.

The *entertainment* profile applicable for example to augmented / virtual reality devices has many similarities to the aforementioned ones; however, less emphasis is given to the app perspective (with exception of an app's robustness (A9) – low robustness of, e.g., games is very frustrating). Much attention is given to the user perspective to satisfy users' needs when using an app to be entertained. The notable exception is the look and feel (U1). While we routinely stress the importance of a native look and feel of an app, entertainment apps often come with a highly individual user interface. Particularly games often do not adhere to a platform's interface standards at all but provide custom elements to create an *immersive* atmosphere.

Although the app-enablement of cars is just at its beginning and uncertainty exists regarding the best approach for secure and reliable platforms [Man+18], we propose a *vehicle* profile to illustrate a weighting that is less similar to the former three than those are to each other. Save for the long-term-feasibility (I7), less weight could be given to the infrastructure perspective. Regarding development, we consider the UI design (D5) to be particularly important, so that apps for cars can be aligned with existing infotainment systems. Moreover, testing of such apps (D6) is vital, as even in non-security critical areas app crashes and malfunctions are highly undesirable for their distraction alone. In the app perspective, we would put weight on access to hardware (A1), which would be unlike the hardware apps typically have access to. Highest importance must be given to security (A8). Particularly an app that has been given wide hardware access must not

be exploitable to gain access to a car's internal functions. Additionally, the degree of mobility (A10) should ideally be very high.

Finally, due to the rise of IoT applications with interfaces to consumer usage, we deem a *smart home* profile to be utile. Even though this field is also still emerging and the weights might need adjustments in the near future, an initial assessment can be made. Due to the very high heterogeneity, adequate support of possible target platforms (I2) is essential. For the same reason, special emphasis should be put on long-term feasibility (I7) at this stage. Development aspects can get somewhat less focus but for a powerful IDE and a low preparation time, enabling a rapid start with trying out functionality – and a low penalty for changing to another framework if necessary. An app requires profound hardware access (A1), good connectivity with a wealth of other devices (A3), and a high level of security (A8).

Weight profiles are not limited to typical settings in corporate app development. One of our experts suggested that weighting would also be useful if students work on development projects. Weighting would then be aligned with the curriculum. Thus, there could for example be a CS-123 "Mobile App Development Laboratory" profile. It would be a specialisation from the smartphone profile that discards – in this setting – superfluous criteria such as I4 (Distribution Channels), I5 (Monetisation), I6 (Internationalisation), and D8 (Configuration Management).

7.5 Evaluation Study

According to the taxonomy by [MEK15], three main approaches to develop apps can be distinguished (although recently introduced frameworks such as React Native and NativeScript are blurring the lines between them):

1. app employing a runtime-environment, subdivided into (Progressive) Web Apps, hybrid apps (similar to Web Apps but wrapped in installable containers), and self-contained runtimes,
2. generative approaches, subdivided into model-driven software development and transpiling of existing apps, and
3. native development.

Obviously, only the first two qualify for cross-platform development. The following evaluation compares a heterogeneous selection of frameworks in order to demonstrate the applicability of the proposed evaluation scheme to different development approaches. In particular, native app development is contrasted to the hybrid app framework PhoneGap as well as to React Native and to (Progressive) Web Apps. PhoneGap was chosen due to its perennial prominence and popularity as leading cross-platform development tool using hybrid apps [Sta18a]. As a relatively new approach, React Native aims to combine the advantages of native UI elements with the familiarity of JavaScript among web developers [Fac18]. Web Apps on the other end of the spectrum bridge

the gap to traditional web development – especially with regard to the recently introduced concept of *progressive* Web Apps, which intensify the integration of web and smartphone applications.

This evaluation does *not* aim for a comprehensive survey of the investigated frameworks but rather serves as a benchmark for our evaluation criteria. Nevertheless, it can be used by researchers to scrutinize our criteria; for practitioners is can serve as a starting point. More detailed comparisons of specific frameworks are for example provided by [DM15] or [HHH15].

7.5.1 Method

The criteria presented in Section 7.3 have been assessed by several experts, both academic researchers involved in mobile app research and practitioners with experience in cross-platform development tools. Their feedback has been incorporated in the improved criteria description.

To demonstrate the applicability of our criteria catalogue and weighting scheme, we perform an evaluation study. Resulting from the recent emergence of devices, barely any (commercial or academic) framework exists that allows for cross-platform development within one or across multiple device classes regarding novel mobile devices (cf. Section 7.2). Our evaluation study therefore focuses on cross-platform smartphone frameworks to demonstrate the applicability of our weighted criteria approach, which can be backed with empirical and theoretical work. For better assessment, the corresponding criteria (or rather their short identifiers) are given in brackets when discussing aspects of a frameworks through Subsections 7.5.2 to 7.5.5.

As a hands-on scenario of a cross-platform app, we consider the example of a typical business app that performs data manipulation tasks for field service workers. Salespersons need ubiquitous access to the company's information systems to support their daily work. For example, they are often away on business and frequently experience context switches in sales talks with customers or while travelling. Retrieving up-to-date information such as current inventory levels is essential for decision making. Also, performing tasks such as order placement or master data management can be accomplished efficiently. Using a mobile cross-platform app, field service workers can use a device of their choice and benefit from digitized business processes.

As elaborated in Section 7.4, the chosen weights presented in Table 7.4 (p. 158) are therefore not inalterable but adapted to this specific use case. Arguably, they represent a suitable weight profile regarding business apps for company-internal usage based on the following considerations. Previous studies have shown that cross-platform approaches are often driven by IT departments to enable efficient development for multiple target platforms [Res14]. From an infrastructure perspective, this means that open and extensible approaches are considered to be particularly important. Concerning long-term feasibility, the dominance of Android and iOS as main players on the mobile operating system market [Fv17] has created a stabilized smartphone ecosystem. Consequently, distribution channels are mostly limited to the respective platform-specific app stores, which offer a broad set of features such as limiting the deployment to a geographic region or offering multiple language versions.

App developers want to use existing standards and previous knowledge to progress quickly with the task at hand [Res14]. Until now, apps are commonly developed in small development teams. Hence, the organisational aspects of software engineering practices such as scalability, maintainability, and the integration in team-oriented development processes have low priority [Res14]. With increasing variety of devices and complexity of the apps itself, effective testing of app artefacts becomes more important when dealing with essential business activities. A targeted delivery of related apps (e.g., language-specific variants or functional variability) is, however, negligible for the given scenario. Most important, UI design using characteristic platform widgets and interaction patterns seems to be an ongoing challenge for cross-platform framework. Beyond company-internal apps, it may even become more important for "standing out from the mass of apps" [AK14].

On the application side, access to a broad rage of device functionalities is frequently requested by practitioners. This trend is facilitated by the convergence of input and output capabilities in the matured smartphone market. In general, apps are mostly used for entertainment and communication purposes [LL17]; thus, business integration and security issues are specific requirements for digitisation projects allowing for the transmission of sensitive data to mobile devices.

In 2016, worldwide mobile usage has already surpassed desktop usage [Sta16], a trend which is potentially accelerated by the plethora of upcoming wearable devices. Therefore, smartphones cannot be treated as merely displaying web content from anywhere – a platform-specific look and feel as well as performance considerations remain important topics for mobile app development. Finally, user authentication is often required for apps because a central backend or cloud environment often serves as content provider for the application or is used for additional services such as synchronisation across devices.

According to this scenario and the derived weight profile, the following subsections discuss the suitability of PWAs, PhoneGap, and React Native in contrast to traditional native app development. We do not justify each score separately (which would require at least $32 * 5 = 160$ sentences clumsily filled with numbers). Rather, we address particularly noteworthy parts of the evaluation, especially if scores do not intuitively make sense in the light of the frameworks and based on above considerations.

7.5.2 (Progressive) Web Apps

Web Apps are web sites created with web technologies – HTML5, Cascading Style Sheets (CSS), and JavaScript (JS) – that are optimized for mobile usage. They are run inside a browser, which makes them compatible with any device providing a browser that supports the features of employed versions of HTML, CSS, and JS. Consequently, Web Apps integrate well with traditional web development tooling (I1 (License), I2 (Target Platforms), I3 (Development Platforms)), yet app distribution cannot be controlled as no centralised app store exists (I4 (Distribution Channels)). Furthermore, the web development foundation constraints its "app-like" behaviour, resulting in

Table 7.4: Comparison of Frameworks and Device Class Weight Profiles for the Exemplary Scenario

			Smartphone Comparison					Category Weights (%)				
Criterion		**Weight (%)**	**Web Apps**	**PWAs**	**PhoneGap**	**React Native**	**Native apps**	**Tablets**	**Entertainment**	**Wearables**	**Vehicle**	**Smart Home**
I1	License	5	5		5	5	5	5	6	5	3	5
I2	Target Platforms	6	5	4	5	4	1	5	6	5	4	7
I3	Development Platforms	2	4		5	4	2	2	2	1	1	1
I4	Distribution Channels	2	5		3	3	4	2	3	4	3	3
I5	Monetisation	1	0		3	3	5	2	1	1	2	2
I6	Internationalisation	1	1		3	3	5	2	2	2	0	1
I7	Long-term Feasibility	5	5		5	3	4	5	3	3	5	5
D1	Development Environment	7	4		5	4	5	7	7	5	5	6
D2	Preparation Time	7	5		4	4	3	7	7	5	1	5
D3	Scalability	2	3		3	4	3	2	3	2	3	2
D4	Development Process Fit	2	3		3	3	2	2	3	1	3	2
D5	UI Design	4	3		3	2	4	4	5	5	6	2
D6	Testing	3	3		4	3	5	3	3	3	6	3
D7	Continuous Delivery	3	5		5	3	3	3	3	4	3	2
D8	Configuration Management	1	0		0	0	3	1	1	2	2	2
D9	Maintainability	2	2		4	4	2	2	2	1	3	2
D10	Extensibility	2	5		5	2	5	2	2	2	1	2
D11	Custom Code Integration	2	0		3	3	5	2	2	1	0	0
D12	Pace of Development	4	3		4	3	0	4	3	3	2	4
A1	Hardware Access	4	2		4	3	5	3	1	6	4	6
A2	Platform Functionality	5	2		4	3	5	5	3	2	2	3
A3	Connected Devices	3	0		2	2	5	2	1	5	3	7
A4	Input Heterogeneity	1	3		4	4	5	3	3	2	2	2
A5	Output Heterogeneity	1	3		4	4	5	1	1	6	3	4
A6	App Life Cycle	2	0	2	4	4	5	3	3	3	3	2
A7	System Integration	3	3		3	3	5	3	3	1	2	1
A8	Security	3	0		0	0	3	4	1	3	7	5
A9	Robustness	2	2		4	2	3	1	4	3	2	1
A10	Degree of Mobility	1	1		1	3	5	1	0	4	5	0
U1	Look and Feel	5	1	2	3	4	5	4	2	4	5	3
U2	Performance	4	2	3	2	3	5	3	6	3	3	2
U3	Usage Patterns	2	0	2	2	2	2	3	4	3	3	4
U4	User Authentication	3	0		0	0	1	2	4	0	1	4
Weighted Score			2.87	2.98	3.59	3.11	3.73					

recurring features of apps for cloud-based services to be re-implemented manually, e.g., regarding internationalisation, payments, or user authentication (I6 (Internationalisation), D12 (Pace of Development), U4 (Development Process Fit)). As smartphones become computationally more powerful and web development standards evolve to incorporate new (hardware) features in public APIs of browser environments, the long-term feasibility (I7) of this approach can be assured.

The recent development has led to so-called Progressive Web Apps (PWA), which need to be distinguished from "classic" Web Apps. While PWAs are also created using web technologies, their possibilities go beyond what was possible so far [MBG18]. According to Google, the key characteristics to provide a better user experience are to create *reliable*, *fast*, and *engaging* applications [Goo18h]. Using *Service Workers*, PWAs can be added to a smartphone's home screen *without* the need to install them like native apps. In addition, recent standards for in-browser storage are used to load app logic and previous content from the device and provide functional applications even in situations with unavailable or unstable network connection. They might thereby close the gap between web site and native apps, providing a contender for the unifier of mobile development [BMG17].

With regard to developing apps, an immense community of developers exists as standard web development skills are required; many tutorials and profound tool support are available to learn (D1 (Development Environment), D2 (Preparation Time)). However, this flexibility also limits the goal-oriented creation of apps. The structure of source code and UI development completely depend on the developers, for instance requiring boilerplate code. Frameworks can provide guidance by structuring the components of the app and consistently applying the concepts of PWAs (D3 (Scalability), D4 (Development Process Fit), D5 (UI Design)). For example, the Ionic framework [Ion18] is based on the common JS library Angular and focuses on the fast creation of Web Apps though a large variety of pre-defined components for UI design and interactions. Whereas several techniques for testing JavaScript exist, desktop browsers can only inadequately emulate the characteristics of mobile devices and mobile in-browser debugging of the complete app life cycle is complicated (D6 (Testing)). On the other hand, established tool chains (for instance build tools such as Grunt or Gradle) can be used to assemble Web Apps in a continuous delivery process, especially regarding the increasing complexity of app product lines (D7 (Continuous Delivery), D8 (Configuration Management)).

The Ionic framework and similar libraries simplify the development of Web Apps through modular components that can be reused in multiple projects; in addition, they are extensible using third-party plug-ins (D10). The execution of native code is, however, generally unsupported within a browser environment[4] (D11 (Custom Code Integration)). Instead, device components can be accessed via HTML5 APIs such as Media Capture Stream and Battery Status, which are varyingly supported by mobile browsers (A1 (Hardware Access), A2 (Platform Functionality)) and depend

[4]Current ambitions of introducing further high-level languages to browsers using binary instructions such as WebAssembly (`http://webassembly.org/`) open up interesting perspectives to native in-browser development, but cannot be considered a feasible alternative, yet.

on the platform vendors' willingness to support these standards [Mob15]. Considering input and output mechanisms, keyboard and gesture support are well established through JS events. These are based on traditional web page behaviour, thus providing only limited support for novel mechanisms such as voice-based interfaces (A4 (Input Heterogeneity), A5 (Output Heterogeneity), U1 (Look and Feel)). Web Apps and PWAs are inherently bound to JavaScript engines, which are unavailable on most wearable devices; also, browser environments are not designed to interoperate with connected devices (A3 (Connected Devices), A10 (Degree of Mobility)).

As stated before, all aspects regarding system integration, security, and robustness have to be built manually based on web technology without platform-specific abstraction (A7 (System Integration), A8 (Security), A9 (Robustness)). The largest advantages of PWAs are related to the usage perspective. By storing application code and previous content on the device instead of fully reloading a web site, the perceived performance is drastically improved [Goo18g]. Through service workers running in the background even after "leaving" the Web App, the application life cycle is better supported (A6). Also, the application can save user-related characteristics to the local device and instantly adapt to the users' preferences when reopening the PWA (U2 (Performance), U3 (Usage Patterns)).

7.5.3 PhoneGap

The first stable release of PhoneGap was developed in 2009, just two years after the introduction of the first smartphones such as Apple's iPhone. Since then, it has evolved to one of the top-used cross-platform development tools [Dav09; Sta18a]. PhoneGap and its open-source foundation Apache Cordova are representatives of the hybrid app development approach. Essentially, a regular Web App is developed with HTML5 and JavaScript and subsequently wrapped in a container that uses a *Web view* component for rendering the content without browser controls. In addition, the framework provides a bridge to access native device functionality through a common JavaScript API (A1 (Hardware Access), A2 (Platform Functionality)). The platform-specific wrappers allow for packaging installable apps for all major smartphone platforms. These can be distributed via regular app stores and utilize their general monetisation and internationalisation features except for deeply integrated features such as in-app purchases (I1-I6).

Because of the underlying app content, most of the freedoms and restrictions of Web Apps apply likewise. The framework's structure is well documented and many tutorials are provided by the community to quickly gain momentum and start custom development from a minimal running app skeleton. Consequently, the developer can create the app using any methodology, modular structure, and web development environment (D1-D5, D12 (Pace of Development)).

Due to subtle differences between browser engines on desktop computers and smartphones, hybrid apps with embedded WebView components are more complex to test than Web Apps. Also, running a WebView interlinked with custom app code introduces new security risks [Luo+11]. To support the testing of intermediate app prototypes, PhoneGap offers a remote debugging interface

that connects to actual devices (D6 (Testing), A8 (Security)). Furthermore, the delivery of the resulting apps is simplified through a cloud-based deployment service to build app packages without locally installed SDKs (D7 (Continuous Delivery)). Through the extensive API as level of abstraction, differences across platforms are counterbalanced, which increases the overall speed of development and ensures the maintainability when platform implementations evolve (D9 (Maintainability), D12 (Pace of Development). A large set of plug-ins exist to augment the PhoneGap core functionality with additional features (D10 (Extensibility)). In contrast to Web Apps, this also includes native code components that can be added to the bridge component in order to extend the JS-accessible API to further device hardware (D11 (Custom Code Integration)). Consequently, various input and output capabilities are supported. The *Event API* and *Device API* additionally allow for managing the overall app life cycle (A4-A6) [Apa15; Ado15].

Similar to Web Apps, the approach is limited to smartphones and tablets due to the required JavaScript engine (A10 (Degree of Mobility)). However, third-party plug-ins exist to connect PhoneGap apps to external devices such as Wear OS smartwatches [Gar18] (A3 (Connected Devices)). With regard to the native look and feel, PhoneGap apps not automatically comply with individual platform guidelines but provide a generic mobile appearance (U1 (Look and Feel)). All app functionality needs to be implemented manually (A7 (System Integration), A8 (Security), A9 (Robustness), U3 (Usage Patterns), U4 (User Authentication)). Finally, the additional functionality through the wrapper component incurs a performance overhead (U2) as, e.g., analysed by [DM15].

7.5.4 React Native

Traditionally, the decision of implementing apps using either a hybrid approach or native development splits the community of developers into two camps. Besides many subordinate differences, a long-lasting controversy revolves around sacrificing a convincing platform-aligned appearance for the benefit of using web technologies well-known by a large community of developers. Several papers have investigated this topic and works such as the present article aim to guide developers in choosing an approach adequate to the project-specific requirements [MMM16; QGZ17; AF14]. However, a variety of frameworks such as NativeScript and Flutter have recently emerged that try to bridge this chasm by creating native UI components while specifying the app with JavaScript [Pro18; Goo18d]. A popular framework, React Native, is backed by Facebook and was presented in 2015 [Occ15]. Using the API of the JavaScript framework React (also called ReactJS), the whole app, including view elements, is specified using JavaScript and a template syntax called JSX. Instead of rendering the content in a browser component, the frameworks is based on a runtime approach that uses a JavaScript engine to execute business logic but transforms UI-related code into commands to the native UI elements.

React Native is distributed under the permissive MIT license (A1) and currently targets the two major platforms Android and iOS (A2). Developers can freely choose their development platform and environment that supports JavaScript (I3 (Development Platform), D1 (Development

Environment)). As the resulting artefacts are installable app packages, capabilities and restrictions regarding distribution channels and app store features such as monetisation and internationalisation are similar to hybrid apps (I4-I6). However, due to the relative youth of the framework, long-term reliability (I7) has not yet established and the developer community is still comparably small. The current trend towards JavaScript frameworks with native UI components might change this assessment in the future. Also, being backed by Facebook reduces the risk of discontinuation before the framework has matured.

Developers experimenting with React Native benefit from an extensive documentation and profit from previous knowledge of ReactJS (D2 (Preparation Time)). Due to the highly component-centred architecture of React Native, apps can be easily subdivided, which improves development and long-term maintainability (D3 (Scalability), D4 (Development Process Fit), D9 (Maintainability)). Generally, this structure is favourable for including third-party extensions; yet, comparatively few of them exist (D10 (Extensibility)). On the other hand, the composition of components within the reactive programming paradigm complicates UI development and testing because app interactions are hard to simulate within and no visual editor supports the custom JSX notation (D5 (Ui Design), D6 (Testing), D7 (Continuous Delivery). Consequently, the advantages of using JavaScript cannot yet be fully utilized in terms of development speed, mainly because of the relatively recent introduction of the framework and supporting tools (D12 (Pace of Development)).

With regard to app capabilities, 34 APIs exist to access platform functionality and hardware features (A1, A2), although some cover only individual platforms. Supporting established JavaScript input / output events allows for a decent coverage of user interaction possibilities (A4 (Input Heterogeneity),A5 (Output Heterogeneity)). Due to its underlying native foundation, integrating custom native code is also possible (D11 (Custom Code Integration)) and third-party components provide Message APIs to interact with connected devices on a case-by-case basis, e.g., Wear OS smartwatches (A3 (Connected Devices)). The overall app life cycle as well as integration and security characteristics are comparable to PhoneGap apps with native components but for an additional layer of abstraction (A6 (App Life Cycle), A7 (System Integration), A8 (Security)). Similar to other JavaScript approaches, functionality and usage patterns need to be re-implemented manually and the novelty of the approach restricts the range of predefined components (U3 (Usage Patterns), U4 (User Authentication), A9 (Robustness)). However, the runtime approach provides additional flexibility for future development, especially creating connectors to map the React Native APIs to non-smartphone platforms and thus support a much larger range of devices compared to web views (A10 (Degree of Mobility)).

In the resulting apps, the native UI components achieve an appearance well-adapted to the target platform. Also, a comparably low performance overhead is induced by the runtime because it can delegate performance-heavy UI computations to the native environment (U1 (Look and Feel), U2 (Performance)).

7.5.5 Native Apps

Although technically not a cross-platform approach, it makes sense to compare frameworks with the baseline of native apps to highlights the potential of cross-platform app development. In particular, the native approach does not achieve a perfect score: Despite individual platform capabilities being fully exploitable, drawbacks result from the separate development of multiple apps. Moreover, development in the native languages and using the typical environment for native development is not necessarily more efficient; in fact, even the performance of a native app might be challenged by one that undergoes runtime optimization, as has been discussed[5] for desktop applications written in Java [Goe05]. Besides, using native development might feel clumsy compared to the *convention-over-configuration* approach typical for modern frameworks [VS11].

Obviously, the target platforms are limited to only one, whereas the app itself may be developed using various development platforms or technologies (I2 (Target Platforms), I3 (Development Platforms)). Using the freely distributed platform SDKs, a full integration of apps with the respective app store infrastructure can be achieved, including runtime interactions such as in-app payments and updates (I1 (License), I4 (Distribution Channels), I5 (Monetisation), I6 (Internationalisation)). Because iOS and Android have emerged as stable duopoly of smartphone operating system vendors, long-time reliability is ensured [Fv17]. Yet, even within such ecosystems, technological changes occur over time, e.g., introducing new programming languages such as Swift for iOS or Kotlin for Android (I7 (Long-term Feasibility)).

With regard to the development perspective, knowledge of these programming languages is mandated but in general, platform vendors provide detailed documentation of the APIs and best practices to the large community of developers (D1 (Development Environment), D2 (Preparation Time)). In addition, platform-adapted IDEs often support the development process through visual editors for UI design (D5) or suitable tools for testing, bundling, and deploying apps (D6 (Testing), D7 (Continuous Delivery)). On the other hand, this flexibility requires the developer to decide on appropriate processes to develop, scale, and maintain the app (D3 (Scalability), D4 (Development Process Fit), D9 (Maintainability)). For example, developers can use platform and app store features to develop multiple app versions such as regional language translations or theming, but the actual integration of provided low-level functionality needs to be performed by hand. As different programming languages and platform characteristics prohibit the reuse of code across multiple platforms, the redundant implementation of apps leads to a very inefficient pace of development (D11 (Custom Code Integration)).

Native apps can access all possible features of a given platform (A1-A7). Again, this flexibility is provided on a low level of platform interfaces and developers have the responsibility to adequately use the provided features. For example, platforms allow for a fine-grained control over app

[5] Benchmarking just-in-time compiled programs is an extremely hard endeavour [GBE07]. For native apps, research on runtime optimization is an open task.

permissions or network connectivity, yet it is up to the developers to exploit these capabilities and react to app-external changes of context. Similarly, the app state can be monitored to integrate typical usage patterns and provide a pleasant user experience (A8 (Security), A9 (Robustness), U3 (Usage Patterns), U4 (User Authentication)).

Finally, native app development ensures that visual appearance and user interactions can be fully aligned with the respective platform guidelines (U1 (Look and Feel)). At the same time, developers can achieve the best performance on computationally restricted mobile devices as overhead of cross-platform abstractions through frameworks or runtimes is avoided (U2 (Performance)).

7.5.6 Intermediate Conclusions

In this section, we have demonstrated how the proposed criteria catalogue can be put into practice by performing an evaluation study. While focussing on business apps for smartphones, we compared different development paradigms for mobile apps with the help of representative and widespread frameworks. More specific, native development for multiple platforms is contrasted to purely browser-based Web Apps and enhanced Progressive Web Apps using the Ionic Framework, hybrid apps with PhoneGap / Apache Cordova, and the runtime-based approach React Native.

For the given scenario, it can be concluded that native app development supports the widest set of features but comes at the cost of multiple redundant implementations. Though simple to develop, Web Apps do not provide an adequate solution for cross-platform apps – yet Progressive Web Apps enable more app-like behaviour and significantly reduce the drawbacks of in-browser apps on supported platforms. Hybrid app frameworks such as PhoneGap on the one hand outperform the previous approaches by combining the ease of web development using technologies such as JavaScript while at the same time creating installable apps with access to hardware sensors and platform features. On the other hand, native app appearance and performance are unobtainable and critically noticed by users [AHN16]. Finally, trending runtime-based frameworks such as React Native aim to solve the usage drawbacks of hybrid apps through native UI components while still being programmed using JavaScript. However, due to their novelty these frameworks have not yet established a large community to provide third-party libraries and equivalent coverage of functionality.

We want to stress that this evaluation is largely influenced by the scenario-specific weight profile and based on the specific frameworks' capabilities. Changing the weights would arguably lead to different recommendations. In fact, having no eminent *winner* of the evaluation underlines the benefits that situation-specific weights provide. This also has implications for research and practice: Scientific assessment of cross-platform app development is far from being complete. At the same time, practitioners are undoubtedly provided with a choice of *good* option; finding the optimal is strongly dependent on context. We recommend to do at least individual weighting, if feasible also additional assessment.

Moreover, the evaluation must be seen in the light of ongoing developments. For example, novel frameworks such as React Native might soon be on a par with long-standing frameworks when the community of developers grows and openly shares components for reuse in other projects. Also, target platforms evolve such that PWAs might be better supported in the future; thus further blurring the lines between apps and the Web. Our evaluation study should therefore rather be seen as a present-day snapshot of the mobile app development landscape instead of a *final* assessment. We by no means claim to provide a definite ranking of frameworks. On the contrary, such universal evidence does not exist but strongly relies on assumptions derived from the application domain, company guidelines, team members' experience, and project-specific requirements.

In order to provide the necessary flexibility for customised evaluations, the weight profiles can be used to tailor the comprehensive list of criteria to the respective use case as described in Section 7.4. Moreover, our catalogue of criteria can be applied to different classes of mobile devices and considers the varying importance of specific criteria. Consequently, this work can be used by practitioners as well as researchers for systematically selecting a suitable cross-platform development framework.

7.6 Discussion

Although most feedback by the experts consulted to examine (a prior iteration of) our catalogue is already incorporated in the presented set of criteria, some overarching topics are discussed in the following. In addition, we revisit literature gaps, limitations, and implications on further research.

7.6.1 Assessment

To validate our evaluation framework and the criteria of the catalogue, we contacted several experts from academia and practice with experience in the field of mobile app development to scrutinize the approach. More specific, open-ended questions were asked to assess the following requirements for each criterion[6]:

- **Comprehensibility**: Are descriptions clear even for people without cross-platform proficiency?
- **Unambiguousness**: Is it impossible to wrongly interpret criteria or the way how frameworks should be assessed according to them?
- **Adequacy**: Do the criteria really cover important aspects for selecting a framework?
- **Completeness**: Are the criteria collectively exhaustive and cover all relevant aspects of cross-platform framework selection?

[6]We roughly follow typical assessment criteria used in requirements engineering [Som11, p. 94]. These are handy for cases in which a comprehensive specification is desired.

- **Consistency**: Are descriptions free from contradictions? Are criteria mutually exclusive?
- **Verifiability**: Can the criteria realistically be used to assess cross-platform development frameworks? Are the descriptions specific enough to be operationalisable?

Additional questions dealt with the following overarching aspects:

- Are all criteria operationalisable? This includes whether we realistically describe them and whether they are not too generic.
- Do our criteria align with the "business routine" in software development? In particular, do they fit with different development methodologies (e.g., agile vs. traditional) and paradigms? And does the catalogue reflect app development practices in teams (e.g., team size, roles etc.)?
- Besides smartphones and tablets, does our catalogue sufficiently cover requirements towards app development for novel and future app-enabled devices such as smart watches, cars, smart glasses, VR devices etc.? If not, which changes or even which new criteria are needed?
- Does the catalogue constitute a good balance between practical relevance and rigorous work on the criteria?

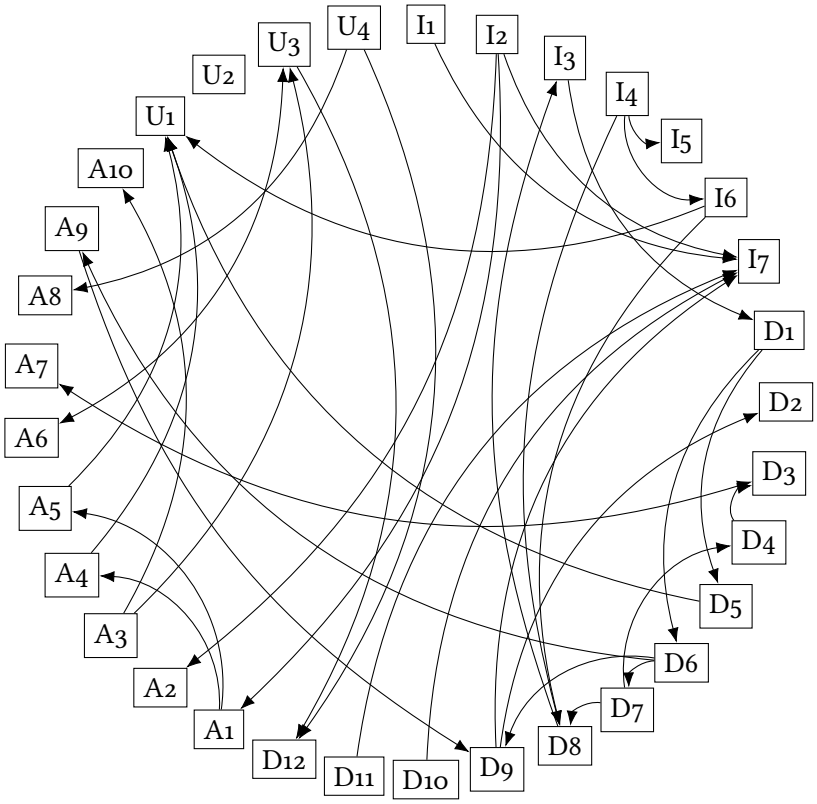
Overall, we received very positive feedback on the criteria we deem important for evaluating cross-platform frameworks. The feedback has resulted in several clarifications to the criteria descriptions and are incorporated in the criteria catalogue presented in Section 7.3. The experts noted that particularly for rather generic criteria such as D12 (Pace of Development) disjunction with other criteria needs to be scrutinized. Undoubtedly, full independence of criteria is unobtainable even if direct overlaps are avoided to the degree achieved by us; in the example of D12, its relationship to D2 (Preparation Time) is evident. Further influences between criteria in the presented catalogue are visualized in Figure 7.1.

The directed edges in Figure 7.1 can be read as "having an *inherent* influence", not as "having a *constituting* influence". This means that no criterion is a superordinate or marks a kind of prerequisite for another one. Rather, some criteria share characteristics that despite all strive for cohesion are not independent in typical settings in practice.

Consider, e.g., (I2) Supported Target Platforms. The supported platforms, and the way they are supported for example regarding different versions, cannot fully be untied from the (I7) Long-term Feasibility. Undoubtedly, unsatisfying platform support will hardly go along with a good long-term feasibility. However, the other way around, very good platform support does not mandate long-term feasibility, which is dependent on many other factors. Additionally, (A1) Access to Device-specific Hardware and (A2) Access to Platform-specific Functionality are influenced by I2. The worse the platform support is, the less likely it is to meet satisfying levels of hardware and platform access. Again, this is a unidirectional relationship.

Explicating such interdependencies should make it easier to assess and advance our framework. For practical usage, there should hardly be any consequences. However, with any future changes to the framework keeping the high cohesion of criteria needs to be an explicit aim.

Figure 7.1: Visualisation of Main Dependencies Among Evaluation Criteria



Cross-platform app development for mobile devices extends beyond "traditional" smartphones and tablets. With novel classes of mobile devices such as smartwatches and smart TVs reaching a more widespread adoption and further ones such as connected cars, smart glasses, and augmented / virtual reality devices foreseeable [RM18], new apps might be developed for multiple platforms within such a device class or even bridging different heterogeneous devices. Therefore, our experts were further asked whether the existing set of criteria satisfies the additional requirements or what changes need to be made. Although none of the interviewed experts had previous practical experience with novel app-enabled devices from a developer perspective, to their perception the presented catalogue is apt to these new challenges.

The applicability of our criteria catalogue for real projects is an important consideration to assess whether we have managed to achieve a good balance between practical relevance and rigorous work on the criteria. In addition, the criteria catalogue needs to align with typical business routines. For example, development methodologies such as agile approaches or typical team structures such as team size and role distribution influence the ability to follow the approach presented in this work. Responses indicate that our catalogue is well applicable and can easily be tailored to a project-specific configuration. Not all criteria are equally important; therefore, the weighting approach presents a suitable solution to filter and prioritize the comprehensive set of possible criteria and adapt it to company- or project-specific requirements. One expert

stated it would be helpful to have an additional checklist in order to simplify the assessment of the score for individual criteria. This was also backed by another expert who expressed concerns that some criteria require much effort for assessment (such as the robustness, A9) or are hard to verify (such as the performance, U2). We are aware of the practical benefits of a checklist; however, the fast-moving mobile domain would require a constantly updated list of current and upcoming features for a broad set of frameworks. Instead, the list of criteria relevant for cross-platform frameworks has stabilized over time and is a long-term contribution to the field of research. Moreover, we have noted when project-specific requirements should be considered.

7.6.2 Ongoing Demand for Research

The comprehensive literature search that serves as foundation for our criteria catalogue revealed five gaps that call for further research in the field of assessing mobile cross-platform frameworks.

First, the selection of criteria often appears to be chosen in an ad-hoc manner and rarely covers all four perspectives nor an exhaustive list of criteria within one specific perspective. For instance, [Rd12] focus on the development and app perspectives but do not mention essential criteria such as the effort to set up (D2) and continuously maintain (D9) developed apps. Our weighting approach aims to guide the selection process by focusing on important criteria from the *large* catalogue. However, how to create adequate weight profiles or setting custom weights are still open questions. We have provided the tools but how to use them most effectively needs to be identified. On the one hand, the domain of (novel) mobile devices still evolves quickly and many new devices are proposed. The identified device classes might therefore change in the future and evolutionary effects of convergence (e.g., fitness devices and smartwatches share commonalities such as health tracking capabilities) or divergence (e.g., augmented and virtual reality devices might develop characteristic features when more devices appear) might be observed. On the other hand, the chosen weights in this work are convincingly argued but need to be validated through real-world projects and are open for revisions. Because of the limited availability of case studies on novel mobile devices, the overall prioritisation of criteria for different device classes needs further research (nevertheless, project-specific adaptations are always possible).

Second, a need for common benchmarks arises from the multitude of individual and incomparable evaluation studies. Researches put significant effort in the assessment of cross-platform frameworks, yet various parameters limit the comparability of their results:

- **Features**: Various software and hardware features can be considered for framework comparisons. No uniform set of features that are expected has been described.
- **Metrics**: Measuring the performance of feature usage is complex. Sensor modes (e.g., network- or hardware-based location detection), data retrieval methods (getting the last known value vs. waiting for the next update), and the amount of system- or app-specific setup instructions (event listeners, etc.) have a great impact on the outcome when measuring

execution times and resource utilization. However, benchmarks with detailed specification of constituent metrics are usually omitted and thus limit reproducibility.

- **Platforms**: Commonly, the distinction is made between the two major operating systems Android and iOS but not all studies cover both (e.g., [QGZ17]) such that results cannot be transferred. Moreover, different versions of each may qualify as distinct platforms if the internal (e.g., performance improvements by using different browser engines) or visible characteristics (e.g., the introduction of material design affecting usability) have significantly changed [El-+17].
- **Devices**: Emulated and physical devices exhibit a large variety of hardware characteristics which affect the outcome. In addition, current settings such as energy modes as well as contextual influences (e.g., network connectivity) or background tasks influence the application behaviour.
- **Framework architecture**: The underlying architecture of a framework can further complicate the comparison. For example, the performance of traditional and reactive handling of UI components is hard to assess objectively and frameworks may provide custom events for accessing sensors such as the camera which influences available metrics.

As a result, studies produce vastly different – sometimes surprising or contradicting – results, even for quantitative and seemingly objective performance measures. For instance, surprisingly slow execution times for NativeScript's video playback feature observed by [Cor+18] cannot clearly be attributed to the device, the platform version, or the framework itself for lack of repetition. Also, studies tend to either specialize on a large set of criteria and a limited set of evaluated frameworks (e.g., [QGZ17]) or vice versa (e.g., [UB16]). Given the rapid emergence of new frameworks and the degeneration into insignificance of others, a stable set of common, repeatable benchmarks would be beneficial to enable more comparable results among different evaluation studies. For example, energy consumption is well researched for established frameworks such as PhoneGap, Sencha Touch, and Titanium [CG17; Dal+13] but there is no simple solution to compare these results with a measurement using a recent framework such as React Native. Automating the testing procedure to compare a large variety of devices with different platform versions as, e.g., performed by [QGZ17] is a worthwhile step towards this aim. However, the set of features and procedures to build such a benchmark suite need to be further researched. Also, it is not obvious whether an isolated examination of hardware or software features is sufficient or whether (and which) *app scenarios* (e.g., [BMG17]) are better suited to assess a framework under realistic conditions.

Third, frameworks are nowadays designed to provide similar applications for different platforms and thus different users. With the advent of wearables and other app-enabled devices, app ecosystems are likely to change. For example, the same user might own multiple devices and use them in combination or subsequently depending on personal preferences [RM16]. Frameworks can possibly adapt to this by reusing source code of one existing app [El-+17], applying multi-level code generation [UB16; RK19], or using other techniques related to the evolution of cross-platform

software. Evaluating a framework for such a usage scenario is, however, barely considered and subject to further research. In addition, research on usability and UI design needs to scrutinize whether it is desirable that the interaction with several design classes should converge. Dependent apps for different types of devices that are unified as much as possible while keeping traits specific to (and desired for) a class would make a class-spanning cross-platform development framework even more complex than it already is. At the same time, a framework that provides such functionality would dramatically simplify developers' life.

Besides the user-centred aspects, one expert also suggested that system integration (A7) might become (even) more important in the future, possibly even suggesting to split up the criterion. While using cloud services is essential for many apps already, the embedding of them in whole ecosystems of services and applications will likely require more attention with the expected convergence of mobile computing and IoT. When a multitude of devices provide functionality from the same app ecosystem, aspects of re-use, partitioning and distribution could also require more attention.

Finally, whereas security and data privacy issues are important especially in a business context, studies on mobile security suggest that current apps created using cross-platform tools are commonly not designed with security in mind. For example, an analysis on Apache Cordova apps by [WVN17] revealed that best practices for the framework are mostly ignored. Integrating and evaluating the security of cross-platform apps is another challenge that has not been studied systematically and mandates further research.

7.6.3 Limitations

Although we deem our evaluation framework theoretically sound and usable in practice, opportunities for further research exist. Being based on existing literature and validated by experts with cross-platform experience, the criteria catalogue has a solid foundation and is expected to provide a comprehensive and rather stable set of criteria. However, the technological evolution of app-enabled devices and platforms is hard to foresee.

For example, Tesla initially announced an own SDK but reconsidered this approach due to security concerns [Lam16]. New kinds of devices are designed with a focus on app-enablement – or not. Regarding platform evolution, Wear OS [Goo18j] *might* unify development for wearables or at least consolidate different streams. Alternatively, ecosystems such as the Universal Windows Platform [Mic17] might establish themselves as integrated platforms that simplify the development for multiple devices. Also, it remains to be seen whether the success of Web technology and plethora of JavaScript frameworks for bridging the heterogeneity of devices will extend towards new device classes. So-called *instant apps* can be run without installation on smartphones [Gan16] and might also contribute to future changes. Yet for now, most novel mobile devices do not support JavaScript engines such as WebKit (e.g., due to hardware constraints) and require different approaches to achieve cross-platform compatibility.

Therefore, while the criteria catalogue can be expected to be relatively stable, keeping up with the technological progress will remain an inherent limitation of any work on assessing mobile computing technology.

Regarding the established weight profiles, limitations have already been mentioned in Subsection 7.6.2. More research is needed to gain empirical insights in suitable weights. Moreover, it would be utile to provide better support in criteria-specific evaluations. For examples, what would be the expected interval in which measurements are to be found for quantitative criteria? What would be sufficient expectations for qualitative ones? How would this be transferred to the score, and how could it be ensured that progress is reflected in these scores? Moreover, which ranges of scores could be expected in general, and which scores for assessment are sufficient and realistic? We have put these questions as boundaries rather than as tasks for future research for now, as answering them in a generalizable manner, aiming at the same soundness as our criteria catalogue as such, will be next to impossible.

On a more general level, it needs to be questioned whether full ecosystems similar to the current situation for smartphones will emerge for all device classes. The multitude of upcoming devices makes cross-platform development much more difficult, especially when considering the interrelations of multiple devices used in combination by the same user. A cloud-based middleware, mirroring techniques, or other "remote" approaches could solve issues such as low performance, hardware heterogeneity, and security without even relying on device-installed apps directly [Gal+16; KK16]. Nonetheless, the criteria catalogue established in this work is flexible enough to deal with a wide range of technological bases which enable cross-platform capabilities.

7.6.4 Future Work

Although we are confident that we have reached the goal of providing not *another* but the definitive framework for evaluation of cross-platform app development approaches, we will continue with our work on related topics.

One of the experts suggested that for better operationalisation of own assessments, it would be helpful to have a per-criteria checklist. Remember for instance criterion I2 (Target Platforms). The checklist could comprise Android, iOS, and Windows Mobile for smartphones. However, whether it should not contain more operating systems would be a matter of discussion, as would be how the checklist should be kept up to date. Moreover, if you think of other criteria, it can be extremely hard to propose a relatively simple list. For example, D4 (Development Process Fit) can hardly be broken down into single items that can be checked (or not). Thus, we deem research on such checklists, and in general on assessment advice (How to do it? Single choice, multiple choice, multiple select?) a target of our future work.

Ongoing research concentrates on engineering cross-platform frameworks for novel mobile devices and approaches for developing apps across multiple device classes. Besides numerous technical considerations, the prevalent conceptual challenges relate to suitable abstractions for

developing apps for devices with heterogeneous input and output capabilities as well as different capabilities. Also, multi-device interactions become important when using different devices sequentially or concurrently depending on personal preferences or usage context. In the domain of data-driven business apps, model-driven approaches such as MAML [RK18a; Rie18] and MD^2 [HMK13] with high levels of abstraction are promising for extending them towards new device classes such as smartwatches. However, it is still very hard to image proper abstractions for different domains such as home automation apps for devices that fall under the umbrella of smart home technology.

In addition to the two concrete topics, we will continue with our work on building the theory of modern mobile computing. This quest will be built upon a close scrutinisation of work in practice and strive to provide rigour where fast-pace developments predominate.

7.7 Conclusion

In this article, we have proposed an evaluation framework for cross-platform app development approaches. Building on previous frameworks and being broken down into sound abstract criteria, it sets out to be *the* way of assessing cross-platform development frameworks for all situations in which app-enabled devices play a role.

We have shown that much literature exists yet few works have gone the lengths of providing comprehensive, holistic frameworks. Moreover, the complexity introduced by novel mobile devices is reflected in many works yet needed to be grasped and built into assessment criteria. We have provided and, in much detail, explained our criteria catalogue, which provides a synthesis of the literature. Criteria are grouped into four perspectives: infrastructure, development, app, and usage. To provide means for a tailored, customised assessment, we have proposed the usage of weight profiles. With the help of these, a criteria-based assessment can be adapted to whatever situation is concretely met. In fact, one assessment can even be used to cater for multiple decisions.

We have demonstrated the feasibility of our framework with an evaluation study. This study has been done by applying the criteria catalogue to several app development approaches, namely (Progressive) Web Apps, PhoneGap, React Native, and native apps for comparison. The framework has been proven handy. Moreover, we have provided exemplary weight profiles. Our results have been assessed by several experts from the field of mobile computing.

We have identified much need for future research, most notably regarding the weighting, the understanding of the technological progress, and the emergence of device ecosystems. We will continue to work on solving these challenges. Eventually, we hope to be able to provide further synthesis articles such as this. The theory on modern mobile computing ought to be extended!

Acknowledgements

We would like to thank the experts who contributed to the assessment of our criteria catalogue. Detailed feedback was received from

- Henning Heitkötter, SAP SE
- Gregor Kurpiel, itemis AG
- Spyridon Xanthopoulos, University of Macedonia

Moreover, we would like to thank the three anonymous JSS reviewer who pointed out several options for improving our work.

References

[Ado15] Adobe Systems Inc. *PhoneGap Documentation*. 2015. URL: http://docs.phonegap.com (visited on 09/12/2018).

[AF14] Esteban Angulo and Xavier Ferre. "A Case Study on Cross-Platform Development Frameworks for Mobile Applications and UX". In: *Proceedings of the XV International Conference on Human Computer Interaction*. Interacción '14. Puerto de la Cruz, Tenerife, Spain: ACM, 2014, 27:1–27:8. DOI: 10.1145/2662253.2662280.

[AGB12] Morten Andersen-Gott, Gheorghita Ghinea, and Bendik Bygstad. "Why do commercial companies contribute to open source software?" In: *International Journal of Information Management* 32.2 (2012), pp. 106–117. DOI: 10.1016/j.ijinfomgt.2011.10.003.

[AHN16] V. Ahti, S. Hyrynsalmi, and O. Nevalainen. "An evaluation framework for cross-platform mobile app development tools: A case analysis of Adobe PhoneGap framework". In: *ACM International Conference Proceeding Series* 1164 (2016). DOI: 10.1145/2983468.2983484.

[Aho15] Jukka Ahola. "Challenges in Android Wear Application Development". In: *Engineering the Web in the Big Data Era*. Ed. by Philipp Cimiano et al. Cham: Springer International Publishing, 2015, pp. 601–604.

[AK14] Suyesh Amatya and Arianit Kurti. "Cross-Platform Mobile Development: Challenges and Opportunities". In: *ICT Innovations 2013*. Vol. 231. Springer, 2014, pp. 219–229.

[Ala+17] Mussab Alaa et al. "A review of smart home applications based on Internet of Things". In: *Journal of Network and Computer Applications* 97 (2017), pp. 48–65. DOI: 10.1016/j.jnca.2017.08.017.

[Alh+14] Kati Alha et al. "Free-to-play games: Professionals' perspectives". In: *Proceedings of nordic DiGRA* 2014 (2014).

[Apa15] Apache Software Foundation. *Apache Cordova Documentation*. 2015. URL: https://cordova.apache.org/docs/en/ (visited on 09/12/2018).

[App18a] Apple Inc. *iOS Human Interface Guidelines*. 2018. URL: https://developer.apple.com/design/human-interface-guidelines/ios/ (visited on 09/05/2018).

[App18b] Apple Inc. *Wallet - Apple Developer*. 2018. URL: https://developer.apple.com/wallet/ (visited on 09/11/2018).

[Arp+16] Daniel Arp et al. "Bat in the Mobile: A Study on Ultrasonic Device Tracking. Computer Science Report 2016-02". In: *Institute of System Security, Technische Universität Braunschweig* (2016), pp. 1–31.

[Aut18] Automatic Labs. *Automatic: Connect Your Car to Your Digital Life*. 2018. URL: https://automatic.com/ (visited on 09/10/2018).

[BB57] George M. Beal and Joe M. Bohlen. *The diffusion process*. Agricultural Experiment Station, Iowa State College, 1957.

[BB96] F Hutton Barron and Bruce E Barrett. "Decision quality using ranked attribute weights". In: *Management science* 42.11 (1996), pp. 1515–1523.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[BEP16] Federico Botella, Pedro Escribano, and Antonio Peñalver. "Selecting the Best Mobile Framework for Developing Web and Hybrid Mobile Apps". In: *Proceedings of the XVII International Conference on Human Computer Interaction*. Interacción '16. New York, NY, USA: ACM, 2016, 40:1–40:4. DOI: 10.1145/2998626.2998648.

[BG18] Andreas Biørn-Hansen and Gheorghita Ghinea. "Bridging the Gap: Investigating Device-Feature Exposure in Cross-Platform Development". In: *Proceedings of the 51st Hawaii International Conference on System Sciences*. ScholarSpace, 2018, pp. 5717–5724.

[Bis06] Judith Bishop. "Multi-platform User Interface Construction: A Challenge for Software Engineering-in-the-small". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. ACM, 2006, pp. 751–760. DOI: 10.1145/1134285.1134404.

[BMG17] Andreas Biørn-Hansen, Tim A. Majchrzak, and Tor-Morten Grønli. "Progressive Web Apps: The Possible Web-Native Unifier for Mobile Development". In: *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST)*. INSTICC. SciTePress, 2017, pp. 344–351. DOI: 10.5220/0006353703440351.

[BMG18] Andreas Biørn-Hansen, Tim A. Majchrzak, and Tor-Morten Grønli. "Progressive Web Apps for the Unified Development of Mobile Applications". In: *Revised Selected Papers WEBIST 2017*. Ed. by Tim A. Majchrzak et al. Vol. 322. Lecture Notes in Business Information Processing (LNBIP). Springer, 2018, pp. 64–86.

[Bou15] Gil Bouhnick. *A List of All Operating Systems Running on Smartwatches*. 2015. URL: http://www.mobilespoon.net/2015/03/a-list-of-all-operating-systems-running.html (visited on 09/12/2018).

[Bri+17] Mitch A Brisebois et al. *Method and system for customizing a mobile application using a web-based interface*. US Patent 9,836,446. Dec. 2017.

[CAB15] S. Charkaoui, Z. Adraoui, and E. H. Benlahmar. "Cross-platform mobile development approaches". In: *Colloquium in Information Science and Technology, CIST* 2015-January.January (2015). DOI: 10.1109/CIST.2014.7016616.

[Car15] Jamie Carter. *Which is the best Internet of Things platform?* 2015. URL: http://www.techradar.com/news/-1302416 (visited on 05/31/2016).

[CG15] M. Ciman and O. Gaggi. "Measuring energy consumption of cross-platform frameworks for mobile applications". In: *LNBIP* 226 (2015), pp. 331–346. DOI: 10.1007/978-3-319-27030-2_21.

[CG17] Matteo Ciman and Ombretta Gaggi. "An empirical analysis of energy consumption of cross-platform frameworks for mobile development". In: *Pervasive and Mobile Computing* (Oct. 2017). DOI: 10.1016/j.pmcj.2016.10.004.

[CGG14] M. Ciman, O. Gaggi, and N. Gonzo. "Cross-platform mobile development: A study on apps with animations". In: *Proc. ACM Symposium on Applied Computing*. 2014. DOI: 10.1145/2554850.2555104.

[Chm13] J. Chmielewski. "Towards an architecture for future internet applications". In: *Lecture Notes in Computer Science* 7858 (2013), pp. 214–219. DOI: 10.1007/978-3-642-38082-2_18.

[Chu+12] Byung-Gon Chun et al. "Mobius: Unified messaging and data serving for mobile apps". In: *Proceedings of the 10th international conference on Mobile systems, applications, and services - MobiSys '12*. Ed. by Nigel Davies, Srinivasan Seshan, and Lin Zhong. New York, New York, USA: ACM Press, 2012, pp. 141–154. DOI: 10.1145/2307636.2307650.

[Cor+18] Leonardo Corbalan et al. "Development Frameworks for Mobile Devices: A Comparative Study About Energy Consumption". In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '18. New York, NY, USA: ACM, 2018, pp. 191–201. DOI: 10.1145/3197231.3197242.

[Dal+13] I. Dalmasso et al. "Survey, comparison and evaluation of cross platform mobile application development tools". In: *Proc. 9th IWCMC*. 2013. DOI: 10.1109/IWCMC.2013.6583580.

[Das18] Dash Labs, Inc. *Dash - Smarter Driving, Every Day*. 2018. URL: https://dash.by/ (visited on 09/10/2018).

[Dav09] Lidija Davis. *PhoneGap: People's Choice Winner at Web 2.0 Expo Launch Pad.* 2009. URL: http://readwrite.com/2009/04/02/phone_gap (visited on 05/31/2016).

[DC14] Sunil G. Dewan and Lei-da Chen. "Mobile Payment Adoption in the US: A Cross-industry, Crossplatform Solution". In: *Journal of Information Privacy and Security* 1.2 (2014), pp. 4–28. DOI: 10.1080/15536548.2005.10855765.

[Del+18] L. Delia et al. "Approaches to mobile application development: Comparative performance analysis". English. In: *Proc. Computing Conference 2017*. Vol. 2018-January. Institute of Electrical and Electronics Engineers Inc., 2018, pp. 652–659. DOI: 10.1109/SAI.2017.8252165.

[DHH13] S. Durach, U. Higgen, and M. Huebler. "Smart automotive apps: An approach to context-driven applications". In: *LNEE* 200.VOL. 12 (2013), pp. 187–195. DOI: 10.1007/978-3-642-33838-0-17.

[DM12] Sebastian Dyck and Tim A. Majchrzak. "Identifying Common Characteristics in Fundamental, Integrated, and Agile Software Development Methodologies". In: *Proc. 45th Hawaii International Conference on Systems Science (HICSS-45)*. IEEE Computer Society, 2012, pp. 5299–5308.

[DM15] S. Dhillon and Q. H. Mahmoud. "An evaluation framework for cross-platform mobile application development tools". In: *Software – Prac. and Exp.* 45.10 (2015), pp. 1331–1357. DOI: 10.1002/spe.2286.

[Dob12] Alex Dobie. *Why you'll never have the latest version of Android.* 2012. URL: http://www.androidcentral.com/why-you-ll-never-have-latest-version-android (visited on 09/12/2018).

[Don08] Jonathan Donner. "Research approaches to mobile use in the developing world: A review of the literature". In: *The information society* 24.3 (2008), pp. 140–159.

[Dor18] Tim Dorr. *Tesla Model S JSON API.* 2018. URL: http://docs.timdorr.apiary.io (visited on 09/12/2018).

[Dou15] Adam Doud. *How important is cross-platform wearable support?* 2015. URL: http://pocketnow.com/2015/05/10/cross-platform-wearable-support (visited on 05/31/2016).

[DRB15] M. Deindl, M. Roscher, and M. Birkmeier. "An architecture vision for an open service cloud for the smart car". In: *Green Energy and Technology* 203 (2015), pp. 281–295. DOI: 10.1007/978-3-319-13194-8_15.

[El-+14] W. S. El-Kassas et al. "ICPMD: Integrated cross-platform mobile development solution". In: *Proc. 9th ICCES*. 2014. DOI: 10.1109/ICCES.2014.7030977.

[El-+17] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* 8.2 (2017), pp. 163–190. DOI: 10.1016/j.asej.2015.08.004.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *Proc. of the 12th WEBIST*. SciTePress, 2016, pp. 307–316.

[EVP01] J. Eisenstein, J. Vanderdonckt, and A. Puerta. "Applying model-based techniques to the development of UIs for mobile computers". In: *International Conference on Intelligent User Interfaces, Proceedings* (2001).

[Fac18] Facebook Inc. *React Native - A framework for building native apps using React*. https://facebook.github.io/react-native/. 2018.

[Fan+14] R. Fantacci et al. "Short paper: Overcoming IoT fragmentation through standard gateway architecture". In: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. Vol. 00. Mar. 2014, pp. 181–182. DOI: 10.1109/WF-IoT.2014.6803149.

[Fer+18] C.M.S. Ferreira et al. "An evaluation of cross-platform frameworks for multimedia mobile applications development". Portuguese. In: *IEEE Latin America Transactions* 16.4 (2018), pp. 1206–1212. DOI: 10.1109/TLA.2018.8362158.

[Fit06] Brian Fitzgerald. "The Transformation of Open Source Software". In: *MIS quarterly* 30.3 (2006), pp. 587–598.

[Fri14] Peter Friese. *applause*. 2014. URL: https://github.com/applause/.

[Fv17] Amy Ann Forni and Rob van der Meulen. *Gartner Says Worldwide Sales of Smartphones Grew 9 Percent in First Quarter of 2017*. 2017. URL: https://www.gartner.com/newsroom/id/3725117.

[Gal+16] A. Gallidabino et al. "On the Architecture of Liquid Software: Technology Alternatives and Design Space". In: *WICSA*. 2016, pp. 122–127. DOI: 10.1109/WICSA.2016.14.

[Gan16] Suresh Ganapathy. *Introducing Android Instant Apps*. 2016. URL: http://android-developers.blogspot.no/2016/05/android-instant-apps-evolving-apps.html.

[Gar18] Trent Gardner. *Android Wear Cordova Plugin*. 2018. URL: https://github.com/tgardner/cordova-androidwear (visited on 09/05/2018).

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *SIGPLAN Not.* 42.10 (Oct. 2007), pp. 57–76. DOI: 10.1145/1297105.1297033.

[GK91] G. K. Gill and C. F. Kemerer. "Cyclomatic complexity density and software maintenance productivity". In: *IEEE Transactions on Software Engineering* 17.12 (Dec. 1991), pp. 1284–1288. DOI: 10.1109/32.106988.

[Goe05] Brian Goetz. *Urban performance legends, revisited.* https://www.ibm.com/developerworks/library/j-jtp09275/. 2005.

[Goo18a] Google LLC. *Android Developers - design for Android.* 2018. URL: https://developer.android.com/design/ (visited on 09/05/2018).

[Goo18b] Google LLC. *Android TV.* 2018. URL: https://www.android.com/tv/ (visited on 05/31/2016).

[Goo18c] Google LLC. *Behavior changes: all apps.* 2018. URL: https://developer.android.com/about/versions/pie/android-9.0-changes-all (visited on 09/10/2018).

[Goo18d] Google LLC. *Flutter - beautiful native apps in record time.* 2018. URL: https://flutter.io/ (visited on 09/05/2018).

[Goo18e] Google LLC. *Google AdMob.* 2018. URL: https://www.google.com/admob/ (visited on 09/05/2018).

[Goo18f] Google LLC. *J2ObjC.* 2018. URL: http://j2objc.org/ (visited on 09/05/2018).

[Goo18g] Google LLC. *Progressive Web Apps.* 2018. URL: https://developers.google.com/web/progressive-web-apps/ (visited on 04/03/2018).

[Goo18h] Google LLC. *Requesting permissions at runtime.* 2018. URL: https://developer.android.com/training/permissions/requesting.html (visited on 01/14/2018).

[Goo18i] Google LLC. *Understand the Activity Lifecycle.* 2018. URL: https://developer.android.com/guide/components/activities/activity-lifecycle (visited on 09/13/2018).

[Goo18j] Google LLC. *Wear OS - Android Developers.* 2018. URL: https://developer.android.com/wear/index.html.

[Hbb18] HbbTV. *HbbTV Overview.* 2018. URL: https://www.hbbtv.org/overview/ (visited on 09/12/2018).

[HC16] Kuo-Lun Hsiao and Chia-Chen Chen. "What drives in-app purchase intention for mobile games? An examination of perceived values and loyalty". In: *Electronic commerce research and applications* 16 (2016), pp. 18–29.

[HEE13] S.R. Humayoun, S. Ehrhart, and A. Ebert. "Developing mobile apps using cross-platform frameworks: A case study". English. In: *Lecture Notes in Computer Science* 8004 LNCS.PART 1 (2013), pp. 371–380. DOI: 10.1007/978-3-642-39232-0_41.

[Hei+14] Henning Heitkötter et al. "Comparison of Mobile Web Frameworks". In: *LNBIP*. Vol. 189. Springer, 2014, pp. 119–137.

[HHH15] A. Hudli, S. Hudli, and R. Hudli. "An evaluation framework for selection of mobile app development platform". In: *Proc. 3rd MobileDeLi*. 2015. DOI: 10.1145/2846661.2846678.

[HHM12] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. "Comparing Cross-platform Development Approaches for Mobile Applications". In: *Proceedings 8th WEBIST*. SciTePress, 2012, pp. 299–311.

[HHM13] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. "Evaluating Cross-Platform Development Approaches for Mobile Applications". In: *LNBIP*. Vol. 140. Springer, 2013, pp. 120–138.

[HKM15] Henning Heitkötter, Herbert Kuchen, and Tim A. Majchrzak. "Extending a model-driven cross-platform development approach for business apps". In: *Science of Computer Programming* 97, Part 1.0 (2015), pp. 31–36. DOI: 10.1016/j.scico.2013.11.013.

[HM13] Henning Heitkötter and Tim A. Majchrzak. "Cross-Platform Development of Business Apps with MD2". In: *DESRIST*. Ed. by Jan vom Brocke et al. Vol. 7939. Lecture Notes in Computer Science. Springer, 2013, pp. 405–411. DOI: 10.1007/978-3-642-38827-9_29.

[HMK13] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. "Cross-platform model-driven development of mobile applications with MD2". In: *Proc. SAC '13*. ACM, 2013, pp. 526–533.

[Hob80] Benjamin F Hobbs. "A comparison of weighting methods in power plant siting". In: *Decision Sciences* 11.4 (1980), pp. 725–737.

[Hor16] David Horsley. *Beyond Touch: Tomorrow's Devices Will Use MEMS Ultrasound to Hear Your Gestures*. 2016. URL: https://spectrum.ieee.org/semiconductors/devices/beyond-touch-tomorrows-devices-will-use-mems-ultrasound-to-hear-your-gestures.

[Ion18] Ionic. *Build amazing native apps and progressive web apps with Ionic Framework and Angular*. 2018. URL: https://ionicframework.com/ (visited on 04/03/2018).

[Jac99] Ivar Jacobson. *The unified software development process*. Pearson Education India, 1999.

[Jak13] Ben Jakuben. *Why Developing Apps for Android is Fun*. 2013. URL: http://blog.teamtreehouse.com/why-developing-apps-for-android-is-fun.

[JB13] Slinger Jansen and Ewoud Bloemendal. "Defining App Stores: The Role of Curated Marketplaces in Software Ecosystems". In: *Software Business. From Physical Products to Software Services and Solutions. ICSOB 2013*. Ed. by Georg Herzwurm and Tiziana Margaria. Vol. 150. Lecture Notes in Business Information Processing. Springer, 2013, pp. 195–206. DOI: 10.1007/978-3-642-39336-5_19.

[JET18] Xiaoping Jia, Aline Ebone, and Yongshan Tan. "A Performance Evaluation of Cross-platform Mobile Application Development Approaches". In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '18. New York, NY, USA: ACM, 2018, pp. 92–93. DOI: 10.1145/3197231.3197252.

[Jie+15] G. Jie et al. "Cross-Platform Android/iOS-Based Smart Switch Control Middleware in a Digital Home". In: *Mobile Information Systems* 2015 (2015). DOI: 10.1155/2015/627859.

[Kel+12] Patrick Gage Kelley et al. "A Conundrum of Permissions: Installing Applications on an Android Smartphone". In: *Financial Cryptography and Data Security: FC 2012 Workshops, USEC and WECSR*. Ed. by Jim Blyth, Sven Dietrich, and L. Jean Camp. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 68–79. DOI: 10.1007/978-3-642-34638-5_6.

[Kim+16] H. Kim et al. "Wearable device control platform technology for network application development". In: *Mobile Information Systems* 2016 (2016). DOI: 10.1155/2016/3038515.

[KK16] István Koren and Ralf Klamma. "The Direwolf Inside You: End User Development for Heterogeneous Web of Things Appliances". In: *Proceedings 16th International Conference on Web Engineering (ICWE)*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Cham: Springer International Publishing, 2016, pp. 484–491. DOI: 10.1007/978-3-319-38791-8_35. URL: http://dx.doi.org/10.1007/978-3-319-38791-8_35.

[KS05] Rex Bryan Kline and Ahmed Seffah. "Evaluation of integrated software development environments: Challenges and results from three empirical studies". In: *International Journal of Human-Computer Studies* 63.6 (2005), pp. 607–627. DOI: 10.1016/j.ijhcs.2005.05.002.

[Kun+14] Michael Kunz et al. "Analyzing Recent Trends in Enterprise Identity Management". In: *25th International Workshop on Database and Expert Systems Applications*. 2014, pp. 273–277. DOI: 10.1109/DEXA.2014.62.

[LA17] Mohamed Lachgar and Abdelmounaïm Abdali. "Decision Framework for Mobile Development Methods". In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 8.2 (2017). DOI: 10.14569/IJACSA.2017.080215.

[Lam16] Fred Lambert. *Tesla is moving away from an SDK*. 2016. URL: http://9to5mac.com/2016/01/28/tesla-sdk-iphone-apps-mirror/.

[Lat+16] M. Latif et al. "Cross platform approach for mobile application development: A survey". In: *International Conference on Information Technology for Organizations Development, IT4OD'16* (2016). DOI: 10.1109/IT4OD.2016.7479278.

[LG 18] LG Electronics. *WebOS TV Developers*. 2018. URL: http://webostv.developer.lge.com/ (visited on 09/12/2018).

[Lil+17] Georgios Lilis et al. “Towards the next generation of intelligent building: An assessment study of current automation and future IoT based systems with a proposal for transitional design”. In: *Sustainable Cities and Society* 28 (2017), pp. 473–481. DOI: 10.1016/j.scs.2016.08.019.

[Lin18] Linux Foundation. *Tizen*. 2018. URL: https://www.tizen.org (visited on 09/12/2018).

[LL15] A. De Luca and J. Lindqvist. “Is secure and usable smartphone authentication asking too much?” In: *Computer* 48.5 (2015), pp. 64–68. DOI: 10.1109/MC.2015.134.

[LL16] Renju Liu and Felix Xiaozhu Lin. “Understanding the Characteristics of Android Wear OS”. In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’16. Singapore, Singapore: ACM, 2016, pp. 151–164. DOI: 10.1145/2906388.2906398.

[LL17] Adam Lella and Andrew Lipsman. *The 2017 U.S. Mobile App Report*. 2017. URL: https://www.comscore.com/Insights/Presentations-and-Whitepapers/2017/The-2017-US-Mobile-App-Report.

[Luo+11] Tongbo Luo et al. “Attacks on WebView in the Android system”. In: *Proceedings of the 27th Annual Computer Security Applications Conference*. 2011, pp. 343–352.

[LW13] Olivier Le Goaer and Sacha Waltham. “Yet another DSL for cross-platforms mobile development”. In: *Proc. of the First Workshop on the Globalization of Domain Specific Languages*. ACM. 2013, pp. 28–33.

[LZI18] Xiaoyu Li, Xia Zhao, and Lakshmi Iyer. “Investigating of In-app Advertising Features’ Impact on Effective Clicks for Different Advertising Formats”. In: *24th Americas Conference on Information Systems, AMCIS 2018*. 2018.

[Man+18] Amit Kr Mandal et al. “Vulnerability Analysis of Android Auto Infotainment Apps”. In: *15th ACM International Conference on Computing Frontiers*. CF ’18. ACM, 2018, pp. 183–190. DOI: 10.1145/3203217.3203278.

[MBG18] Tim A. Majchrzak, Andreas Biørn-Hansen, and Tor-Morten Grønli. “Progressive Web Apps: the Definite Approach to Cross-Platform Development?” In: *Proceedings 51th Hawaii International Conference on Systems Science (HICSS-51)*. AIS Electronic Library (AISeL), 2018.

[MD13] Murtaza Mir and Brian Dangerfield. “Propagating a digital divide: Diffusion of mobile telecommunication services in Pakistan”. In: *Technological Forecasting and Social Change* 80.5 (2013), pp. 992–1001. DOI: 10.1016/j.techfore.2012.08.006.

[MEK15] Tim A. Majchrzak, Jan Ernsting, and Herbert Kuchen. “Achieving Business Practicability of Model-Driven Cross-Platform Apps”. In: *Open Journal of Information Systems (OJIS)* 2.2 (2015), pp. 3–14.

[Meu+00] Matthew L Meuter et al. "Self-service technologies: understanding customer satisfaction with technology-based service encounters". In: *Journal of marketing* 64.3 (2000), pp. 50–64.

[MH13] Tim A. Majchrzak and Henning Heitkötter. "Status Quo and Best Practices of App Development in Regional Companies". In: *Lecture Notes in Computer Science*. Vol. 189. Springer, 2013, pp. 189–206.

[Mic17] Microsoft Inc. *Develop apps for the Universal Windows Platform (UWP)*. 2017. URL: https://docs.microsoft.com/en-us/visualstudio/cross-platform/develop-apps-for-the-universal-windows-platform-uwp.

[Mic18] Microsoft Corp. *Microsoft Band*. 2018. URL: https://www.microsoft.com/en-us/band/ (visited on 09/10/2018).

[MMM16] I. T. Mercado, N. Munaiah, and A. Meneely. "The impact of cross-platform development approaches for mobile applications from the user's perspective". In: *WAMA 2016 - Proceedings of the International Workshop on App Market Analytics, co-located with FSE 2016* (2016). DOI: 10.1145/2993259.2993268.

[Mob15] MobileHTML5. *Mobile HTML5 compatibility*. 2015. URL: http://mobilehtml5.org/ (visited on 09/12/2018).

[Moj18] Moj.io Inc. *Mojio - Connected Car Platform*. 2018. URL: https://www.moj.io/ (visited on 09/10/2018).

[MS15] Tim A. Majchrzak and Matthias Schulte. "Context-Dependent Testing of Applications for Mobile Devices". In: *Open Journal of Web Technologies (OJWT)* 2.1 (2015), pp. 27–39.

[MSK15] Farouk Messaoudi, Gwendal Simon, and Adlen Ksentini. "Dissecting Games Engines: The Case of Unity3D". In: *Proceedings of the 2015 International Workshop on Network and Systems Support for Games*. NetGames '15. Zagreb, Croatia: IEEE Press, 2015, 4:1–4:6.

[MWA15] Tim A. Majchrzak, Stephanie Wolf, and Puja Abbassi. "Comparing the Capabilities of Mobile Platforms for Business App Development". In: *LNBIP*. Vol. 232. Springer, 2015, pp. 70–88. DOI: 10.1007/978-3-319-24366-5_6.

[Nan+17] Vijayakumar Nanjappan et al. "Clothing-based wearable sensors for unobtrusive interactions with mobile devices". In: *2017 International SoC Design Conference (ISOCC)*. IEEE, 2017, pp. 139–140. DOI: 10.1109/ISOCC.2017.8368837.

[NBN14] M. Noreikis, P. Butkus, and J. K. Nurminen. "In-vehicle application for multimodal route planning and analysis". In: *Proc. IEEE 3rd CloudNet*. 2014. DOI: 10.1109/CloudNet.2014.6969020.

[Ng+14] Yi Ying Ng et al. "Which Android app store can be trusted in China?" In: *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*. IEEE. 2014, pp. 509–518.

[NJE17] Timothy Neate, Matt Jones, and Michael Evans. "Cross-device Media: A Review of Second Screening and Multi-device Television". In: *Personal Ubiquitous Comput* 21.2 (2017), pp. 391–405. DOI: 10.1007/s00779-017-1016-2.

[Occ15] Tom Occhino. *React Native: Bringing modern web techniques to mobile*. 2015. URL: https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/ (visited on 04/05/2018).

[OT12] Julian Ohrt and Volker Turau. "Cross-Platform Development Tools for Smartphone Applications". In: *IEEE Computer* 45.9 (2012), pp. 72–79. DOI: 10.1109/MC.2012.121.

[Par98] Donn B. Parker. *Fighting Computer Crime: A New Framework for Protecting Information*. New York, NY, USA: John Wiley & Sons, Inc, 1998.

[Pén+12] Thierry Pénard et al. "Comparing the determinants of internet and cell phone use in Africa: evidence from Gabon". In: *Communications & Strategies* 86 (2012), pp. 65–83.

[PG15a] E. Perakakis and G. Ghinea. "A proposed model for cross-platform web 3D applications on Smart TV systems". In: *Proc. 20th Web3D*. 2015. DOI: 10.1145/2775292.2778303.

[PG15b] E. Perakakis and G. Ghinea. "HTML5 Technologies for Effective Cross-Platform Interactive/Smart TV Advertising". In: *IEEE Trans. HMS* 45.4 (2015), pp. 534–539. DOI: 10.1109/THMS.2015.2401975.

[Pro18] Progress Software Corp. *How NativeScript works*. 2018. URL: https://docs.nativescript.org/core-concepts/technical-overview (visited on 09/05/2018).

[PSC12] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. "Comparison of cross-platform mobile development tools". In: *Proc. 16th ICIN*. IEEE, 2012, pp. 179–186. DOI: 10.1109/ICIN.2012.6376023.

[QG14] M. Quaresma and R. Gonçalves. "Usability analysis of smartphone applications for drivers". In: *Lecture Notes in Computer Science* 8517 (2014), pp. 352–362. DOI: 10.1007/978-3-319-07668-3_34.

[QGZ17] P. Que, X. Guo, and M. Zhu. "A Comprehensive Comparison between Hybrid and Native App Paradigms". English. In: *Proceedings - 2016 8th International Conference on Computational Intelligence and Communication Networks, CICN 2016*. Ed. by Tomar G.S. IEEE, 2017, pp. 611–614. DOI: 10.1109/CICN.2016.125.

[QSM09] QSM. *Function Point Languages Table: Version 5.0*. 2009. URL: http://www.qsm.com/resources/function-point-languages-table (visited on 05/27/2016).

[Rd12] Andre Ribeiro and Alberto Rodrigues da Silva. "Survey on Cross-Platforms and Languages for Mobile Apps". In: *Eighth International Conference on the Quality of Information and Communications Technology* (2012), pp. 255–260. DOI: 10.1109/QUATIC.2012.56.

[Res14] Research2guidance. *Cross-Platform Tool Benchmarking 2014*. 2014. URL: http://research2guidance.com/product/cross-platform-tool-benchmarking-2014/.

[Rev18] Florent Revest. *AsteroidOS*. 2018. URL: http://asteroidos.org/ (visited on 09/12/2018).

[Rie18] Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *Hawaii International Conference on System Sciences (HICSS-51)*. 2018, pp. 5725–5734.

[RK18a] Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *Computer Languages, Systems & Structures* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001.

[RK18b] Christoph Rieger and Herbert Kuchen. "Towards Model-Driven Business Apps for Wearables". In: *Mobile Web and Intelligent Information Systems*. Ed. by Muhammad Younas et al. Cham: Springer International Publishing, 2018, pp. 3–17. DOI: 10.1007/978-3-319-97163-6_1.

[RK19] Christoph Rieger and Herbert Kuchen. "A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes". In: *Hawaii International Conference on System Sciences (HICSS-52)*. Maui, Hawaii, USA, 2019, pp. 7431–7440.

[RM16] Christoph Rieger and Tim A. Majchrzak. "Weighted Evaluation Framework for Cross-Platform App Development Approaches". In: *Information Systems: Development, Research, Applications, Education: 9th SIGSAND/PLAIS EuroSymposium*. Ed. by Stanislaw Wrycza. Springer, 2016, pp. 18–39. DOI: 10.1007/978-3-319-46642-2_2.

[RM18] Christoph Rieger and Tim A. Majchrzak. "A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle". In: *Web Information Systems and Technologies*. Ed. by Tim A. Majchrzak et al. Cham: Springer International Publishing, 2018, pp. 202–220.

[RM19] Christoph Rieger and Tim A. Majchrzak. "Towards the Definitive Evaluation Framework for Cross-Platform App Development Approaches". In: *Journal of Systems and Software (JSS)* (2019). DOI: 10.1016/j.jss.2019.04.001.

[Roy70] Winston W. Royce. "The Development of Large Software Systems". In: *Proc. IEEE WESCON 1970*. IEEE CS, 1970, pp. 328–338.

[RP12] S. Rodriguez Garzon and M. Poguntke. "The personal adaptive in-car HMI: Integration of external applications for personalized use". In: *LNCS* 7138 (2012), pp. 35–46. DOI: 10.1007/978-3-642-28509-7_5.

[RPP14] Reza Rawassizadeh, Blaine A. Price, and Marian Petre. "Wearables: Has the Age of Smartwatches Finally Arrived?" In: *Commun. ACM* 58.1 (Dec. 2014), pp. 45–47. DOI: 10.1145/2629633.

[RT12] C. P. Rahul Raj and Seshu Babu Tolety. "A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach". In: *2012 Annual IEEE India Conference (INDICON)*. 2012, pp. 625–629. DOI: 10.1109/INDCON.2012.6420693.

[Rus15] Alex Russell. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. 2015. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/ (visited on 08/16/2017).

[Ryu+14] D. Ryu et al. "A serious game design for english education on Smart TV platform". In: *Pro. ISCE*. 2014. DOI: 10.1109/ISCE.2014.6884479.

[Sam14] Samsung. *Samsung: Let's talk about the design of the Galaxy Note Edge*. 2014. URL: https://news.samsung.com/global/lets-talk-about-the-design-of-the-galaxy-note-edge.

[Sam18] Samsung. *TOAST - Samsung Developers*. 2018. URL: https://developer.samsung.com/tv/develop/extension-libraries/toast/ (visited on 09/12/2018).

[SAW94] B. Schilit, N. Adams, and R. Want. "Context-Aware Computing Applications". In: *Proc. of the 1994 1st WMCSA*. IEEE CS, 1994, pp. 85–90.

[Sey+15] T. Seyed et al. "SoD-toolkit: A toolkit for interactively prototyping and developing multi-sensor, multi-device environments". In: *Proceedings of the 2015 ACM International Conference on Interactive Tabletops and Surfaces, ITS 2015* (2015). DOI: 10.1145/2817721.2817750.

[She88] Martin Shepperd. "A critique of cyclomatic complexity as a software metric". English. In: *Software Engineering Journal* 3 (2 Mar. 1988), 30–36(6). DOI: 10.1049/sej.1988.0003.

[SK13] A. Sommer and S. Krusche. "Evaluation of cross-platform frameworks for mobile applications". In: *LNI* P-215 (2013).

[SKS14] R. N. Sansour, N. Kafri, and M. N. Sabha. "A survey on mobile multimedia application development frameworks". In: *Proc. ICMCS*. 2014. DOI: 10.1109/ICMCS.2014.6911207.

[SMP12] Liyanage C. De Silva, Chamin Morikawa, and Iskandar M. Petra. "State of the art of smart homes". In: *Engineering Applications of Artificial Intelligence* 25.7 (2012), pp. 1313–1321. DOI: 10.1016/j.engappai.2012.05.002.

[Soh+15] H.-J. Sohn et al. "Quality evaluation criteria based on open source mobile HTML5 UI Framework for development of cross-platform". In: *IJSEIA* 9.6 (2015), pp. 1–12. DOI: 10.14257/ijseia.2015.9.6.01.

[Som11] Ian Sommerville. *Software Engineering*. 9th. Pearson, 2011.

[Sta16] StatCounter. *Mobile and tablet internet usage exceeds desktop for first time worldwide*. 2016. URL: http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide (visited on 09/05/2018).

[Sta18a] Stack Overflow. *Developer Survey Results*. 2018. URL: https://insights.stackoverflow.com/survey/2018 (visited on 11/05/2018).

[Sta18b] Statista Inc. *Statista*. 2018. URL: http://www.statista.com/ (visited on 09/12/2018).

[SV14] Stijn Schuermans and Michael Vakulenko. *Apps for connected cars? Your mileage may vary*. 2014. URL: http://trendscan.info/blog/2014/04/28/apps-for-connected-cars-your-mileage-may-vary/ (visited on 09/12/2018).

[Tan16] Ailie K. Y. Tang. "Mobile app monetization: App business models in the digital era". In: *International Journal of Innovation, Management and Technology* 7.5 (2016), p. 224.

[UB16] Eric Umuhoza and Marco Brambilla. "Model Driven Development Approaches for Mobile Applications: A Survey". In: *Mobile Web and Intelligent Information Systems: 13th International Conference*. Ed. by Muhammad Younas et al. Springer International, 2016, pp. 93–107. DOI: 10.1007/978-3-319-44215-0_8.

[Uni18] Unity Technologies. *Unity Game Engine*. 2018. URL: https://unity3d.com/de (visited on 09/12/2018).

[VJ17] T. Vilček and T. Jakopec. "Comparative analysis of tools for development of native and hybrid mobile applications". In: *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2017, pp. 1516–1521. DOI: 10.23919/MIPRO.2017.7973662.

[VS11] I. P. Vuksanovic and B. Sudarevic. "Use of web application frameworks in the development of small applications". In: *2011 Proceedings of the 34th International Convention MIPRO*. May 2011, pp. 458–462.

[VSB13] G. Vitols, I. Smits, and O. Bogdanov. "Cross-platform solution for development of mobile applications". English. In: *ICEIS 2013 - Proceedings of the 15th International Conference on Enterprise Information Systems*. Vol. 2. 2013, pp. 273–277.

[Wag17] L. Wagner. "Turbocharging the web". In: *IEEE Spectrum* 54.12 (Dec. 2017), pp. 48–53. DOI: 10.1109/MSPEC.2017.8118483.

[Was10] Anthony I. Wasserman. "Software engineering issues for mobile application development". In: *Proc. FoSER '10*. Ed. by Gruia-Catalin Roman and Kevin Sullivan. 2010, p. 397. DOI: 10.1145/1882362.1882443.

[Wat+17] Takuya Watanabe et al. "Understanding the Origins of Mobile App Vulnerabilities: A Large-scale Measurement Study of Free and Paid Apps". In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 14–24. DOI: 10.1109/MSR.2017.23.

[Wol13] F. Wolf. "Will vehicles go the mobile way? Merits and challenges arising by car-apps". In: *Proc. 10th ICINCO*. Vol. 2. 2013.

[WVN15] M. Willocx, J. Vossaert, and V. Naessens. "A Quantitative Assessment of Performance in Mobile App Development Tools". In: *Proc. 3rd Int. Conf. on Mobile Services*. 2015. DOI: 10.1109/MobServ.2015.68.

[WVN17] Michiel Willocx, Jan Vossaert, and Vincent Naessens. "Security Analysis of Cordova Applications in Google Play". In: *12th International Conference on Availability, Reliability and Security*. ARES '17. ACM, 2017, 46:1–46:7. DOI: 10.1145/3098954.3103162.

[XBM18] XBMC Foundation. *Third-party forks and derivatives*. 2018. URL: http://kodi.wiki/view/Third-party_forks_and_derivatives.

[Xie12] J. Xie. "Research on key technologies base Unity3D game engine". In: *2012 7th International Conference on Computer Science Education (ICCSE)*. July 2012, pp. 695–699. DOI: 10.1109/ICCSE.2012.6295169.

[XX13] Spyros Xanthopoulos and Stelios Xinogalos. "A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications". In: *Proc. 6th BCI*. ACM, 2013, pp. 213–220. DOI: 10.1145/2490257.2490292.

[Zha+11] J. Zhang et al. "USink: Smartphone-based moible sink for wireless sensor networks". In: *Proc. CCNC'2011*. 2011. DOI: 10.1109/CCNC.2011.5766639.

[ZZ12] Kevin Xiaoguo Zhu and Zach Zhizhong Zhou. "Research note—Lock-in strategy in software competition: Open-source software vs. proprietary software". In: *Information Systems Research* 23.2 (2012), pp. 536–545.

8

Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons

Table 8.1: Fact sheet for publication P2

Title	Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons
Authors	Fabian Wrede[1] Christoph Rieger[1] Herbert Kuchen[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2019
Publication Outlet	Journal of Supercomputing (SUPE)
Copyright	Springer International Publishing
Full Citation	Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *Journal of Supercomputing (SUPE)* (2019). DOI: 10.1007/s11227-019-02825-6

Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons

Fabian Wrede Christoph Rieger Herbert Kuchen

Keywords: algorithmic skeletons, parallel programming, high-performance computing, model-driven development, domain-specific language

Abstract: Parallel programming can be difficult and error-prone, in particular if low-level optimizations are required in order to reach high performance in complex environments such as multi-core clusters using MPI and OpenMP. One approach to overcome these issues is based on *Algorithmic Skeletons*. These are predefined patterns which are implemented in parallel and can be composed by application programmers without taking care of low-level programming aspects. Support for algorithmic skeletons is typically provided as a library. However, optimizations are hard to implement in this setting and programming might still be tedious because of required boiler plate code.
Thus, we propose a domain-specific language (DSL) for algorithmic skeletons that performs optimizations and generates low-level C++ code. Our experimental results on four benchmarks show that the models are significantly shorter and that the execution time and speedup of the generated code often outperforms equivalent library implementations using the Muenster Skeleton Library (Muesli).

8.1 Introduction

Parallel programming for high-performance computing (HPC) is a difficult endeavor, which requires expertise in different frameworks and programming languages. For example, if the code targets a multi-core cluster environment, the programmer needs to know a framework for shared-memory architectures, such as OpenMP [CJv08], for distributed-memory architectures, such as MPI [GLS14], and possibly even for accelerators, such as CUDA [Nic+08] or OpenCL [SGS10] for GPUs. Especially if advanced performance tuning is required, basic programming skills might not be sufficient. Moreover, using different frameworks in combination can lead to subtle errors, which are difficult to find and resolve.

One approach to solve this problem is based on predefined, typical parallel-programming patterns [Col91] such as `map`, `fold/reduce`, and `zip`. In addition to these data-parallel skeletons, there are task-parallel skeletons, such as `pipeline` or `farm`, and communication skeletons, such as `gather` or `scatter`. In order to use a general-purpose skeleton, a function providing the application-specific computation logic can be passed as an argument. This function is then executed in parallel on all elements of a data structure or data stream.

Thus, by using algorithmic skeletons, low-level details become transparent to the programmer and he or she does not have to consider them or even does not need to know anything about the underlying frameworks. Moreover, algorithmic skeletons make sure that common parallel-programming errors, such as deadlocks and race conditions, do not occur.

There are several libraries, which provide algorithmic skeletons (see Section 8.2). A library implementation of algorithmic skeletons makes certain optimizations hard to implement, such as re-arranging skeleton calls, since on the level of C++ and using a combination of low-level frameworks, relevant and irrelevant features are hard to distinguish at runtime. In the present paper, we show how to avoid these drawbacks by generating C++ code from a dedicated domain-specific language (DSL) model. Moreover, we demonstrate how such a DSL can facilitate the efficient development of parallel programs by reducing the language to the essential core features and offering useful validation for models.

Our paper is structured as follows: first, we give an overview about different libraries for algorithmic skeletons in Section 8.2. In Section 8.3, our DSL is described and Section 8.4 shows how the language constructs are transformed into C++ code. The results of four benchmark applications are presented in Section 8.5. In Section 8.6, we conclude and point out future work.

8.2 Related Work

Algorithmic skeletons for parallel programming are mostly provided as libraries. One instance is the Muenster Skeleton Library (Muesli), a C++ library which is used as a reference for the implementation of the presented approach. Muesli offers distributed data structures and the skeletons are member functions of them. Data-parallel skeletons implemented in Muesli are, e.g., `map` , `fold` , `zip` , `mapStencil` and variants of these. Muesli works on multi-core and multi-GPU clusters [EK12; EK17].

Other well-known libraries are for example FastFlow, which focuses on task-parallel skeletons and stream parallelism for multi-core systems [Ald+10], and eSkel, which provides skeletons for C and MPI [Ben+05].

Two libraries we want to examine more closely here are SkePU2 [ELK17] and SkeTo [ME10], since they incorporate similar concepts as presented in this paper. SkePU2 includes a source-to-source compiler based on Clang and LLVM. A program written with SkePU can always be compiled into a sequential program. A precompiler transforms the program for parallel execution, e.g. adding the `__device__` keyword to functions. SkePU uses custom C++ attributes, which the precompiler recognizes and transforms accordingly.

SkeTo is a library for distributed-memory environments, which focuses on optimizations such as fusion transformations – i.e., combining two skeleton invocations into one and hence reducing the overhead for function calls and the amount of data which is passed between skeletons. The implementation is based on expression templates [Vel95], a metaprogramming technique. By using expression templates it is for example possible to a avoid temporary variables, which are required for complex expressions.

Lately, the concepts of model-driven and generative software development have gained attraction in academia and practice, mainly because of the expected benefits in development speed, software quality, and reduction of redundant code [SV06]. In addition, DSLs allow for better reuse

and readability of models – targeted at both the modeling domain and user experience – while at the same time reducing the complexity through appropriate abstractions [MHS05]. In the domain of high-performance programming, few approaches have been presented in literature that adopt DSLs. Almorsy and Grundy [AG15] have presented a graphical notation to ease the shift from sequential to parallel implementations of existing software for CPU and GPU clusters. Anderson et al. [And+17] have extended the language Julia which is designed for scientific computing and partly aligned with the MATLAB notation. By applying optimization techniques such as parallelization, fusion, hoisting, and vectorization, the generated code significantly improves the computation. In contrast, our approach focuses on the parallelization on clusters of compute nodes.

There are also DSLs for parallel programming with skeletons or parallel patterns, which are embedded into other languages, or enable the creation of embedded DSLs, such as SPar [Gri+17] for C++ or Delite [Suj+14] for Scala. SPar uses C++ annotations and therefore, it is possible to parallelize an existing code base without much effort. Additionally, all features of the host language can easily be used. If a code base already exists, even a less complex standalone DSL requires more effort for a re-implementation. However, by reducing the language to core features in a standalone DSL the complexity can be easier handled by inexperienced programmers.

Danelutto et al. [DTK16] propose a DSL for designing parallel programs based on parallel patterns, which also allows for optimization and rewriting of the pattern composition. The DSL focuses on the management of non-functional properties such as performance, security, or power consumption. Based on the model and non-functional properties, a template providing an optimized pattern composition for the FastFlow library is generated, which the programmer can use to implement the application.

Since this work presents an own language for parallel programming, we want to highlight that some upcoming and established languages aim to benefit from parallel code execution as well. For example, Chapel has built-in concepts for parallel programming such as *forall loops* and the *cobegin statement*, which allows for starting independent tasks [CCZ07]. Also, the C++ standard includes extensions for parallel programming since version C++17 [Sta15]. For example, the algorithms `for_each` and `transform` are comparable to the presented skeletons `map(InPlace)`. However, these capabilities are restricted to a single (potentially multi-core) node and do not support clusters. Another related approach, which provides a source-to-source compiler, is Bones [NC15]. It takes sequential C code and transforms it into code for GPUs but again, it does not support distributed systems.

8.3 A Domain-Specific Language for High-Level Parallel Programming

Many existing approaches to high-level parallel programming provide parallel constructs in the form of a library. As pointed out in the introduction, this causes some limitations such as the difficulty to implement optimizations and a higher entry barrier for inexperienced programmers. Thus, we propose a DSL named *Musket* (Muenster Skeleton Tool for High-Performance Code Generation) to tackle these limitations and generate optimized code.

8.3.1 Benefits of Generating High-Performance Code

The main drawback of using libraries for high-performance computing is the fact that library calls are included in arbitrarily complex code of a host language such as C++. Besides introducing some performance overhead, a library is always restricted by the host language's syntax. In contrast, important design decisions such a the syntax and structure of code can be selected purposefully when building a DSL. For example, different algorithmic skeletons as major domain concept for parallelization can be integrated as keywords in the designed language and recognized by the editing component. Consequently, the program specification is more readable for novice users who want to apply their domain knowledge.

A code generator can easily analyze the DSL features based on a formalized meta model and produce optimized code for different hardware configurations. In the domain of high-level parallel programming using algorithmic skeletons, parallelism can be built into the structure of the language such that the user does not need to cater for parallelism-specific implementations. Required transformations can be provided by the framework developers, for example when applying an algorithmic skeleton to a distributed data structure. In addition, this level of abstraction increases the readability for users who do not need to know the details of (potentially multiple) target platforms but can focus on the high-level sequence of activities.

With regard to framework developers who are concerned with efficient program execution, DSLs introduce additional flexibility. The abstract syntax of the parallel program can be analyzed and modified in order to optimize the generated high-performance code for the target hardware. In particular, recurring – and potentially inefficient – patterns of high-level user code can be transformed to hardware-specific low-level implementations by applying rewrite rules as described in [Ste+15]. For example, *map fusion* may be applied to combine multiple transformations on the same data structure instead of applying them consecutively (cf. Section 8.4.2).

Moreover, a DSL-based approach can be extended to additional platforms in the future by supplying new generator implementations – without changing the input programs. Compared to customizing compilers, DSL creation frameworks such as Xtext further support in creating usable

editing components with features such as syntax highlighting and meaningful model[1] validation [Bet13].

8.3.2 Language Overview

The Musket DSL targets rather inexperienced programmers who want to use algorithmic skeletons to quickly write high-performance programs that run on heterogeneous clusters. Therefore, a syntax similar to C++ was chosen to align with a familiar programming language that is common for high-performance scenarios such as simulating physical or biological systems. However, a Musket model is more structured than an arbitrary C++ program and provides four main sections which are described in more detail in the following[2]. The DSL was created using the Xtext language development framework which uses an EBNF-like grammar to specify the language syntax and derives a corresponding Ecore meta model [The18]. Furthermore, a parser as well as an editor component are generated which integrate with the Eclipse ecosystem for subsequent code generation. Consequently, common features of an integrated development environment (IDE) such as syntax highlighting, auto-completion, and validation are available and have been customized to provide contextual modeling support.

Meta-Information

The header of a Musket model consists of meta information that guides the generation process. On the one hand, target *platforms* and the compiler optimization *mode* can be chosen for convenient debugging of the program. More important, the configuration of *cores* and *processes* is used by the generator to optimize the code for a distributed execution on a high-performance cluster. For example, the setup of distributed data structures, the parallel execution of skeletons, and the intra-cluster communication of calculation results are then automatically managed. An exemplary model for a matrix multiplication according to the algorithm described in [EK17] is depicted in Listing **??**.

Data Structure Declaration

Because of the distributed execution of the program, all global data structures are declared upfront and distributed to the different compute nodes. Also, global constants can be defined in this block to easily parametrize the program (lines 6-10).

Musket currently supports several primitive data types (*float, double, integer*, and *boolean*). *Array* and *matrix* collection types also exist and are defined using the C++ template style, e.g. `matrix<double,512,512,dist> table;` . This definition contains the type and dimension of the collection, and also provides a keyword indicating whether the collection should be present

[1] The model-driven software development community prefers the notion DSL *model* rather than DSL *program*.

[2] Due to the lack of space, only the overall structure and the main concepts of the language are presented here and an excerpt of the DSL is given in Figure **??**. The full DSL specification can be found in our code repository [WR18].

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

const int dim = 16384;

matrix<float,dim,dim,dist> as = {1.0f};
matrix<float,dim,dim,dist> bs = {0.001f};
matrix<float,dim,dim,dist> cs = {0.0f};

float dotProduct(int i, int j, float Cij){
  float sum = Cij;

  for (int k = 0; k < cs.columnsLocal(); k++) {
    sum += as[[i,k]] * bs[[k,j]];
  }

  return sum;
}

main{
  as.shiftPartitionsHorizontally((int a) -> int {return -a;});
  bs.shiftPartitionsVertically((int a) -> int {return -a;});

  for (int i = 0; i < as.blocksInRow(); ++i) {
    cs.map<localIndex, inPlace>(dotProduct());
    as.shiftPartitionsHorizontally((int a)-> int {return -1;});
    bs.shiftPartitionsVertically((int a) -> int {return -1;});
  }

  as.shiftPartitionsHorizontally((int a) -> int {return a;});
  bs.shiftPartitionsVertically((int a) -> int {return a;});
}
```

Listing 8.1: Musket model for matrix multiplication.

on all nodes (*copy*), distributed across the nodes (*dist*, *rowDist*, or *columnDist*), or instantiated depending on the context (*loc*). The explicit distinction lets the user control the partitioning of a data structure by means of a user function (see subsection 8.3.2). To simplify the handling of distributed data structures, collections can be accessed either using their global index (e.g. `table[42]`) or the local index within the current partition (e.g. `table[[42]]`). Moreover, primitive and collection types can be composed into custom *struct* types.

User Function Declaration

The third section of a Musket program consists of custom user functions which specify behavior to be executed on each node within skeleton calls (such as the `dotProduct` function in lines 12-20). Therefore, a wide variety of calculations such as arithmetic and boolean expressions can be directly expressed in the DSL. In addition to assignments and skeleton applications, different control structures such as *sequential composition*, *if statements*, and *for loops* are available. Moreover, the modeler can use C++ functions from the standard library or call arbitrary external C++ functions (which are, however, not considered for the optimizations described in Section 8.4.2).

Within functions, users can access globally available data structures (declared in the previous section) or create local variables to store temporary calculation results which are not available to other processes. The sophisticated validation capabilities allow for instant feedback to the user when errors are introduced in the model. For example, type inference aims to statically analyze the resulting data type of expressions or type casts, and thus warns the user before vainly starting the generation process.

Main Program Declaration

Finally, the overall sequence of activities in the program is described in the *main* block (lines 22-34 in Listing **??**). Besides the possible control structures and expressions described in the previous paragraphs, *skeleton functions* are the main features to write high-level parallel code. Currently, *map*, *fold*, *gather*, *scatter*, and *shift partition* skeletons are implemented in multiple variants. In general, they are applied to a distributed data structure and may take additional arguments such as the previously defined user functions. For convenience and code readability reasons, the user can instead specify a lambda abstraction for simple operations e.g. `(int a)-> int {return -a;}` .

An excerpt of the Musket grammar concerning the main program declaration is depicted in Listing **??** using the EBNF notation[3]. A *map* skeleton applies a user function to a each element of the data structure (either returning a new collection or updating values in place depending on the *SkeletonOption*). A *fold* skeleton (also known as reduce pattern) takes a user function and the identity value of the operation and folds pairs of elements in the collection into a single value. For performance reasons, both skeletons can be combined into a *mapFold* skeleton (see Section 8.4.2). The *zip* skeleton joins two data structures of the same size using the provided user function. The

[3]The Xtext representation of the full DSL is available in our code repository [WR18].

```
MainBlock ::=  // cf. Section 3.2.4
 'main' '{' {MainFunctionStatement} '}'

MainFunctionStatement ::=
 MusketControlStructure |  // For loop and if clause variants
 MusketStatement ';'

MusketStatement ::=
 MusketVariable |          // Variable declarations
 MusketAssignment |        // Assigning values to variables
 SkeletonExpression |      // Arithmetic and boolean expressions
 FunctionCall              // Function calls without assignment

SkeletonExpression ::= CollectionObjectRef '.' Skeleton

Skeleton:           // available algorithmic skeletons
 'map'  SkeletonOptions '(' MapFunction ')' |
 'fold' SkeletonOptions '(' IdentityValue ',' FoldFunction ')' |
 'mapFold' SkeletonOptions
     '(' MapFunction ',' IdentityValue ',' FoldFunction ')' |
 'zip' SkeletonOptions '(' ObjectRef ',' UserFunction ')' |
 'gather'  '(' ')' |
 'scatter' '(' ')' |
 'shiftHorizontally' '(' UserFunction ')' |
 'shiftVertically'   '(' UserFunction ')'
// alternative skeleton representations omitted

SkeletonOptions ::= ['<' SkeletonOption {',' SkeletonOption} '>']
SkeletonOption  ::= index | localIndex | inPlace

MapFunction ::= UserFunction
FoldFunction ::= UserFunction

UserFunction ::=
 FunctionCall |     // reference to user function (cf. 3.2.3)
 LambdaFunction     // inline definition of functions (cf. 3.2.4)
```

Listing 8.2: Excerpt of the Musket DSL in EBNF notation.

gather and *scatter* skeletons are used to transfer objects with different distribution strategies. Finally, *shift* skeletons can be applied in order to re-distribute rows / columns of distributed matrices between computation nodes.

Again, multiple validators have been implemented to ensure that the types and amount of parameters passed into skeletons match. Meaningful error messages such as depicted in Figure 8.1 can be instantly provided while writing the program instead of relying on cryptic failure descriptions when compiling the generated code.

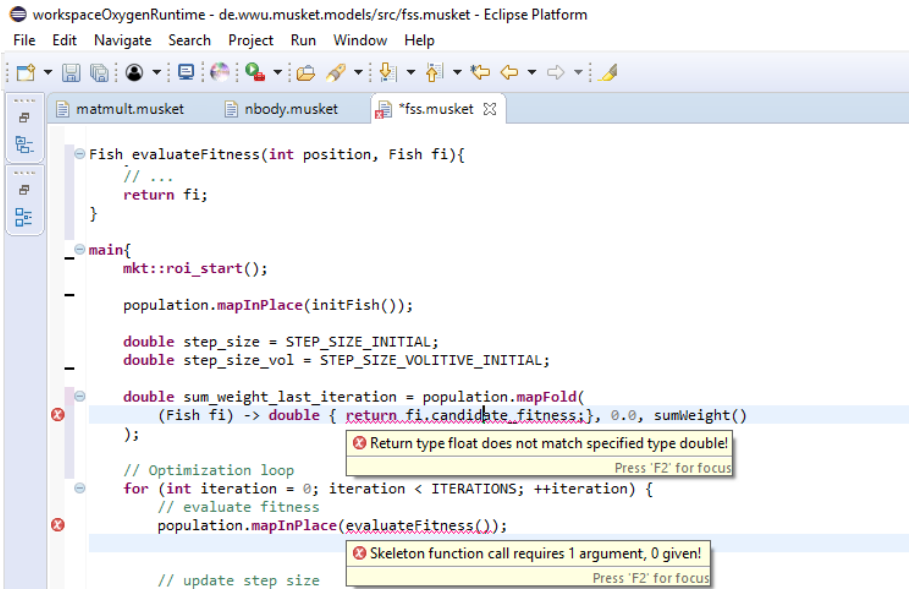


Figure 8.1: Integration of Custom Validation Errors in the Eclipse IDE

To sum up, the Musket DSL represents a subset of the C++ language in order to handle the complexities of generating parallelism-aware and hardware-optimized code. With only few additions such as distribution modes, local/global collection access, and predefined skeleton functions, a transformation of otherwise regular C++ code into distributed programs which are executable in a cluster environment can be achieved.

8.4 Code Generation for Multi-Core Clusters

In the following section, we demonstrate how certain language constructs are transformed into C++ code. We cover the data structures, data-parallel skeletons, as well as selected specific functions provided by the language. In general, we tried to generate code, which is still readable and makes use of modern features of C++11, 14, and 17.

Further, the way to generate code as described in the following is only one possibility. The approach of using a DSL allows for generating very different implementations to achieve the same behavior. It becomes also possible to consider cost models or descriptions of the target

hardware and guide the generation accordingly. This also includes the generation of code for different architectures. By adding an additional generator, the same language and models can be reused to, for example, generate code for GPUs.

8.4.1 Data Structures

In general, all distributed data structures are represented as wrapper classes around a `std::vector`. Based on the number of processes configured in the model and the distribution type, the size of the local vectors is calculated. Consequently, also for matrices the values are stored only in one vector. When an element in a matrix is accessed, the index is calculated accordingly.

Even though the size of the data structure is known when the code is generated, we decided to use `std::vector` over `std::array`. This is mostly because of the more efficient *move* operation for vectors: for some skeletons, intermediate buffers for sending and receiving data are required and we found vectors to be more efficient when data is moved from temporary buffers to the main vector.

Structs, which are defined in the model, are transformed into C++ structs. Additionally, a default constructor is generated, which initializes all members to default values. Moreover, the generation approach could be used to generate code for different data layouts. At the moment, the data is generated as array of structs, but it could be transformed to struct of arrays or any hybrid representation, which can increase the performance regarding data access and vectorization.

Moreover, there are collection functions, such as `show` and `size`, which can be invoked on data structures. Where possible, these function calls are already evaluated during the generation. For example, the global or local size is known for distributed data structures so that the function call can be replaced by the fixed value.

8.4.2 Model Transformation

The generation approach enables a re-writing step of skeleton calls by performing a model-to-model transformations before the actual generation. In this transformation, certain sequences of skeleton calls can be re-written in a more efficient way [Kuco4]. For example, one or more skeleton calls can be combined through *map fusion*. This is the case for several calls of `map` on the same data structure. The sequence `a.mapInPlace(f); a.mapInPlace(g);` can be joined to `a.mapInPlace(g ∘f);`. For the generated code, this is one less parallel loop, which can save time for synchronization and intermediate data storage.

Also, different skeletons such as `map` and `fold` can be combined. In terms of the presented DSL, `a.mapInPlace(f); x = a.fold(0, g);` can be joined to `x = a.mapFold(f, 0, g);`. In the generated code this results in one parallel for loop with reduction and a call to `MPI_Allreduce` instead of two loops and the MPI call. Moreover, the intermediate result does not need to be stored in the result data structure. However, this transformation would only be valid if `a` was not used in any subsequent skeleton calls. Using static analysis of the model's abstract syntax tree,

such transformations can be specifically targeted, for instance to optimize specific combinations of skeleton and user function.

8.4.3 Custom Reduction

The implementation of the fold skeleton is based on a straightforward sequence of the OpenMP pragma `#pragmaomp parallel for simd reduction` for performing a local reduction in each thread, followed by an `MPI_Allreduce` for combining the local intermediate results. MPI requires a function with the following signature `void f(void *in, void *inout, int *len, MPI_Datatype *dptr)`, which can then be used in reduction operations. By generating this reduction function, it is possible to avoid the combination of a gather operation followed by a second local fold.

8.4.4 User Functions

The generation approach allows for generating user functions in different ways, while they can be expressed at a single point in the model. Moreover, the context in which the function is called can be considered during the generation step, e.g. the function might be generated differently if is used in a `map map_in_place` skeleton. Further examples are the generation for different platforms, e.g. clusters with or without GPUs or generating different functions based on a single user functions as described in subsection 8.4.3. To this respect, the generation approach provides a rather convenient possibility to provide alternative code for the same model.

8.4.5 Specific Musket Functions

There are some additional functions provided by Musket, which are not part of the standard library, such as `rand`. If the `rand` function is used in the model, random engines and distribution objects are generated in the beginning of the main function, so that they can be re-used without additional overhead. The actual call to `rand` is generated as `rand_dist[thread_id](random_engines ↩ [thread_id])`, thus it can be used as a part of an expression. Consequently, the DSL conveniently reduces the amount of boilerplate code, since the function can simply be used in the model without, for example, creating an object which creates the random engines on construction.

8.4.6 Build Files

The generation approach offers additional convenience for programmers. In addition to the source and header files, we also generate a CMake file and scripts to build and run the application as well as Slurm job files [YJG03]. Consequently, there is no effort required for the setup and build process, which lowers the entry threshold to parallel programming for inexperienced programmers.

8.5 Benchmarks

We used four benchmark applications to test our approach: calculation of the Frobenius norm, Nbody simulation, matrix multiplication, and Fish School Search (FSS). In the following subsections, we demonstrate the models, compare them to the C++ implementations with Muesli, and analyze the execution times for both. All execution times are presented in Table 8.2. The code has been compiled with g++7.3.0 and OpenMPI 3.1.1. Each node of the cluster we have used for the benchmark is equipped with two Intel Xeon E5-2680 v3 CPUs (12 cores each, 30MiB shared L3 cache per CPU and 256KiB L2 cache per core) and 7200MiB memory per node. Hyper-Threading has been disabled.

Table 8.2: Execution times of the benchmark applications (in seconds)

			Execution times (s)		
Benchmark	**Nodes**	**Cores**	**Muesli**	**Musket**	**Speedup**
Frobenius norm	1	1	9.8932	2.0513	4.8228
	1	6	2.1389	0.6906	3.0973
	1	12	1.4890	0.6520	2.2837
	1	18	1.5508	0.6366	2.4361
	1	24	1.6015	0.7240	2.2120
	4	1	2.4793	0.5193	4.7743
	4	6	0.5308	0.2308	2.2996
	4	12	0.3925	0.2034	1.9298
	4	18	0.3943	0.1967	2.0043
	4	24	0.3915	0.1960	1.9970
	16	1	0.6703	0.1364	4.9136
	16	6	0.1497	0.0706	2.1212
	16	12	0.1040	0.0575	1.8093
	16	18	0.1018	0.0538	1.8937
	16	24	0.1060	0.0528	2.0089
Nbody simulation	1	1	7422.7370	7568.5646	0.9807
	1	6	1234.074	1249.3490	0.9878
	1	12	616.9001	624.8900	0.9872
	1	18	411.3290	416.6970	0.9871
	1	24	309.0764	312.6062	0.9887
	4	1	1850.179	1923.3303	0.9620
	4	6	308.7439	312.5329	0.9879
	4	12	154.3643	156.2796	0.9877

Benchmark	Nodes	Cores	Execution times (s)		Speedup
			Muesli	Musket	
	4	18	102.9115	104.2059	0.9876
	4	24	77.7512	78.4770	0.9908
	16	1	473.8626	469.3563	1.0096
	16	6	77.2279	78.1771	0.9879
	16	12	38.6245	39.1056	0.9877
	16	18	25.7638	26.0871	0.9876
	16	24	23.1896	19.6905	1.1777
Matrix	1	1	83529.6618	17802.0678	4.6921
Multiplication	1	6	14726.9833	2430.8756	6.0583
	1	12	7700.4190	1269.3401	6.0665
	1	18	5185.8780	953.9736	5.4361
	1	24	4758.7190	711.8798	6.6847
	4	1	19245.4000	4043.8573	4.7592
	4	6	3578.7240	588.7502	6.0785
	4	12	2086.8870	275.8081	7.5664
	4	18	1549.4470	199.2162	7.7777
	4	24	1376.8620	164.2069	8.3849
	16	1	3655.1590	787.6939	4.6403
	16	6	729.3905	97.0805	7.5133
	16	12	413.3354	44.4174	9.3057
	16	18	252.2129	32.3545	7.7953
	16	24	224.8478	26.2930	8.5516
Fish School Search	1	1	916.3965	762.9235	1.2012
	1	6	158.2332	136.2039	1.1617
	1	12	81.0045	72.4273	1.1184
	1	18	54.8713	52.2537	1.0501
	1	24	45.5823	43.0435	1.0590
	4	1	211.6765	182.9391	1.1571
	4	6	40.1817	33.2093	1.2100
	4	12	20.9549	18.1227	1.1563
	4	18	15.2139	13.7305	1.1080
	4	24	13.5301	12.0980	1.1184
	16	1	54.9478	46.2524	1.1880

			Execution times (s)		
Benchmark	**Nodes**	**Cores**	**Muesli**	**Musket**	**Speedup**
	16	6	11.7779	9.4939	1.2406
	16	12	7.1947	6.1798	1.1642
	16	18	5.9784	5.4158	1.1039
	16	24	5.7644	5.2615	1.0956

8.5.1 Frobenius Norm

The calculation of the Frobenius norm for matrices consists of three steps. First, all values are squared, then all values are summed up, and finally, the square root of the sum yields the result. We used a 32768×32768 matrix with double precision values. The model is presented in Listing **??**.

There is one matrix defined as a distributed data structure. Since all user functions are written as lambda expressions, there is no need for separately defined user functions. Within the main block, the logic for the program is defined. First, in this case, the matrix is initialized with arbitrary values. The calculation of the Frobenius norm is modelled in lines 14-16. The functions `roi_start` and `roi_end` are merely for benchmark purposes and trigger the generation of timer functions. In line 20, the Musket function `print` is used, so that the result is only printed once by the main process.

The results for this benchmark already reveal some interesting insights, even though the complexity of the program is rather low, which is also the reason why the program does not scale very well, when increasing the number of cores per node.

Musket achieves good speedups compared to the Muesli implementation. There are multiple effects that lead to the observed results. First of all, for the generated code GCC is able to vectorize the loops performing the `map` and `fold` operations. Auto-vectorization is however not possible for the Muesli implementation. Additionally, Muesli does not consider configurations with one process or one thread as special cases, but relies on the fact that MPI routines also work for one process and that all used OpenMP pragmas are ignored for sequential programs. In contrast, Musket checks the configuration and generates the code accordingly. Consequently, there are no MPI routines in the generated program and less operations are required regarding the management of the data structures, if there is only one process. When a data structure is created in Muesli, the global and local sizes have to be calculated, the new memory has to be allocated etc. In Musket, the data structures are defined in the model, and all required information, such as the size, can be generated and therefore need not be calculated during program execution.

To give a perspective on the effort of writing a parallel program, we want to point to the lines of code for the Musket model, the Muesli implementation, and the generated source file. We did not use the lambda notation for Musket for counting the lines of code, since Muesli requires functors in certain situations and in this way the results become more comparable. While the Musket

```
1  #config PLATFORM CPU
2  #config PROCESSES 4
3  #config CORES 24
4  #config MODE release
5
6  matrix<double,32768,32768,dist> as;
7
8  main{
9    // init
10   as.map<index, inPlace>((int x, int y, double a) -> double {return (double) x + y + 1.5;});
11
12   mkt::roi_start(); // Timer start
13
14   as.map<inPlace>((double a) -> double {return a * a;});
15   double fn = as.fold(0.0, (double a, double b) -> double {return a + b;});
16   fn = std::sqrt(fn);
17
18   mkt::roi_end(); // Timer end
19
20   mkt::print("Frobenius norm is %.5f.\n", fn);
21 }
```

Listing 8.3: Model for Frobenius norm calculation

model consists of 20 lines, the Muesli implementation has 45 lines and the generated source code 398. Thus, we conclude that the DSL provides a concise way to express a parallel program.

Another aspect to mention here is the skeleton fusion optimization. The lines 14 and 15 could also be written as `double fn = as.mapFold(square(), 0.0, sum());`, if we assume that the lambda expressions correspond to the respective functions. In the generated code, this would reduce the two loops for the map and fold operations into a single loop. The execution times in Table 8.2 do not reflect this optimization to keep the results comparable, because Muesli does not offer a combined `mapFold` skeleton. As an example, for a configuration with 4 nodes and 24 cores, the execution time has been 0.07s with skeleton fusion, which corresponds to a speedup of 2.68 compared to the Musket implementation without skeleton fusion.

8.5.2 Nbody Simulation

In the case of the Nbody simulation with 500.000 particles over five time steps the execution times for both implementations are rather similar. For most configurations, the Musket generated code is slightly slower, while for the configuration with 16 nodes and 24 cores per node there is a speedup of 1.18. The benchmark does not allow for much optimizations. Transformations such as skeleton fusion are not applicable and the user function used in the `map` skeleton contains function calls as well as multiple branches, which prevent efficient vectorization.

In an alternative implementation of the Musket generator, we investigated the effects of inlining of user functions to avoid overhead for function calls. Contrary to intuition, this approach is

not blindly applicable as can be seen for the Nbody simulation where inlining has not been advantageous. For a configuration with 4 nodes and 24 cores, the execution time was even 87.08s compared to 78.48s for the current approach. We have simulated the behavior of the application with Valgrind's cachegrind and callgrind tools [NS07]. The number of L3 cache misses as well as the amount of unnecessary data loaded to cache is higher than for the Muesli implementation. Consequently, we refrain from applying loop unrolling and inlining by default and propagate these optimizations to the subsequent compilation step.

In terms of complexity, both implementations have about the same size. The Musket model consists of 77 lines of code, whereas the Muesli program consists of 84 lines. The generated code consists of 335 lines.

8.5.3 Matrix Multiplication

The matrix multiplication benchmark shows a case, in which massive improvements become possible due to the code generation approach. We have performed the matrix multiplication with matrices of 16.384×16.384 single precision values, but first, we have to emphasize again that Musket targets rather inexperienced programmers. The benchmark is set up in such a way that the second matrix for the multiplication is not transposed. Hence, the data is stored row major, but the user function iterates column-wise through the matrix, which leads to inefficient memory accesses. The model is shown in Listing **??** in Section 8.3.

In the Muesli implementation, the compiler is not able to vectorize the calculation and to optimize the memory access. However, this is the case for the generated program, which leads to significant shorter execution times. The speedups for configurations with multiple nodes and cores range between 6.08 and 9.31.

The model has 42 lines of code, while the Muesli implementation has 74. Again, the effort for implementing the benchmark has been reduced. In comparison, the generated code has 542 lines.

8.5.4 Fish School Search

The FSS benchmark showcases a complex and more real-world example of a parallel program. FSS is a swarm-intelligent meta-heuristic to solve hard optimization problems [Bas+08]. The model has 244 lines of code and the Muesli implementation even 623, which is a reduction of about 61%. The generated code consists of 866 lines of code. A detailed discussion of the implementation with Muesli can be found in [WMK18].

Listing **??** shows excerpts of the FSS model. Since Musket also supports distributed data structures of complex types, – which can include arrays – it becomes very convenient to work with. The struct for *Fish* is defined in lines 9–19 and the distributed array is defined in line 21. The fact that complex types are allowed in distributed data structures highlights the benefit of the fused `mapFold` skeleton. For one operator called *collective volitive movement*, it is necessary to calculate the sum of the weight of all fish. Line 26 shows how this can be conveniently done

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

const int NUMBER_OF_FISH = 2048;
const int DIMENSIONS = 512;

struct Fish{
  array<double,DIMENSIONS,loc> position;
  double fitness;
  array<double,DIMENSIONS,loc> candidate_position;
  double candidate_fitness;
  array<double,DIMENSIONS,loc> displacement;
  double fitness_variation;
  double weight;
  array<double,DIMENSIONS,loc> best_position;
  double best_fitness;
};

array<Fish,NUMBER_OF_FISH,dist> population;
// [...]

main{
  // [...]
  double sum_weight = population.mapFold(
      (Fish fi) -> double {return fi.weight;}, 0.0,
      (double a, double b) -> double {return a + b;});
  // [...]
}
```

Listing 8.4: Excerpt of the Musket model for Fish School Search

by invoking the `mapFold` skeleton on the `population` array. In the map part, the weight of each fish is taken and in the fold part the sum is calculated. In the generated, code this is efficiently performed in a single parallelized loop. The execution times show a slight improvement for most configurations with speedups up to 1.24.

8.6 Conclusions and Future Work

In this paper, we have proposed a DSL for parallel programming, which is based on algorithmic skeletons. Regarding the execution times, the generated code can offer significant speedups (of up to 9) compared to the library-based approach Muesli for most benchmarks and configurations. Furthermore, the benchmark applications have shown that the DSL offers a convenient and concise way to express applications. Based on a single model, it becomes possible to generate different implementations to achieve the same behavior.

For future work on multi-core clusters, this leads to the problem of selecting the best (i.e., fastest) alternative. This could be achieved by considering cost models or descriptions of the target hardware or by comparing alternatives experimentally. At the same time, we are investigating more model transformations to identify performance potentials in the interplay of user functions and skeletons in order to further optimize the generated code.

In addition, current work focuses on implementing a corresponding generator for multi-GPU clusters. Consequently, the modeler can express the desired program using the Musket DSL and generate optimized code for multiple platforms using the same Musket models.

References

[AG15] M. Almorsy and J. Grundy. “Supporting Scientists in Re-engineering Sequential Programs to Parallel Using Model-Driven Engineering”. In: *2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*. IEEE, 2015, pp. 1–8. DOI: 10.1109/SE4HPCS.2015.8.

[Ald+10] M. Aldinucci et al. “Efficient Streaming Applications on Multi-core with FastFlow: The Biosequence Alignment Test-bed”. In: *Parallel computing: from multicores and GPU’s to petascale*. Ed. by B. Chapman et al. Vol. 19. Advances in Parallel Computing. Amsterdam: IOS Press, 2010. DOI: 10.3233/978-1-60750-530-3-273.

[And+17] T. A. Anderson et al. “Parallelizing Julia with a Non-Invasive DSL”. In: *31st European Conference on Object-Oriented Programming (ECOOP ’17)*. Ed. by P. Müller. Vol. 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 4:1–4:29. DOI: 10.4230/LIPIcs.ECOOP.2017.4. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7269.

[Bas+08] C. J. A. Bastos-Filho et al. "A Novel Search Algorithm Based on Fish School Behavior". In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC '08)*. IEEE, 2008, pp. 2646–2651. DOI: 10.1109/ICSMC.2008.4811695.

[Ben+05] A. Benoit et al. "Flexible Skeletal Programming with eSkel". In: *Proceedings of the 11th International Euro-Par Conference on Parallel Processing (Euro-Par '05)*. Ed. by J. C. Cunha and P. D. Medeiros. Vol. 3648. Lecture Notes in Computer Science. Springer, 2005, pp. 761–770. DOI: 10.1007/11549468{\textunderscore}83.

[Bet13] L. Bettini. *Implementing Domain-specific Languages with Xtext and Xtend*. Community experience distilled. Birmingham, UK: Packt Pub, 2013.

[CCZ07] B. L. Chamberlain, D. Callahan, and H. P. Zima. "Parallel Programmability and the Chapel Language". In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[CJv08] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Scientific and Engineering Computation. Cambridge, MA, USA: MIT Press, 2008.

[Col91] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[DTK16] M. Danelutto, M. Torquati, and P. Kilpatrick. "A DSL Based Toolchain for Design Space Exploration in Structured Parallel Programming". In: *Procedia Computer Science* 80 (2016). International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA, pp. 1519–1530. DOI: https://doi.org/10.1016/j.procs.2016.05.477. URL: http://www.sciencedirect.com/science/article/pii/S1877050916309620.

[EK12] S. Ernsting and H. Kuchen. "Algorithmic Skeletons for Multi-core, Multi-GPU Systems and Clusters". In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138. DOI: 10.1504/IJHPCN.2012.046370.

[EK17] S. Ernsting and H. Kuchen. "Data Parallel Algorithmic Skeletons with Accelerator Support". In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299. DOI: 10.1007/s10766--016--0416--7.

[ELK17] A. Ernstsson, L. Li, and C. Kessler. "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems". In: *International Journal of Parallel Programming* (2017). DOI: 10.1007/s10766-017-0490-5.

[GLS14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. 3rd ed. Scientific and Engineering Computation. Cambridge, MA, USA: MIT Press, 2014.

[Gri+17] D. Griebler et al. “SPar: A DSL for high-level and productive stream parallelism”. In: *Parallel Processing Letters* 27.01 (2017), p. 1740005.

[Kuc04] H. Kuchen. “Optimizing Sequences of Skeleton Calls”. In: *Domain-Specific Program Generation*. Ed. by C. Lengauer et al. Vol. 3016. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 254–274.

[ME10] K. Matsuzaki and K. Emoto. “Implementing Fusion-Equipped Parallel Skeletons by Expression Templates”. In: *Implementation and Application of Functional Languages*. Ed. by M. T. Morazán and S. Scholz. Berlin, Heidelberg: Springer, 2010, pp. 72–89.

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. “When and How to Develop Domain-specific Languages”. In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[NC15] C. Nugteren and H. Corporaal. “Bones: An Automatic Skeleton-based C-to-CUDA Compiler for GPUs”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (2015), p. 35.

[Nic+08] J. Nickolls et al. “Scalable Parallel Programming with CUDA”. In: *Queue* 6.2 (2008), pp. 40–53. DOI: 10.1145/1365490.1365500.

[NS07] N. Nethercote and J. Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[SGS10] J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science and Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[Sta15] ISO Standard. *Programming Languages – Technical Specification for C++ Extensions for Parallelism*. Standard ISO/IEC TS 19570:2015. Geneva, Switzerland: International Organization for Standardization, 2015. URL: https://www.iso.org/standard/65241.html.

[Ste+15] M. Steuwer et al. “Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’15. Vancouver, BC, Canada: ACM, 2015, pp. 205–217. DOI: 10.1145/2784731.2784754. URL: http://doi.acm.org/10.1145/2784731.2784754.

[Suj+14] Arvind K Sujeeth et al. “Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014), p. 134.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development*. Chichester: John Wiley & Sons, 2006.

[The18] The Eclipse Foundation. *Xtext Documentation*. https://eclipse.org/Xtext/documentation/. 2018. (Visited on 03/29/2018).

[Vel95] T. Veldhuizen. "Expression Templates". In: *C++ Report* 7.5 (1995), pp. 26–31.

[WMK18] F. Wrede, B. Menezes, and H. Kuchen. "Fish School Search with Algorithmic Skeletons". In: *International Journal of Parallel Programming* (2018). DOI: 10.1007/s10766-018-0564-z. URL: %5Curl%7Bhttps://doi.org/10.1007/s10766-018-0564-z%7D.

[WR18] F. Wrede and C. Rieger. *Musket Material Respository*. https://github.com/wwu-pi/musket-material. 2018. URL: https://github.com/wwu-pi/musket-material.

[WRK19] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *Journal of Supercomputing (SUPE)* (2019). DOI: 10.1007/s11227-019-02825-6.

[YJG03] A. B. Yoo, M. A. Jette, and M. Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Berlin, Heidelberg: Springer, 2003, pp. 44–60.

9

A Process-Oriented Modeling Approach for Graphical Development of Mobile Business Apps

Table 9.1: Fact sheet for publication P3

Title	A Process-Oriented Modeling Approach for Graphical Development of Mobile Business Apps
Authors	Christoph Rieger[1] Herbert Kuchen[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2018
Publication Outlet	Computer Languages, Systems & Structures
Copyright	Elsevier (CC BY-NC-ND 4.0)
Full Citation	Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *Computer Languages, Systems & Structures* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001

A Process-Oriented Modeling Approach for Graphical Development of Mobile Business Apps

Christoph Rieger Herbert Kuchen

Keywords: Graphical DSL, Mobile Application, Business App, Model-driven Software Development, Data Model Inference

Abstract: Mobile app development is an activity predominantly performed by software developers. Domain experts and future users are merely considered in early development phases as source of requirements or consulted for evaluating the resulting product. In the domain of business apps, many cross-platform programming frameworks exist but approaches also targeted at non-technical users are rare. Existing graphical notations for describing apps either lack the simplicity to be understandable by domain experts or are not expressive enough to support automated processing. The MAML framework is proposed as model-driven approach for describing mobile apps in a platform-agnostic fashion not only for software developers but also for process modelers and domain experts. Data, views, business logic, and user interactions are jointly modeled from a process perspective using a graphical domain-specific language. To aggregate multiple use cases and provide advanced modeling support, an inference mechanism is utilized to deduce a global data model. Through model transformations, native apps are then automatically generated for multiple platforms without manual programming. Our approach is compared to the IFML notation in an observational study, with promising results regarding readability and usability.

9.1 Introduction

A decade after Apple triggered the trend towards smartphones with its first iPhone, mobile devices and apps have been widely adopted. Business apps usually cover small-scale tasks and support the digitalization of processes which benefit from the increased mobility and availability of ubiquitous devices. For example, salespersons can access company data from a customer's location, expenses can be followed up remotely, and employees can submit requests for vacations not only from their office desk.

Until now, app development remains a task predominantly executed by programmers, often considering other stakeholders and future users primarily in requirements engineering phases upfront implementation. However, the research institute Gartner predicted that within a few years, more than half of all company-internal business apps will be created using codeless tools [Rv14]. Many frameworks for *programming* mobile apps have emerged over the past years and cross-platform approaches allow for a large user base with low development efforts (an overview is given in [PSC12; CAB14; MEK15]). Several commercial platforms provide cross-platform capabilities but usually focus on source code transformations, partly supported by graphical editors for designing individual views (e.g., [Esp17; Xam17]).

Modeling approaches that focus on platform-agnostic representations of mobile apps are rarely used in practice. Model-driven software development using a domain-specific language (DSL) bears the advantage of transforming a concise specification of the target application into a software product (semi-) automatically [MHS05]. DSLs are generally suited to cover a well-defined scope with sensible abstractions for inherent domain concepts and increase productivity of developers compared to general purpose languages (GPL) [KMC12]. Several textual DSLs for mobile apps have been presented in literature (e.g., [HV11; JJ15; HM13]), although not all of them fully automate code generation [UB16]. However, *textual* DSLs provide only minor benefits to *non-technical users* because they still feel like programming [ZS09].

Graphical modeling is therefore particularly suitable to describe apps by stakeholders with strong domain knowledge in order to better match the software product with their tacit requirements [BKF14; Mel+16]. A suitable notation mandates a platform-independent design to also consider emerging mobile devices such as wearables, smart home applications, and in-vehicle apps.

To model sequences of activities, a wide variety of general-purpose process modeling notations such as Business Process Model and Notation (BPMN) exists [Obj11]. Usually, those are not detailed enough to cover mobile-specific aspects and can hardly be interpreted by code generators from a technical point of view. In contrast, technical notations such as the Interaction Flow Modeling Language (IFML) are too complex to be understood by domain experts and require software engineering knowledge [Obj15a; Żył15].

Moreover, the editor component is of major importance for the usability of a graphical modeling approach. Comparisons for graphical notations such as the Unified Modeling Language (UML) show that editors for the same notation differ significantly in modeling effort, learnability, and memory load for the user [SIK15]. Editor development is a challenge in itself [Buc12; El +] and even the presence of a metamodel can only support the syntax of the resulting notation [Kol+17].

This article aims to alleviate the aforementioned problems by presenting the Münster App Modeling Language (MAML; pronounced 'mammal'), a *graphical DSL* for describing business apps that tackles the trade-off between technical complexity and graphical oversimplification in order to be understandable not only for software developers but also for domain experts and process modelers. *Model transformations* allow for a fully automatic generation of native smartphone apps for the Android and iOS platform from the specified graphical model without manually writing code. Besides the technical necessity of such a global model for code generation, it enables *advanced modeling support* for the graphical editor.

Four main research questions are addressed to investigate the potential of data model inference in the context of model-driven code generation approaches for mobile business apps:

(RQ1) How can user-oriented specification of business app functionality be achieved using a graphical modeling notation?

(RQ2) Is the modular subdivision of mobile app functionality feasible from a technical perspective with regard to the recombination of partial data models?

(RQ3) What additional support can data model inference provide for users to create semantically correct models?

(RQ4) Does the process-oriented subdivision of functionality help non-technical users in understanding and creating mobile app models?

Extending on previous work presented at SAC 2017 [Rie17a], this article presents the data model inference approach using a the scenario of an inventory management app and focuses on the benefits of such a mechanism for modeling environments regarding semantic semantic and validation (RQ3). In addition, the evaluation of MAML is extended to investigate the perceived usefulness of data model inference specifically for non-technical users (RQ4).

After presenting related work in **??**, the proposed DSL and framework are presented in **??**. **??** discusses the approach and presents evaluation results from a usability study before concluding in **??**.

9.2 Related work

Since model-driven and cross-platform development of apps has been a topic for a few years now, there is plenty of scientific work on the general topic. However, only few graphical modeling approaches with subsequent code generation of mobile apps exist. Especially with regard to a workflow-related level of abstraction, related work on business process modeling is also considered in the following.

Cross-platform mobile apps Developing mobile apps that run on multiple platforms can be achieved using different approaches. El-Kassas et al. [El-+15] distinguish three major categories: *compiling* existing source code from a legacy application or different platform such as in [Cha+17], *interpreting* a single code base through a runtime or virtual machine such as Apache Cordova for developing hybrid apps [Apa16], and *model-driven* generation of native app source code from a common representation. With regard to the latter category, various academic and commercial frameworks exist [UB16]. Only few of them, such as Mobl [HV11], PIMAR [Vau+14], and AXIOM [JJ15] cover the full spectrum of runtime behavior and structure of an app; often providing a custom textual DSL for this means. The work in this article is based on MD² which focuses on the generation of business apps (i.e. form-based, data-driven apps interacting with back-end systems [MEK15]). The input model is likewise specified using a platform-independent, textual DSL [HM13]. After preprocessing the model, code generators transform it into native platform source code for the Android and iOS platform [ME15; Ern+16]. Despite reducing development effort and complexity through domain-specific concepts, textual DSLs often feel like programming

to non-technical users [Żył15]. In order to include those stakeholders and potential end users in the development of the app product, we consider graphical modeling of app contents [ROS14; A N+16].

Companies such as BiznessApps [Biz16] and Bubble [Bub16] (see **??**) promise codeless creation of apps using detailed configurators and web-based user interface editors. Compared to these approaches, our work on the MAML framework abstracts from concrete interface specifications by focusing on a process-centric and platform-agnostic representation. The level of abstraction is therefore significantly higher such that modelers can focus on the logical flow of actions instead of positioning user interface (UI) elements on screen. The commercial WebRatio Mobile Platform [Ace+15] is closest to the work presented in this article by generating apps from a graphically edited model (see **??**). It uses a multi-viewpoint combination of IFML, Unified Modeling Language (UML) class diagrams, and custom notations. This negatively influences learnability and comprehensibility for non-technical users who are first introduced to these technical notations (see **??**). In contrast, we use an integrated modeling approach for specifying the sequence of process steps together with data objects and abstract UI information in a single model.

Process modeling notations Graphical notations for supporting business operations have been developed for several purposes, including domains such as business process compliance [Knu+13] and data integration [Hit17]. Specifically for mobile business applications, only few approaches exist. For example, AppInventor encourages novices to create apps by combining building blocks of a visual programming language [Wol11], and Puzzle enables visual development of mobile applications within a mobile environment itself [DP12]. As semi-graphical approach, RAPPT represents a model-driven approach that mixes a graphical DSL for process flows with a textual DSL for programming business logic [Bar+15]. Although all of them aim at simplifying the actual programming work, they are often designed for novice or experienced developers and neglect non-technical stakeholders.

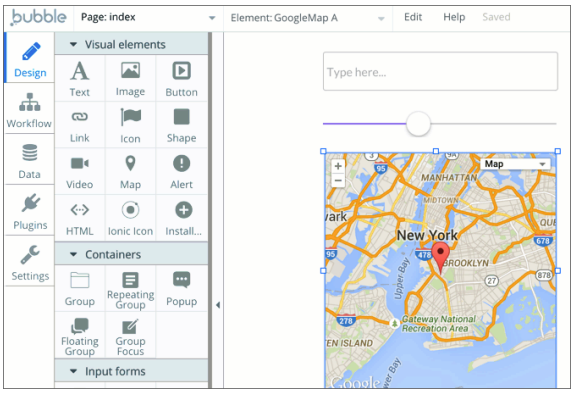


Figure 9.1: Bubble's App Configurator [Bub16]

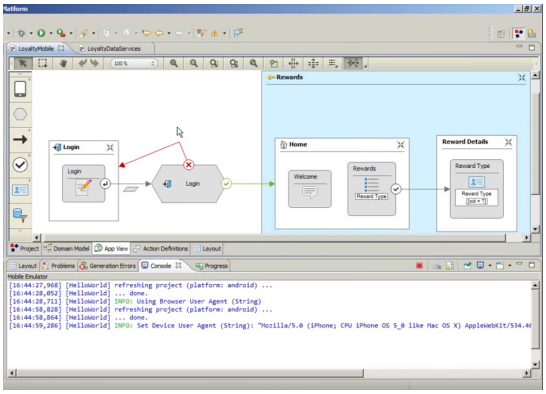


Figure 9.2: WebRatio Mobile Platform [Ace+15]

In contrast, general-purpose modeling notations exist to describe applications and process flows. The UML is the most prominent example for software developers, providing a set of standards to define the structure and runtime behavior of an application [Obj15b]. In particular, the IFML notation (succeeding WebML) represents user interactions within a software system [Obj15a]. To visualize process flows, a variety of modeling notations exist, e.g. BPMN [Obj11], Event-driven Process Chains (EPC) [van99], YAWL [vt05], or flowcharts [Int85]. As noted by Schalles [Sch13], process modeling languages dealing with behavioral aspects are generally more usable than system modeling approaches that focus on structural concepts. However, process modeling notations often lack the technical specificity required for automated processing such as app creation. For example, the YAWL notation is constructed according to workflow management patterns [vt05] but initially did not provide a data perspective and does not support the specification of task types that could be used for app view templates.

Approaches specific to the Web or mobile domain, for example providing elements for mobile-specific functionality, rarely reach beyond UI modeling. Some related work explicitly tries to incorporate non-technical users, e.g. through a subdivision of models in content, form, navigation, and workflow viewpoints [Cha+16]. Franzago et al. [FMM14] use a similar approach (separating data, UI, navigation, and business logic) for collaborative multi-viewpoint modeling. Also, a combination of multiple general-purpose notations can be used to describe all relevant application perspectives, for example in the model-driven approach by Koch et al. [Koc+08] based on multiple UML standards. Others use existing modeling notations such as statecharts [Fra+15] or extend them for mobile purposes, e.g. UML to model contextual information in mobile distributed systems [SW07], IFML with mobile-specific elements [BMU14], or BPMN to orchestrate web services [BDF09]. Yet, technical modeling notations are often considered as complex to understand for domain experts [Fra+06]. Cognitive studies have been conducted, e.g. for WebML [Gra+15], and problems such as symbol overload are aggravated by introducing multiple notations to represent the full range of data, business logic, UI, and user interaction modeling. As an example of integrating different application perspectives, Trætteberg and Krogstie [TK08] combine BPMN models with additional elements for representing the UI of desktop applications.

These works are mainly used to specify applications, not to automatically transform models in functional app source code. However, we deem process models to be a suitable level of abstraction for creating cross-platform apps with MAML.

Model aggregation Further relevant fields of research for identifying commonalities in models include model differentiation techniques and schema matching of database structures [Sut+16; EI10]. According to the classification by Rahm [RB01], a schema-only and constraint-based approach on element level is suitable for the inference of a data model from multiple input models. For the given problem of partial models, we focus on an additive and name-based approach, although more sophisticated strategies such as ontology-based approaches may later improve the uncovering of modeling inconsistencies [LGJ07]. An application related to mobile devices, but

focused on data visualization instead of its manipulation, is MobiMash for graphically creating mashup apps by configuring the representation and orchestration of data services [Cap+12].

In the context of metamodeling, reverse engineering approaches to track metamodel evolution have to cope with similar problems of inferring object structures [Liu+12] and finding correspondences from model instances [Jav+08]. López-Fernández et al. [Lóp+15] presented a related idea of building a metamodel in a bottom-up fashion from sample model fragments.

9.3 MAML Framework

The MAML framework consists of a graphical domain-specific language to specify mobile business apps as well as required tooling such as a model editor and code generators [Rie17b]. After presenting the overall design goals, these components and their interaction for seamless app generation are explained in the following.

9.3.1 Language Design Principles

The MAML DSL is built around five main principles:

Domain expert focus Besides software developers, MAML should also be usable by non-technical users with knowledge about the circumstances of the target application, including for example process modelers and domain expert with limited software engineering experience. This requirement is relevant both for the notation itself and the surrounding tooling in order to engage with MAML without steep learning curve.

Data-driven process MAML focuses on the domain of business apps. However, a process perspective is adopted by graphically describing an app as a sequence of processing steps performed on one or several data objects. This distinguishes MAML from similar approaches which provide far lower abstractions from the actual app development process.

Modularization Traditional software engineering promotes a separation of concerns with patterns such as Model-View-Controller (MVC) [Gam+95]. To accommodate for the domain experts' mindset, MAML-based apps are specified with data structures, process steps, and screen visualization combined in a single model. In addition, mobile apps are typically designed to solve small and specific tasks. Therefore, processes are modeled as *Use Case*s, units of useful functionality with a self-contained set of behaviors and interactions performed by the app user [Obj15b].

Declarative description Because of its platform-agnostic nature, MAML use cases describe *what* modifications should be performed on data objects. An imperative specification of *how* to perform the operation on a specific mobile device is not part of the model. The concrete

representation depends on the target platform and suitable defaults are automatically provided during source code generation.

Automatic cross-platform app generation An essential feature of the MAML framework is the automatic transformation of graphical models into fully-functional apps for multiple platforms. Cross-platform code generation constitutes an extensible approach to reach a large user base equipped with heterogeneous mobile devices. It is therefore mandatory that MAML models contain all technical details regarding app structure and runtime behavior in order to achieve a "zero coding" solution.

9.3.2 Language Overview

For a fictitious inventory management system, the administration of item data and stock levels represents a typical business process. In MAML, this task is modeled in a *use case* as depicted in **??**. The model contains a sequence of activities, from a *start event* (labeled with (1) in **??**) towards one or several *end events* (2). In the beginning, a *data source* (3) specifies the data type of the manipulated objects and whether they are only saved locally on the device or managed by the remote backend system. Data can then be modified through a pre-defined set of (arrow-shaped) *interaction process elements* (4), for instance to *select/create/update/display/delete entities*, show *popup messages*, or access device functionalities such as the *camera* and starting a phone *call*. It should be noted that no device-specific assumptions on the appearance of a step can be made from this platform-agnostic description of models. The code generator instead provides representations and functionalities according to the platform capabilities, e.g., display a *select entity* step using a list of all available objects as well as functionality to filter or search for specific entries. Furthermore, *automated process elements* (5) represent invisible processing steps without user interaction, e.g., interacting with RESTful *web services*, *including* other models for process reuse, or navigating through the object graph (*transform*).

The navigation between connected process steps happens using an automatically created "Continue" button. Alternative captions can be specified along the *process connectors* (6). Many workflows are nonlinear, therefore process flows can be branched out using an *XOR* element (7). To decide which path is chosen, a condition can either be evaluated automatically based on specified expressions, or let the app user decide by providing buttons for each possible process path (such as in the example).

The rectangular elements represent the data objects which are displayed within a respective process step. *Attributes* (8) consist of a cardinality indicator, a name and the respective data type. Besides pre-defined data types such as *String, Integer, Float, PhoneNumber, Location* etc., enumerations and custom types can be defined. Consequently, attributes may be nested over multiple levels in order to further describe the content of such a custom data type. *Labels* (9)

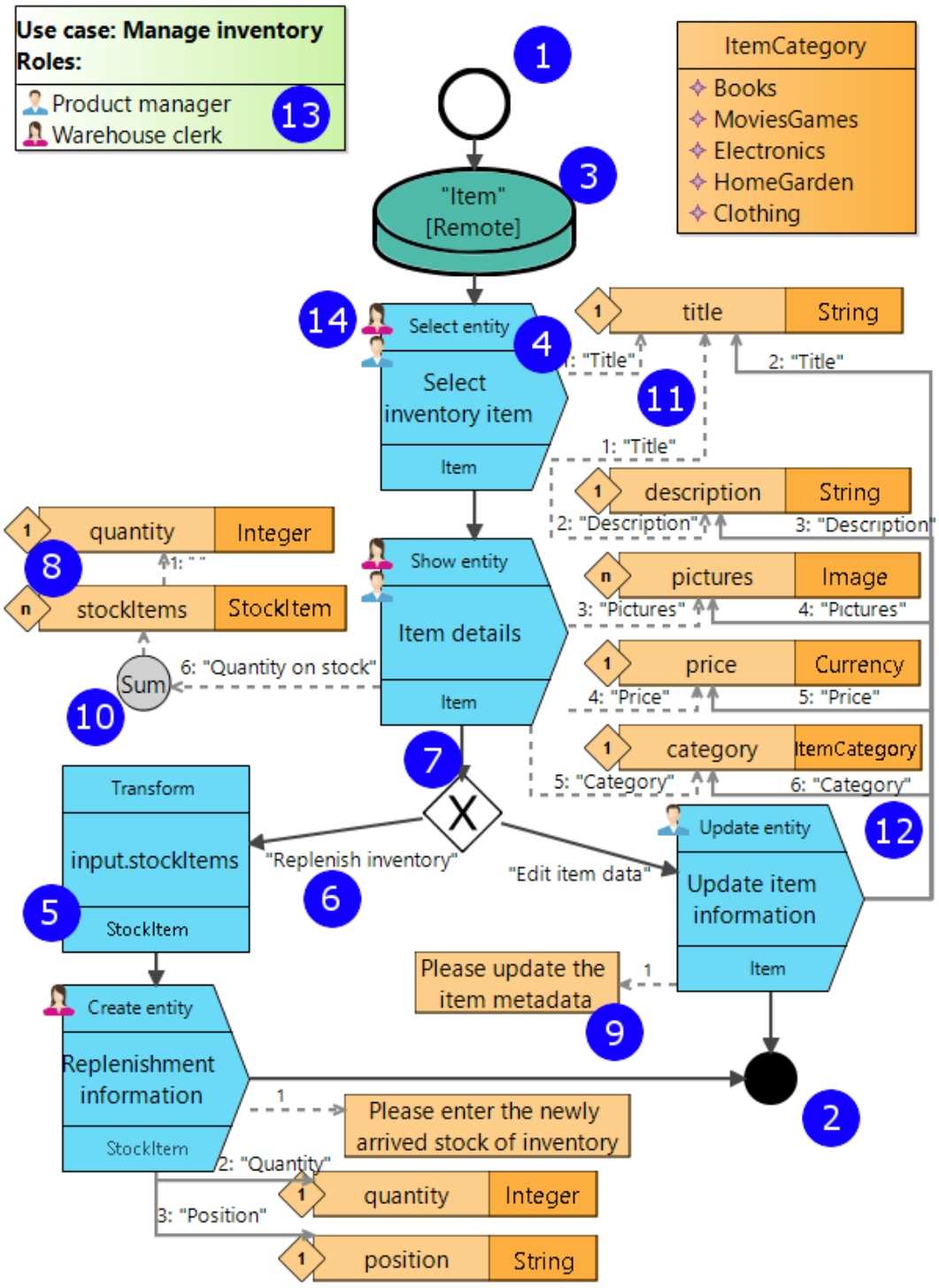


Figure 9.3: MAML Model With Sample Use Case “Manage inventory”

provide explanatory text to be displayed on screen, and *computed attributes* (10) are used to calculate derived values at runtime based on other attributes.

In MAML, only those attributes are modeled which will be used in a particular process step, for instance for including it in the graphical user interface or using its value in web service calls. Although this concept seems surprising to programmers expecting fully specified data models, it simplifies the integrated modeling of process flows and data. Attributes which are neither updated nor read do not contribute to the resulting app and are therefore ignored [1].

Two types of connectors exist for attaching data to process steps: Dotted arrows represent a *reading relationship* (11) whereas solid arrows signify a *modifying relationship* (12) with regard to the target element. This refers not only to attributes displayed either as read-only text or editable input field. The interpretation also applies in a wider sense, for instance concerning web service calls in which the server "reads" an input parameter and "modifies" information in the app through its response.

Every connector which is connected to an interaction process element also specifies an order of appearance and a field description. For convenience, a human-readable field description is derived from the attribute name by default but can be modified. To reduce the amount of elements to model, multiple connectors may point to the same UI element from different sources (given their data types match). Alternatively, to avoid confusing connections across larger models, UI elements may instead be duplicated to different positions in the model and will automatically be matched in the inference process (see **??**).

Finally, MAML supports a multi-role concept because many business processes such as approval workflows involve different people or departments. The modeler can specify arbitrary role names (13) and annotate them to the respective interaction process elements (14). If the assigned role changes, the generated app automatically terminates the process for the first app user, transfers modified data objects, and informs the subsequent user about an open workflow instance in his app.

9.3.3 Data-Model Inference

As described in the previous paragraphs, each process step within a model refers only to the attributes needed within this particular step. Besides avoiding redundancies of separately specified data models, this proceeding suits the mindset of non-technical users who focus on the task's process and consider separate data models "technical clutter". Also, due to the modular development, modelers might not even be aware of particular data fields required in other processes. Nevertheless, a global data model is a technical requirement for performing code generation. Therefore, the global view on all data structures in the resulting app is inferred from partial models on multiple levels: for each process element individually, then for the whole use case, and finally across multiple use cases[2].

[1] Such attributes may, however, be required for manually written business logic after the code generation step. In this case, respective attributes can be likewise introduced in the generated source code.

[2] However, semantic reasoning such as inferring generalization relationships and fuzzy matching techniques for identifying "similar" attributes are beyond the scope of this work.

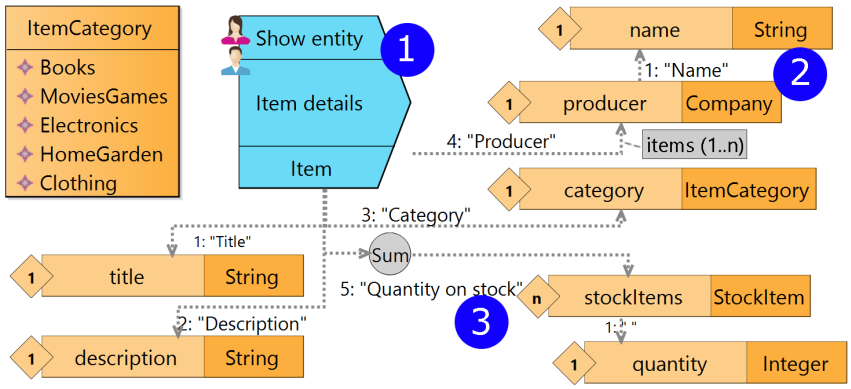


Figure 9.4: Exemplary Partial MAML Model

Partial Data-Model Inference

In MAML, each process step effectively represents a partial data model describing the main data type of the step with the connected attributes. The data model inference approach therefore constructs a directed graph structure with vertices representing data types and edges denoting attribute relationships between two types.[3] For clarification, the term *attribute* is further used to refer to the UI element in a MAML model whereas *property* signifies the corresponding member of the UML class diagram representation.

Firstly, all data types of the process element and its attached attributes are collected. **??** depicts a sample process element with the set of (indirectly) related data types *Item, Company, ItemCategory, StockItem, String*, and *Integer*. For each non-primitive data type in this set, a class is created in the partial data model (including enumerations such as *ItemCategory*). Secondly, relationships between those data types are identified. Three origins need to be considered (cf. numbered circles in **??**):

(1) A relationship may exist between a process element and an attached attribute.

(2) A nested attribute adds a relationship to the nesting attribute's data type.

(3) An attribute may be transitively connected to the process element through computed attributes, but still refers to the process element's data type.

The identified relationships are transformed into properties of the containing class by distinguishing the following four cases.

Primitive An attribute with primitive data type is converted to a single- or multi-valued property of the source data type. For the exemplary process step depicted above, *description*, *name*, *quantity* and *title*, are added as properties to the respective classes (see **??**).

[3]It should be noted that the implemented algorithm [Rie16] is more complex than sketched here for conveying the idea in a concise way.

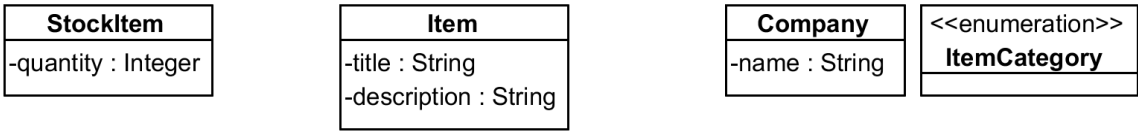


Figure 9.5: Class Diagram of Inferred Primitive Types

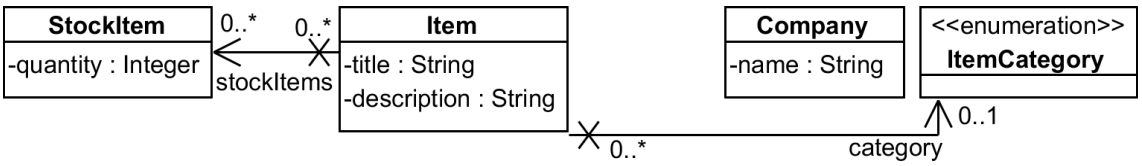


Figure 9.6: Class Diagram Including Inferred Unidirectional Relationships

Unidirectional Attribute connections are usually represented as unidirectional relationships. Because multiple connections may be present in different parts of the MAML model, the opposite direction is unknown and must be interpreted with unrestricted o..* cardinality, where $i..j = \{i, i+1, ..., j\}$ and $*$ represents infinity. For example, the *stockItems* property in **??** holds a collection of *StockItem* objects but may itself be referenced by multiple *StockItem*s in other parts of the application.

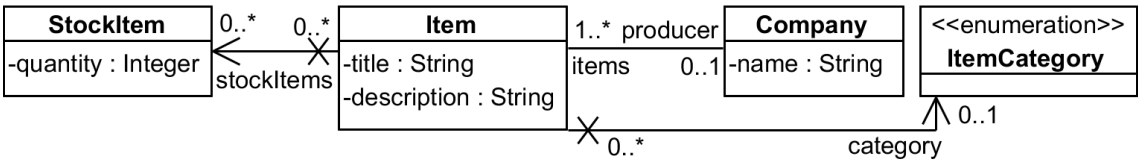


Figure 9.7: Class Diagram Including Inferred Bidirectional Relationships

Bidirectional In contrast to the previous case, explicit bidirectional relationships in the model are transformed to named properties in both classes. For instance, the *producer* attribute in **??** represents a multi-valued property and is converted to the association shown in **??**. In addition to creating access methods for consistent updates and deletions, potential cardinality restrictions need to be enforced in the generated code (e.g., forbid the removal of the last related *item* to comply with the minimum cardinality in the *company* object).

Singleton An attribute of a singleton data type (not depicted in the example) is a variant of the unidirectional scenario in which the opposite cardinality is known to be restricted to o..1 (i.e., either the property is set in the single instance of the referencing class or not) [Gam+95].

Merging Partial Data Models

After creating the graph data structure for an individual process element, the multitude of partial data models needs to be merged both for the whole process within a use case and the overall app

product consisting of multiple use cases. Because of the additive merging process, partial models can be merged iteratively in arbitrary order.

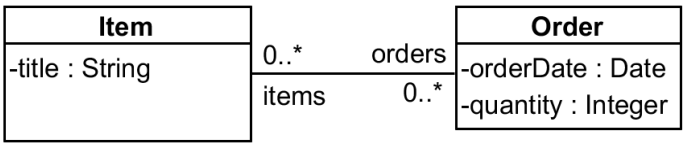


Figure 9.8: Class Diagram of a Second MAML Model

To continue the example of the previous section, the partial data model depicted in **??** is merged with another partial model depicted in **??**. In a first step, the union of both sets of data types is created such that the global data model contains the types *Item, Company, ItemCategory, StockItem, Order, String*, and *Integer*.

Next, each edge of both partial models is added to the global data model if it introduces a new name or cardinality to the respective combination of source and target data type. With this name-based matching strategy, specifying multiple associations between the same pair of data types is unproblematic and will result in the same edge of the type graph. MAML modelers are therefore free to either link to the same attribute from different origins in the model or clone the modeled attribute for better readability. For the exemplary models, this results in the complete global data model depicted in **??**.

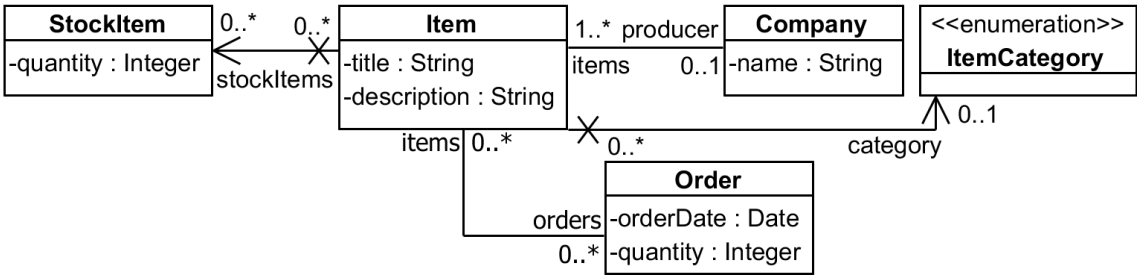


Figure 9.9: Inferred Global Data Model

However, two types of modeling errors may occur when merging these data structures as visualized in **??**:

(1) A *type error* exists if any source data type has two properties of the same name pointing to different target data types. Automatically resolving this error is impossible and the modeler needs to be informed.

(2) A *cardinality conflict* exists if two properties with the same name differ with regard to their cardinality for any pair of data types. In this case, the modeler should be warned, but automatic resolution is possible.

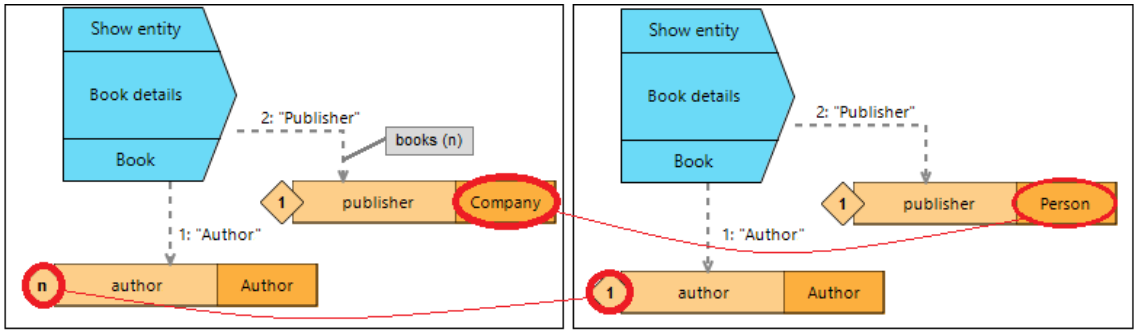


Figure 9.10: Visualization of Modeling Errors [Rie16]

In case of cardinality conflicts, the resulting cardinality for each side of the association is calculated as the union between the conflicting cardinalities (ignoring not navigable ends) to avoid invalidating existing data. Regarding the inferred directionality, the merged association is bidirectional if any of the affected relationships is modeled to be bidirectional. In the example class diagram depicted in **??**, the cardinality $0..1 \cup 0..* = 0..*$ is assigned to b and $1..*$ is assigned to a (as $0..*$ is not navigable in this direction).

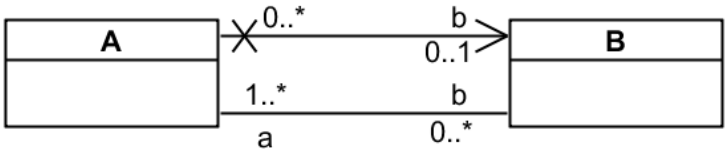


Figure 9.11: Exemplary Model With Cardinality Error

To sum up, this inference algorithm can be applied to partial models of different granularity in order to derive a global data schema and simultaneously validate the individual models.

9.3.4 Modeling support

The developed editor for MAML is based on Eclipse Sirius and the underlying Eclipse Modeling Framework [The16b]. Beyond the structural constrains imposed by the DSL's metamodel, semantic

restrictions need to be considered for creating valid models [SBS14]. Language workbenches for developing DSLs usually provide additional features to help end users model "correct" models [Erd+13]. For example, in the Xtext framework for developing textual DSLs, *validators* can be defined to programmatically apply rules to model elements and *scope providers* allow for context-sensitive filtering of references [Bet13]. Eclipse Sirius, as generic tool for graphical editor development, could be called an "editor workbench" and provides similar mechanisms for controlling the modeled result. These features can be classified into *validation* of the model itself and *semantic services* of the editor [Erd+13].

Supporting Model Validation

Different systems for categorizing the severity of faults are commonly used in software development, usually comprising a fatal error state, one or more levels of error severity regarding the implications on functionality, and one level of trivial issues [Zha+14; Guj+15]. Using the aforementioned data model inference approach, the developed tool support for MAML includes rules on corresponding levels of severity.

Error Blockers with regard to model editors signify the inability to execute or process the model. In addition to the graph structure which describes the partial data models of **??**, the respective *attribute* in the MAML model is also tracked. Unresolvable type errors identified by the merge algorithm (cf. **??**) can therefore be traced back to all affected model elements. This allows for a visual feedback of semantic discrepancies directly within the model.

Warning Without further differentiating severity levels, warnings can be regarded as subsuming all model inaccuracies that are not technically incorrect but influence the processing result in probably unintended ways. For example, the self-resolved cardinality conflict in **??** might have unintended consequences in views that initially expected single-valued elements. Further, omissions such as defining but not assigning roles are considered as unintentional error. Obviously, the modeler also needs to be notified about warnings, ideally presenting possible alternatives to repair the modeling mistake.

Hint Trivial issues do not affect the model outcome but are likewise displayed next to the respective model element in order to inform the user about potentially unwanted effects or consistency issues. In MAML, naming conventions, e.g., lowercasing attribute names, can seamlessly be integrated as hints in the editor.

Providing Semantic Services

In addition to validating models, global data model inference allows for semantic support within and across MAML models.

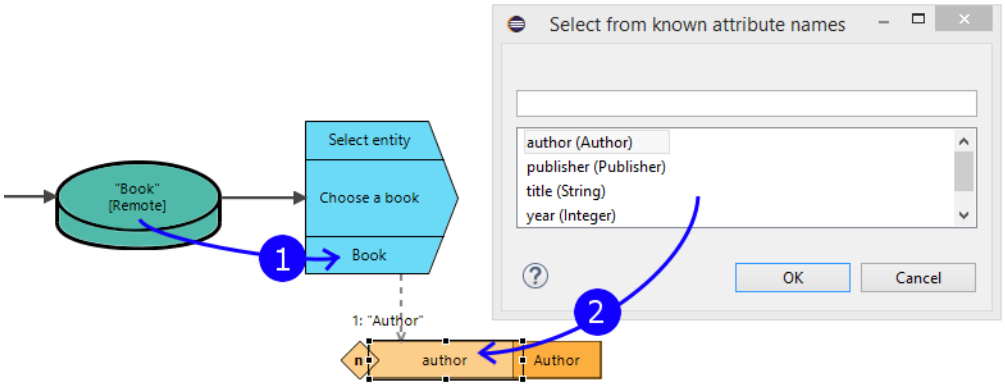


Figure 9.12: Examples for Semantic Editing Services

Continuous semantic assessment The best level of modeling support is preventing the appearance of erroneous conditions in the first place. Invisible to the user, semantic discrepancies are checked when modifying or adding elements in the model. Because of the inferred knowledge, introducing semantic errors such as attaching incorrect data types is denied – similar to scope providers that provide a context-dependent set of valid elements. Also, textual expressions for specifying branching conditions of *XOR* elements or navigating through the data structure in a *Transform* element (cf. **??**) are evaluated at runtime. Beyond the syntactic correctness, the correspondence to elements in the data model is immediately checked.

This real-time validation capability is also important for another type of continuous modeling support built into MAML: At the bottom of each process element, the data type of the manipulated objects is automatically inserted as mental assistance and additional guidance for non-experienced modelers (labeled (1) in **??**). Whenever process flow elements are (re-)connected, the validity of the process chain is checked. For example, merging multiple branches that operate on different data types is automatically disallowed.

Context-sensitive suggestions Finally, the inference results are used to reduce modeling effort similar to intelligent code completion features known from textual editors. For instance, matching attributes are suggested for process elements in case the source data type is known from other steps in the model or different use cases within the same app (see (2) in **??**). Similarly, matching data types and cardinalities can automatically be completed for existing attributes.

To sum up, using the data inference mechanism not only prepares the MAML model for automatic code generation as described in the following section. It also strengthens the model's consistency while at the same time reducing the overall modeling effort. The graphical modeling environment thus evolves beyond the stage of just providing the infrastructure for modeling "shapes filled with text" into a context-aware tool that promptly reevaluates the semantic validity while models are built or modified.

9.3.5 App generation

One of the distinguishing features of the MAML framework is its automatic generation of apps for multiple target platforms in order to eliminate the need for manual programming. MAML relies on a model-driven software development approach and the graphical models are used as sole input for the app generation.

The current approach is based on previous work in the domain of business apps, notably the textual DSL MD² (see **??**). Although both languages share common overall concepts, MD² is programming-oriented and follows a MVC-layered approach instead of the presented separation by functionality. For further details on MD² language features the reader is referred to [HM13; ME15].

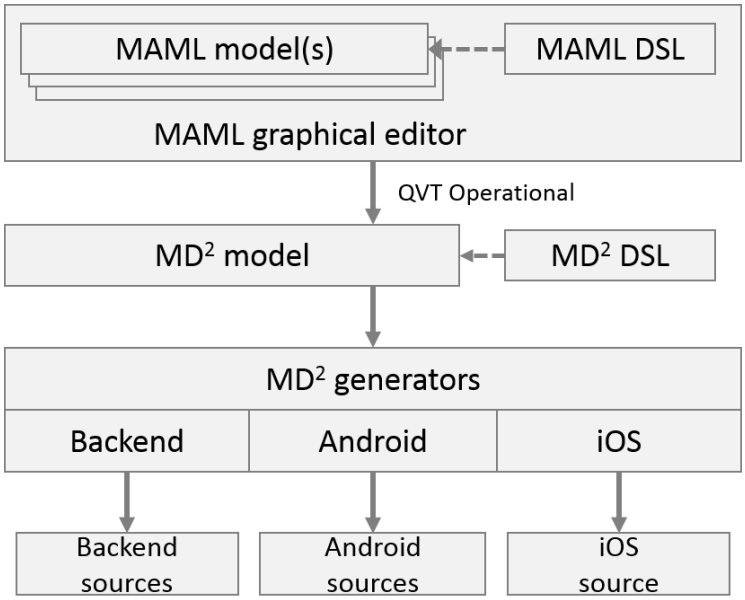


Figure 9.13: MAML Generation Process [Rie17a]

A two-step generation process was established (cf. **??**): First, a model-to-model transformation translates all related MAML model instances to one MD² model instance. This transformation is based on mappings between the metamodels of MAML and MD², specified using the QVT Operational framework [The16a]. Because both DSLs are designed for the same domain of business apps, many concepts such as roles and process flows are present in both notations and can be transformed unambiguously. In addition, there are MD² concepts such as content providers or view layouts which have no correspondence because MAML operates on a higher level of abstraction. If mandatory for MD² models, default representations are created for these concepts during transformation.

Second, existing MD² generators perform the model-to-code transformation and output the app source code for the target platforms, in particular Java code for Android and Swift code for

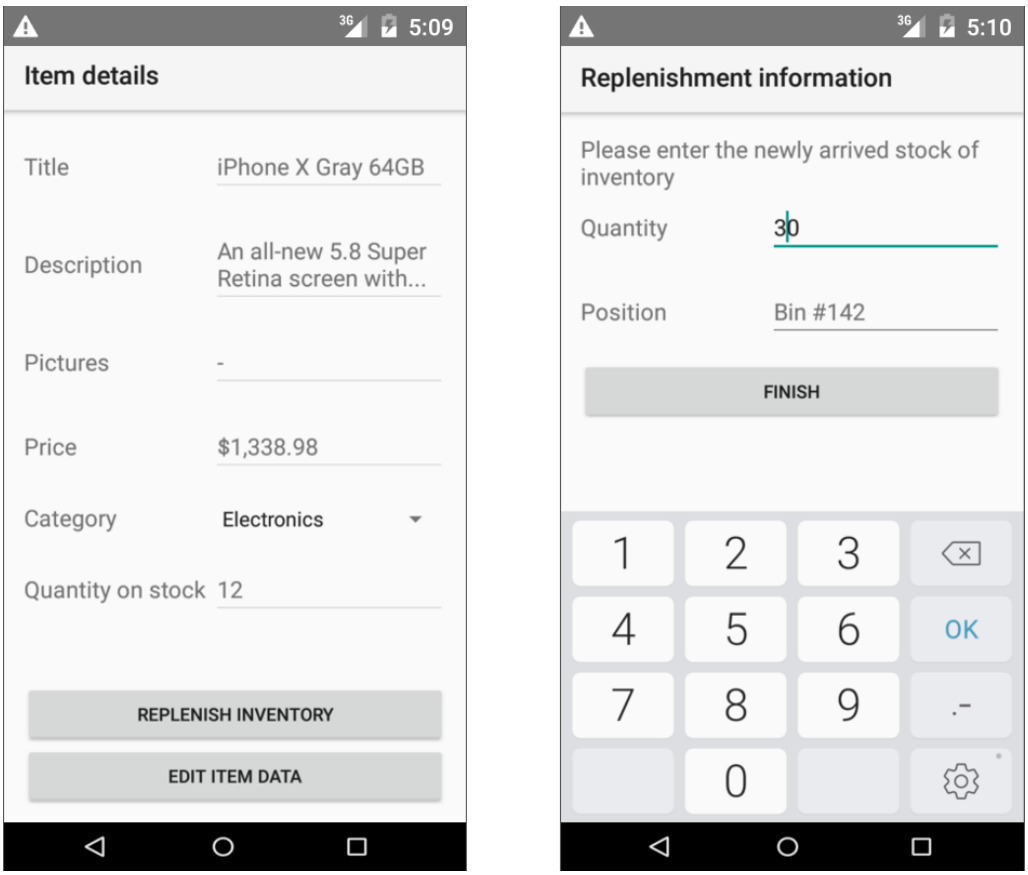


Figure 9.14: Generated Android App Screenshots

iOS. ?? presents exemplary screenshots of the resulting Android app for the first process steps shown in ??.

9.4 Evaluation and Discussion

With regard to RQ1 and RQ2, the design of MAML shows the possibility of replacing the software engineering focused separation of concerns in mobile app models by a business perspective using functional subdivision. The MAML framework also aims for zero-code generation of cross-platform apps from the graphical model, thus requiring a minimum of technical specificity to be interpretable by the data model inference process and subsequent generators. In this context, business processes are seen as reasonable level of abstraction. It is, however, not intended to create yet another workflow management system with the presented framework. Currently, variability in the process flow is deliberately limited to XOR elements and deemed sufficient due to the sequential display on smartphone screens.

There are other process modeling languages such as EPC with a high degree of usability [Sch13] but which rely on humans to interpret the meaning of models. Yet, a rigid representation of

technicalities such as information propagation through parameter declarations might alienate potential users with limited experience in software development. Finding a suitable trade-off between technical detailedness and visual simplicity [Sch13] was therefore essential for designing MAML's modeling notation. This conflict can be alleviated by embedding domain knowledge and aligning with existing standard notations [MHS05]. With regard to modeling complexity, the MAML DSL is positioned between BPMN 2.0 and IFML.

However, a platform-agnostic notation which is at the same time designed for mobile apps is not a contradiction. Conceptual differences exist both with regard to the characteristics of mobile platforms in general as well as the usage context. Regarding mobile specifics, the platforms are characterized by a heterogeneity of input and output mechanisms even among devices of the same platform – much more than desktop- or browser-based applications which rely on established mouse and keyboard inputs. For instance, smartphones typically provide capacitive displays but at the same time support different modes of touch input (e.g., multi-touch gestures or pressure-sensitive "3D touch" on Apple devices[4]) and can be controlled using additional hardware buttons or voice commands [RM17]. Device capabilities differ with regard to sensors such as gyroscopes for motion detection, determination of position via GPS, or front/back cameras. Also, seamless interaction with operating system functionality such as starting phone calls is possible [San+14]. Moreover, the usage behavior of mobile apps is reflected in the design of the DSL. For example, splitting apps by use cases and modeling workflow and data perspectives in a combined model is feasible for the scope of mobile processes but becomes unpractical for enterprise workflows such as production processes. The suitability of partial data models might be questioned in large-scale models, but with regard to app development, fully specified data models are often not pre-existing and need to be newly created. Although the notation can of course be used in non-mobile development scenarios which make use of MAML's independence from concrete UI representations, these mobile specifics were considered during its development in order to create an understandable notation primarily for describing mobile apps.

Compared to other modeling notations, MAML models have the advantage of being self-contained and do not rely on a synchronization with external information. In contrast, IFML models are connected to other UML standards and require multiple models for data, business logic, and user interaction to be interpreted together. In addition, all pieces of information are presented in an integrated model. MAML models can therefore also be seen as a means of communication, facilitating the discussion between involved stakeholders concerning the structure, interactions, and data of an app product. We do not claim that this design is better under all circumstances. However, we argue that the task-oriented approach chosen for MAML is reasonable for the purpose of small-scale processes performed in apps.

With regard to a comparable scope, IFML is the most similar approach to ours and can be compared to gain insights regarding RQ4. Although BPMN and further simplified notations [Mar+11]

[4] https://developer.apple.com/ios/3d-touch/

seem more usable at first sight, they lack significant capabilities to represent all perspectives of app development. Also, a notation with more elements does not necessarily lead to a degradation of its usability [Sch13]. To compare IFML with MAML, a qualitative, observational study was performed with 26 (mostly student) participants in individual sessions of 90 minutes duration. None of the participants had previous knowledge of either modeling language.

The first part of the study evaluated the readability and understandability of each notation. Therefore, one MAML and one IFML model, both depicting similar scenarios (cf. online material [Rie17b]), were shown to the participants (in random order to avoid bias) without prior introduction to the notation. Additionally, a System Usability Scale (SUS) questionnaire was answered for each notation, comprising ten questions on a 5-point scale between strong disagreement and strong agreement. The result is obtained by converting and scaling the responses according to the method presented by Brooke [Bro96].

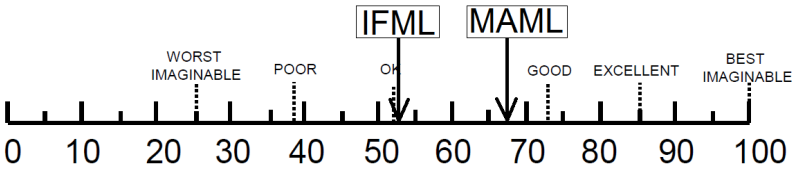


Figure 9.15: SUS Ratings for IFML and MAML [Rie17a]

Overall, MAML surpasses IFML with 66.8 to 52.8 points (of maximum 100) regarding its readability without prior knowledge. It should be noted that this score does not represent a percentage value but can instead be interpreted according to the adjective rating scale depicted in **??** as proposed by Bangor et al. [BKM09].

To gain additional insights, the participants were clustered in the three groups of software developers (11), process modelers (9), and domain experts (6), according to their personal background in programming and process modeling. Although differences can be observed, all groups rated MAML better, as summarized by **??**.

Table 9.2: SUS Scores by Participant Group

Group	**IFML score (σ)**	**MAML score (σ)**
Software developers	45.91 (23.6)	64.09 (17.3)
Process modelers	64.17 (19.0)	69.44 (12.0)
Domain experts	48.33 (24.5)	67.92 (18.7)
Overall	52.79 (23.0)	66.83 (15.6)

In more detail, aspects regarding the ease of understanding and complexity of the notation scored better for MAML, particularly for the group of domain experts without knowledge of programming or process models. This reflects the observations in which seven participants

emphasized MAML's simplicity and reduction of "technical clutter". Interestingly, even software developers generally preferred MAML's process-oriented approach. Only two participants critically noted the absence of explicit data models, supporting the idea of integrated process modeling. The comparatively low baseline for the group of software developers might result from a general disinclination to specify models for software products due to the complexity of technical notations, supplementary effort to the "actual" development, or experienced problems of keeping both artifacts synchronized over time [Pet13]. Because of its model-driven approach that considers source code as derived artifact, these issues do not apply to MAML.

Data model inference as demonstrated in MAML alleviates three problems of graphical modeling: First, the modeler is disburdened of separately maintaining a redundant data model. On-the-fly inference of the underlying data model is particularly helpful for process modelers and domain experts without programming experience to quickly get started with modeling the actual app behavior. The study confirms this, as the inference mechanism and enhanced modeling support was welcomed as helpful guidance during the sessions, and scores favor MAML regarding consistency and explanatory power (RQ4).

Second, the availability of fine-grained partial data models (per process flow element) allows for improved security. Generators may tailor app-specific data models by splitting the global data model according to the required fields. This not only reduces the volume of transferred data but allows for automatically generated validation mechanisms to control data access permissions.

Third, improved modeling support as described in **??** is only possible if interrelations between elements are accessible to the editor component (RQ3). Many graphical editors focus on the usability of placing and connecting elements on the canvas but lack the ability to provide context-sensitive modeling support similar to integrated development environments for textual programming languages. Further sophisticated inference techniques relying on ontology-based matching [Noy04] may be extensions to further improve model consistency.

When comparing the answers of domain experts and technical users (developers and process modelers together) for the SUS questionnaire (see **??**; answers rescaled to an [0;4] interval), the contrasting approaches of using a technical (IFML) or a process-oriented (MAML) separation of models become apparent. Of course, a net effect of modeling support through the inference mechanism cannot be measured as it is intertwined with the graphical syntax of the notation itself (e.g., inferred type hints for process elements). However, responses are significantly higher for domain experts assessing whether the MAML notation is wieldy usable (+1.17 compared to IFML), fast to learn (+1.17), and self-descriptive (+1.00). All three aspects relate to the ability to understand and apply the notation in modeling tasks. For technical users, the largest differences are found in the self-learnability (+1.05), perceived consistency (+0.80), and pace of learning (+0.70), also indicating that the process-oriented modeling approach does not harm their technical understanding but is less essential than the emphasis on correctly applying the notation as expected in RQ3.

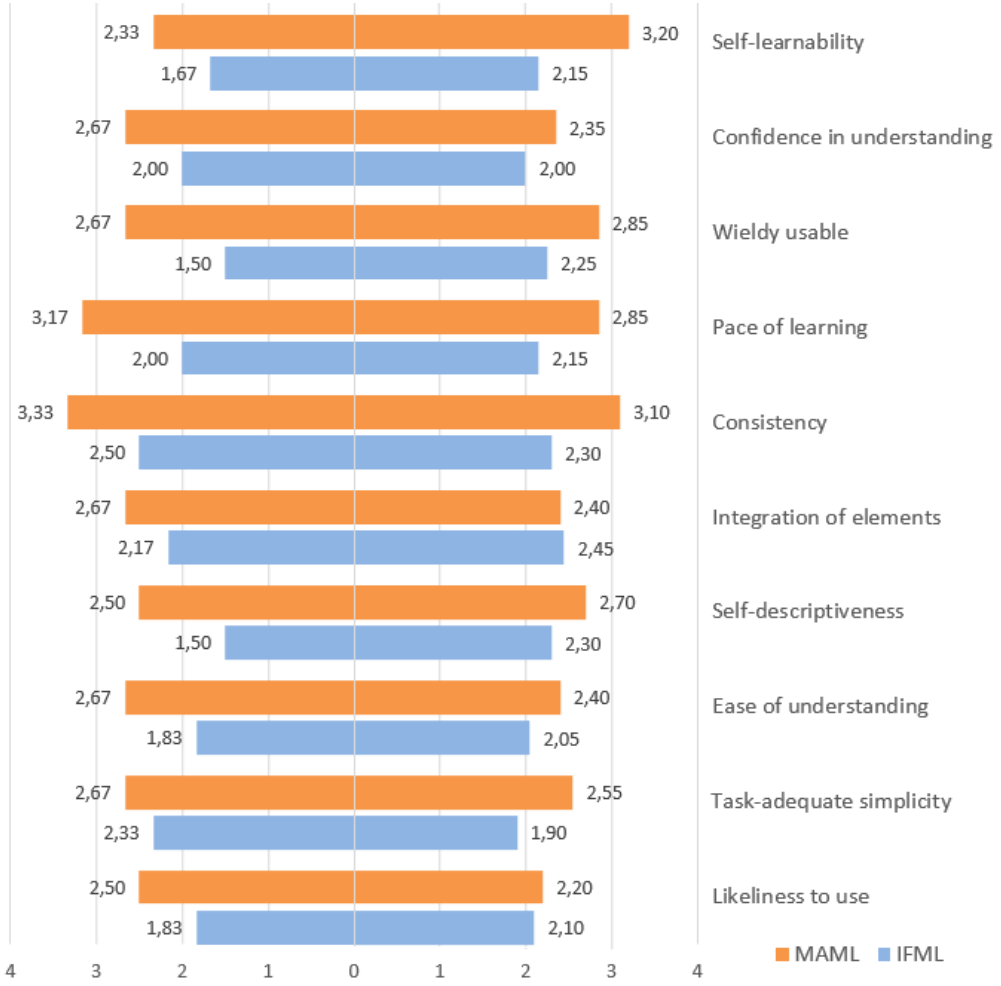


Figure 9.16: SUS Answers for Domain Experts (left) and Technical Users (right)

Regarding app generation, all steps can be executed without additional configuration. The intermediate transformation step is however no inherent limitation of the framework. Future generators targeting new platforms may just as well generate code based on MAML models directly. Apart from reusing existing MD² generators developed in the last years, the intermediate transformation adds a technical representation with detailed possibilities of configuration such as UI element styling [MEK15]. Users accustomed to MD² may therefore adapt the default representations in the created textual model according to their needs.

The usability study also revealed potentials for further refinement: Although modeling activities were mostly performed correctly, the lack of a screen-oriented representation similar to IFML's nested element structure was mentioned by four participants. A generic and read-only preview of a possible outcome on screen would improve this issue. Also, the wording of some elements proved to be not clear enough. For instance, some data type names are hard to understand for domain experts without further explanation and may be replaced by symbols for a more intuitive understanding.

9.5 Conclusion and Outlook

In this article, the MAML framework was proposed to model mobile apps using a declarative graphical DSL. In contrast to the current practice of graphically configuring user interfaces by positioning UI elements on a screen-like canvas, MAML focuses on a process-centric definition of business apps. Using a sequence of platform-agnostic process elements, the notation aligns with the business perspective of managing processes and data flows, and makes app development accessible to domain users without software engineering experience. The approach is based on existing work on cross-platform business app generation and uses model-driven techniques to transform the models first into an intermediate textual representation before generating platform-specific source code. In particular, a data model inference mechanism was presented that enables real-time validation and consistency checks on partial data models, and overcomes the need for explicitly modeling a global data schema. Moreover, the inferred data can be used to provide contextual modeling support and enhanced semantic validation in the editor component. An empirical evaluation study supports the advantage of MAML over the related technical IFML notation, specifically with regard to its readability by domain experts. MAML therefore achieves the desired balance of abstracting programming-heavy tasks to understandable process flows while keeping the technical expressiveness required for automatic source code generation for multiple target platforms.

The study results support the benefit of MAML and the participants can be seen as realistic sample for app-experienced adults in the general workforce. Still, the amount of student participants may pose a threat to validity and more extensive studies are necessary to confirm these results. Applying the prototype to real-world use cases might reveal further need for improvements and at the same time constitutes future work. Programmers may want to deviate from the default configuration used for automatic app generation, for example by performing manual changes to the intermediate representation. Further research needs to be done on how to avoid interference of generated and custom DSL content in the context of iterative development with frequent re-generation cycles.

Finally, the principles of MAML can be applied to mobile devices beyond smartphones and tablets. Regarding the emergence of novel app-enabled devices such as smartwatches, interesting questions arise on transferring model-driven development approaches to different device classes. Ideally, best practices can be found for reusing the same input models for generating mobile business apps on heterogeneous devices with different capabilities and user interaction patterns.

References

[A N+16] A. Namoun et al. “Exploring Mobile End User Development: Existing Use and Design Factors”. In: *IEEE Transactions on Software Engineering* 42.10 (2016), pp. 960–976. DOI: 10.1109/TSE.2016.2532873.

[Ace+15] Roberto Acerbis et al. “Model-driven Development of Cross-platform Mobile Applications with WebRatio and IFML”. In: *International Conference on Mobile Software Engineering and Systems*. MOBILESoft. IEEE, 2015, pp. 170–171. DOI: 10.1109/MobileSoft.2015.49.

[Apa16] Apache Software Foundation. *Apache Cordova Documentation*. 2016. URL: https://cordova.apache.org/docs/en/latest/guide/overview/.

[Bar+15] Scott Barnett et al. “A Multi-view Framework for Generating Mobile Apps”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 305–306. DOI: 10.1109/VLHCC.2015.7357239.

[BDF09] M. Brambilla, M. Dosmi, and P. Fraternali. “Model-driven engineering of service orchestrations”. In: *World Congress on Services (SERVICES)* (2009). DOI: 10.1109/SERVICES-I.2009.94.

[Bet13] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Community experience distilled. Birmingham, UK: Packt Pub, 2013.

[Biz16] Bizness Apps. *Mobile App Maker – Bizness Apps*. 2016. URL: http://biznessapps.com/.

[BKF14] R. Breu, A. Kuntzmann-Combelles, and M. Felderer. “New Perspectives on Software Quality”. In: *IEEE Software* 31.1 (2014), pp. 32–38. DOI: 10.1109/MS.2014.9.

[BKM09] Aaron Bangor, Philip Kortum, and James Miller. “Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale”. In: *J. Usability Studies* 4.3 (2009), pp. 114–123.

[BMU14] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. “Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End”. In: *Lecture Notes in Computer Science* 8640 (2014), pp. 176–191. DOI: 10.1007/978-3-319-10359-4_15.

[Bro96] John Brooke. “SUS - A quick and dirty usability scale”. In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7.

[Bub16] Bubble Group. *Bubble - Visual Programming*. 2016. URL: https://bubble.is/.

[Buc12] T. Buchmann. “Valkyrie: A UML-based Model-driven Environment for Model-driven Software Engineering”. In: *International Conference on Software Paradigm Trends (ICSOFT)*. 2012, pp. 147–157. DOI: 10.5220/0004027401470157.

[CAB14] S. Charkaoui, Z. Adraoui, and E. H. Benlahmar. “Cross-platform mobile development approaches”. In: *Colloquium in Information Science and Technology, CIST* (2014), pp. 188–191. DOI: 10.1109/CIST.2014.7016616.

[Cap+12] Cinzia Cappiello et al. “MobiMash: End User Development for Mobile Mashups”. In: *Annual Conference on World Wide Web Companion (WWW)* (2012), pp. 473–474. DOI: 10.1145/2187980.2188083.

[Cha+16] Moharram Challenger et al., eds. *A Model-Driven Engineering Technique for Developing Composite Content Applications: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany*. 2016. DOI: 10.4230/OASIcs.SLATE.2016.11.

[Cha+17] Sanchit Chadha et al. "Facilitating the development of cross-platform software via automated code synthesis from web-based programming resources". In: *Computer Languages, Systems & Structures* 48 (2017), pp. 3–19. DOI: 10.1016/j.cl.2016.08.005.

[DP12] Jose Danado and Fabio Paternò. "Puzzle: A Visual-Based Environment for End User Development in Touch-Based Mobile Phones". In: *Human-Centered Software Engineering (HCSE)*. Ed. by Marco Winckler, Peter Forbrig, and Regina Bernhaupt. Springer, 2012, pp. 199–216. DOI: 10.1007/978-3-642-34347-6_12.

[EI10] Hidekazu Enjo and Junichi Iijima. "Towards Class Diagram Algebra for Composing Data Models". In: *Conference on New Trends in Software Methodologies, Tools and Techniques (SoMeT)*. IOS Press, 2010, pp. 112–133.

[El +] Amine El Kouhen et al. *Evaluation of Modeling Tools Adaptation*. URL: https://hal.archives-ouvertes.fr/hal-00706701.

[El-+15] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* (2015). DOI: 10.1016/j.asej.2015.08.004.

[Erd+13] S. Erdweg et al. "The state of the art in language workbenches: Conclusions from the language workbench challenge". In: *Lecture Notes in Computer Science* 8225 LNCS (2013), pp. 197–217. DOI: 10.1007/978-3-319-02654-1_11.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *International Conference on Web Information Systems and Technologies (WEBIST)* (2016), pp. 307–316. DOI: 10.5220/0005862103070316.

[Esp17] David Esperalta. *DecSoft - App Builder*. 2017. URL: https://www.davidesperalta.com/appbuilder.

[FMM14] Mirco Franzago, Henry Muccini, and Ivano Malavolta. "Towards a Collaborative Framework for the Design and Development of Data-intensive Mobile Applications". In: *International Conference on Mobile Software Engineering and Systems*. MOBILESoft. ACM, 2014, pp. 58–61. DOI: 10.1145/2593902.2593917.

[Fra+06] R. B. France et al. "Model-Driven Development Using UML 2.0: Promises and Pitfalls". In: *Computer* 39.2 (2006), pp. 59–66. DOI: 10.1109/MC.2006.65.

[Fra+15] Rita Francese et al. "Model-Driven Development for Multi-platform Mobile Applications". In: *Product-Focused Software Process Improvement (PROFES)*. Ed. by Pekka Abrahamsson et al. Springer, 2015, pp. 61–67. DOI: 10.1007/978-3-319-26844-6_5.

[Gam+95] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1995.

[Gra+15] David Granada et al. "Analysing the cognitive effectiveness of the WebML visual notation". In: *Software & Systems Modeling* (2015). DOI: 10.1007/s10270-014-0447-8.

[Guj+15] S. Gujral et al. "Classifying bug severity using dictionary based approach". In: *International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*. Feb. 2015, pp. 599–602. DOI: 10.1109/ABLAZE.2015.7154933.

[Hit17] Hitachi Vantara Corp. *Data Integration - Kettle*. 2017. URL: http://pentaho.com/product/data-integration.

[HM13] H. Heitkötter and T. A. Majchrzak. "Cross-Platform Development of Business Apps with MD^2". In: *Intl. Conf. on Design Science at the Intersection of Physical and Virtual Design (DESRIST)*. Vol. 7939. LNBIP. Springer, 2013, pp. 405–411. DOI: 10.1007/978-3-642-38827-9_29.

[HV11] Zef Hemel and Eelco Visser. "Declaratively Programming the Mobile Web with Mobl". In: *International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA. ACM, 2011, pp. 695–712. DOI: 10.1145/2048066.2048121.

[Int85] International Organization for Standardization. *ISO 5807:1985*. 1985.

[Jav+08] Faizan Javed et al. "MARS: A metamodel recovery system using grammar inference". In: *Information and Software Technology* 50.9-10 (2008), pp. 948–968. DOI: 10.1016/j.infsof.2007.08.003.

[JJ15] Christopher Jones and Xiaoping Jia. "Using a Domain Specific Language for Lightweight Model-Driven Development". In: *ENASE 2014*. 2015, pp. 46–62. DOI: 10.1007/978-3-642-23391-3.

[KMC12] Tomaž Kosar, Marjan Mernik, and Jeffrey C. Carver. "Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments". In: *Empirical Software Engineering* 17.3 (2012), pp. 276–304. DOI: 10.1007/s10664-011-9172-x.

[Knu+13] David Knuplesch et al. "Visual Modeling of Business Process Compliance Rules with the Support of Multiple Perspectives". In: *Conceptual Modeling (ER)*. Ed. by Wilfred Ng, Veda C. Storey, and Juan C. Trujillo. Springer, 2013, pp. 106–120. DOI: 10.1007/978-3-642-41924-9_10.

[Koc+08] Nora Koch et al. "UML-Based Web Engineering". In: *Web Engineering: Modelling and Implementing Web Applications*. Ed. by Gustavo Rossi et al. Human-Computer Interaction Series. Springer, 2008, pp. 157–191. DOI: 10.1007/978-1-84628-923-1_7.

[Kol+17] Dimitrios S. Kolovos et al. "Eugenia: Towards disciplined and automated development of GMF-based graphical model editors". In: *Software & Systems Modeling* 16.1 (2017), pp. 229–255. DOI: 10.1007/s10270-015-0455-3.

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. "DSMDiff: a differentiation tool for domain-specific models". In: *European Journal of Information Systems* 16.4 (2007), pp. 349–361. DOI: 10.1057/palgrave.ejis.3000685.

[Liu+12] Qichao Liu et al. "Application of Metamodel Inference with Large-Scale Metamodels". In: *International Journal of Software and Informatics* 6.2 (2012), pp. 201–231.

[Lóp+15] Jesús J. López-Fernández et al. "Example-driven meta-model development". In: *Software & Systems Modeling* 14.4 (2015), pp. 1323–1347. DOI: 10.1007/s10270-013-0392-y.

[Mar+11] J. S. Martínez et al. "Isastur Modeler: A tool for BPMN MUSIM". In: *Iberian Conference on Information Systems and Technologies (CISTI)*. June 2011, pp. 1–6.

[ME15] T. A. Majchrzak and J. Ernsting. "Reengineering an Approach to Model-Driven Development of Business Apps". In: *SIGSAND/PLAIS EuroSymposium*. 2015, pp. 15–31. DOI: 10.1007/978-3-319-24366-5_2.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14. DOI: 10.19210/OJIS_2015v2i2n02_Majchrzak.

[Mel+16] Santiago Meliá et al. "Comparison of a textual versus a graphical notation for the maintainability of MDE domain models: An empirical pilot study". In: *Software Quality Journal* 24.3 (2016), pp. 709–735. DOI: 10.1007/s11219-015-9299-x.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[Noy04] N. F. Noy. "Semantic integration: A survey of ontology-based approaches". In: *SIGMOD Record* 33.4 (2004), pp. 65–70. DOI: 10.1145/1041410.1041421.

[Obj11] Object Management Group. *Business Process Model and Notation 2.0*. 2011.

[Obj15a] Object Management Group. *Interaction Flow Modeling Language 1.0*. 2015.

[Obj15b] Object Management Group. *Unified Modeling Language 2.5*. 2015.

[Pet13] Marian Petre. "UML in Practice". In: *International Conference on Software Engineering*. ICSE. IEEE, 2013, pp. 722–731. DOI: 10.1109/ICSE.2013.6606618.

[PSC12] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. "Comparison of cross-platform mobile development tools". In: *16th International Conference on Intelligence in Next Generation Networks (ICIN)*. 2012, pp. 179–186. DOI: 10.1109/ICIN.2012.6376023.

[RB01] E. Rahm and P. A. Bernstein. "A survey of approaches to automatic schema matching". In: *VLDB Journal* 10.4 (2001), pp. 334–350. DOI: 10.1007/s007780100057.

[Rie16] Christoph Rieger. "A Data Model Inference Algorithm for Schemaless Process Modelling". In: *Working Papers, European Research Center for Information Systems No. 29*. Ed. by Jörg Becker et al. ERCIS, University of Münster, 2016, pp. 1–17.

[Rie17a] Christoph Rieger. "Business Apps with MAML: A Model-driven Approach to Process-oriented Mobile App Development". In: *Symposium on Applied Computing*. SAC. Marrakech, Morocco: ACM, 2017, pp. 1599–1606. DOI: 10.1145/3019612.3019746.

[Rie17b] Christoph Rieger. *MAML Respository*. https://github.com/wwu-pi/maml. 2017.

[RK18] Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *Computer Languages, Systems & Structures* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001.

[RM17] Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-enabled Devices". In: *International Conference on Web Information Systems and Technologies (WEBIST)*. 2017, pp. 332–339.

[ROS14] Jean Michel Rouly, Jonathan D. Orbeck, and Eugene Syriani. "Usability and Suitability Survey of Features in Visual IDEs for Non-Programmers". In: *Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU. ACM, 2014, pp. 31–42. DOI: 10.1145/2688204.2688207.

[Rv14] Janessa Rivera and Rob van der Meulen. *Gartner Says By 2018, More Than 50 Percent of Users Will Use a Tablet or Smartphone First for All Online Activities*. 2014. URL: http://www.gartner.com/newsroom/id/2939217.

[San+14] Zohreh Sanaei et al. "Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges". In: *IEEE Communications Surveys & Tutorials* 16.1 (2014), pp. 369–392. DOI: 10.1109/SURV.2013.050113.00090.

[SBS14] Martin Smuts, Clayton Burger, and Brenda Scholtz. "Composite, Real-time Validation for Business Process Modelling". In: *The Southern African Institute for Computer Scientist and Information Technologists Annual Conference*. Ed. by C. de Villiers et al. 2014, pp. 93–103. DOI: 10.1145/2664591.2664603.

[Sch13] Christian Schalles. *Usability evaluation of modeling languages*. Springer Fachmedien Wiesbaden, 2013. DOI: 10.1007/978-3-658-00051-6.

[SIK15] Safdar Aqeel Safdar, Muhammad Zohaib Iqbal, and Muhammad Uzair Khan. "Empirical Evaluation of UML Modeling Tools–A Controlled Experiment". In: *Modelling Foundations and Applications*. Ed. by Gabriele Taentzer and Francis Bordeleau. Vol. 9153. Lecture Notes in Computer Science. Springer, 2015, pp. 33–44. DOI: 10.1007/978-3-319-21151-0_3.

[Sut+16] E. Sutanta et al. "Survey: Models and prototypes of schema matching". In: *International Journal of Electrical and Computer Engineering* 6.3 (2016), pp. 1011–1022. DOI: 10.11591/ijece.v6i3.9789.

[SW07] C. Simons and G. Wirtz. "Modeling context in mobile distributed systems with the UML". In: *Journal of Visual Languages and Computing* 18.4 (2007), pp. 420–439. DOI: 10.1016/j.jvlc.2007.07.001.

[The16a] The Eclipse Foundation. *Model-to-Model Transformation*. 2016. URL: https://projects.eclipse.org/projects/modeling.mmt.

[The16b] The Eclipse Foundation. *Sirius*. 2016. URL: https://eclipse.org/sirius/.

[TK08] Hallvard Trætteberg and John Krogstie. "Enhancing the Usability of BPM-Solutions by Combining Process and User-Interface Modelling". In: *The Practice of Enterprise Modeling: First IFIP WG 8.1 Working Conference (PoEM)*. Springer, 2008, pp. 86–97. DOI: 10.1007/978-3-540-89218-2_7.

[UB16] Eric Umuhoza and Marco Brambilla. "Model Driven Development Approaches for Mobile Applications: A Survey". In: *Mobile Web and Intelligent Information Systems (MobiWIS)*. Ed. by Muhammad Younas et al. Springer, 2016, pp. 93–107. DOI: 10.1007/978-3-319-44215-0_8.

[van99] W.M.P. van der Aalst. "Formalization and verification of event-driven process chains". In: *Information and Software Technology* 41.10 (1999), pp. 639–650. DOI: 10.1016/S0950-5849(99)00016-6.

[Vau+14] S. Vaupel et al. "Model-driven development of mobile applications allowing role-driven variants". In: *Lecture Notes in Computer Science* 8767 (2014), pp. 1–17.

[vt05] W.M.P. van der Aalst and A.H.M. ter Hofstede. "YAWL: Yet another workflow language". In: *Information Systems* 30.4 (2005), pp. 245–275. DOI: 10.1016/j.is.2004.02.002.

[Wol11] D. Wolber. "App inventor and real-world motivation". In: *ACM Technical Symposium on Computer Science Education (SIGCSE)* (2011). DOI: 10.1145/1953163.1953329.

[Xam17] Xamarin Inc. *Developer Center - Xamarin*. 2017. URL: https://developer.xamarin.com (visited on 11/28/2017).

[Zha+14] X. Zhao et al. "An empirical study of bugs in build process". In: *ACM Symposium on Applied Computing* (2014), pp. 1187–1189. DOI: 10.1145/2554850.2555142.

[ZS09] Uwe Zdun and M. Strembeck. "Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development". In: *European Conference on Pattern Languages of Programs (EuroPLoP)*. 2009, pp. 1–37.

[Żył15] Kamil Żyła. "Perspectives of Simplified Graphical Domain-Specific Languages as Communication Tools in Developing Mobile Systems for Reporting Life-Threatening Situations". In: *Studies in Logic, Grammar and Rhetoric* 43.1 (2015). DOI: 10.1515/slgr-2015-0048.

10

Towards Pluri-Platform Development

Table 10.1: Fact sheet for publication P4

Title	Towards Pluri-Platform Development: Evaluating a Graphical Model-Driven Approach to App Development Across Device Classes
Authors	Christoph Rieger[1] Herbert Kuchen[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2019
Publication Outlet	Lecture Notes in Business Information Processing (LNBIP)
Copyright	Springer International Publishing
Full Citation	Christoph Rieger and Herbert Kuchen. "Towards Pluri-Platform Development: Evaluating a Graphical Model-Driven Approach to App Development Across Device Classes". In: *Towards Integrated Web, Mobile, and IoT Technology*. Ed. by Tim A. Majchrzak, Cristian Mateos, Francesco Poggi, and Tor-Morten Grønli. Vol. 347. Springer International Publishing, 2019, pp. 36–66. DOI: 10.1007/978-3-030-28430-5_3 This version is not for redistribution. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-28430-5_3.

Towards Pluri-Platform Development: Evaluating a Graphical Model-Driven Approach to App Development Across Device Classes

Christoph Rieger Herbert Kuchen

Keywords: Graphical Domain-Specific Language, Model-Driven Software Development, Business App, Cross-Platform

Abstract: The domain of mobile apps encompasses a fast-changing ecosystem of platforms and vendors in which new classes of heterogeneous app-enabled devices are emerging. To digitize everyday work routines, business apps are used by many non-technical users. However, designing apps is mostly done according to traditional software development practices, and further complicated by the variability of device capabilities. To empower non-technical users to participate in the creation of supportive apps, graphical domain-specific languages can be used. Consequently, we propose the Münster App Modeling Language (MAML) to specify business apps through graphical building blocks on a high level of abstraction. In contrast to existing process modelling notations, these models can directly be transformed into apps for multiple platforms across different device classes through code generators without the need for manual programming. To evaluate the comprehensibility and usability of MAML's DSL, two studies were performed with software developers, process modellers, and domain experts.

10.1 Introduction

The opportunities of Model-Driven Software Development (MDSD) with regard to increased efficiency and flexibility have been studied extensively in the past years. The use of MDSD techniques is one approach to counteract the variety of platforms, programming languages, and human-interface guidelines found in the domain of mobile business apps. Several approaches have been researched in academic literature, including MD^2[ME15], Mobl [HV11], and AXIOM [JJ14]. Those approaches provide cross-platform development functionalities with one common model for multiple target platforms using textual domain-specific languages (DSLs) to specify apps. Whereas these approaches significantly ease the development of apps and thus also support current trends such as "Bring your own device" [Tho12], the actual creation of apps is still restricted to users with programming skills [MHS05]. Business apps focus on specific tasks to be accomplished by employees. Therefore, a centralized definition of such processes aligns well with traditional software development practices but may deviate from the end user's needs. Consequently, the introduction of business apps may fall short of improving efficiency. In addition, operating employees have valuable insights into the actual process execution as well as unobvious process exceptions. Giving them a means to shape the software they use in their everyday work routines offers not only the possibility to explicate their tacit knowledge for the

development of best practices, but also actively involves them in the evolution of the enterprise. Instead of participating only in early requirements engineering phases of software development, continuously co-designing such systems may increase the adoption of the resulting application and possibly strengthen their identification with the company [Ful+06]. Mobile app development can thus benefit from the incorporation of people from all levels of the organization and development tools should be understandable to both programmers and domain experts.

The research company Gartner predicted that more than half of all company-internal mobile apps will be built using codeless tools by 2018 [Rv14]. The general trend towards low-code or codeless development of business apps can be supported by the introduction of graphical notations which are particularly suitable to represent the concepts of a data-driven and process-focused domain. However, current approaches often lack the capacity for holistic app modelling, often operating on a low level of abstraction with visual user interface builders or approaches using view templates (e.g., [Xam19; Goo19a]).

In order to advance research in the domain of cross-platform development of mobile apps and investigate opportunities for organizations in a digitized world, this paper presents and evaluates the Münster App Modeling Language (MAML) framework. Rooted in the Eclipse ecosystem, the DSL grammar is defined as an Ecore metamodel, the visual editor is built using the Sirius framework [The19], and technologies such as Xtend are used for the code generation of Android, iOS, and Wear OS apps.

Moreover, the high level of abstraction and automatic inference required for simple-to-use app development opens up opportunities for reusing the same notation for heterogeneous target devices. The terms cross-platform or multi-platform development typically denote the creation of applications for multiple platforms *within the same device class*, for example iOS and Android in the smartphone domain – potentially extended to technically similar tablets. However, the development of apps *across device classes* is not yet tackled systematically in academia or practice. To distinguish the additional requirements and challenges introduced when creating applications across heterogeneous devices, we propose the term *pluri-platform development*.

This article greatly extends the paper [Rie18] presented at HICSS 2018[1]. It has been updated to reflect the latest developments, includes new content based on additional work as well as on the discussions at the conference. Also, it has been amended with a perspective on challenges and opportunities regarding model-driven app development for novel device classes and a study on the suitability of the MAML notation in this context. The remainder of this article is structured as follows: After presenting related work in Section 10.2, MAML's graphical DSL is presented that allows for the visual definition of business apps (Section 10.3). The codeless app creation capabilities are demonstrated using the MAML editor with advanced modelling support and an automated generation of native app source code through a two-step model transformation process. Section 10.4 discusses the setup and results of two usability studies conducted to demonstrate

[1]Please note that verbatim content from the original paper is not explicitly highlighted but for figures and tables already included there.

the potential and intricacies of an integrated app modelling approach for a wide audience of process modellers, domain experts, and programmers. The possibility to extend the approach to heterogeneous devices is covered in Section 10.5. In Section 10.6, the findings and implications of MAML are discussed with regard to model-driven development for heterogeneous app-enabled devices before concluding with a summary and outlook in Section 10.7.

10.2 Related Work

Different approaches to cross-platform mobile app development have been researched. In general, five approaches can be distinguished [MEK15]. Concerning runtime approaches, *mobile webapps* are browser-run web pages optimized for mobile devices but without native user interface (UI) elements, *hybrid approaches* such as Apache Cordova [Apa19] provide a wrapper to web-based apps that allow for accessing device-specific features through interfaces, and *self-contained runtimes* provide separate engines that mimic core system interfaces in which the app runs. In addition, two generative approaches produce native apps, either by *transpiling* apps between programming languages such as J2ObjC [Goo18] to transform Android-based business logic to Apple's language Objective-C, or *model-driven software development* for transforming a common model to code.

With regard to model-driven development, DSLs are used to model mobile apps on a platform-independent level. According to Langlois et al. [LJJ07], DSLs can be classified in textual, graphical, tabular, wizard-based, or domain-specific representations as well as combinations of those. Several frameworks for mobile app development have been developed in the past years, both for scientific and commercial purposes. In the particular domain of business apps – i.e., form-based, data-driven apps interacting with back-end systems [MEK15] – the graphical approach JUSE4Android [dB14] uses annotated UML diagrams to generate the appearance of and navigation within object graphs, and Vaupel et al. [Vau+14] presented an approach focusing around role-driven variants of apps using a visual model representation. Other approaches such as AXIOM [JJ14] and Mobl [HV11] provide textual DSLs to define business logic, user interaction, and user interface in a common model. An extensive overview of current model-driven frameworks is provided by Umuhoza and Brambilla [UB16]. However, current approaches mostly rely on a textual specification which limits the active participation of non-technical users without prior training [Zył15], and graphical approaches are often incapable of covering all structural and behavioural aspects of a mobile app. For generating source code, the work in this paper is based on the Model-Driven Mobile Development (MD^2) framework which also uses a textual DSL for specifying all constituents of a mobile app in a platform-independent manner. After preprocessing the models, native source code is generated for each target platform as described by Majchrzak and Ernsting [ME15]. This intermediate step is, however, automated and requires no intervention by the user (see Subsection 10.3.4).

In contrast to DSLs, several general-purpose modelling notations exist for graphically depicting applications and processes, such as the Unified Modeling Language (UML) with a collection

of interrelated standards for software development. The Interaction Flow Modeling Language (IFML) can be used to model user interactions in mobile apps, especially in combination with the mobile-specific elements introduced as extension by Breu et al. [BKF14]. Process workflows can for example be modelled using BPMN [Obj11], Event-Driven Process Chains [Aal99], or flowcharts [Int85]. However, such notations are often either suitable for generic modelling tasks and remain on a superficial level of detail, or represent rather complex technical notations designed for a target group of programmers [Fra+06]. A trade-off is necessary to balance the ease of use for modellers with the richness of technical details for creating functioning apps. Moody [Moo09] has pointed out principles for the cognitive effectiveness of visual notations and subsequent studies have revealed comprehensibility issues through effects such as symbol overload, e.g., for the WebML notation preceding IFML [Gra+15]. Examples of technical notations in the domain of mobile applications include a UML extension for distributed systems [SW07] and a BPMN extension to orchestrate web services [BDF09]. Nevertheless, the approach presented in this work goes beyond pure process modelling. While IFML is closest to the work in this paper regarding the purpose of modelling user interactions, MAML covers both structural (data model and views) and behavioural (business logic and user interaction) aspects.

Lastly, visual programming languages have been created for several domains such as data integration [Pen17] but few approaches focus specifically on mobile apps. RAPPT combines a graphical notation for specifying processes with a textual DSL [Bar+15], and AppInventor provides a language of graphical building blocks for programming apps [Wol11]. However, non-technical users are usually ignored in the actual development process. Hence, those visual notations do not exploit the potential of including people with in-depth domain knowledge. Considering commercial frameworks, support for visual development of mobile apps varies significantly. In practice, many recent tools are limited to specific components such as back-end systems or content management, or support particular development phases such as prototyping [Pro16]. Start-ups such as Bizness Apps [Biz19] and Bubble Group [Bub19] aim for more holistic development approaches using configurators and web-based editors. Similarly, development environments have started to provide graphical tools for UI development, enhancing the programmatic specification of views by complementary drag and drop editors [Xam19]. The WebRatio Mobile Platform also supports codeless generation of mobile apps through a combination of IFML, other UML standards, and custom notations [Web19]. In contrast, this work focuses on a significantly more abstract and process-centric modelling level as presented in the next section.

10.3 Münster App Modeling Language

At its core, the MAML framework consists of a graphical modelling notation that is described in the following subsections. Contrary to existing notations, its models contain sufficient information to transform them into fully functional mobile apps. The framework also comprises the necessary development tools to design MAML models in a graphical editor and generate apps

without requiring manual programming. The generation process is described in more detail in Subsection 10.3.4.

10.3.1 Language Design Principles

The graphical DSL for MAML is based on five design goals:

Automatic cross-platform app creation: Most important, the whole approach is built around the key concept of codeless app creation. To achieve this, individual models need to be recombined and split according to different roles (see Subsection 10.3.4) and transformed into platform-specific source code. As a consequence, models need to encode technical information such as data fields and interrelations between workflow elements in a machine-interpretable way as opposed to an unstructured composition of shapes filled with text.

Domain expert focus: MAML is explicitly designed with a non-technical user in mind. Process modellers as well as domain experts are encouraged to read, modify, and create new models by themselves. The language should, therefore, not resemble technical specification languages drawn from the software engineering domain but instead provide generally understandable visualizations and meaningful abstractions for app-related concepts.

Data-driven process modelling: The basic idea of business apps to focus on data-driven processes determines the level of abstraction chosen for MAML. In contrast to merely providing editors for visual screen composition as replacement for manually programming user interfaces, MAML models represent a substantially higher level of abstraction. Users of the language concentrate on visualizing the sequence of data processing steps and the concrete representation of affected data items is automatically generated using adequate input/output user interface elements.

Modularization: To engage in modelling activities without advanced knowledge of software architectures, appropriate modularization is important to handle the complexity of apps. MAML embraces the aforementioned process-oriented approach by modelling use cases, i.e., a unit of functionality containing a self-contained set of behaviours and interactions performed by the app user [Obj15]. Combining data model, business logic, and visualization in a single model deviates from traditional software engineering practices which, for instance, often rely on the Model-View-Controller pattern [Gam+95]. In accordance with the domain expert focus, the end user is, however, unburdened from this technical implementation issue.

Declarative description: MAML models consist of platform-agnostic elements, declaratively describing *what* activities need to be performed with the data. The concrete representation in the resulting app is deliberately unspecified to account for different capabilities and usage patterns of each targeted mobile platform. The respective code generator can provide sensible defaults for such platform specifics.

10.3.2 Language Overview

In the following, the key concepts of the MAML DSL are highlighted using the fictitious scenario of a publication management app. A sample process to add a new publication to the system consists of three logical steps: First, the researcher enters some data on the new publication. Then, he can upload the full-text document and optionally revise the corresponding author information. This self-contained set of activities is represented as one model in MAML, the so-called use case, as depicted in Figure 10.1.

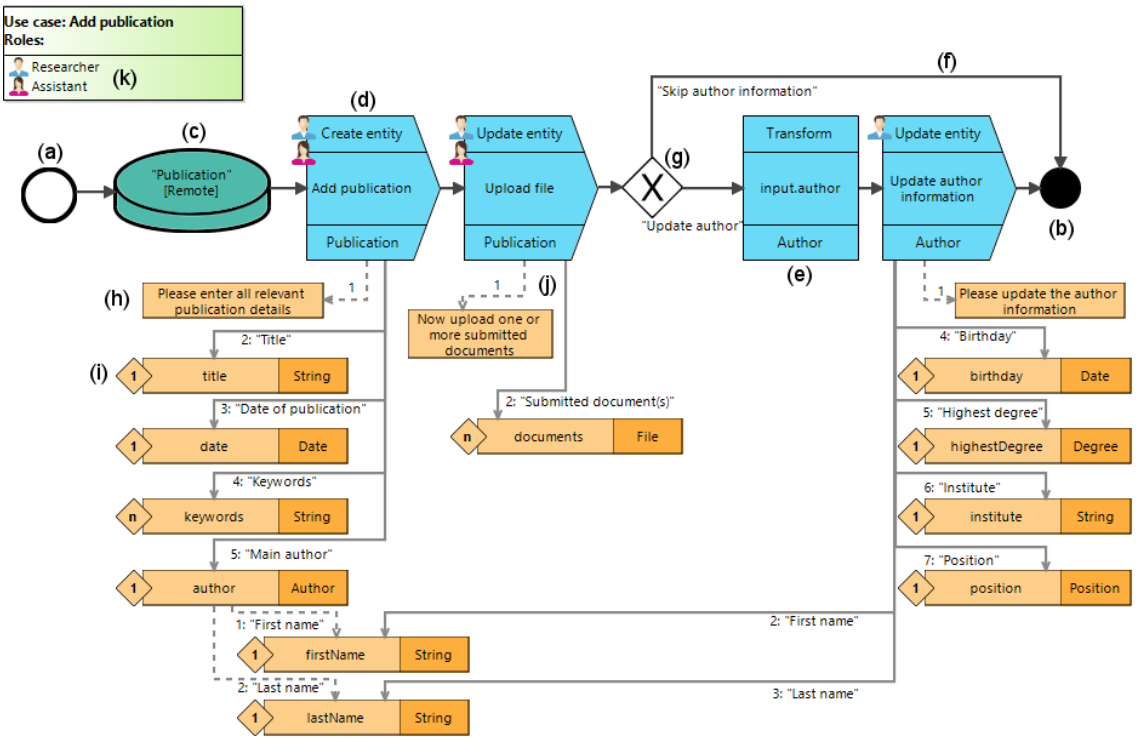


Figure 10.1: MAML Use Case for Adding a Publication to a Review Management System [Rie17]

A model consists of a *start event* (labelled with (a) in Figure 10.1) and a sequence of process flow elements towards an *end event* (b). A *data source* (c) specifies what type of entity is first used in the process, and whether it is only saved locally on the mobile device or managed by the remote back-end system. Then, the modeller can choose from predefined *interaction process elements* (d), for example to *create/show/update/delete* an entity, but also to *display messages*, access device sensors such as the *camera*, or *call* a telephone number. Because of the declarative description, no device-specific assumptions can be made on the appearance of such a step. The generator instead provides default representations and functionalities, e.g., display a *select entity* step using a list of all available objects as well as possibilities for searching or filtering. In addition, *automated process elements* (e) represent steps to be performed without user interaction. Those elements provide the minimum amount of technical specificity in order to navigate between the model objects

(*transform*), request information from *web services*, or *include* other models to reuse existing use cases.

The order of process steps is established using *process connectors* (f), represented by a default "Continue" button unless specified differently along the connector element. *XOR* (g) elements branch out the process flow based on a manual user action by rendering multiple buttons (see differently labelled connectors in Figure 10.1), or automatically by evaluating expressions referring to a property of the considered object.

The lower section of Figure 10.1 contains the data linked to each process step. *Labels* (h) provide explanatory text on screen. *Attributes* (i) are modelled as combination of a name, the data type, and the respective cardinality. Data types such as *String, Integer, Float, PhoneNumber, Location*, etc. are already provided but the user can define additional custom types. To further describe custom-defined types, attributes may be nested over multiple levels (e.g., the "author" type in Figure 10.1 specifies a first name and last name). In addition, *computed attributes* (not depicted in the example) allow for runtime calculations such as counting or summing up other attribute values.

A suitable UI representation is automatically chosen based on the type of *parameter connector* (j): Dotted arrows signify a reading relationship whereas solid arrows represent a modifying relationship. This refers not only to the manifest representation of attributes displayed either as read-only text or editable input field. The interpretation also applies in a wider sense, e.g., regarding web service calls in which the server "reads" an input parameter and "modifies" information through its response. Each connector also specifies an order of appearance and, for attributes, a human-readable caption derived from the attribute name unless manually specified.

Finally, annotating freely definable *roles* (k) to all interactive process elements allows for the coherent visualization of processes that are performed by more than one person, for example in scenarios such as approval workflows. When a role change occurs, the app automatically saves modified data and users with the subsequent role are informed about the open workflow instance in their app.

10.3.3 App Modelling

In contrast to other notations, all of the modelling work is performed in a single type of model, mainly by dragging elements from a palette and arranging them on a large canvas. The modelling environment was developed using the Eclipse Sirius framework [The19] that was extended with domain-specific validation and guidance to provide advanced modelling support for MAML.

Modelling only the information displayed in each process step effectively creates a multitude of partial data models for each process step and for each use case as a whole. Also, attributes may be connected to multiple process elements simultaneously, or can be duplicated to different positions to avoid wide-spread connections across the model. An inference mechanism [Rie17] aggregates and validates the complete data model while modelling. During generation, app-internal and

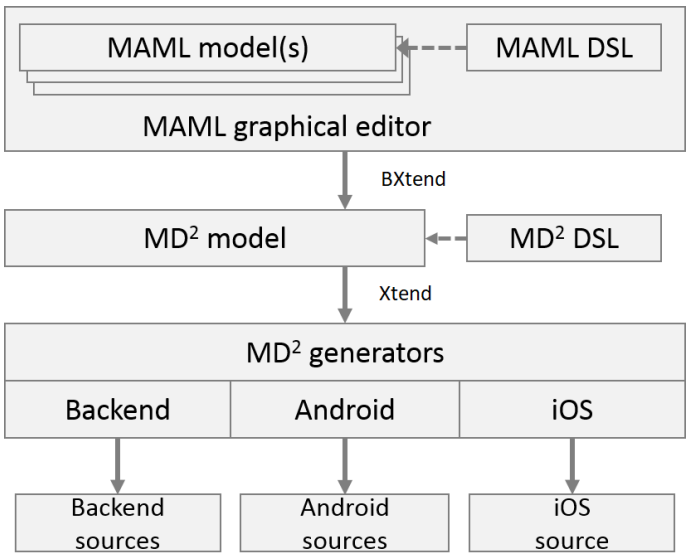


Figure 10.2: MAML App Generation Process (cf. [Rie18])

back-end data stores are automatically created. As a result, the user does not need to specify a distinct global data model and consistency is automatically checked when models change.

Apart from validation rules to prevent users from modelling syntactically incorrect MAML use cases in the first place, additional validity checks have been implemented in order to detect inconsistencies across use cases (based on the inferred data model) as well as potentially unwanted behaviour (e.g., missing role annotations). Moreover, advanced modelling support attempts to provide guidance and overview to the user. For example, the current data type of a process element (lower label of (d) in Figure 10.1) is automatically derived from the preceding elements to improve the user's imagination within the process. Also, suggestions of probable values are provided when adding elements (e.g., known attributes of the originating type when adding UI elements).

10.3.4 App Generation

Technically, MAML relies on and integrates with the Eclipse Modeling Framework (EMF), for example by specifying the DSL's metamodel as an Ecore model. In order to generate apps, the proposed approach reuses previous work on MD²(see Section 10.2). The complete generation process is depicted in Figure 10.2. Because of space constraints, the respective transformations are only sketched next.

First, model transformations are applied to transform graphical MAML models to the textual MD²representation using the BXtend framework [Buc18] and the Xtend language. Amongst other activities, all separately modelled use cases are recombined, a global data model across all use cases is inferred and explicated, and processes are broken down according to the specified roles.

In the subsequent code generation step, previously existing generators in MD² create the actual source code for all supported target platforms.

This is, however, not an inherent limitation of the framework. Newly created generators might just as well generate code directly from the MAML model or use interpreted approaches instead of code generation.

It should be noted that this proceeding differs from approaches such as UML's Model Driven Architecture [Arco1] in that the intermediate representation is still a platform-independent representation but with a more technical focus. Optionally, a modeller has the possibility to modify default representations and configure parts of the application in more detail before source code is generated for each platform. Although the tooling around MAML is still in a prototypical state, it currently supports the generation of Android and iOS apps as well as a Java-based server back-end component. Also, a smartwatch generator for Google's Wear OS platform highlights the applicability to further device classes (cf. Section 10.5). The screenshots in Figure 10.3 depict the generated Android app views for the first process steps of the MAML model depicted in Figure 10.1.

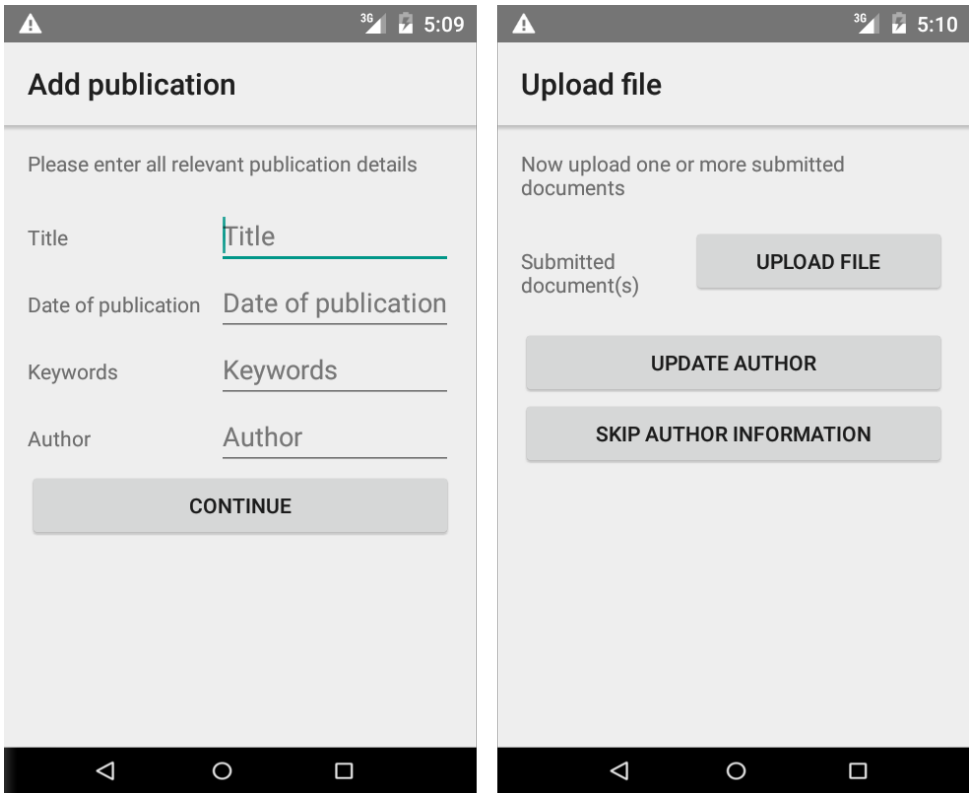


Figure 10.3: Exemplary Screenshots of Generated Android App Views [Rie18]

10.4 Evaluation

As demonstrated, MAML aligns with the goals of automated cross-platform app creation from modular and platform-agnostic app models (cf. Subsection 10.3.1). However, the suitability of data-driven process models with regard to the target audience needed to be evaluated in more detail. Therefore, an observational study was performed to assess the utility of the newly developed language. After describing the general setup in Subsection 10.4.1, the results on comprehensibility and usability of the graphical DSL are presented.

10.4.1 Study Setup

The purpose of the study was to assess MAML's claim to be understandable and applicable by users with different backgrounds, in particular including non-technical users. From the variety of methodologies for usability evaluation, observational interviews according to the think-aloud method were selected as empirical approach [GH02]. Participants were requested to perform realistic tasks with the system under test and urged to verbalize their actions, questions, problems, and general thoughts while executing these tasks. Due to the novelty of MAML which excludes the possibility of comparative tests, this setup focused on obtaining detailed qualitative feedback on usability issues from a group of potential users.

Therefore, 26 individual interviews of around 90 minutes duration were conducted. An interview consisted of three main parts: First, an online questionnaire had to be filled out in order to collect demographic data, previous knowledge in the domains of programming or modelling, and personal usage of mobile devices. Second, a MAML model and an equivalent IFML model were presented to the participants (in random order to avoid bias) to assess the comprehensibility of such models without prior knowledge or introduction. In addition to the verbal explanations, a short 10-question usability questionnaire was filled out to calculate a score according to the System Usability Scale (SUS) [Bro96] for each notation (cf. Subsection 10.4.2). Third, the main part of the interview consisted of four modelling tasks to accomplish using the MAML editor. Finally, the standardized ISONORM questionnaire was used to collect more quantitative feedback, aligned with the seven key requirements of usability according to DIN 9241/110-S [Into6] (cf. Subsection 10.4.3).

To capture the variety of possible usability issues, 71 observational features were identified a priori and structured in six categories of interest: comprehensibility, applying the notation, integration of elements, tool support, effectiveness, and efficiency. In total, over 1500 positive or negative observations were recorded as well as additional usability feedback and proposals for improvement.

Regarding participant selection, 26 potential users in the age range of 20 to 57 years took part in the evaluation. Although they mostly have a university background, technical experience varied widely and none had previous knowledge of IFML or MAML. To further analyse and interpret the results, the participants were categorized in three distinct groups according to their personal

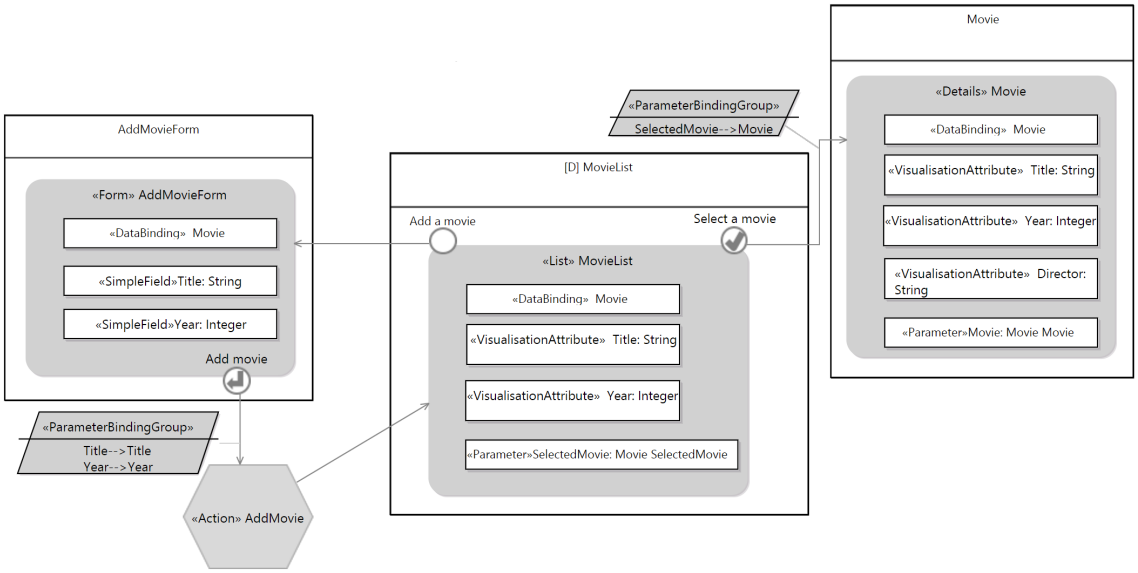


Figure 10.4: IFML Model to Assess the a Priori Comprehensibility of the Notation [Rie18]

background stated in the online questionnaire: 11 software developers have at least medium knowledge in traditional/web/app programming or data modelling, 9 process modellers have at least medium knowledge in process modelling (exceeding their programming skills), and 6 domain experts are experienced in the modelling domain but have no significant technical or process knowledge. Although it is debated whether Virzi's [Vir92] statement of five participants being sufficient to uncover 80% of usability problems in a particular software holds true [SS01], arguably the selected amount of participants in this study is reasonable with regard to finding the majority of grave usability defects for MAML and generally evaluating the design goals.

For their private use, participants stated an average smartphone usage of 19.2 hours per week, out of which 16.3 hours are spent on apps. In contrast, tablet use is rather low with 3.5 hours (3.2 hours for apps), and notebook usage is generally high with 27.5 hours but only 4.7 hours are spent on apps. For business uses, similar patterns can be observed on total / app-only usage per week on smartphones (5.5h / 4.3h), tablets (0.7h / 0.2h), and notebooks (18.2h / 3.7h). Although this sample is too low for generalizable insights, the figures indicate a generally high share of app usage on smartphones and tablets compared to the total usage duration, both for personal and business tasks. In addition, with mean values of 1.81 / 2.12 on a scale between 0 (strongly reduce) and 4 (strongly increase), the participants stated to have no desire of significantly changing their usage volumes of private / business apps.

10.4.2 Comprehensibility Results

Before actively introducing MAML as modelling tool, the participants should explicate their understanding of a given model without prior knowledge. Comprehensibility is an important

characteristic in order to easily communicate app-related concepts via models without the need for extensive training. To compare the results with an existing modelling notation, equivalent IFML (see Figure 10.4) and MAML models [Rie19] of a fictitious movie database app were provided with the task to describe the purpose of the overall model and the particular elements. The monochrome models were shown to the participants on paper in randomized order to avoid bias from priming effects [BC00] and potential influences from a particular software environment.

After each model, participants were asked to answer the SUS questionnaire for the particular notation. This questionnaire has been applied in many contexts since its development in 1986 and can be seen as easy, yet effective, test to determine usability characteristics. Each participant answers ten questions using a five-point Likert-type scale between strong disagreement and strong agreement, which is later converted and scaled to a [0;100] interval according to Brooke [Bro96]. The participants' scores for both languages and the respective standard deviations are depicted in Table 10.2.

Table 10.2: System Usability Scores for IFML and MAML

SUS ratings	**IFML**	**MAML**
All participants	52.79 (σ = 23.0)	66.83 (σ = 15.6)
Software developers	45.91 (σ = 23.6)	64.09 (σ = 17.3)
Process modellers	64.17 (σ = 19.0)	69.44 (σ = 12.0)
Domain experts	48.33 (σ = 24.5)	67.92 (σ = 18.7)

However, it should be noted that the results do not represent percentage values. Instead, an adjective rating scale was proposed by Bangor et al. [BKM09] to interpret the results as depicted in Figure 10.5. The results show that MAML's scores are superior overall as well as for all three groups of participants. In addition, the consistency of scores across all groups supports the design goal of creating a notation which is well understandable for users with different backgrounds. Particularly, domain experts without technical experience expressed a drastic difference in comprehensibility of almost 20 points.

Especially the distinction between domain experts and technical users (developers and process modellers together) is of interest to evaluate the design goal of MAML to be comprehensible

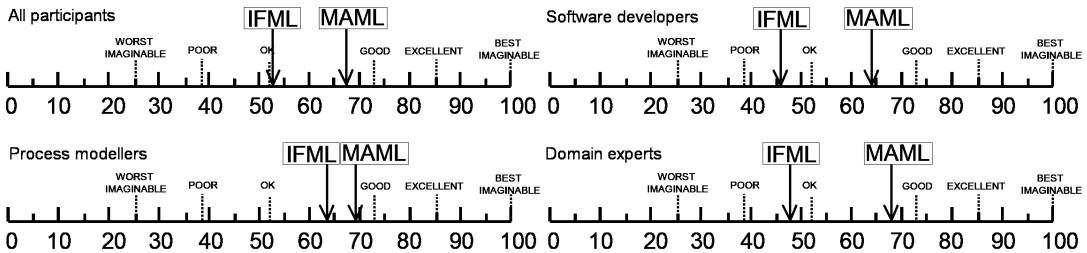


Figure 10.5: SUS Ratings for IFML and MAML [Rie18]

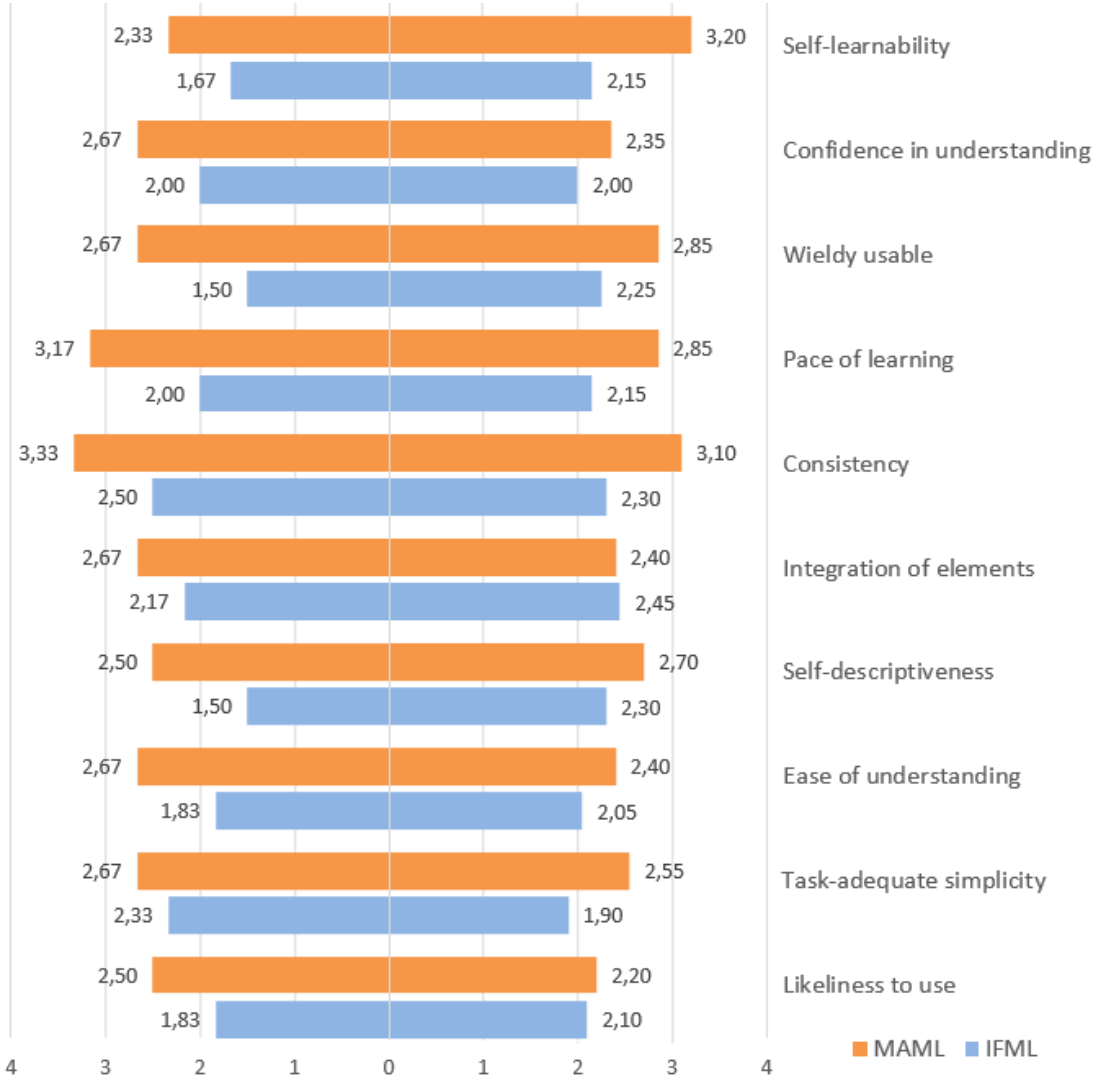


Figure 10.6: SUS Answers for Domain Experts (left) and Technical Users (right) [RK18]

for different user groups. Figure 10.6 breaks down the answers to the 10 questions of the SUS questionnaire (rescaled to a [0;4] interval; 4 denoting strong acceptance). With one exception, responses for MAML are higher than for the technical IFML notation. Moreover, domain experts reacted significantly more positively when assessing the MAML notation as being wieldy usable (+1.17 compared to IFML), fast to learn (+1.17), and self-descriptive (+1.00). Consequently, the understandability and general applicability of the notation is in the focus of domain experts, which aligns well with the intention to use MAML for communicating with potential end users and include them in the development process. The strongest deviations for technical users, in contrast, can be seen in questions regarding self-learnability (+1.05), perceived consistency (+0.80), and pace of learning (+0.70). Conforming with their technical background, these aspects emphasize the correct application of the notation which apparently is perceived as positive in MAML, too.

Considering also the qualitative observations, some interesting insights can be gained. According to the questionnaire results, most of the criticism is related to the categories "easy to understand" and "confidence in the notation". IFML's approach of visually hinting at the outcome through the order of elements and their composition in screen-like boxes was often noted as positive and slightly more intuitive compared to MAML. This argument is not unexpected as the level of abstraction was designed to be higher than a pure visual equivalent of programming activities. Also, the notation is not limited to the few types of mobile devices known by a participant, e.g., smartwatches and smartphones exhibit very different interface and interaction characteristics. Therefore, a fully screen-oriented approach generally contradicts the desired platform-independent design of MAML. However, this is valuable feedback for the future, e.g., improving modelling support by using an additional simulator component to preview the outcome while modelling.

Surprisingly, IFML scores were worst for the group of software developers, although they have knowledge of other UML concepts and diagrams. Despite this apparent familiarity, reasons for the negative assessment of IFML can be found in the amount of "technical clutter", e.g., regarding parameter and data bindings, as well as perceived redundancies and inconsistencies. In contrast, 86% of these participants highlight the clarity of MAML regarding the composition of individual models and 88% are able to sketch a possible appearance of the final app result based on the abstract process specification.

Overall, three in four participants can also transfer knowledge from other modelling notations, e.g., to interpret elements such as data sources. All participants within the process modeller group immediately recognize analogies from other graphical notations such as BPMN, and understand the process-related concepts of MAML. Whereas elements such as data sources (understood by 75% of all participants) and nested attribute structures (83%) are interpreted correctly on an abstract level, comprehensibility drops with regard to technical aspects, e.g., data types (57%) or connector types (43%).

Finally, domain experts also have difficulties to understand the technical aspects of MAML without previous introduction. Although concepts such as cardinalities (0%), data types (25%), and nested object structures (67%) are not initially understood and ignored, all participants in this group are still able to visualize the process steps and main actions of the model. As described in Subsection 10.3.1, further reducing these technical aspects constrains the possibilities to generate code from the model. Some suggestions exist to improve readability, e.g., replacing the textual data type names with visualizations. Nevertheless, MAML is comparatively well understandable. Curiously enough, the sample IFML model is often perceived as being a more detailed technical representation of MAML instead of a notation with equivalent expressiveness.

To sum up, MAML models are favoured by participants from all groups, despite differences in personal background and technical experience. This part of the study is not supposed to discredit IFML but emphasizes their different foci: Whereas IFML covers an extensive set of features and integrates into the UML ecosystem, it is originally designed as generic notation for modelling

user interactions and targeted at technical users. In contrast, the study confirms MAML's design principle of an understandable DSL for the purpose of mobile app modelling.

10.4.3 Usability Results

In addition to the language's comprehensibility, a major part of the study evaluated the actual creation of models by the participants using the developed graphical editor. After a brief ten-minute introduction of the language concepts and the editor environment, four tasks were presented that cover many of MAML's features and concepts. In the hands-on context of a library app (cf. supplementary online material [Rie19]), a first simple model to add a new book to the library requires the combination of core features such as process elements and attributes. Second, participants should model how to borrow a book based on screenshots of the resulting app. This requires more interaction element types, a case distinction, and complex attributes. Third, modelling a summary of charges includes a web service call, exception handling, and calculations. Fourth, a partial model in a multi-role context needed to be altered.

The final evaluation was performed using the ISONORM questionnaire in order to assess the usability according to the ISO 9241-110 standard [Into6]. 35 questions with a scale between -3 and 3 cover the seven criteria of usability as presented in Table 10.3. Again, MAML achieves positive results for every criterion, both for the participant subgroups and in total. Taking the interview observations into account for qualitative feedback, these figures can be evaluated in more detail.

Table 10.3: ISONORM Usability Questionnaire Results for MAML

Criterion	**All participants**	**Software developers**	**Process modellers**	**Domain experts**
Suitability for the task	1.63 (σ = 1.04)	1.36 (σ = 1.13)	1.62 (σ = 1.12)	2.13 (σ = 0.62)
Self-descriptiveness	0.51 (σ = 0.73)	0.62 (σ = 0.62)	0.38 (σ = 1.02)	0.50 (σ = 0.41)
Controllability	2.10 (σ = 0.83)	2.20 (σ = 0.63)	2.02 (σ = 0.63)	2.03 (σ = 1.41)
Conformity with user expectations	1.78 (σ = 0.52)	1.85 (σ = 0.47)	1.64 (σ = 0.47)	1.87 (σ = 0.70)
Error tolerance	0.92 (σ = 0.96)	0.89 (σ = 0.63)	1.11 (σ = 0.81)	0.70 (σ = 1.63)
Suitability for individualisation	1.20 (σ = 0.90)	1.04 (σ = 1.05)	1.42 (σ = 1.02)	1.17 (σ = 0.27)
Suitability for learning	1.83 (σ = 0.67)	2.02 (σ = 0.54)	1.69 (σ = 0.66)	1.70 (σ = 0.90)
Overall score	1.43 (σ = 0.49)	1.43 (σ = 0.46)	1.41 (σ = 0.53)	1.44 (σ = 0.59)

Regarding the *suitability for the task*, observations on the effectiveness and efficiency of the notation show that handling models in the editor is achieved without major problems. 94% of the participants themselves noticed a fast familiarization with the notation, although domain experts are generally more wary when using the software. The deliberately chosen high level of abstraction manifests in 37% of participants describing this approach as uncommon or astonishing (see also Section 10.6). Nevertheless, 67% of the participants state to have an understanding of the resulting app while modelling.

Self-descriptiveness refers to comprehension issues but additionally deals with the correct integration of different elements while modelling. For example, the concept of user roles was introduced to the participants but not the assignment in models. Still, 86% of them intuitively drag and drop role icons on process elements correctly. Furthermore, process exceptions were not explained at all in the introduction but 71% of the participants applied the "error event" element correctly without help. Self-descriptiveness is, however, more limited when dealing with technical issues. Side effects of transitive attributes are only recognized by 43% of process modellers and 25% of domain experts. Model validation or additional modelling support is needed in order to guide the users towards semantically correct models. Similarly, the complexity of modelling web service responses within the use case's data flow poses challenges to 44% of the participants.

The very positive responses for the *controllability* criterion can be explained by the simplistic design of MAML and its tools. All modelling activities are performed in a single model instead of switching between multiple perspectives. In contrast to other notations, all of the modelling work is performed in a single type of model, mainly by dragging elements from a palette and arranging them on a large canvas. Many participants utter remarks such as "the editor does not evoke the impression of a complex tool". In parts, this impression can be attributed to sophisticated modelling support, including live data model inference when connecting elements in the model, validation rules, and suggestions for available data types.

Related to the clarity of possible user actions, the *conformity with user expectations* is also clearly positive. Despite occasional performance issues caused by the prototypical nature of the tools, a consistent handling of the program is confirmed by the participants. Although aspects such as the direction of parameter connections may be interpreted differently (e.g., either a sum refers to attributes or attributes are incoming arguments to the sum function), the consistent use of concepts throughout the notation is easily internalized by the participants.

Regarding *error tolerance* and *suitability for individualisation*, scores are moderate but the prototype was not yet particularly optimized for production-ready stability or performance. Also, an individual appearance was not intended, thus providing only basic capabilities such as resizing and repositioning components. Whereas the editor is very permissive with regard to the order of modelling activities, adding invalid model elements is mostly avoided by syntactic and semantic validity checks, e.g., which elements are valid end points of a connector. Participants appreciate the support of not being able to model invalid constellations. However, criticism arises from disallowing actions without further feedback on why a specific action is invalid. The modelling

environment Sirius is currently not able to provide this information, yet users might benefit more from such dynamic explanations than from traditional help pages.

Finally, *suitability for learning* can be demonstrated best using quotes such as MAML being judged as "a really practical approach", and participants having "fun to experiment with different elements" or being "surprised about what I am actually able to achieve". Using the graphical approach, users can express their ideas and apply concepts consistently to different elements. As mentioned above, many unknown features such as roles or web service interactions can be learned using known drag and drop patterns or read/modify relationships.

10.5 Towards Pluri-Platform Development

The term *cross-platform* as well as actual development frameworks in academia and practice are usually limited to smartphones and sometimes – yet not always – technically similar tablets. Thus, they ignore the differing requirements and capabilities within the variety of novel devices and platforms reaching the mainstream consumer market in the near future. Extending the boundaries of current cross-platform development approaches requires a new scope of target devices which can be subsumed under the term *app-enabled* devices. Following the definition in [RM18], an app-enabled device can be described as being extensible with software that comes in small, interchangeable pieces, which are usually provided by third parties unrelated to the hardware vendor or platform manufacturer, and increase the versatility of the device after its introduction. Although these devices are typically portable or wearable and therefore related to the term *mobile computing*, there are further devices classes with the ability to run apps (e.g., smart TVs).

However, new challenges arise when extending the idea of cross-platform development to app-enabled device classes. In particular, not all approaches mentioned in Section 10.2 are generally capable for this extension as described in the following.

10.5.1 Challenges

From the development and usage perspectives on app development, specific challenges can be identified related to app development across device classes and which can be grouped into four main categories.

Output heterogeneity: The user interface of upcoming device classes enables more flexible and intuitive ways of device interaction compared to the prevalent focus on medium screen sizes between 4" and 10". By design, graphical output is very limited on wearables. On the other hand, smart TVs provide large-scale screens beyond 20". Also, devices can use new techniques for presenting information, e.g., auditive output by smart virtual assistants, or projection through augmented reality (for example using the wind shield in vehicles). In addition, even for screen-based

devices the variability of output increases because of new screen designs with drastically differing pixel density, aspect ratios, and form factors (e.g., round smartwatches) [RM18]. Techniques from the field of adaptive user interfaces may be used to tackle these issues. To achieve this degree of adaptability, specifying user interfaces needs to evolve from a screen-oriented specification of explicitly positioned widgets to a higher level of abstraction which can use semantic information to transform the content to a particular representation.

User input heterogeneity: The device interfaces for entering information by the user also evolve and will use a wider spectrum of possible techniques for user input [RM18]. This ranges from pushing buttons attached to the device, using remote controls, directing pointing devices for graphical user interfaces, tapping on touch screens, and using auxiliary devices (e.g., stylus pens) to hands-free interactions via gestures, voice, or even neural interfaces. To complicate matters, a single device may provide multiple input alternatives for convenience and especially new device classes are often experimenting with different interactions patterns. Again, this complexity calls for a higher level of abstraction when specifying apps by decoupling actual input events from the intended actions of the user interaction.

Device class capabilities: The variability of hardware and software across device classes is also apparent besides the user interface. For example, the miniaturization in wearable devices negatively impacts the computational power and battery capacity. Complex computations may therefore be offloaded to potential companion devices or provided through edge/cloud computing [RZ15]. Sensing capabilities can vary both within and across device classes. In addition, platform operating systems provide different levels of device functionality access and app interoperability, e.g., regarding security issues in vehicles. To avoid the problem of developing for the least common denominator of all targeted devices, suitable replacements for unavailable sensors need to be provided. For example, automatic location detection via GPS sensors can have fallback solutions such as address lookup or manual selection on a map.

Multi-device interaction: Whereas cross-platform approaches often provide self-contained apps with the same functionality for different users (it is fairly uncommon to own multiple smartphones with different platforms), users increasingly own multiple devices of different device classes and aim for interoperable solutions within their ecosystem. This complexity of multi-device interactions for a single user might occur *sequentially* when a user switches to a different device depending on the usage context or user preferences (e.g., reading notifications on a smartwatch and typing the response on a smartphone for convenience). Moreover, a *concurrent* usage of multiple devices for the same task is possible, for instance in a second screening scenario in which one device provides additional information or input/output capabilities for controlling another device [NJE17]. In both cases, fast and reliable synchronization of content is essential in order to seamlessly switch between multiple devices.

10.5.2 Towards Pluri-Platform Development

To emphasize the difference in scope and the respective solution approaches compared to traditional cross-platform development, we propose the term *pluri-platform* development to signify the creation of apps *across* device classes, in contrast to multi-/cross-platform development for several platforms *within* one class of mostly homogeneous devices. Pluri-platform development can, therefore, be understood as an umbrella term for different approaches aiming to bridge the gap between multiple device classes by tackling the challenges of heterogeneous input and output mechanisms, device capabilities, and multi-device interactions. In contrast to cross-platform development, the focus lies on simplification of app creation not just with regard to the representation of user interfaces but also the integration with platform-specific usage patterns and the interaction within a multi-device context. The related research fields of adaptive user interfaces and context-aware interfaces thus only account for a subset of the required solutions to achieve pluri-platform development.

Considering previous literature in this domain, very few works explicitly deal with app development *spanning multiple device classes*, indicating that app development beyond smartphones is not yet approached systematically but on a case-by-case basis. Cross-platform overview papers such as [JM15] typically focus on a single category of devices and apply a very narrow notion of mobile devices. [RM18] provides the only classification that includes novel device classes. Few papers provide a *technical* perspective on apps spanning multiple device classes. Singh and Buford [SB16] describe a cross-device team communication use case for desktop, smartphones, and wearables, and Esakia et al. [ENM15] performed research on the interaction between the Pebble smartwatch and smartphones in computer science courses. In the context of Web-of-Things devices, Koren and Klamma [KK16] propose a middleware approach to integrate data and heterogeneous UI, and Alulema et al. [AIC17] propose a DSL for bridging the presentation layer of heterogeneous devices in combination with web services for incorporating business logic.

With regard to commercial cross-platform products, Xamarin [Xam19] and CocoonJS [Lud19] provide Wear OS support to some extent. Whereas several other frameworks claim to support wearables, this usually only refers to accessing data by the main smartphone application or displaying notifications on already coupled devices.

Together with the increase in devices, new software platforms have appeared, some of which are either related to established operating systems (OS) for other device classes or are newly designed to run on multiple heterogeneous devices. Examples include Android / Wear OS, watchOS, and Tizen. Although these platforms ease the development of apps (e.g., reusing code and libraries), subtle differences exist in the available functionality and general cross-platform challenges remain.

10.5.3 Applicability of Existing Cross-Platform Approaches for Pluri-Platform Development

Different instantiations and practical frameworks may be conceived which extend the approaches to cross-platform development presented in Section 10.2.

A plethora of literature exists in the context of cross- or multi-platform development. Classifications such as in [El-+15] and [MWA15] have identified five main approaches to multi-platform app development which are varyingly suited to the specific challenges of pluri-platform development. With regard to runtime-based approaches, *mobile webapps* – including recently proposed Progressive Web Apps – are mobile-optimized web pages that are accessed using the device's browser and relatively easy to develop using web technologies. However, most novel device types, e.g., the major smartwatch platforms watchOS [App19] and Wear OS by Google [Goo19b], do not provide WebView components or browser engines that allow for the execution of JavaScript code. Consequently, this approach cannot be used for pluri-platform development targeting a broader range of devices.

Hybrid apps are developed similarly using web technologies but are encapsulated in a wrapper component that enables access to device hardware and OS functionality through an API. Although they are distributed as app packages via marketplaces, hybrid apps rely on the same technology and can neither be used for pluri-platform development.

In contrast, a *self-contained runtime* does not depend on the device's browser engine but uses platform-specific libraries provided by the framework developer in order to use native functionality. Of the runtime environment approaches, this is the only one that can be used for pluri-platform development. Although usually based on custom scripting languages, a runtime can also be used as a replacement for inexistent platform functionality. As an example, CocoonJS [Lud19] recreated a restricted WebView engine and, therefore, supports the development of JavaScript-based apps also for the Wear OS platform. However, devices need sufficient computing power to execute the runtime on top of the actual operating system. Also, synergies with regard to user input/output and available hardware/software functionality across heterogeneous devices are dependent on the runtime's API.

Considering generative approaches to cross-platform development, *model-driven software development* has several advantages as it uses textual or graphical models as main artefacts to develop apps and then generates native source code from this platform-neutral specification. Referring to domain-specific concepts allows for a high level of abstraction, for example circumventing issues such as input and output heterogeneity using declarative notations. Arbitrary platforms can be supported by developing respective generators which implement a suitable mapping from descriptive models to native platform-specific implementations.

Finally, *transpiling* approaches use existing application code and transform it into different programming languages. Pluri-platform development using this approach is technically possible as the result is also native code. However, there is more to app development than just the technical

equivalence of code, which also explains the low adoption of this approach by current cross-platform frameworks. For instance, user interfaces behave drastically differently across different device classes, and substantial transformations would be required to identify the contextual patterns from the low-level implementation. It is therefore unlikely that this approach provides a suitable means for pluri-platform development beyond reusing individual components such as business logic.

To sum up, only self-contained runtimes and model-driven approaches are candidates for pluri-platform development, of which the latter additionally benefits from the transformation of domain abstractions to platform-specific implementations.

10.5.4 Evaluation of MAML in a Pluri-Platform Context

The MAML notation condenses the development of apps to a sequence of data manipulation activities. Conceptually, the platform-agnostic nature and high level of abstraction allow for a wider applicability beyond just smartphone platforms. From the variety of novel app-enabled device classes [RM18], smartwatches have so far become most prevalent on the consumer market which offers a multitude of devices and several vendors promoting new platforms. Today's situation resembles the early experimental years after the introduction of the iPhone in 2007 and development using adequate abstractions is needed. Although a smartwatch typically has a touch screen, the screen dimension as well as the user input mechanisms, sensing capabilities, and usage patterns differ from smartphones. Ideally, pluri-platform development approaches can bridge the gap between app development not only for different smartwatches but integrate with existing ecosystems of current app-enabled devices. This is especially beneficial in a multi-device context in which one user owns several devices and can use platform-adapted apps depending on personal preferences or usage contexts.

To investigate the practical opportunities and challenges of pluri-platform development, we developed a new code generator for the Wear OS platform by Google (formerly Android Wear) [Goo19b] which supports the creation of stand-alone apps for respective smartwatches. Consequently, the model-driven foundation of our framework now allows for the combined generation of smartphone and smartwatch source code using the same MAML models as input. More details on the required transformations to represent the desired content on a smartwatch are presented in [RK19a]. Yet, in this work we want to focus on the usability of the notation with regard to the specification of apps across device classes.

Therefore, a second study was conducted in order to validate the previous results in the established smartphone domain and gain insights on the suitability towards other app-enabled device classes. The study was conducted with 23 students from a course on advanced concepts in software engineering of an information systems master's program. Whereas designing applications using MDSD techniques is part of the course contents and knowledge of process modelling notations can be presumed, no previous experience with app development was expected in order to avoid

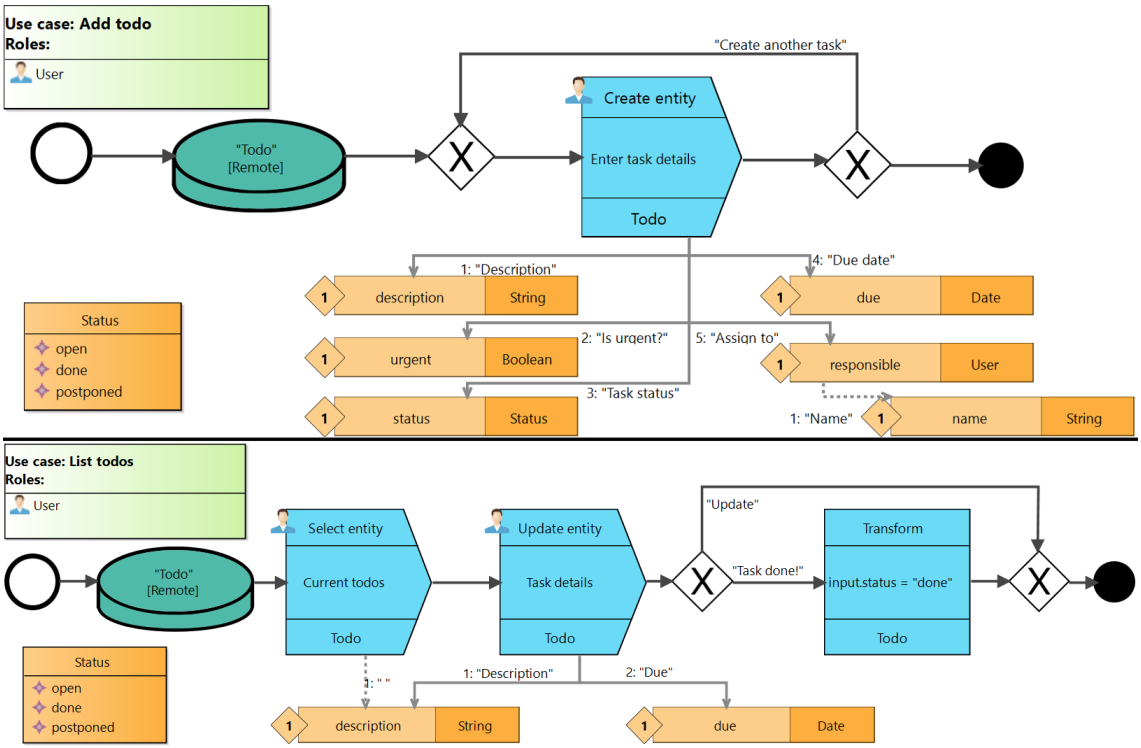


Figure 10.7: Use Cases for Adding and Displaying Items in a To-Do Management System [Rie17]

a bias towards existing frameworks or approaches. This is supported by the average responses regarding experience in the development of web apps (3.26), hybrid apps (4.35), and native apps (4.30) on a 5-point Likert scale.

Using a simple to-do management scenario depicted in Figure 10.7, a 5-minute introduction to the MAML notation was given to explain the two processes of creating a new to-do item in the system and displaying the full list of to-dos with the possibility to update items and ticking off the task. Subsequently, participants were asked to express their conceptions of the resulting apps by sketching smartphone and smartwatch user interfaces complying with these use cases.

Interestingly, 64% of the participants intuitively chose a square representation for the smartwatch screen, which reflects the publicity of the Apple watch. Also, a variety of interaction patterns could be derived from the sketches, for example the representation of repetitive elements as a vertical scrollable list (65%) in contrast to 17% using a horizontal arrangement. From the sketches that hint towards navigation patterns, the master-detail pattern of the "list todos" use case of Figure 10.7 was conceived either via tapping on the element (42%), using an edit button (33%), pressing a hardware button such as the watch crown (8%), swiping to the right hand side (8%), or using a voice command (8%).

As regards the creation of new entities, 35% of the participants imagined a scrollable view containing all attributes, whereas 30% decided for separate input steps for each attribute, 9% utilized the available screen dimensions and distributed the attributes across multiple views with

more than one attribute on each. In addition, 30% allowed the unstructured input of data via voice interface and 26% allowed voice inputs per attribute (the total above 100% results from multiple alternatives combined in the same sketches). It can also be said that a common perception of navigation within a smartphone app has not been established so far: From the identifiable navigation patterns, 53% relied on buttons to continue through the creation process whereas horizontal or vertical swipe gestures were depicted in 20% and 13%, respectively, and 14% decided to use one or multiple hardware buttons for navigation.

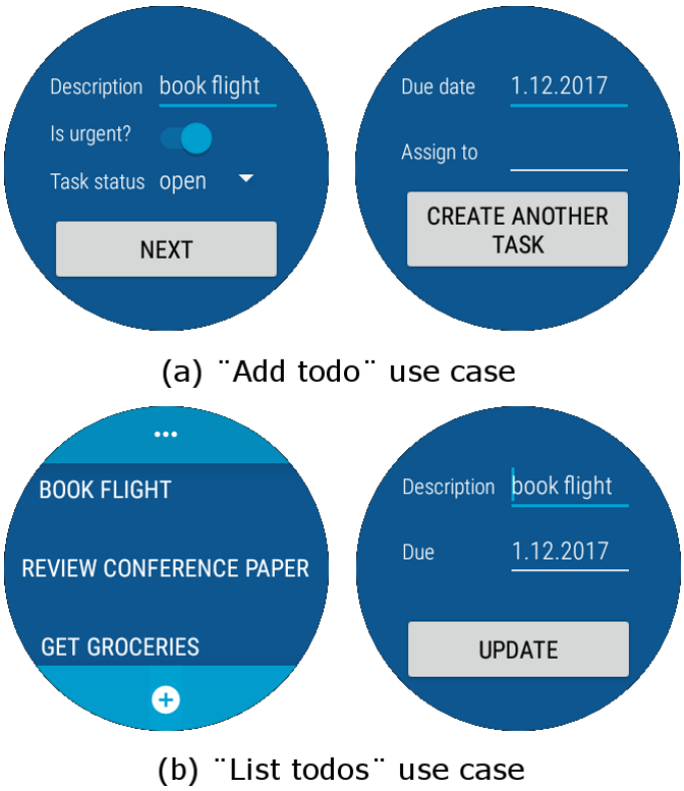


Figure 10.8: Generated Wear OS App for the System Modelled in Figure 10.7 (cf. [RK19a])

The standardized SUS questionnaire was used to triangulate the results with the initial study presented in Section 10.4. The resulting score of 66.85 ($\sigma = 12.9$) aligns very well with the figures depicted in Table 10.2 for software developers and process modellers and reinforces the validity of the previous study. Upon showing the generated app result depicted in Figure 10.8, the participants were asked about their opinion on the smartwatch outcome (using again a 5-point Likert scale). The participants agreed (2.04) that the generated smartwatch app suitably represents the process depicted in the MAML model. Furthermore, they supported the statement (2.39) that the resulting app is functional with regard to the to-do management scenario. The visual appearance of the smartwatch was rated merely with 3.3 which can be explained by the generic transformations and assumptions derived from the abstract process model. Also, the prototypical nature of our generator needs more refinements to choose suitable representations.

Regarding the combined generation of apps for smartwatch and smartphone from the same model, the participants did not feel that the common notation makes app development unnecessarily complex (3.35) and tended to agree that having one notation for both app representations accelerates app development (2.48). When asked about specific durations, the students estimated the required time to build the MAML models with 50 minutes on average, compared to a mean value of 27.3 hours when programming the application natively or with cross-platform programming frameworks. Though the actual development was not performed in this study, these estimates underline the possible economic impact of MDSD to reduce the effort for creating specific applications and thus achieve a faster time to market for new apps or app updates.

10.6 Discussion

In this section, key findings of the proposed MAML framework and subsequent evaluation are discussed with regard to the design objectives and general implications on model-driven software development for mobile applications across device classes. Regarding the principle of data-driven process modelling, using process flows in a graphical notation has shown to be a suitable approach for declaratively designing business apps. Graphical DSLs can also simplify modelling activities for the users of other domains, especially those that benefit from a visual composition of elements such as graph structures. Particularly for MAML, the chosen level of abstraction allows for a much wider usage compared to low-level graphical screen design: Besides the actual app product, models can be used to discuss and communicate small-scale business processes in a more comprehensive way than BPMN or similar process notations through combined modelling of process flows and data structures. In contrast to alternative codeless app development approaches focused on the graphical configuration of UI elements, users do not get distracted by the eventual position of elements on screen but can focus on the task to be accomplished. Moreover, the DSL is platform-agnostic and can thus be used to describe apps for a large variety of mobile devices. Apart from smartphones and tablets, generators for novel device types such as smartwatches or smart glasses may be created in the future based on the same input models.

Second, the challenge of developing a machine-interpretable notation that is understandable both for technical and non-technical users is a balancing act, but the interview observations and consistent scores in the evaluation indicate this design goal was reached. The most significant differences in the participants' modelling results are related to technical accuracy, mostly because of (missing) knowledge about programming or process abstractions. As such issues not always manifest as modelling errors but often happen through oversights, preventing them while keeping a certain joy of use is only achievable using a combined approach: The notation itself should be permissive instead of overly formal. Moreover, clarity (e.g., wording of UI elements) and simplicity of the DSL contribute to manageable models. Most important, however, is the extensive use of modelling support for different levels of experience. Novice users learn from hints (e.g., hover texts and error explanations) whereas advanced users can benefit from domain-specific validation

rules and optional perspectives to preview results of model changes. Particularly for MAML, advanced modelling support is achieved by interpreting the models and inferring a global object structure from a variety of partial data models as described in [Rie17]. Consequently, this feature allows for dynamically generated suggestions such as available data types, implicit reactions such as forbidding illegitimate element connections, and validation of conflicting data types and cardinalities. In general, a model-driven approach with advanced modelling support enables the active involvement of business experts in software development processes and can be regarded as major influencing factor for a successful integration of non-programmers.

Finally, the choice of mixing data model, business logic, and view details in a single model deviates from traditional software engineering practices in order to ease the modelling process for non-technical users. This does not mean that we recommend MAML for all process-oriented modelling tasks. Large business processes are just too complex to be jointly expressed with all data objects in a single model. However, mobile apps with small-scale tasks and processes are well suited to this kind of integrated modelling approach. The evaluation has shown that users appreciate the simplicity of the editor without switching between multiple interrelated models, a major distinction from related approaches to graphical mobile app development. Possibly related to the aforementioned modelling support, not even programmers miss the two-step approach of first specifying a global data model and then separately defining the respective processes. Nevertheless, as potential future extension, an optional view of the inferred data model may be interesting for them to check the modelling result before generation. Similarly, two non-technical users stated the wish for a preview of the resulting screens. However, both suggestions are neither meant to be editable nor mandatory for the app creation process and rather serve as reassuring validation while modelling the use case. It can therefore be said that modelling activities should suit the users' previous experience, potentially ignoring established concepts of technical domains for the greater good of a more comprehensible and seamless modelling environment.

As a result, bringing mobile app modelling to this new level of abstraction not only bridges the gap to the field of business process modelling but can also impact organizations. On the one hand, new technical possibilities arise from process-centric app models. For example, already documented business processes can be used as input for cross-platform development targeting a variety of heterogeneous mobile devices. On the other hand, codeless app generation creates the opportunity for different development methodologies. The distinction between app developer and framework developer can lead to performance benefits and better resource utilization on hardware-constrained devices such as smartphones. Best practices of mobile software development can be adopted by developers with expert knowledge of the respective platforms within the transformations which are then applied consistently throughout all generated apps. It has been shown that structural implementation decisions and even small-scale code refactorings can significantly improve battery consumption and execution times [RMZ17; SBB19]. Also, instead of involving domain experts only in requirements phases before the actual development, an equitable relationship with fast development cycles is possible because changes to the model can be deployed

instantly. Furthermore, future non-technical users may themselves develop applications according to their needs, extending the idea of self-service IT to its actual development. All of these ideas, however, rely on the modelling support provided by the environment, as begun with MAML's data model inference mechanism. Smart software to guide and validate the created models is required instead of simply representing the digital equivalent of a sheet of paper. In the future, graphical editors may evolve beyond just organizing and linking different models, towards tools enabling novel digital ecosystems through supportive technology.

10.7 Conclusion

In this work, a model-driven approach to mobile app development called MAML was presented which focuses around a declarative and platform-agnostic DSL to graphically create mobile business apps. The visual editor component provides advanced modelling support such as suggestions and validation through automatic data model inference. In addition, transformations allow for a codeless generation of app source code for multiple platforms. To evaluate the notation with regard to comprehensibility and usability, an extensive observational study with 26 participants was performed. The results confirm the design goals of achieving a wide-spread comprehensibility of MAML models for different audiences of software developers, process modellers, and domain experts. In comparison to the IFML notation, an equivalent MAML model is perceived as much less complex – in particular by non-technical users – and participants felt a high level of control, thus confidently solving their tasks. Furthermore, we analysed the challenges when extending the cross-platform approach to multiple app-enabled device classes. The applicability of MAML for this so-called *pluri-platform development* was assessed using a second study on a newly developed generator for the Wear OS smartwatch platform. As a result, MAML's approach of describing a mobile app as process-oriented set of use cases reaches a suitable balance between the technical intricacies of cross-platform app development and the simplicity of usage through the high level of abstraction and can be used to create app source code for both device classes from the same input models.

In case of the presented study results, some limitations may threaten their validity. Although a reasonable amount of participants was chosen for the observational interviews, additional evaluations may be carried out after the next iteration of MAML's development. Also, our participants were mostly students which potentially reduces the generalizability of the results. However, their generation of app-experienced adults already participates in the general workforce and can be seen as realistic (albeit not representative) sample. The synthetic examples within the case study were designed to test a wide range of MAML's capabilities and uncover usability issues. Therefore, a real-world application would strengthen the validity of the approach and at the same time represents future work.

Regarding limitations of the approach itself, the chosen level of abstraction requires assumptions on the generic representation of data in the prototype. Possibilities to customize low-level details

such as UI styling for different device classes need to be addressed in future, for example on the level of the intermediate MD²representation. Also, improvements of the generator prototype itself are part of ongoing work to provide a wide set of platform-adapted representations.

The presented process-oriented DSL offers the opportunity for research on a suitable framework structure for pluri-platform development and possible reuse of common transformations among multiple generators. Also, the process of developing such a framework of coupled components through a team with different roles may be investigated to further integrate model-driven techniques with traditional software development. Technically, further iterations on the framework's development are planned in order to provide additional user support, improve performance, and incorporate feedback based on the observed usability issues. Finally, the applicability of our approach to create business apps through model-driven transformations of MAML's platform-agnostic models to further device classes with drastically different UIs such as smart virtual assistants also presents exciting possibilities for future research.

References

[Aal99] W.M.P. van der Aalst. "Formalization and verification of event-driven process chains". In: *Information and Software Technology* 41.10 (1999), pp. 639–650. DOI: 10.1016/S0950-5849(99)00016-6.

[AIC17] D. Alulema, L. Iribarne, and J. Criado. "A DSL for the Development of Heterogeneous Applications". In: *FiCloudW*. 2017, pp. 251–257.

[Apa19] Apache Software Foundation. *Apache Cordova Documentation*. 2019. URL: https://cordova.apache.org/docs/en/latest/ (visited on 02/25/2019).

[App19] Apple Inc. *watchOS*. 2019. URL: www.apple.com/watchos/ (visited on 03/01/2018).

[Arc01] Architecture Board ORMSC. *Model Driven Architecture (MDA): Document number ormsc/2001-07-01*. 2001. URL: http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01 (visited on 12/03/2016).

[Bar+15] Scott Barnett et al. "A Multi-view Framework for Generating Mobile Apps". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 305–306. DOI: 10.1109/VLHCC.2015.7357239.

[BC00] J. A. Bargh and T. L. Chartrand. "Studying the mind in the middle: A practical guide to priming and automaticity research. Handbook of research methods in social psychology". In: *Handbook of research methods in social and personality psychology*. Ed. by C. M. Judd and H. T. Reis. Cambridge University Press, 2000, pp. 253–285.

[BDF09] M. Brambilla, M. Dosmi, and P. Fraternali. "Model-driven engineering of service orchestrations". In: *5th World Congress on Services* (2009). DOI: 10.1109/SERVICES-I.2009.94.

[Biz19] Bizness Apps. *Mobile App Maker | Bizness Apps*. 2019. URL: http://biznessapps.com/ (visited on 02/20/2019).

[BKF14] R. Breu, A. Kuntzmann-Combelles, and M. Felderer. "New Perspectives on Software Quality [Guest editors' introduction]". In: *IEEE Software* 31.1 (2014), pp. 32–38. DOI: 10.1109/MS.2014.9.

[BKM09] Aaron Bangor, Philip Kortum, and James Miller. "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale". In: *J. Usability Studies* 4.3 (2009), pp. 114–123.

[Bro96] John Brooke. "SUS-A quick and dirty usability scale". In: *Usability evaluation in industry*. Ed. by P. W. Jordan et al. Taylor and Francis, 1996, pp. 189–194.

[Bub19] Bubble Group. *Bubble - Visual Programming*. 2019. URL: https://www.bubble.is/ (visited on 02/20/2019).

[Buc18] Thomas Buchmann. "BXtend - A Framework for (Bidirectional) Incremental Model Transformations". In: *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2018. DOI: 10.5220/0006563503360345.

[dB14] L. P. da Silva and F. Brito e Abreu. "Model-driven GUI generation and navigation for Android BIS apps". In: *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2014, pp. 400–407.

[El-+15] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* (2015). DOI: 10.1016/j.asej.2015.08.004.

[ENM15] Andrey Esakia, Shuo Niu, and D. Scott McCrickard. "Augmenting Undergraduate Computer Science Education With Programmable Smartwatches". In: *SIGCSE*. 2015, pp. 66–71. DOI: 10.1145/2676723.2677285.

[Fra+06] R. B. France et al. "Model-Driven Development Using UML 2.0: Promises and Pitfalls". In: *Computer* 39.2 (2006), pp. 59–66. DOI: 10.1109/MC.2006.65.

[Ful+06] J. B. Fuller et al. "Perceived external prestige and internal respect: New insights into the organizational identification process". In: *Human Relations* 59.6 (2006), pp. 815–846. DOI: 10.1177/0018726706067148.

[Gam+95] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995.

[GH02] Günther Gediga and Kai-Christoph Hamborg. "Evaluation in der Software-Ergonomie". In: *Journal of Psychology* 210.1 (2002), pp. 40–57. DOI: 10.1026//0044-3409.210.1.40.

[Goo18] Google Inc. *J2ObjC*. 2018. URL: http://j2objc.org/ (visited on 02/20/2019).

[Goo19a] GoodBarber. *GoodBarber: Make an app*. 2019. URL: https://www.goodbarber.com/ (visited on 02/20/2019).

[Goo19b] Google Inc. *Wear OS by Google Smartwatches*. 2019. URL: https://wearos.google.com/ (visited on 02/20/2019).

[Gra+15] David Granada et al. "Analysing the cognitive effectiveness of the WebML visual notation". In: *Software & Systems Modeling* (2015). DOI: 10.1007/s10270-014-0447-8.

[HV11] Zef Hemel and Eelco Visser. "Declaratively Programming the Mobile Web with Mobl". In: *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2011, pp. 695–712. DOI: 10.1145/2048066.2048121.

[Int06] International Organization for Standardization. *ISO 9241-110:2006*. 2006.

[Int85] International Organization for Standardization. *ISO 5807:1985*. 1985.

[JJ14] Chris Jones and Xiaoping Jia. "The AXIOM Model Framework: Transforming Requirements to Native Code for Cross-platform Mobile Applications". In: *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, 2014.

[JM15] Chakajkla Jesdabodi and Walid Maalej. "Understanding Usage States on Mobile Devices". In: *ACM International Joint Conference on Pervasive and Ubiquitous Computing*. UbiComp. ACM, 2015, pp. 1221–1225. DOI: 10.1145/2750858.2805837.

[KK16] István Koren and Ralf Klamma. "The Direwolf Inside You: End User Development for Heterogeneous Web of Things Appliances". In: *Web Engineering: 16th International Conference, ICWE. Proceedings*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Springer, 2016, pp. 484–491. DOI: 10.1007/978-3-319-38791-8_35.

[LJJ07] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. "DSL classification". In: *The 7th OOPSLA Workshop on Domain-Specific Modeling*. 2007.

[Lud19] Ludei Inc. *Canvas+ Cocoon Documentation*. https://docs.cocoon.io/article/canvas-engine/. 2019. (Visited on 02/20/2019).

[ME15] T. A. Majchrzak and J. Ernsting. "Reengineering an Approach to Model-Driven Development of Business Apps". In: *8th SIGSAND/PLAIS EuroSymposium 2015, Gdansk, Poland*. 2015, pp. 15–31. DOI: 10.1007/978-3-319-24366-5_2.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[Moo09] Daniel Moody. "The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 756–778.

[MWA15] T. A. Majchrzak, S. Wolf, and P. Abbassi. "Comparing the Capabilities of Mobile Platforms for Business App Development". In: *Information Systems: Development, Applications, Education: 8th SIGSAND/PLAIS EuroSymposium.* Ed. by Stanislaw Wrycza. Springer International Publishing, 2015, pp. 70–88. DOI: 10.1007/978-3-319-24366-5_6.

[NJE17] Timothy Neate, Matt Jones, and Michael Evans. "Cross-device Media: A Review of Second Screening and Multi-device Television". In: *Personal Ubiquitous Comput* 21.2 (2017), pp. 391–405. DOI: 10.1007/s00779-017-1016-2.

[Obj11] Object Management Group. *Business Process Model and Notation.* 2011. URL: http://www.omg.org/spec/BPMN/2.0.

[Obj15] Object Management Group. *Unified Modeling Language.* 2015. URL: http://www.omg.org/spec/UML/2.5.

[Pen17] Pentaho Corp. *Data Integration - Kettle.* 2017. URL: http://community.pentaho.com/projects/data-integration/ (visited on 06/15/2017).

[Pro16] Product Hunt. *7 Tools to Help You Build an App Without Writing Code.* 2016. URL: https://medium.com/product-hunt/7-tools-to-help-you-build-an-app-without-writing-code-cb4eb8cfe394 (visited on 06/02/2017).

[Rie17] Christoph Rieger. "Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development". In: *Proceedings of the 32nd Annual ACM Symposium on Applied Computing.* 2017, pp. 1599–1606.

[Rie18] Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *51st Hawaii International Conference on System Sciences (HICSS).* 2018, pp. 5725–5734.

[Rie19] Christoph Rieger. *MAML Code Respository.* 2019. URL: https://github.com/wwu-pi/maml.

[RK18] Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *Computer Languages, Systems & Structures* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001.

[RK19a] Christoph Rieger and Herbert Kuchen. "A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes". In: *52nd Hawaii International Conference on System Sciences (HICSS).* 2019, pp. 7431–7440.

[RK19b] Christoph Rieger and Herbert Kuchen. "Towards Pluri-Platform Development: Evaluating a Graphical Model-Driven Approach to App Development Across Device Classes". In: *Towards Integrated Web, Mobile, and IoT Technology*. Ed. by Tim A. Majchrzak et al. Vol. 347. Springer International Publishing, 2019, pp. 36–66. DOI: 10.1007/978-3-030-28430-5_3.

[RM18] Christoph Rieger and Tim A. Majchrzak. "A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle". In: *Web Information Systems and Technologies*. Ed. by Tim A. Majchrzak et al. Springer International Publishing, 2018, pp. 202–220.

[RMZ17] Ana Rodriguez, Cristian Mateos, and Alejandro Zunino. "Improving scientific application execution on android mobile devices via code refactorings". In: *Software: Practice and Experience* 47.5 (2017), pp. 763–796. DOI: 10.1002/spe.2419.

[Rv14] Janessa Rivera and Rob van der Meulen. *Gartner Says By 2018, More Than 50 Percent of Users Will Use a Tablet or Smartphone First for All Online Activities*. 2014. URL: http://www.gartner.com/newsroom/id/2939217 (visited on 12/03/2016).

[RZ15] Andreas Reiter and Thomas Zefferer. "POWER: A cloud-based mobile augmentation approach for web- and cross-platform applications". In: *CloudNet*. IEEE, 2015, pp. 226–231. DOI: 10.1109/CloudNet.2015.7335313.

[SB16] K. Singh and J. Buford. "Developing WebRTC-based team apps with a cross-platform mobile framework". In: *IEEE CCNC* (2016). DOI: 10.1109/CCNC.2016.7444762.

[SBB19] Hareem Sahar, Abdul A. Bangash, and Mirza O. Beg. "Towards energy aware object-oriented development of android applications". In: *Sustainable Computing: Informatics and Systems* 21 (2019), pp. 28–46. DOI: 10.1016/j.suscom.2018.10.005.

[SS01] Jared Spool and Will Schroeder. "Testing Web Sites: Five Users is Nowhere Near Enough". In: *CHI '01 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2001, pp. 285–286. DOI: 10.1145/634067.634236.

[SW07] C. Simons and G. Wirtz. "Modeling context in mobile distributed systems with the UML". In: *Journal of Visual Languages and Computing* 18.4 (2007), pp. 420–439. DOI: 10.1016/j.jvlc.2007.07.001.

[The19] The Eclipse Foundation. *Sirius*. 2019. URL: https://eclipse.org/sirius/ (visited on 02/20/2019).

[Tho12] Gordon Thomson. "BYOD: enabling the chaos". In: *Network Security* 2012.2 (2012), pp. 5–8. DOI: 10.1016/S1353-4858(12)70013-2.

[UB16] Eric Umuhoza and Marco Brambilla. "Model Driven Development Approaches for Mobile Applications: A Survey". In: *Mobile Web and Intelligent Information Systems: 13th International Conference*. Springer International Publishing, 2016, pp. 93–107. DOI: 10.1007/978-3-319-44215-0_8.

[Vau+14] S. Vaupel et al. “Model-driven development of mobile applications allowing role-driven variants”. In: *Lecture Notes in Computer Science* 8767 (2014), pp. 1–17.

[Vir92] Robert A. Virzi. “Refining the Test Phase of Usability Evaluation: How Many Subjects is Enough?” In: *Hum. Factors* 34.4 (1992), pp. 457–468.

[Web19] WebRatio. *WebRatio*. 2019. URL: http://www.webratio.com.

[Wol11] D. Wolber. “App inventor and real-world motivation”. In: *42nd ACM Technical Symposium on Computer Science Education (SIGCSE)* (2011). DOI: 10.1145/1953163.1953329.

[Xam19] Xamarin Inc. *Developer Center - Xamarin*. 2019. URL: https://developer.xamarin.com (visited on 02/20/2019).

[Zył15] Kamil Zyła. “Perspectives of Simplified Graphical Domain-Specific Languages as Communication Tools in Developing Mobile Systems for Reporting Life-Threatening Situations”. In: *Studies in Logic, Grammar and Rhetoric* 43.1 (2015). DOI: 10.1515/slgr-2015-0048.

11

Musket: A Domain-Specific Language for High-Level Parallel Programming with Algorithmic Skeletons

Table 11.1: Fact sheet for publication P5

Title	Musket: A Domain-Specific Language for High-Level Parallel Programming with Algorithmic Skeletons
Authors	Christoph Rieger[1] Fabian Wrede[1] Herbert Kuchen[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2019
Conference	34th Annual ACM Symposium on Applied Computing (SAC)
Copyright	ACM
Full Citation	Christoph Rieger, Fabian Wrede, and Herbert Kuchen. "Musket: A Domain-specific Language for High-level Parallel Programming with Algorithmic Skeletons". In: *34th ACM/SIGAPP Symposium on Applied Computing (SAC)*. Limassol, Cyprus: ACM, 2019, pp. 1534–1543. DOI: 10.1145/3297280.3297434 This version is not for redistribution. The final authenticated version is available online at https://doi.org/10.1145/3297280.3297434.

Musket: A Domain-Specific Language for High-Level Parallel Programming with Algorithmic Skeletons

Christoph Rieger Fabian Wrede Herbert Kuchen

Keywords: Domain-Specific Language, Algorithmic Skeletons, High-level Parallel Programming, High Performance Computing

Abstract: Parallel programming for an infrastructure of multi-core or many-core clusters is a challenge for developers without experience in this domain. Developers need to use several libraries such as MPI, OpenMP, and CUDA to efficiently use the hardware which may include additional accelerators such as GPUs. Also, performing low-level optimizations is required in order to reach high performance. One approach to overcome these issues is the concept of *Algorithmic Skeletons*. These are instances of typical patterns for parallel programming, such as map, fold, and zip, which can simply be composed by an application programmer without taking care of low-level programming aspects.

We propose a domain-specific language called Musket that includes algorithmic skeletons as domain abstractions which seamlessly integrate with sequential code while aligning with the C++ programming language for fast learnability. For improved usability, the editing component validates the correctness of models and provides solution hints in the integrated development environment. From the naïve program specification, automatic transformations are applied in order to optimize the code for parallel execution. Subsequently, low-level C++ programs are generated which are optimized for multi-core parallelism on a cluster infrastructure.

We evaluate the language using benchmark models written in our DSL and compare the execution time and speedup achieved through model preprocessing and code generation. Our experimental results show that the performance of Musket programs can be significantly improved through intermediate optimizations. The DSL approach thus simplifies multi-core application development and enables performance optimizations through model transformations.

11.1 Introduction

Parallel programming is a difficult and error-prone endeavour. There are several pitfalls such as deadlocks or race conditions, which lead to errors that are often very difficult to understand and track for inexperienced programmers. Moreover, in the area of High Performance Computing (HPC), programs are typically executed on clusters, which consist of multiple nodes, i.e., separate computers, which in turn are equipped with multi-core CPUs and possibly as well with accelerators such as Graphical Processing Units (GPUs). Consequently, multiple frameworks have to be used in order to fully utilize the available hardware. These frameworks might for example comprise OpenMP for multi-core CPUs [CJv08], MPI for multiple nodes [GLS14], and CUDA for GPUs [Nic+08]. While these frameworks can already be difficult to use on their own, it is even more challenging, when the frameworks are used in conjunction.

At the same time, HPC programs are often written in a scientific context, for example by physicists or biologists, who need to run complex simulations or to solve numerical problems. Since it requires a lot of time to develop knowledge in the mentioned frameworks and additional knowledge regarding the potentially required low-level performance tuning of applications, we propose a domain-specific language (DSL), which is simple to use, offers good performance, shields the user from low-level tasks such as memory management, and does not require any kind of low-level performance tuning.

The main focus of the language is the concept of Algorithmic Skeletons [Col91] which represent parametric implementations of well-known patterns for functional programming such as map, fold (a.k.a. reduce), and zip. Applied to distributed data structures, skeletons can be utilized in conjunction with a custom user function to manipulate the data in parallel using multiple computation nodes. Consequently, high performance clusters with multi-core computation nodes and potentially multiple GPUs can be utilized to capacity. In contrast to traditional parallel code, the programming approach using skeletons becomes more structured: the programmer has to define data structures, user functions, and the main program as a composition of skeletons. At the same time, this approach is less error-prone because all low-level details are invisible for the programmer and common errors such as deadlocks do not occur in the sequential specification of activities.

The paper is structured as follows: We will first further outline the basic principles of parallel programming with algorithmic skeletons in Section 11.2. Afterwards, related work is described in Section 11.3 to show which alternative approaches exist. Section 11.4 and Section 11.5 outline the main contributions of the paper, i.e., the design of the DSL and how code optimizations are performed. The approach is evaluated in Section 11.6 before concluding in Section 11.7.

11.2 Foundations

Parallel programming differs significantly from sequential programming and introduces new pitfalls, such as deadlocks and race conditions. Consequently, it requires time and effort to learn this new way to write programs. Additionally, even more effort is required for performing optimizations in order to obtain efficient applications and finally, all those skills have to be obtained for different frameworks, such as MPI for distributed memory architectures, OpenMP for shared-memory architectures, and CUDA for GPUs.

Cole [Col91] originally proposed *Algorithmic Skeletons* to overcome these issues. Algorithmic Skeletons are well-known parallel patterns, which can be regarded as templates and composable building blocks. The programmer decomposes the program into a sequence of skeleton calls and provides a custom *user function*, which is executed within the chosen skeleton. Consequently, the programmer does not need to worry about the underlying implementation of the skeleton.

This has multiple advantages: First, the structured approach, i.e., selecting the skeleton composition and providing operations, guides novice application programmers towards parallel

programming. Second, the framework ensures that a respective transformation is automatically applied to the data elements without dealing with the actual implementation of the skeleton and potential optimizations. Moreover, the skeletons can be provided for multiple architectures. This facilitates performance portability. In theory, a code base, which can work on a certain architecture, e.g., multi-core cluster, could also run on a different architecture, e.g., a GPU cluster.

Algorithmic skeletons can be clustered into different groups. A common distinction is made between *data-parallel* and *task-parallel* skeletons [GL10]. Data-parallel skeletons are for example *map* and *fold* (a.k.a. reduce). Map applies a given function to each element of a data structure, such as converting a list of strings to upper case. Fold reduces elements of a data structure to a single element, such as calculating the sum of a list of numbers. Task-parallel skeletons are for example *pipeline* (sequential execution of tasks in which elements in different stages can be handled in parallel) and *farm* (a master node distributing workload across a number of parallel workers).

Many libraries which provide algorithmic skeletons offer specific containers for data elements. For example, the Muenster Skeleton Library (Muesli) [EK17] and SkePU [ELK17] are two C++ libraries for algorithmic skeletons. In Muesli, these data structures are required to utilize multiple nodes and skeletons are offered as member functions of those data structures. In SkePU, data is organized in so-called smart containers which perform tasks such as software caching and memory management, and aim to optimize communication at runtime.

11.3 Related Work

There are multiple approaches to introduce algorithmic skeletons in the specification of parallel programs, typically as a library or a DSL. Libraries can be clustered based on different criteria, such as the programming paradigm and the language. An overview can be found in [GL10] and we can unfortunately only mention selected examples. In contrast to the most prominent solutions, which are either libraries or internal DSLs, with Musket we focus on an external DSL for HPC computing with algorithmic skeletons.

FastFlow is a library that targets stream-parallel skeletons for C++ [Ald+10]. Data streams are for instance encountered in video processing. The most important skeletons are the pipeline and the farm, which may additionally have a feedback loop. These skeletons can be nested to depict very complex patterns. FastFlow has also been extended to work with clusters and additional accelerators.

SkePU is a C++ template library that provides common skeletons such as map, reduce, mapOverlap etc. [ELK17]. All skeletons are implemented with multiple back-ends, i.e., sequential, OpenMP, CUDA, and OpenCL. Additionally, SkePU offers the already mentioned smart containers and a tunable context-aware selection of skeleton implementations. For example, a suitable implementation is selected based on the operand size.

The Muenster Skeleton Library (Muesli) is also a C++ library [Kuc02; CK10; EK12; EK17]. Muesli's first-class entities are distributed data structures, which enable the distribution of elements among multiple processes. Algorithmic skeletons are provided as member functions of these data structures. The skeletons are implemented with OpenMP for multi-core clusters and with CUDA to support clusters with multi-GPU nodes. As skeletons, map, fold, zip, and mapStencil are available in different variants, to, e.g., enable manipulating data structures in place instead of returning a new instance.

In contrast to the aforementioned library examples, a domain-specific language is an alternative approach to introduce algorithmic skeletons. Such a language can either be embedded into a host language (internal DSL) or be designed as stand-alone language (external DSL). A DSL approach by Janjic et al. [Jan+16] simplifies the parallelisation of legacy C++ code by providing semi-automated refactoring support. Danelutto et al. [DTK16] also propose a DSL, which allows the setup of FastFlow programs. The skeleton composition can be expressed with the DSL and the generated output is a template for FastFlow code, which can be completed with custom user functions. The advantage of the DSL is that the pattern composition can be rewritten to a functionally equivalent but more efficient composition and that it is optimized based on non-functional properties, such as power consumption. In contrast, Musket directly generates optimized C++ code so that the programmer is not required to implement user functions in a separate second step.

A related approach, which uses an internal DSL, is SPar [Gri+17]. The advantage is that the approach is non-invasive. An existing code base is annotated with standard C++11 attributes. Subsequently, these SPar attributes are used to perform a source-to-source transformation. The output is a fully functional parallel program using the already mentioned FastFlow library. In contrast, our approach uses an external DSL which allows for more flexible transformation logic during code generation.

Besides libraries and DSLs for algorithmic skeletons, other areas of research are also relevant in the context of this work. Steuwer et al. [Ste+15] developed rewrite rules for parallel patterns, i.e., formalized transformations for algorithmic skeletons. The author primarily focused on GPU programming. While the results show that rewriting high-level parallel expressions can lead to very good performance, the authors did not consider the aspect of expressing such models or programs in an easy and concise way.

There are also general purpose languages that focus on parallel and distributed programming such as Chapel [CCZ07]. Chapel's syntax is close to C, C++, and Fortran so that it becomes easy to learn for users who are familiar with those languages. In order to support parallel programming, Chapel has built-in types, statements, control flows, etc. For example, the basic *locale* type can represent a node in a cluster environment, which can be used to manage the data distribution onto physical machines.

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

const int DIM = 16384;

matrix<float,DIM,DIM,dist> as = {1.0f};
matrix<float,DIM,DIM,dist> bs = {0.001f};
matrix<float,DIM,DIM,dist> cs = {0.0f};

float dotProduct(int i, int j, float Cij){
 float sum = Cij;
 for (int k=0; k < cs.columnsLocal(); k++){
  sum += as[[i,k]] * bs[[k,j]];
 }
 return sum;
}

main{
 as.shiftPartitionsHorizontally((int a) -> int {return -a;});
 bs.shiftPartitionsVertically((int a) -> int {return -a;});

 for (int i = 0; i < as.blocksInRow(); ++i){
  cs.map<localIndex, inPlace>(dotProduct());
  as.shiftPartitionsHorizontally((int a)-> int {return -1;});
  bs.shiftPartitionsVertically((int a) -> int {return -1;});
 }

 as.shiftPartitionsHorizontally((int a) -> int {return a;});
 bs.shiftPartitionsVertically((int a) -> int {return a;});
}
```

Listing 11.1: Musket Model for Matrix Multiplication Using the Cannon Algorithm

11.4 Musket DSL

To alleviate the problems of other solutions to high-level parallel programming – especially the simplification for potential users and the optimization for low-level program execution – we propose a DSL named *Musket* (Muenster Skeleton Tool for High-Performance Code Generation) which generates optimized C++ code that runs on heterogeneous clusters.

11.4.1 Language Structure

The Musket DSL targets programmers who want to use algorithmic skeletons to quickly write distributed parallel programs but have limited specific knowledge of low-level parallel programming. Therefore, a syntax similar to C++ was chosen to align with a programming language that is common for high-performance scenarios such as simulating physical or biological systems.

The DSL was created using the Xtext language development framework which uses an EBNF-like grammar to specify the language syntax and derives a corresponding Ecore meta model [The18]. In contrast to working with C++ directly, e.g., through template metaprogramming [Cza98], the DSL approach offers more flexibility for the design and analysis of programming constructs, especially with regard to automatic transformations.

To guide users in creating a valid parallel program, a Musket model is more structured than an arbitrary C++ program and provides four main sections which are described in more detail in the following. The language core is therefore a subset of the C++ language scope, enriched with domain-specific concepts. However, arbitrary functions can be called from within the model such that the full scope of C++ is indirectly supported (of course, optimizations based on static analysis are then limited due to unforeseeable side effects). Some major domain-specific additions are presented in **??**; for the full language specification the reader is referred to the code repository [WR18].

Furthermore, the Xtext framework generates a parser as well as an editor component which integrate with the Eclipse ecosystem for subsequent code generation. Consequently, common features of an IDE such as syntax highlighting, auto-completion, and validation are implemented and have been customized to provide contextual modelling support as discussed in Subsection 11.4.1.

The following subsections describe the structure of Musket models based on an example for matrix multiplication according to the Cannon algorithm (described in more detail in [EK17]) as depicted in **??**. First, the input matrices are distributed between the processes. Second, the matrix multiplication is performed by subsequent parallel multiplications of submatrices. Between each computation step, the input submatrices are shifted between the processes (thus, transferring the local data partition to the subsequent computation node) such that eventually each process has computed the correct submatrix of the final result.

Meta Information

In order to support the generation process and optimize models for the target hardware, a few pieces of meta information are contained within the model headers (lines 1-4). On the one hand, the target *platforms* (sequential, multi-core CPU, or different generator implementations) and the compiler optimization *mode* can be chosen for convenient experimentation and debugging of the program. Moreover, the configuration of *cores* and *processes* is important input for the generator to optimize the code for a distributed execution on a high-performance cluster of a given size. For example, the creation and distribution of data structures, the parallel execution of skeletons, and the intra-cluster communication of calculation results are then automatically pre-configured and managed.

Data Structure Declaration

To highlight the visibility of data structures which can be accessed during the distributed execution of the program, globally available arrays and matrices are declared before the actual application behaviour in the model (lines 8-10). Also, global constants can be defined in this block to easily parametrize the program (line 6).

Currently, Musket supports several primitive data types (*float, double, integer*, and *boolean*) for variable declarations and global constants. *Array* and *matrix* collection types are defined using the C++ template style, e.g., `matrix<double,512,512,dist> table;` . This definition contains the type and dimension of the collection, and also provides a keyword indicating whether the values should be present on all nodes (*copy*), distributed across the nodes in equally sized blocks (*dist*), by row (*rowDist*), column-wise (*columnDist*), or locally instantiated depending on the surrounding context (*loc*). These specific distribution strategies account for cases in which an algorithm (defined within the user functions; see Subsection 11.4.1) relies on a specific partitioning scheme of the data. In order to ease the handling of elements within a distributed data structure, collection elements can be accessed either using their global index (e.g., `table[42]`) or the local index within the current partition (e.g., `table[[42]]`).

Moreover, primitive and collection types can be composed into custom *struct* types to define objects with nested attributes. Although object-oriented programming and structs are generally available in C++, the actual parallel computation needs to be performed on plain arrays of primitive values both for performance reasons and limitations of the computational nodes (e.g., considering data transfer between nodes or between host memory and GPU memory). Therefore, structs in Musket are transformed by the generator to respective low-level representations.

User Function Declaration

As described in Section 11.2, algorithmic skeletons are usually based on custom user functions performing the actual calculation on each node. To separate these individual calculations from the overall program flow, the next section of a Musket model contains the user functions (such as the `dotProduct` function in lines 12-18). A wide variety of statements can be directly expressed in the DSL, including *arithmetic* and *boolean expressions*, *type casts*, and *assignments*. In addition, different control flow structures such as *sequential composition*, *if statements*, and *for loops* are available. Beyond the provided functionality, the modeller can use C++ functions from the standard library or call arbitrary external C++ functions but these are neither further analysed nor optimized.

Moreover, several so-called *CollectionFunction*s have been implemented as helper functions to determine the local/global amount of rows, columns, and values within a collection structure (e.g., the locally available amount of columns in line 14 of **??**). *MusketFunction*s additionally provide parallelism-aware functionality such as random number generation or printing of values/ collections.

Within a user function, users can access all globally available data structures (declared in the previous section) or create local variables to store temporary calculation results which are not available to other processes. The system automatically deals with the allocation of memory and the transfer of data to the computation nodes.

Main Program Declaration

The composition of skeletons into a full program is depicted in lines 20-32 of **??**. The *main* block may contain the same control structures and expressions as described in the previous paragraphs. In addition, *skeletons* are the main features to write high-level parallel code. In general, skeletons are applied to a distributed data structure and may take additional arguments such as the previously defined user functions. For convenience and code readability reasons, the user can alternatively specify a lambda abstraction for simple operations, e.g., `(int a)-> int {return -a;}`.

Currently, *map*, *fold*, *zip*, and *shift partition* skeletons are implemented in multiple variants as explained in the following. An excerpt of the Musket DSL specification concerning the main program declaration is depicted in **??**.

Map The basic *map* skeleton applies a user function to each element of a data structure and returns a new data structure of equal size. In order to avoid this memory-intensive copying step, the *mapInPlace* variant directly updates the input data structure. Also, *mapIndex* and *mapLocalIndex* represent variants of context-dependent operations that make use of the current index – within the global collection or the current partition, respectively – as additional input. Consequently, the user function receives one (for arrays) or two (for matrices) additional parameters. A combination of both variants into a context-dependent map operation which operates in place is also possible (e.g., line 25 in **??**).

Fold The *fold* skeleton, also known as reduce pattern, joins all elements of a collection according to a specified associative aggregation function (e.g., sum, min, max). In order to achieve a parallel execution by subsequently folding two elements over multiple levels, an identity value needs to be provided that does not alter the calculation. Again, a variant includes the current index as additional parameter.

Zip The *zip* skeleton takes two data structures of equal size as input and applies a user function to each pair of values, thus effectively merging two collections into one. Possible variants are the same as for the map skeleton, i.e., passing the current index (*zipIndex*/*zipLocalIndex*) and updating the first data structure with the result of the user function (*zipInPlace*).

Shift partition The *shift partitions horizontally* and *shift partitions vertically* skeletons simplify the data transfer between computation nodes by sending the content of the local partition to another computation node in bulk instead of transferring individual values on demand. For

```
MainFunctionStatement:
MusketControlStructure | MusketStatement';';

MusketStatement:
MusketVariable |   //variable declarations
MusketAssignment | //variable assignment
Expression |       //arithmetic expressions
FunctionCall       //function calls
;

Expression: // skeleton calls
 obj=[CollectionObject] '.' skel=Skeleton
;

Skeleton: // available algorithmic skeletons
 {MapSkeleton} 'map'
     ('<' options+=MapOption (',' options+=MapOption)* '>')?
     '(' param=SkeletonParameterInput ')' |
 {FoldSkeleton} 'fold'
     ('<' options+=FoldOption (',' options+=FoldOption)* '>')? '(' identity=Expression ',' ↩
     param=SkeletonParameterInput ')'|
 {MapFoldSkeleton} 'mapFold'
     ('<' options+=FoldOption (',' options+=FoldOption)* '>')? '(' ↩
     mapFunction=SkeletonParameterInput ',' identity=Expression ',' ↩
     param=SkeletonParameterInput ')' |
 {ZipSkeleton} 'zip'
     ('<' options+=ZipOption (',' options+=ZipOption)* '>')?
     '(' zipWith=ObjectRef ',' param=SkeletonParameterInput ')' |
 {ShiftPartitionsHorizontallySkeleton}
     'shiftHorizontally' '(' param=SkeletonParameterInput ')' |
 {ShiftPartitionsVerticallySkeleton}
     'shiftVertically' '(' param=SkeletonParameterInput ')'
; // alternative representations omitted

enum MapOption: index | localIndex |inPlace;
enum FoldOption: index;
enum ZipOption: index | localIndex |inPlace;

SkeletonParameterInput:
InternalFunctionCall | // user function ref.
LambdaFunction         // inline definition
;
```

Listing 11.2: Musket DSL Specification of Skeleton Expressions

example, the Cannon algorithm for matrix multiplication (cf. **??**) makes use of this pattern in order to perform a series of partial computations on specified subsets of the data.

The arguments of skeletons are partially applied functions [JP01]. The partially applied arguments are the user-defined parameters within the skeleton call which are first bound to the referenced user function. The remaining arguments are supplied by the respective skeleton variant, in particular one or two index parameters (for arrays/matrices) as well as the actual value of the collection element.

As an example, the skeleton call on a matrix of float values `aMatrix.map<index>(calc(42));` complies with the user function `float calc(int b, int i, int j, float Aij)` which specifies a parameter *b* (bound to a fixed value of 42), two integers (for the matrix indices), and the respective element value *Aij* from the `aMatrix` collection.

Consequently, the correspondence of user functions and skeleton calls can be validated with regard to type and number of parameters while modelling (see next subsection). However, partial function application only exists within the DSL and is transformed to plain method calls during the generation phase for performance reasons. Functions can thus be rewritten more extensively and their explication in source code reduces dynamic calls at runtime.

Modelling Support

The creation of models is significantly eased through customised modelling support. The Xtext framework already provides a suitable integration with the Eclipse IDE with features such as syntactical validation, code auto-completion, and syntax highlighting. Domain-specific validation logic was implemented to add modelling support for the dynamic semantics as depicted in Figure 11.1:

Type checking Users can define variables and collection data structures within the model based on primitive and custom data types. A type checking mechanism then aims to statically analyse the resulting data type of assignments, function calls, and expressions by recursively checking the parameter and return types along the abstract syntax tree of the nested expressions. Consequently, the user can be warned about type errors already in the IDE before vainly starting the generation process. To align with the C++ syntax, an explicit type specification is still possible.

User function conformity The combination of skeleton variant and user function is analysed to match the amount and type of input parameters to be passed during execution. Also, return types of user functions are checked against the expected skeleton output.

Data type conformity Validators ensure that skeleton functions can only be called on applicable data structures. Also, assignments are checked to match the resulting type of expressions.

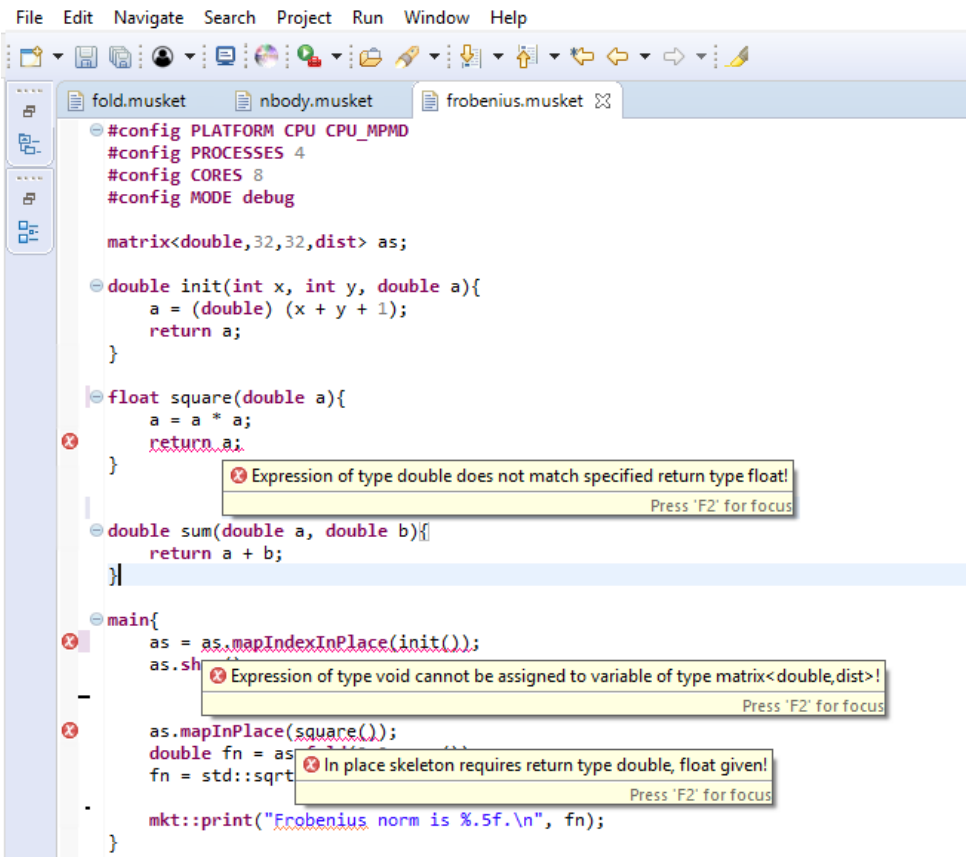


Figure 11.1: Integration of Model Validation in the Eclipse IDE

Together, the sophisticated validation capabilities allow for instant contextual feedback to the user when errors are introduced in the model. Thus, the Musket framework prepones the identification of model inconsistencies, before executing the application on a cluster environment which frequently incurs queue times or enforces a limited quota allocation.

11.4.2 Benefits of Generating High-Performance Code

Using libraries for high-performance computing introduces a performance overhead resulting from runtime calls and calculations as described in [WRK18] with up to 29% overhead on the real-world Fish School Search (FSS) example (cf. Subsection 11.6.3). From a usage perspective, the main drawback of using libraries is the restriction to the host language's syntax. First-class concepts of the particular domain need to be implemented using available capabilities (e.g., using C++ template programming to hide computational details or macro functionality workarounds to rewrite the final representation). In contrast, the structure and syntax of a DSL are important design decisions and can be selected purposefully to the requirements of the targeted users. For example, algorithmic skeletons as major domain concept for parallelisation can be integrated as

keywords in the designed language and handled by the editing component. Consequently, the program specification is more readable for novice users in this domain.

In addition, a library is reactive with regard to operations called by the user. Semantic errors can only be identified at runtime without custom tools (which are typically not provided for libraries), causing the repeated effort of building the application and its deployment on the cluster environment. Although it can be integrated in arbitrarily complex code of a language such as C++, this flexibility complicates the analysis of models and limits the internal optimization of programs to the execution of a particular task.

Instead, a code generator for a DSL can easily analyse and preprocess model characteristics based on a formal meta model and produce hardware-optimized code. Appropriate transformations can be provided by the framework developers, e.g., when applying an algorithmic skeleton to different distributed data structures. In addition, this separation of high-level program specification and low-level parallel execution increases the readability for users who do not need to know the details of (potentially multiple) target platforms but can focus on the abstract sequence of activities.

From a long term perspective, a DSL-based approach can be extended to cover additional platforms in the future by supplying new generator implementations – without need for changing the input programs. Compared to customizing compilers, DSL creation frameworks such as Xtext further support in creating usable editing components with features such as syntax highlighting and meaningful model validation [Bet13].

With regard to framework developers who are concerned with efficient program execution, DSLs introduce additional flexibility. The abstract syntax of the parallel program can be analysed and modified in order to optimize the generated high-performance code for the target hardware. In particular, inefficient patterns of user code can be transformed to hardware-specific low-level implementations by applying rewrite rules as described next.

11.5 Model Transformation

The primary aim of Musket is the creation of low-level code for parallel application execution. In this context, a DSL allows for the optimization of user models in order to increase performance. Moreover, the modeller can be disburdened through reasonable default behaviour as well as abstract constructs which can be rewritten via model preprocessing. Within the generation process, input models are first validated by the Xtext framework for syntactical correctness as well as custom validation logic to ensure semantically valid models and provide practical modelling support as described in the previous section. Subsequently, a series of self-contained transformations are executed which modify the original model. These transformations either optimize the model provided by the user or simplify the model in order to reduce the complexity within the generator (e.g., when using alternative representations for the same concept such as lambda abstractions instead of user functions).

```
1  struct Pos { double x, y; }
2  Pos origin;
3  array<Pos,512,dist> positions, result;
4
5  Pos moveTowards(Pos current, Pos target){
6    current.x = (current.x + target.x) / 2;
7    current.y = (current.y + target.y) / 2;
8    return current;
9  }
10 [...]
11 positions.map<inPlace>(moveTowards(origin));
12 result = positions.map(moveTowards(origin));
```

Listing 11.3: Musket Example for User Function Transformation

11.5.1 Map Fusion

Map fusion describes the process of combining multiple sequential mapping operations on the same data structure into a single combined call according to the relation $map\ g \circ map\ h \rightarrow map(g \circ h)$ [Ste+15]. This concept promises a performance increase because an intermediate synchronization between multiple skeleton calls is avoided and computations can operate on the already loaded data.

After checking that two map operations operate on the same data – and do not access other values within the data structure which potentially corrupts the outcome (e.g., pure functions) –, the skeleton is automatically rewritten to use a new combined function.

11.5.2 Skeleton Fusion

Continuing the idea of fusing user functions, this concept can be extended to the recombination of skeleton calls themselves. One example is the fusion of the map and fold operation to a *mapFold* (a.k.a. map reduce) skeleton. This combined call is more efficient because a temporary data structure after the mapping operation can be omitted and folding the result can be applied directly. Again, the transformation considers subsequent usages of user functions and data structures to prevent side effects on the remaining program.

11.5.3 User Function Transformation

In order to simplify the development of a parallel program, the structured approach encourages the user to write algorithms using a functional style of dedicated functionality that can be reused across the program. Although a user function can be syntactically and semantically valid, its performance highly depends on the context of where it is called.

In particular, generic map skeletons return a new data structure but the *inPlace* modifier alters the original data. In the latter case, features such as call-by-reference for parameter passing can

```
1 Pos moveTowardsMap(Pos current, const Pos& target){
2   current.x = (current.x + target.x) / 2;
3   current.y = (current.y + target.y) / 2;
4   return current;
5 }
6
7 void moveTowardsMapInPlace(Pos& current, const Pos& target){
8   current.x = (current.x + target.x) / 2;
9   current.y = (current.y + target.y) / 2;
10 }
```

Listing 11.4: Generated Functions After Transformation

be used (applying this transformation on generic map calls might lead to data corruption). To provide an example: For the skeleton in line 12 of **??**, the naïve implementation would use the function as is (using call-by-value parameter access). Through static analysis it can be observed that the `target` parameter is never changed, therefore it can be safely replaced by a constant reference (**??**; top). In contrast, the mapInPlace skeleton in line 11 returns the modified input parameter and to be stored in the original variable. Therefore, a transformation is applied during preprocessing which rewrites applicable functions to a call-by-reference scheme and avoid the costly copy operation (**??**; bottom). As a result, the same user function might result in different C++ functions.

11.5.4 Automated Data Distribution

Besides actual performance optimizations, the availability of an intermediate transformation also opens up the opportunity to mediate modelling complexities and simplify data handling. In the particular case of cluster environments, a user with little experience in parallel programming is often not well aware of the distribution strategies of data structures. Of course, data from remote nodes is fetched ad hoc within user functions but depending on the algorithm, changes of the distribution strategy can significantly reduce the communication overhead of retrieving values multiple times from remote nodes. Traditionally, an explicit data gathering of all current values of a distributed data structure and scattering of a coherent data structure into blocks for distribution to all computation nodes are required. Oversights only manifest as runtime errors or incorrect results. In contrast, automatic management of the data when assigning values to differently distributed variables can be applied during preprocessing and transformed automatically to include necessary data transfers.

In addition, memory management improvements are possible using knowledge of the main program control flow (which is not possible in libraries). For example, the map skeleton in line 12 of **??** can be transformed into a mapInPlace skeleton if there is no further read access to the `positions` array in order to benefit from those transformations (cf. Subsection 11.5.3).

To sum up, the preprocessing phase allows for optimizations regarding performance improvements and user support. The result of the separate transformations still complies to the Musket meta model, thus enabling a modular development. Consequently, current and future transformations can selectively identify and optimize specific patterns within the code – without modifying the actual input model. This highlights the benefit of DSLs in contrast to library-based approaches which are tightly coupled with the actual user code and bound to the features and syntax within the host programming language.

The subsequent code generation phase deals with the conversion into actual C++ source code that runs on high-performance clusters. In general, each generator can then further choose a suitable architecture for the specific target platform and include required boilerplate code while reusing the same platform-independent Musket model. The implementation is beyond the scope of this paper; more details and a comparison with a library-based approach for high-level parallel programming can be found in [WRK18].

11.6 Evaluation and Discussion

Previous research has shown that the performance of programs written in Musket can compete with equivalent library implementations such as Muesli [WRK18] and many articles exist on benchmarking those against hand-written implementations (e.g., [WMK18] for the FSS benchmark presented in Subsection 11.6.3). We want to highlight that our preprocessing approach can lead to improved execution times by, first, isolating certain optimizations in hand-written code and second, by using two Musket benchmarks, which include different optimizations. The first of these benchmarks is the calculation of the Frobenius norm and the second benchmark is a simplified version of the metaheuristic Fish School Search (FSS), which only performs the individual movement. We have executed the benchmarks on a multi-core cluster with four nodes and two Intel Xeon E5-2680 v3 CPUs without Hyper-Threading. Each CPU has twelve cores so that each node has a total of 24 cores. An open-source tool chain with g++ and OpenMPI has been used to compile the code. For the hand-written optimizations we used g++7.3.0 and OpenMPI 3.1.1 and for the Frobenius Norm and FSS benchmarks we used g++6.4.0 and OpenMPI 2.1.2. Platform-specific optimizations have been enabled with the -O3 and -march=haswell flags.

11.6.1 Optimizations

All benchmarks work on an array that contains 100,000 data objects, of which in turn contains another array with 25,000 single precision numbers. The first optimization considered is *map fusion*. The code consists of two consecutive parallel loops, which first set all values to two and then square each value. The handling of each element is parallelised, while the elements of the inner arrays are handled sequentially as the implementation of a user function in Musket would be expressed. Fusing both loops into one loop can reduce the execution time from 1.58s to 1.37s

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

matrix<double,65536,65536,dist> A;

main{
 // init
 A.map<index,inPlace>((int x,int y,double a)
      -> double {return (double) x+y+1.5;});

 // Frobenius calculation
 A.map<inPlace>((double a)
      -> double {return a * a;});
 double fn = A.fold(0.0, (double a,double b)
      -> double {return a + b;});
 fn = std::sqrt(fn);

 mkt::print("Frobenius norm is %.5f.", fn);
}
```

Listing 11.5: Musket Model for Frobenius Norm Benchmark

(speedup of 1.15) for a parallel execution with 24 cores. *Skeleton fusion* of a map and a fold skeleton also leads to an improvement in execution time. For squaring all values and calculating the sum, an implementation with two consecutive loops takes 0.38s whereas performing the calculation in one loop lasts 0.3s (speedup of 1.27).

The last benchmark focuses on the difference regarding call-by-value and call-by-reference. The benchmark is a combination of the ones mentioned above, i.e., all values are set to two, squared, and finally the sum is calculated. First, these operations are performed by functions with a call-by-value function call, which mimics the generation of a `mapInPlace` skeleton with a naïve generation of the user function. The execution takes 4.18s. With the transformation of the user function to use a call-by-reference strategy and avoid unnecessary copy operations, the execution time is reduced to 1.41s (speedup of 2.96). Consequently, map fusion and skeleton fusion optimizations as implemented in the Musket preprocessing can lead to significant speedups and the transformation of user functions allows for the convenience to define a user function once, but to use it in different skeletons with good performance.

11.6.2 Frobenius Norm

The Frobenius norm calculation is defined as:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} {a_{ij}}^2}$$

Consequently, two skeleton calls are required to calculate the result. First, all values of the matrix have to be squared using the map skeleton and second, the sum of all values has to be calculated using the fold skeleton. The corresponding model is shown in **??**, the actual norm calculation is performed in **??** and Line 14.

In fact, those two skeleton calls can be fused into a single call to the `mapFold` skeleton. In the CPU generator output, this leads to a single OpenMP parallel for-loop instead of first writing the squared value back in the data structure and afterwards calculating the sum. For a $65,536 \times 65,536$ matrix, the execution times on one and four nodes with the hardware and configuration described above are depicted in Figure 11.2. On a single node, fusing skeleton calls achieves speedups of 2.39, 2.91, and 3.24 on 1, 12, and 24 cores, respectively. The scenario with four computation nodes results in speedups of 2.33, 2.94, and 2.67, which can be explained by the communication overhead when increasing the level of parallelisation.

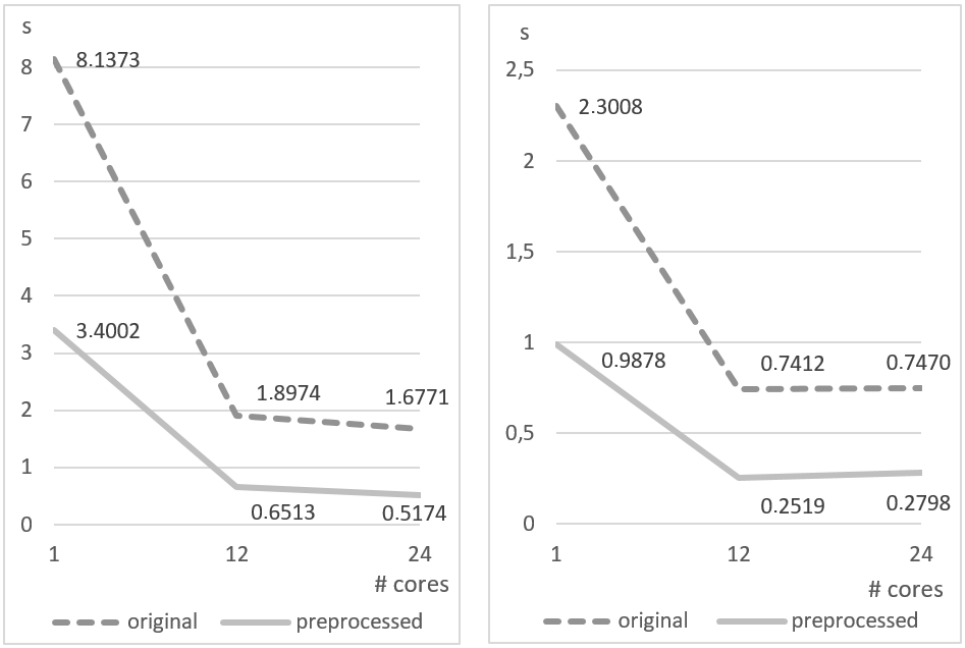


Figure 11.2: Frobenius Benchmark Execution Times for the Original/Preprocessed Model Using 1, 12, and 24 Cores on (a) 1 Node and (b) 4 Nodes

11.6.3 Fish School Search (FSS)

FSS is a metaheuristic that can be used to find good solutions for complex optimization problems, for which it is too time consuming to use exact methods. FSS is inspired by the behaviour of a fish school. Each fish represents one solution in the search space. FSS itself consists of multiple operators. For our purpose, it is sufficient to only use the so-called individual movement operator. The idea of this operator is that each fish moves to a random new location and only stays there if the found solution is better than the current one. So there is no interaction within the swarm,

yet, which would be the case for further operators, such as volitive and instinctive movement operators. An extract of the model can be found in **??**.

With the FSS benchmark, we wanted to analyse the behaviour of the optimization regarding the generation of skeletons with functions that make use of call-by-value and call-by-reference strategies for parameter passing. In general, it should be possible to combine any user function with a skeleton as long as the return type, the argument types and the number of arguments match the requirements of the skeleton. Moreover, from a usability perspective the signatures should not have to vary depending on the usage context. This means in turn that the user does not have distinguish between call-by-value and call-by-reference in the DSL.

However, this makes a difference regarding the performance in the generated C++ code. If the user function is generated as depicted in the benchmark model, call-by-value is used, which leads to copying of the fish objects when the function is called. For this model, this leads to an execution time of 68.6676 s on a single node with 24 cores. However, the DSL allows for further analysing the context of the skeleton call. In this case, the map variant `mapInPlace` is used, which directly changes the values in the data structure `population` . Using the transformations described in Subsection 11.5.3, it is possible to reference the fish object and directly perform the changes instead of copying it and returning the modified value. By applying those changes automatically during preprocessing, it is possible to reduce the execution time to 56.3726 s (speedup of 1.22). Similar results can be achieved when using different amounts of cores (speedups of 1.13 on 1 core and 1.19 on 12 cores) and adding computation nodes (speedups 1.13, 1.16, and 1.17 on 4 nodes).

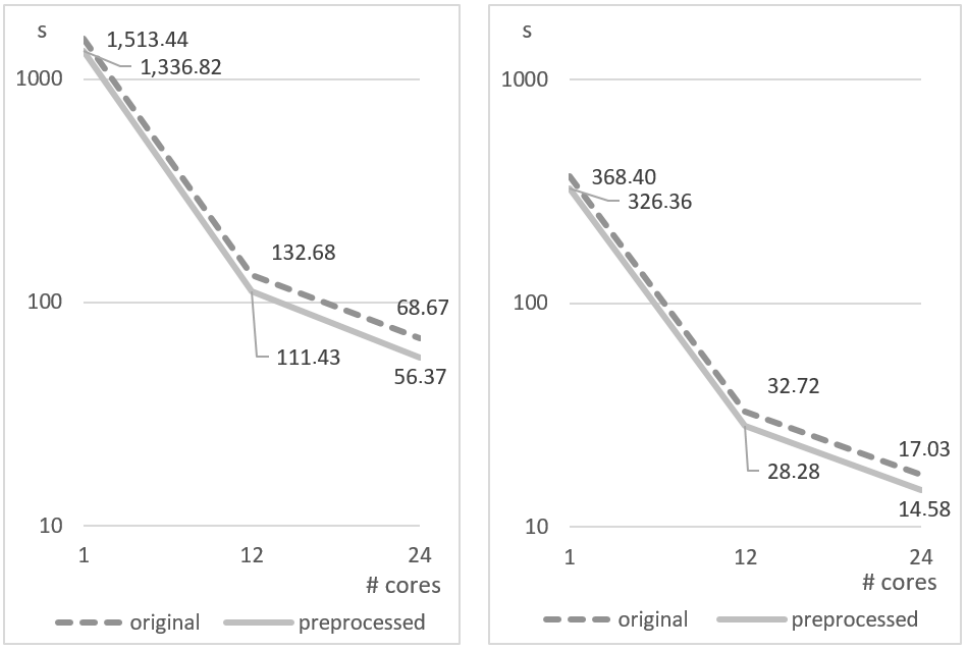


Figure 11.3: FSS Benchmark Execution Times for the Original/Preprocessed Model Using 1, 12, and 24 Cores on (a) 1 Node and (b) 4 Nodes

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

const int NUMBER_OF_FISH = 4096;
const int ITERATIONS = 5000;
const int DIMENSIONS = 1024;

struct Fish{
 array<double,DIMENSIONS,loc> position, candidate_position, best_position;
 double fitness, candidate_fitness, best_fitness;
};

array<Fish,NUMBER_OF_FISH,dist> population;

Fish movement(double step_size, Fish fi){
 for (int i = 0; i < DIMENSIONS; ++i) {
  double rand_factor= mkt::rand(-1.0, 1.0);
  double direction= rand_factor * step_size * (UPPER_BOUND-LOWER_BOUND);
  double new_value=fi.position[i]+direction;
  [...]
  fi.candidate_position[i] = new_value;
 }

 // fitness function
 double sum = 0.0;
 for (int j = 0; j < DIMENSIONS; ++j) {
  double value= fi.candidate_position[j];
  sum += (std::pow(value, 2) - 10 * std::cos(2 * PI * value));
 }
 fi.candidate_fitness= -(10*DIMENSIONS+sum);

 // update values
 if(fi.candidate_fitness>fi.fitness){[...]}
 return fi;
}

main{
 [...]
 for (int iteration = 0; iteration < ITERATIONS; ++iteration) {
  [...]
  population.map<inPlace>(movement(step_size));
 }
 [...]
}
```

Listing 11.6: Musket Model for Fish School Search Benchmark

11.6.4 Modelling Support

In general, it can be observed that DSL models are more concise. Even for Musket with a syntax intentionally close to C++, this can be observed. For example, the complete FSS model with all operators has 244 lines of code. Compared to the library-based approach Muesli (cf. Section 11.3) with 623 lines, a reduction by 61% can be achieved. There have been further comparisons between Musket and Muesli in [WR18], which show similar results. Although this figure is not significant in itself, less complicated models indicate a higher level of domain abstraction. Together with extensive modelling support, the user is guided towards syntactically and semantically correct models. Static code analysis detects erroneous interrelations between model components (e.g., the usage of inappropriate user functions for a given skeleton) before generating low-level code.

The correct application of complex domain concepts can be seen as a further advantage of DSLs – often overshadowed by other benefits such as performance gains. For example, the implicit interoperability of differently distributed data structures described in Subsection 11.5.4 provides a suitable domain abstraction by inferring the required communication pattern when assigning values. Supporting this transformation within the generator tool chain helps the user to focus on the actual problem at hand and reduces manual rework if models are modified later. Potentially, future versions of Musket could even support a more radical approach of completely eliminating the explicit declaration of the distribution strategy and let the system manage the availability of required data on the nodes based on the control flow of the program and a cost model of hardware performance.

Regarding limitations of this work, the presented functionality is already implemented but there is room for improvement. For instance, the DSL could be extended to support arbitrarily nested data structures or dynamically sized arrays. Additional transformations through detailed analysis are desired – yet, they can be seamlessly incorporated through the modular structure of transformations during preprocessing. Also, we currently do not consider transformations that represent mutually exclusive optimization strategies.

11.7 Conclusion and Outlook

Exploiting the potential of parallel code execution for complex calculations is still a challenge for many developers coming from application domains such as biology or physics. In this work, we have presented a textual DSL called Musket which is designed to support developers with programming knowledge by automatically transforming code in familiar C++ style into parallel programs. Based on the concept of algorithmic skeletons, code is generated for execution on high-performance clusters. In contrast to existing libraries, the first-order integration of distributed data structures and algorithmic skeletons in the language design allows for a minimum of additional complexity when designing the models and the generation of hardware-adapted low-level source code with little performance overhead. Using a preprocessing phase with model-to-

model transformations, performance optimizations can be applied to patterns of non-optimal code which cannot be detected by compilers. For example, map fusion and skeleton fusion techniques can merge computation steps across multiple user functions, and further transformations can optimize functions with regard to their usage context in specific skeleton variants. Our evaluation shows that significant performance gains can be achieved for synthetic benchmarks such as the Frobenius norm (speedup up to 3.24) as well as more complex and real-world applications such as the Fish School Search metaheuristic (speedup up to 1.12). In addition, modelling support is achieved by a seamless integration with state-of-the-art IDE features and extensive syntactic and semantic validation which enables clear and contextual feedback to the user during modelling.

Regarding future work, further performance gains are possible when using GPUs for the data-parallel algorithmic skeletons mentioned in this paper. Therefore, extending the generation capabilities towards parallel code execution on multi-GPU clusters from the same Musket input models represents current work in progress.

References

[Ald+10] M. Aldinucci et al. "Efficient Streaming Applications on Multi-core with FastFlow: The Biosequence Alignment Test-bed". In: *Advances in Parallel Computing* 19 (2010). Ed. by B. Chapman et al. DOI: 10.3233/978-1-60750-530-3-273.

[Bet13] L. Bettini. *Implementing Domain-specific Languages with Xtext and Xtend*. Community experience distilled. Birmingham, UK: Packt Pub, 2013.

[CCZ07] B. L. Chamberlain, D. Callahan, and H. P. Zima. "Parallel Programmability and the Chapel Language". In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[CJv08] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Scientific and Engineering Computation. Cambridge, MA, USA: MIT Press, 2008.

[CK10] P. Ciechanowicz and H. Kuchen. "Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures". In: *Proceedings of the HPPC '10*. IEEE, 2010, pp. 108–113. DOI: 10.1109/HPCC.2010.23.

[Col91] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[Cza98] Krzysztof Czarnecki. "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models". Dissertation. Ilmenau: Technical University of Ilmenau, 1998.

[DTK16] M. Danelutto, M. Torquati, and P. Kilpatrick. "A DSL Based Toolchain for Design Space Exploration in Structured Parallel Programming". In: *Procedia Computer Science* 80 (2016), pp. 1519–1530. DOI: https://doi.org/10.1016/j.procs.2016.05.477. URL: http://www.sciencedirect.com/science/article/pii/S1877050916309620.

[EK12] S. Ernsting and H. Kuchen. "Algorithmic Skeletons for Multi-core, Multi-GPU Systems and Clusters". In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138. DOI: 10.1504/IJHPCN.2012.046370.

[EK17] S. Ernsting and H. Kuchen. "Data Parallel Algorithmic Skeletons with Accelerator Support". In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299. DOI: 10.1007/s10766-016-0416-7.

[ELK17] A. Ernstsson, L. Li, and C. Kessler. "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems". In: *International Journal of Parallel Programming* (2017). DOI: 10.1007/s10766-017-0490-5.

[GL10] H. González-Vélez and M. Leyton. "A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers". In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160. DOI: 10.1002/spe.1026.

[GLS14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. 3rd. Scientific and Engineering Computation. Cambridge, USA: MIT Press, 2014.

[Gri+17] D. Griebler et al. "SPar: A DSL for high-level and productive stream parallelism". In: *PPL* 27.01 (2017), p. 1740005.

[Jan+16] V. Janjic et al. "RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications". In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 2016, pp. 288–295. DOI: 10.1109/PDP.2016.122.

[JP01] Jaakko Järvi and Gary Powell. "Side effects and partial function application in C++". In: *MPOOL Workshop at ECOOP*. 2001, pp. 43–60.

[Kuc02] H. Kuchen. "A Skeleton Library". In: *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*. Ed. by B. Monien and R. Feldmann. Vol. 2400. LNCS. Springer, 2002, pp. 620–629.

[Nic+08] J. Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53. DOI: 10.1145/1365490.1365500.

[RWK19] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. "Musket: A Domain-specific Language for High-level Parallel Programming with Algorithmic Skeletons". In: *34th ACM/SIGAPP Symposium on Applied Computing (SAC)*. Limassol, Cyprus: ACM, 2019, pp. 1534–1543. DOI: 10.1145/3297280.3297434.

[Ste+15] M. Steuwer et al. "Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code". In: *ICFP '15*. Vancouver, BC, Canada: ACM, 2015, pp. 205–217. DOI: 10.1145/2784731.2784754.

[The18] The Eclipse Foundation. *Xtext Documentation*. 2018. URL: https://eclipse.org/Xtext/documentation/ (visited on 03/29/2018).

[WMK18] F. Wrede, B. Menezes, and H. Kuchen. "Fish School Search with Algorithmic Skeletons". In: *International Journal of Parallel Programming* (2018). DOI: 10.1007/s10766-018-0564-z.

[WR18] F. Wrede and C. Rieger. *Musket Material Respository*. 2018. URL: https://github.com/wwu-pi/musket-material.

[WRK18] F. Wrede, C. Rieger, and H. Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *High-Level Parallel Programming and Applications (HLPP '18)*. Orléans, France, 2018.

12

A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes

Table 12.1: Fact sheet for publication P6

Title	A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes
Authors	Christoph Rieger[1] Herbert Kuchen[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2019
Conference	52nd Hawaii International Conference on System Sciences (HICSS)
Copyright	CC BY-NC-ND 4.0
Full Citation	Christoph Rieger and Herbert Kuchen. "A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes". In: *52nd Hawaii International Conference on System Sciences.* Maui, Hawaii, USA, 2019, pp. 7431–7440

A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes

Christoph Rieger Herbert Kuchen

Abstract: App development has gained importance since the advent of smartphones to enable the ubiquitous access to information. Until now, multi- or cross-platform approaches are usually limited to different platforms for smartphones and tablets. With the recent trend towards app-enabled mobile devices, a plethora of heterogeneous devices such as smartwatches and smart TVs continues to emerge. For app developers, the situation resembles the early days of smartphones but worsened by the widely differing hardware, platform capabilities, and usage patterns. In order to tackle the identified challenges of app development beyond the boundaries of individual device classes, a systematic process built on the model-driven paradigm is presented. In addition, we demonstrate its applicability using the MAML framework to create interoperable business apps for both smartphones and smartwatches from a common, platform-independent model.

12.1 Introduction

With the rising importance of smartphones as pervasive mobile devices, small and task-oriented pieces of software called apps have emerged as new artifacts in software development. A major challenge of mobile apps lies in the diversity of devices and platforms, although ten years after the introduction of the iPhone, a duopoly of Android and iOS dominates the mobile operating system (OS) market. Recently, the market for *app-enabled* devices has become much more diverse as new device classes such as smartwatches and smart TVs become available to a broader audience. In the near future, further device classes – including smart personal assistants, smart glasses, and vehicles – are expected to reach the mainstream market. Within each device class, a multitude of devices exists with different hardware capabilities and platforms for which third-party apps can be provided.

As app-enablement does not automatically entail compatibility and portability, developing apps for these devices poses various challenges. The current situation resembles the early phase of exploration after the introduction of smartphones. Different approaches set out to bridge device-specific differences but development for heterogeneous app-enabled devices is a far more complex endeavor than creating apps for several platforms of a common device class. In addition, the term *cross-platform* as well as actual frameworks are usually limited to smartphones and sometimes – yet not always – technically similar tablets, ignoring the differing requirements and capabilities within the variety of other app-enabled devices. Therefore, an efficient method for developing truly cross-platform apps across the boundaries of individual device classes is required.

Focusing on the domain of *business apps* – i.e., form-based, data-driven apps interacting with back-end systems [MEK15] –, we aim to simplify app development across multiple device classes by investigating three main research questions:

1. Which existing cross-platform approaches have the potential to be generalized in order to support a broad range of app-enabled devices?
2. How can a model-driven approach be structured to cater for peculiarities of multiple devices classes using platform-independent input models?
3. Is the domain-specific Münster App Modeling Language (MAML) applicable to different device classes using the proposed process model?

The remainder of this article follows these contributions. Section 12.2 highlights challenges arising from app development across device classes and analyzes current cross-platform approaches. Then, our proposal for a process model is presented in Section 12.3 and exemplified in Section 12.4 by extending the previously smartphone-centered MAML framework to create stand-alone apps for Wear OS smartwatches. Section 12.5 presents an evaluation of the presented model-driven approach and Section 12.6 discusses the applicability with regard to different scenarios. Finally, we revisit related work on app development combining different device classes in Section 12.7, before concluding in Section 12.8.

12.2 Cross-platform app development

To extend the boundaries of current cross-platform development approaches, we broaden its scope to *app-enabled* devices, i.e., a device extensible with software that comes in small, interchangeable pieces which are usually provided by third parties unrelated to the hardware vendor or platform manufacturer and increase the versatility of the device after its introduction [RM17]. Although these devices are typically mobile or wearable and therefore related to the term *mobile computing*, there are further device classes with the ability to run apps (e.g., smart TVs). Because of the adaptation to heterogeneous device classes, new challenges arise, to which most approaches are not suited.

12.2.1 Challenges

Challenges related to app development across device classes can be grouped in four main categories [RK18b].

Output heterogeneity Whereas most mobile apps are designed for screen sizes between 4" and 10", novel device classes vary greatly in terms of screen size (e.g., smartwatches $\leq$ 3" and smart TVs $\geq$ 20") or provide completely different means of output such as audio or projection. Even for screen-based devices, variability increases beyond "known" issues such as device orientation and

pixel density: novel devices bring along completely different aspect ratios (e.g., ribbon-like fitness devices worn around the wrist) and form factors (e.g., round smartwatches) [RM17]. Therefore, the information output needs to be described on a high level of abstraction beyond a particular screen layout.

User input heterogeneity Correspondingly, novel app-enabled devices have different characteristics for user inputs which span from pushing hardware buttons to clicking on a graphical User Interface (UI), tapping on touch screens, using auxiliary devices (e.g., stylus pens), and interacting via voice commands [RM17]. Moreover, multiple input alternatives may be available on one device and used depending on user preferences, usage context, or established interactions patterns of the respective platform. Again, this complexity calls for a higher level of abstraction that focuses on intended user actions rather than particular input events.

Device class capabilities Beyond user interfaces, hardware and software variability between different device classes are challenging. For example, the miniaturization in modern devices such as Wearables limits computational power and battery capacity. Complex computations may therefore be performed either on the device, offloaded to potential companion devices, or provided through edge/cloud computing [RZ15]. Also, sensing capabilities can vary widely. Suitable replacements for unavailable sensors need to be provided, e.g., using manual map selection instead of GPS sensors.

Multi-device interaction Whereas cross-platform approaches often focus on providing the same functionality across devices of different users, additional complexity arises from multi-device interactions *per user*. This might occur *sequentially* when a user switches to a different device depending on the usage context or user preferences, e.g., using an app with smart glasses while walking and switching to the in-vehicle app when boarding a car. Alternatively, a *concurrent* usage of multiple devices for the same task is possible, for instance in second screening scenarios in which one device provides additional information or input/output capabilities [NJE17]. In both cases, fast and reliable synchronization of content is essential.

12.2.2 Adequacy of Existing Approaches

A plethora of literature exists in the context of cross- or multi-platform development[1]. Classifications such as in [El-+15] and [MEK15] have identified five main approaches to multi-platform app development which are varyingly suited to the specific challenges of cross-platform development spanning different device classes.

With regard to runtime-based approaches, *mobile Web apps* – including recently proposed Progressive Web Apps – are mobile-optimized Web pages that are accessed using the device's browser and relatively easy to develop using Web technologies. However, most novel device types, e.g., the major smartwatch platforms WatchOS by Apple and Wear OS by Google (formerly

[1] We use these terms synonymously in this paper, although "multi-platform" appears more often in papers combining Web and mobile platforms.

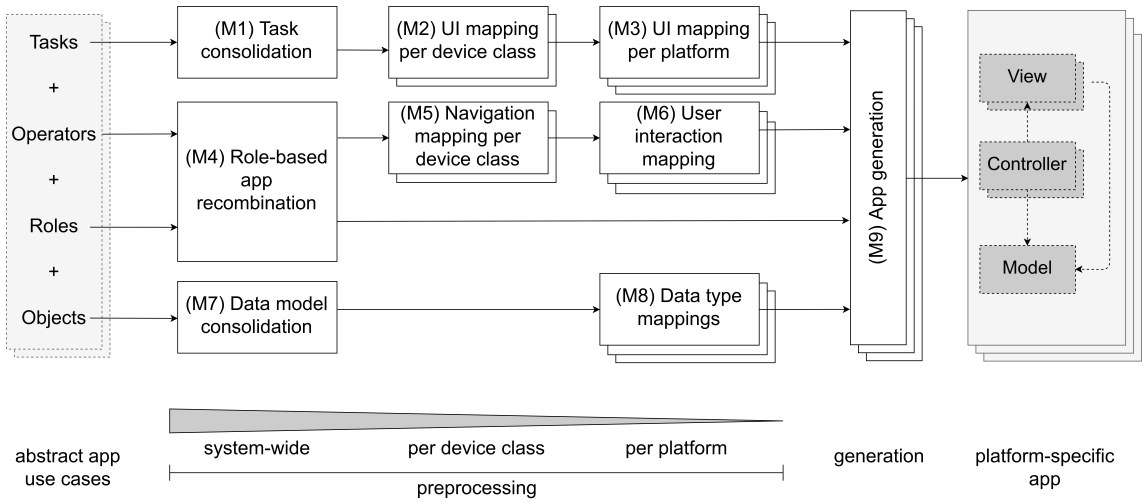


Figure 12.1: Process Model for Cross Device Class App Development

Android Wear), do not provide Web views or browser engines that allow for HTML rendering and the execution of JavaScript code. This approach therefore cannot currently be used for targeting a broader range of devices.

Hybrid apps are developed similarly using Web technologies but are encapsulated in a wrapper component that allows accessing device hardware and OS functionality through an Application Programming Interface (API) and building app packages for installation via app marketplaces. As they rely on the same technology, hybrid apps can neither be used beyond smartphones and tablets.

In contrast, apps using a *self-contained runtime* do not depend on the device's browser engine but use platform-specific libraries to use native UI components. Of the runtime environment approaches, this is the only one that can be used for developing truly cross-platform apps. Although usually based on custom scripting languages, a runtime can also be used as a replacement for inexistent platform functionality. As an example, CocoonJS[2] recreated a restricted Webview engine and therefore supports the development of JavaScript-based apps also for Android and iOS. However, synergies with regard to user input/output and available hardware/software functionality across heterogeneous devices are limited by the runtime's API.

Considering generative approaches to cross-platform development, *model-driven software development* has several advantages as it uses textual or graphical models as main artifacts to develop apps and then generates native source code from this platform-neutral specification. Modeling custom domain-specific concepts allows for a high level of abstraction, for example circumventing issues such as input and output heterogeneity using declarative notations. Arbitrary platforms can be supported by developing respective generators which implement a suitable mapping from descriptive models to native platform-specific implementations.

[2]https://docs.cocoon.io/article/canvas-engine/

Finally, *transpiling* approaches use existing application code and transform it to different programming languages. Bridging device classes using this approach is technically feasible as the result is also native code. However, there is more to app development than just the technical equivalence of code, which also explains the low adoption of this approach by current cross-platform frameworks. For instance, user interfaces behave drastically differently across different device classes, and substantial transformations would be required. It is therefore unlikely that this approach is viable for bridging device classes beyond reusing individual components such as business logic.

To sum up, only self-contained runtimes and model-driven approaches are candidates for device class spanning app development, of which the latter additionally benefits from the transformation of domain abstractions to platform-specific implementations.

12.3 A Process Model for App Development Across Device Classes

Although not all types of apps make sense on all types of devices, our focus lies on *business apps* whose workflow-like concepts of subsequent data manipulation activities can be transferred to the particular user experiences on heterogeneous devices. However, the complexity of developing apps across device classes requires a structured approach in order to manage the variability of hardware and software capabilities. Based on the applicability of current cross-platform development techniques, we propose an extended model-driven approach to develop business apps as depicted in Figure 12.1.

In order to structure the resulting application and foster comprehensibility for all development stakeholders, the domain-specific notation should be designed with modularity in mind and allow to specify different *use cases*, i.e., units of functionality comprising a self-contained set of behaviors and interactions performed by the app user [Obj15]. In a workflow context, a use case can also be interpreted from a user task model perspective [SCK07]. The semantically compliant ConcurTaskTree notation for example consists of four elements [Pat00]:

- the abstract and descriptive *task* descriptions which together form the use case's functionality,
- *operator*s defining the allowed sequences of executed tasks, representing an abstract notion of navigation actions within a use case
- the user *role*s that are allowed to interact with the system (and might differ per task), and
- the data *objects* to interact with in each task.

From such a descriptive representation, mappings need to be defined in order to reach platform-adapted app output. To handle the transformation complexity, we propose a step-wise refinement of input models. First, system-wide transformations need to be applied to consolidate multiple

modular and independently modeled use cases (M1, M4, M7). Second, the model is preprocessed according to each targeted device class (M2, M5). The extent of these transformations highly depends on the degree of maturity of the respective device class. For example, common layout patterns such as tabs or vertical content scrolling may be observed frequently and smartphone hardware converges with regard to sensors and screen dimensions. Third, platform-specific preprocessing is required, similar to today's model-driven cross-platform approaches (M3, M6, M8). Only in this step, a final mapping of abstract UI elements to concrete widgets based on a choice of possible representations for the given task can be specified. Finally, the code generation is executed which outputs the platform-specific app artifacts (M9). The Model-View-Controller (MVC) pattern or derived patterns such as Model-View-ViewModel (MVVM) are suited as high-level structure of the resulting applications [SW14]. In addition, reference architectures applicable to platforms of the same device class can be used to maintain consistency in this final step, thus easing the development of new generators and the addition of features across existing platform generators [Ern+16].

Using the proposed model, the challenges of app development spanning device classes can be tackled:

Resolving output heterogeneity Based on the descriptive use case models with a high level of abstraction, the main activity related to task consolidation (M1) is the specification of views from the logical task units of functionality. If required, generic model preprocessing and simplification activities are also performed, e.g., the resolution of shorthand definitions and references within the models. Subsequently, mapping the UI per device class (M2) is achieved by applying two types of transformations that do not modify the content but specify a target layout for the views adapted to the device class: On the one hand, the presentation of information can be *layouted* according to the available screen sizes, for example by choosing an appropriate appearance – one could think of tabular vs. graphical representations – or leave out complementary details, if necessary. On the other hand, the content can be *re-formatted* by an adaptive UI according to usual interaction patterns, e.g., to present large amounts of information through scrolling, subdivided into multiple subsequent steps, or as a hierarchical structure [EVP01].

Finally, a platform-specific UI mapping (M3) defines concrete UI elements and the actual user interactions within a view. Activities within this mapping include the selection of suitable widgets, their arrangement within the previously specified layout container as well as bidirectional data bindings and validation.

To exemplify the difference to the previous device class preprocessing, an "item selection" task within a use case might for instance be mapped to an abstract list selection for all tablet platforms. The concrete representation such as a horizontal card-based layout for Windows or a vertical list view for iOS is then specified in the platform-specific mapping stage.

Resolving input heterogeneity User inputs are not only important for entering data but also to navigate within the app. The large variety of device class specific (e.g., remote controls for TVs), platform-specific (e.g., Android's hardware buttons), and even device-specific input events

(e.g., Apple's 3D touch gestures) is a hurdle for efficient modeling. Instead, user inputs should be described as *intended actions* for completing a particular task. Based on the possible sequences of tasks, navigation paths can be established (M5), including the initial task selection when opening the app, back-and-forth navigation between views, and conditional process flows resulting from user decisions.

In the platform-specific mapping (M6), it is then possible to perform a mapping of actions to actual input mechanisms. This is similar to the decoupling mechanism for user inputs in MVC architectures described in [CFS16] but on a higher level of abstraction. For instance, a "back" action can be linked to a hardware button, displayed in a navigation bar on screen, bound to a right-swipe gesture in Wear OS, or recognized by a spoken keyword. Some novel device classes such as smart cars are in early experimental stage; mappings such as specific gestures then represent preliminary design decisions suitable for the range of possible apps and in accordance with vendor guidelines. As noted above, repetitive platform-specific transformations can of course be shifted to a more generic layer in the future when commonalities become apparent.

Managing device capabilities Whereas tasks, operators, and roles have no interdependencies between use cases, a global data layer needs to be established from different data models (M7). Data model inference can validate the compliance of different use cases and also provide additional modeling support for the editing tools of the domain-specific language (DSL) [RK18a]. Subsequently, non-primitive data types (e.g., dates and colors) need to be mapped to available platform concepts for output to the user and back-end communication (M8).

In the context of business apps, there are usually no complex computation tasks to be performed on the device itself. Derived attribute values may be computed at runtime but if necessary, computations may be offloaded to remote computation providers. With regard to sensors, a progression in functionality should be aimed for: the same app would function on many devices while providing the highest level of functionality achievable on the given hardware. For instance, location information is easily retrieved if a GPS sensor is available but generic approaches need to cater for adequate fall-back mechanisms such as manual selection of addresses on a map or address lookup.

Enabling multi-device interaction In multi-user scenarios, a recombination of tasks needs to be performed to account for distinct user roles (M4). This mapping modifies the sequence of tasks to cater for the interruption of activities and the automatic transmission of application state to the subsequent role. As a result of this decomposition and bundling of use cases, either one app is created that supports all user roles (depending on the logged-in user) or different apps are generated per role.

In addition, synchronization is essential both for the sequential and concurrent usage of apps on multiple heterogeneous devices. In contrast to "traditional" process-oriented apps, information not only needs to be propagated between devices after a given task has been completed but also intermediate states of data or even a live synchronization capability including incomplete user inputs (e.g., per character) is required. Also, the workflow state itself must be captured in order

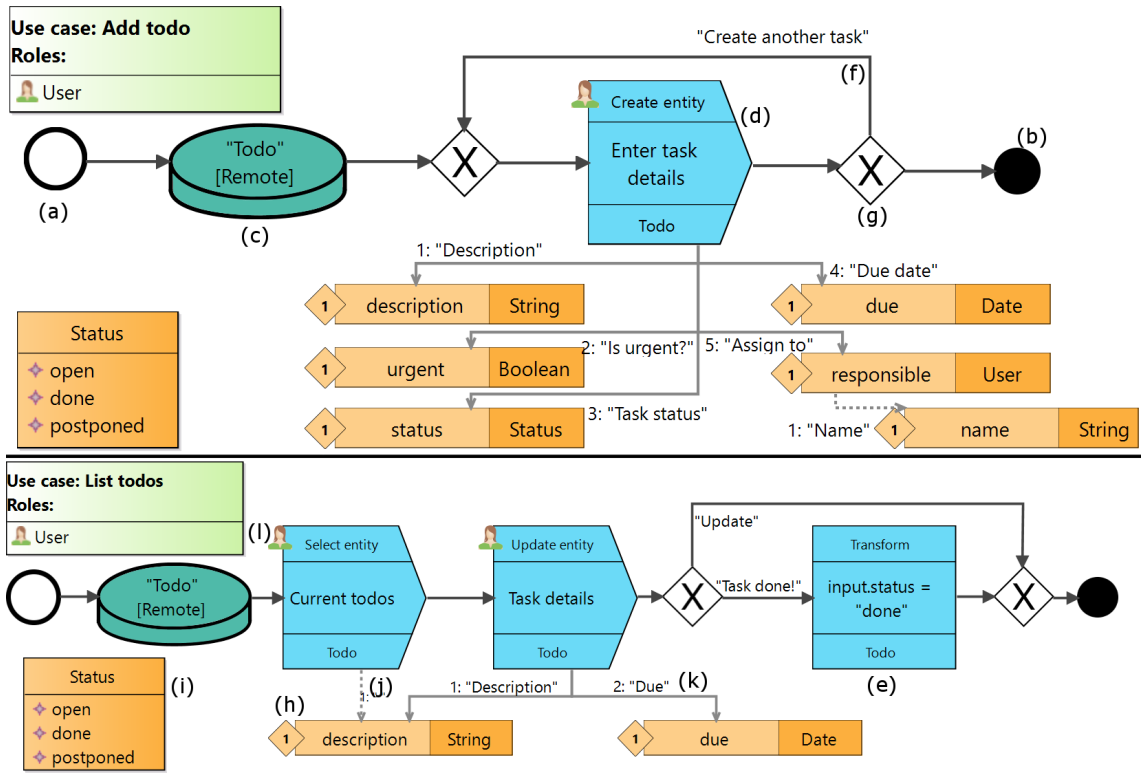


Figure 12.2: Sample MAML Models for a To-Do List Use Case

to pass the current process instance to other concurrently used devices as well as to different users when role changes occur. Consequently, the back-end component also needs to manage the relationship of users and devices (either via central device registration or tracking on which device the user is currently logged in) and provide push-based or pull-based update mechanisms.

12.4 Realizing Cross Device Class Apps

In this section, the open-source MAML framework[3] is briefly introduced to demonstrate the applicability of the proposed process. This domain-specific notation was chosen because of its fit for business apps and its high-level abstraction [RK18a; Rie18].

12.4.1 The MAML Framework

MAML is a graphical DSL for specifying business apps and based on five main design goals [Rie18]:

- *Automatic cross-platform app creation* by transforming a graphical model to fully functional source code for multiple platforms.
- *Domain expert focus* to allow non-technical stakeholders to create, alter, or communicate about an app using the actual models.

[3]MAML code repository: https://github.com/wwu-pi/maml

- *Data-driven process modeling* specifies the application domain on a high level of abstraction by interpreting data manipulation as a process.
- *Modularization* of activities in distinct use cases helps for maintenance, e.g., for domain experts.
- *Declarative description* of the complete app, including necessary specifications of data model, business logic, user interactions, and UI views.

We illustrate the notation and applicability of MAML for cross device class app development using the common scenario of a to-do management app. The system can be represented with two *use cases* depicted in the screenshots of the implemented drag&drop editor component depicted in Figure 12.2. In the first use case, one or more tasks are created with corresponding attributes such as due date and responsibility assignment. The second use case lists all current to-dos and upon selection of an item shows editable details and the possibility to complete the task. The sequence of process steps is modeled between a *start event* (labeled with (a) in Figure 12.2) and *end events* (b). A *data source* (c) specifies the main type of the manipulated entity which is either stored locally or on a remote back-end system. Subsequently, *interaction process elements* (d) are used to *create/show/update/delete* an entity, but also to *display messages* or access device sensors. Note that this strictly declarative description does not make assumptions on the appearance on a device. To perform steps without user interaction, *automated process elements* (e) can be used to *transform* data values or navigate between objects, request information from *web services*, or *include* existing use cases for model reuse. *Process connectors* signify the order of process steps, represented by a default "Continue" action unless specified differently (f). *Condition*s branch out the process flow based on a manual user action (using differently labeled connectors; (g)), or automatically by evaluating expressions referring to the considered object (not visualized in the example).

In addition, the use case models contain the data linked to each process step. *Attributes* (h) are modeled as combination of a name, the data type, and the respective cardinality. Several data types such as *String, Integer, Float, PhoneNumber, Location* etc. are already provided but the modeler can define additional custom types which are further described using nested attributes (e.g., the "User" type in Figure 12.2 specifies a "name" attribute). Moreover, the modeler can create *enumeration* types (i) with lists of possible values. A suitable UI representation is automatically chosen based on the *parameter connector*: Dotted arrows (j) signify a reading relationship whereas solid arrows (k) represent a modifying relationship. This refers not only to the manifest representation of attributes displayed either as read-only text or editable input field. The interpretation also applies in an abstract sense, e.g., regarding web services which *read* input parameters and *modify* information through their response. Each connector also specifies an order of appearance and a human-readable caption.

Finally, annotating freely definable *roles* (l) to process elements allows modeling processes

with several persons involved, e.g., in scenarios such as approval workflows. When a role change occurs, the app automatically informs eligible users about the open process instance.

12.4.2 Business Apps for Smartphone and Smartwatch with MAML

Figure 12.3: Screenshots of the Resulting Wear OS Smartwatch App

As proof of concept for this work, the MAML framework (which supports app generation for Android and iOS smartphones from the same input models [Rie18]), was extended by a smartwatch generator for Wear OS and prepared for further app-enabled device classes. Although the utilized DSL is designed to be platform-independent, existing generators focused on transformations towards smartphone apps. Therefore, the process model described in the previous section was applied to structure and separate the growing generation capabilities – using the BXtend framework and Xtend language for implementing the model transformations [Buc18]. Because of space constraints, the respective transformations are only sketched next.

Regarding *system-wide preprocessing*, separate MAML input models are recombined into a single app project. The task consolidation transformation (M1) merges multiple use cases into a single app project and prepares required app-internal components, e.g., web service calls. A role-based

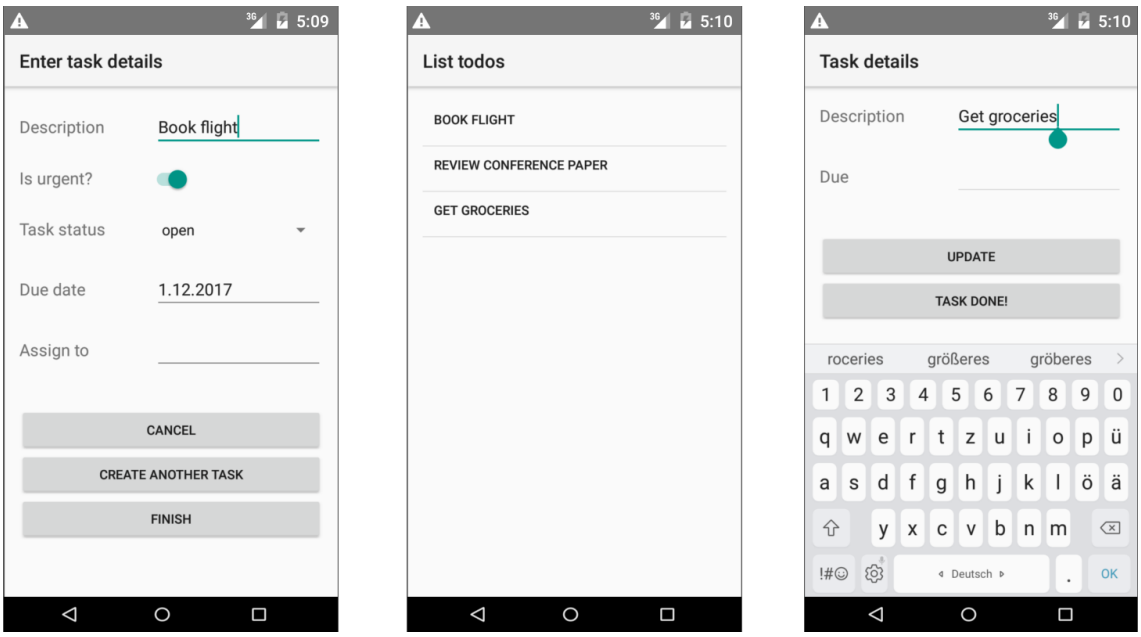


Figure 12.4: Screenshots of the Resulting Android Smartphone App

app recombination mapping (M4) separates each use case into multiple process sequences which are performed by the same role in order to generate distinct apps. Because no explicit data modeling is required in MAML, a common data model (M7) needs to be inferred both within use cases and for the overall app project (cf. [RK18a]). The output and all further preprocessing is performed based on a second, textual, DSL called MD^2 [MEK15] with more detailed specifications, especially with regard to the view layer.

Subsequently, *device class specific* preprocessing is performed by repartitioning views to fit the amount of fields to the available screen sizes (M2). For example, to-do elements can be created within a single view on a smartphone, but the smartwatch implementation splits this in two subsequent steps to avoid scrolling. Also, suitable layouts (e.g., tab or scroll-based) are selected based on the modeled task type – still using abstract UI elements. Regarding the navigation mapping (M5), an additional start-up view is included for all smartphone apps to let the user choose a startable use case when entering the app.

Finally, concrete widgets are introduced in the *platform-specific preprocessing* of UI elements (M3), for instance exploiting the round design of a smartwatch by a curved list widget. Default actions are transformed to prominently displayed buttons in the navigation bar in iOS, whereas Wear OS uses so-called ActionDrawers[4], e.g., to trigger the "Add todo" use case ("+" icon in Figure 12.3) while reading through the list of to-dos. In the user interaction mapping (M6), the navigation through the process is mapped to buttons or specific patterns such as the NavigationDrawer[o] in Wear OS.

[4]https://designguidelines.withgoogle.com/wearos/components/

During the generation phase (M9) of MAML-based apps, an MVC separation of concerns is established with data bindings to view elements [Ern+16] which results in structured apps that may be customized with individual code. For the sample app model represented in Figure 12.2, resulting app screenshots of the generated smartwatch and smartphone apps are depicted in Figure 12.3 and **??**.

12.5 Evaluation

To evaluate MAML as common notation to develop both smartphone and smartwatch apps, a study was performed with 23 Master students attending a course on model-driven software development. Their previous experience with app development was limited, with an average response value of 3.26 regarding Web apps (on a 5-level Likert scale from expert to little experience) as well as hybrid and native apps with 4.35 and 4.3, respectively. After presenting the to-do app scenario depicted in Figure 12.2 together with a short introduction to MAML, the participants were asked to sketch a suitable smartwatch application. 83% of them imagined a scrollable list to display to-dos (65% vertically, 17% horizontally), and 30% added an action button to create new to-dos from there. Of those participants who visualized view transitions, accessing task details was mostly considered by tapping on the list element on screen (42%), other participants opted for dedicated buttons (33%), hardware buttons (8%), swipe gestures (8%), and speech interfaces (84%).

Regarding the representation of task details/creation forms, participants modeled different view designs such as a single, scrollable view (35%), individual steps for each attribute (30%), a separation into multiple views according to available space (9%), or voice input (30%). Again, navigating back and forth in the dialog was envisaged either using on-screen/physical buttons (35% / 8%) or swipe gestures (22%). This diversity of interfaces reflects the variety of interaction modes exhibited by today's smartwatches.

Subsequently, the generated apps (Figure 12.4 and **??**) were presented to the participants to compare the implementation with their expectations. The participants agreed that the generated interface suitably represents the modeled process (2.04) and that the app is functional (2.39). The moderated visual appeal (3.30) is a drawback resulting from the generic nature of business apps our generator has to cater for. However, the participants agreed that using model-driven transformations on a single input model drastically accelerates the development speed (2.48), estimating an average drop from 27 hours of manually programming a mobile application to 41 minutes for creating the equivalent MAML models. Overall, the participants of this preliminary study did not perceive particular complexities resulting from the implicit specification of multiple apps with MAML (3.35) and rated the approach suitable for creating apps across device classes (2.55).

Furthermore, a 10-question System Usability Scale (SUS) questionnaire was filled out to calculate a usability score (on a [0;100] interval) [Bro96]. Compared to an earlier evaluation of MAML for smartphone app generation reaching a score of 66.83 [Rie18], similar results could now be

obtained with regard to smartwatch with a 66.85 score ($\sigma = 12.95$). The perceived usability of MAML therefore barely depends on the tested devices which underlines the objective of a platform-agnostic representation. Using the proposed process model, the vision of model-driven app development for a broad range of devices can be put into practice while avoiding the time-intensive repetitive adaptation of high-level models during generator implementation.

12.6 Discussion

The process model presented in Figure 12.1 does not aim for an unchangeable, all-encompassing set of transformation rules but describes the required transformation steps broken down into individual activities in a structured manner. Likewise, the approach supports a wide range of technical implementations regarding input and output of transformations which can be realized using appropriate technologies. In this regard, our approach differs from the Model-Driven Architecture (MDA) by not enforcing a strict assignment of *platform-independent* and *platform-specific* models to the respective steps. A platform-specific intermediate model might be omitted completely if the final preprocessing is executed within the generator component. Moreover, depending on the design of the chosen DSL, all transformations might be performed according to a single meta model or using multiple representations with different granularity. In the example of MAML as abstract representation, the subsequent transformations are implemented using another DSL, e.g., providing more detailed view specifications. The commonly cited Cameleon Reference Framework (CRF) [Cal+02] similarly describes a step-wise refinement from task-oriented specification to abstract, concrete, and final UI elements. However, it focuses solely on the interface aspect (i.e., the top row in Figure 12.1) as compared to the additional perspectives on navigation, business logic, and data layer.

The main stakeholders of the approach include not only software engineers and developers in the field of mobile apps (working mainly on the transformations) but also process modelers and non-technical users (capturing requirements and providing the abstract representation). Additional stakeholders are integrated in specific transformations as needed, for example requesting design expertise for the platform-adapted UI mapping, and human-computer interaction experts for navigating through the resulting apps. The responsibility of maintaining the transformation chain can be assigned to teams focused either on individual refinement steps of the preprocessing (vertical subdivision) or according to a functional domain (horizontal subdivision).

It is clear that a particular app does not need to make sense on all possible target platforms but the additional effort of generating apps for all devices at once using the same input models is marginal. In return, the potential of upcoming app-enabled devices is tapped and may provide opportunities for enhancing productivity as illustrated with three exemplary use cases in the following.

1) Dashboard apps: Many teams track key performance indicators for monitoring progress, motivating team members, and broadcasting daily achievements. Besides entering this information

via personal devices, dashboards can be displayed on large screens in the office. Heterogeneous team member equipment is a known challenge in cross-platform development, but now intensified because of new devices for visualization (smart TV) or notifications (Wearables).

2) Logistics apps: Order picking in warehouses is an activity that is mainly coordinated using information systems. Operating hands-free using Augmented Reality (AR) devices instead of traditional scanners significantly improves the productivity of employees and first commercial products for the logistics industry already exist[5]. Similarly, technicians can benefit from up-to-date information both before and during repair works.

3) Field service apps: Salespersons need flexible access to their company's information systems from different contexts, e.g., while traveling or in sales talks with customers. The freedom to choose from multiple owned devices (e.g., smartphone, tablet, or smartwatch) for communication, notifications, or information access helps them to accomplish tasks more efficiently.

A limitation of the integrated approach from abstract specification to source code lies in the tight coupling of the app development process for various platforms, unless work can be coordinated among different teams as mentioned above. In addition, the allocation of specific mapping activities to the presented phases of the process model is final but a continuous process. Depending on the heterogeneity or convergence within a device class over time, common platform characteristics may be shifted to earlier mappings in the process chain. As the proposed process model is not limited to a specific technology, further instantiations are desirable to validate the approach. Regarding limitations of the the newly created Wear OS generator, the implementation is still in a prototypical state and we are working on improving the UI/UX design of its output. However, it is integrated in the MAML framework and already contributes functional apps for the new platform target.

12.7 Related Work

Whereas a plethora of literature exists in the context of cross-/multi-platform development, previous work on app development *spanning multiple device classes* is much more scarce. Although an extensive literature search was performed, not a single paper out of almost 300 initial results provided a comprehensive theory on app development across device classes. This shows that app development beyond smartphones is not yet approached systematically but on a case-by-case basis.

Cross-platform overview papers such as [JM15] typically focus on a single category of devices and apply a very narrow notion of mobile devices, e.g., when Humayoun et al. [HEE13] examine "the diversity in smart-devices (i.e., smartphones and tablets)". [RM17] provides the only classification beyond those "classical" mobile devices. To complicate matters, some papers mention that there are novel devices as visionary outlook but do not detail on actual app development challenges

[5] http://www.smartpick.be/

[Umu15; Dag+16]. In contrast to "liquid software" aiming for portability across heterogeneous platforms [Gal+16], model-driven approaches can integrate with arbitrary infrastructures.

Only few papers provide a technical perspective on novel app-enabled devices: Singh and Buford [SB16] describe a cross-device team communication use case for desktop, smartphones, and Wearables, and Esakia et al. [ENM15] performed research on Pebble smartwatch and smartphone interaction in computer science courses. Some authors consider specific combinations of devices, for example Neate et al. [NJE17] who analyze second screening apps that combine smart TVs with additional content on smartphones or tablets, and Koren and Klamma [KK16] who propose a middleware approach to integrate data and UI of heterogeneous Web of Things devices. Zhang et al. [Zha+17] present an architecture for the future of "transparent computing" using virtual apps on lightweight devices equipped with UEFI firmware interface but do not focus on the apps themselves. Finally, a few papers on mobile (cloud) computing might be applicable to app development across device classes but do not explicitly mention this potential.

With regard to commercial cross-platform products, Xamarin[6] and CocoonJS[o] are two runtime-based approaches that provide Wear OS support to some extent. Whereas several other frameworks claim to support Wearables, this usually only refers to data access or the display of notifications by the main smartphone application on coupled devices. Only the domain of gaming apps exhibits more diversity. Unity3D[7], an engine for 2D/3D games, supports 29 platforms including smartphones, smart TVs, consoles, and AR devices.

While related to research fields such as self-adaptive UI or model-driven UI development [ABY16], our work concentrates on the automated generation of all app perspectives for a specific platform without need for dynamic UI adaptation at runtime. Context-awareness and adaptation (e.g., to screen sizes) are therefore complementary considerations when designing mappings for a device class or specific platform.

12.8 Conclusion and Outlook

The field of modern mobile computing does not show signs of less rapid progress and novel app-enabled devices are constantly emerging. New challenges resulting from the heterogeneity of input and output mechanisms, device class capabilities, and multi-device interaction require consideration by app developers. In this article, we have analyzed existing cross-platform development approaches for efficiently creating apps for heterogeneous devices. We propose a systematic process model based on the model-driven paradigm which supports the inclusion of novel app-enabled devices. The applicability of this process is demonstrated using the MAML framework that utilizes a graphical DSL for generating business apps for smartphones and now also smartwatches. Because of its high level of abstraction, the challenges of app development

[6] https://developer.xamarin.com
[7] https://unity3d.com

across device classes can be tackled through a multi-step preprocessing of input models towards platform-specific code generation.

Although the smartwatch generator for MAML demonstrates the required concepts, real-world adoption is pending. Future work concentrates on generating apps for other platforms to validate the generalizability of the proposed model, especially for devices without touchscreen such as smart personal assistants.

References

[ABY16] P. A. Akiki, A. K. Bandara, and Y. Yu. "Engineering Adaptive Model-Driven User Interfaces". In: *IEEE Transactions on Software Engineering* 42.12 (Dec. 2016), pp. 1118–1147. DOI: 10.1109/TSE.2016.2553035.

[Bro96] John Brooke. "SUS-A quick and dirty usability scale". In: *Usability evaluation in industry*. 1996, pp. 189–194.

[Buc18] Thomas Buchmann. "BXtend - A Framework for (Bidirectional) Incremental Model Transformations". In: *MODELSWARD*. 2018. DOI: 10.5220/0006563503360345.

[Cal+02] Gaëlle Calvary et al. "The CAMELEON reference framework". In: *D1.1* (2002).

[CFS16] Alessandro Carcangiu, Gianni Fenu, and Lucio Davide Spano. "A design pattern for multimodal and multidevice user interfaces". In: *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM. 2016, pp. 177–182.

[Dag+16] Jan C. Dageförde et al. "Generating App Product Lines in a Model-Driven Cross-Platform Development Approach". In: *HICSS*. 2016, pp. 5803–5812. DOI: 10.1109/HICSS.2016.718.

[El-+15] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* (2015).

[ENM15] Andrey Esakia, Shuo Niu, and D. Scott McCrickard. "Augmenting Undergraduate Computer Science Education With Programmable Smartwatches". In: *SIGCSE*. 2015, pp. 66–71. DOI: 10.1145/2676723.2677285.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *WEBIST* (2016), pp. 307–316. DOI: 10.5220/0005862103070316.

[EVP01] J. Eisenstein, J. Vanderdonckt, and A. Puerta. "Applying model-based techniques to the development of UIs for mobile computers". In: *IUI* (2001).

[Gal+16] A. Gallidabino et al. "On the Architecture of Liquid Software: Technology Alternatives and Design Space". In: *WICSA*. Apr. 2016, pp. 122–127. DOI: 10.1109/WICSA.2016.14.

[HEE13] Shah Rukh Humayoun, Stefan Ehrhart, and Achim Ebert. "Developing Mobile Apps Using Cross-Platform Frameworks: A Case Study". In: *HCI International*. 2013, pp. 371–380. DOI: 10.1007/978-3-642-39232-0_41.

[JM15] Chakajkla Jesdabodi and Walid Maalej. "Understanding Usage States on Mobile Devices". In: *Int. Joint Conf. on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 1221–1225. DOI: 10.1145/2750858.2805837.

[KK16] István Koren and Ralf Klamma. "The Direwolf Inside You: End User Development for Heterogeneous Web of Things Appliances". In: *ICWE* (2016), pp. 484–491. DOI: 10.1007/978-3-319-38791-8_35.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14. DOI: 10.19210/OJIS_2015v2i2n02_Majchrzak.

[NJE17] Timothy Neate, Matt Jones, and Michael Evans. "Cross-device Media: A Review of Second Screening and Multi-device Television". In: *Personal and Ubiquitous Computing* 21.2 (2017), pp. 391–405. DOI: 10.1007/s00779-017-1016-2.

[Obj15] Object Management Group. *Unified Modeling Language 2.5*. 2015.

[Pat00] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2000. DOI: 10.1007/978-1-4471-0445-2.

[Rie18] Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *HICSS*. 2018, pp. 5725–5734.

[RK18a] Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *COMLAN* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001.

[RK18b] Christoph Rieger and Herbert Kuchen. "Towards Model-Driven Business Apps for Wearables". In: *Mobile Web and Intelligent Information Systems*. Ed. by Muhammad Younas et al. Springer, 2018, pp. 3–17.

[RK19] Christoph Rieger and Herbert Kuchen. "A Model-Driven Cross-Platform App Development Process for Heterogeneous Device Classes". In: *52nd Hawaii International Conference on System Sciences*. Maui, Hawaii, USA, 2019, pp. 7431–7440.

[RM17] Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-Enabled Devices". In: *WEBIST*. 2017, pp. 332–339.

[RZ15] Andreas Reiter and Thomas Zefferer. "POWER: A cloud-based mobile augmentation approach for web- and cross-platform applications". In: *CloudNet*. IEEE, 2015, pp. 226–231. DOI: 10.1109/CloudNet.2015.7335313.

[SB16] K. Singh and J. Buford. "Developing WebRTC-based team apps with a cross-platform mobile framework". In: *IEEE CCNC* (2016). DOI: 10.1109/CCNC.2016.7444762.

[SCK07] Daniel Sinnig, Patrice Chalin, and Ferhat Khendek. "Common Semantics for Use Cases and Task Models". In: *Integrated Formal Methods*. Ed. by Jim Davies and Jeremy Gibbons. Vol. 4591. LNCS. Springer, 2007, pp. 579–598. DOI: 10.1007/978-3-540-73210-5_30.

[SW14] A. Syromiatnikov and D. Weyns. "A Journey through the Land of Model-View-Design Patterns". In: *WICSA*. 2014, pp. 21–30. DOI: 10.1109/WICSA.2014.13.

[Umu15] Eric Umuhoza. "Domain-specific modeling and code generation for cross-platform multi-device mobile apps". In: *CEUR Proceedings* 1499 (2015).

[Zha+17] Yaoxue Zhang et al. "Transparent Computing: A Promising Network Computing Paradigm". In: *CISE* 19.1 (2017), pp. 7–20. DOI: 10.1109/MCSE.2017.17.

13

Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons

Table 13.1: Fact sheet for publication P7

Title	Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons
Authors	Fabian Wrede[1] Christoph Rieger[1] Herbert Kuchen[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2018
Conference	International Symposium on High-Level Parallel Programming and Applications (HLPP)
Copyright	CC BY 4.0
Full Citation	Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *High-Level Parallel Programming and Applications (HLPP)*. Orleans, France, 2018

Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons

Fabian Wrede Christoph Rieger Herbert Kuchen

Keywords: algorithmic skeletons, parallel programming, high-performance computing, model-driven development, domain-specific language

Abstract: Parallel programming can be difficult and error-prone, in particular if low-level optimizations are required in order to reach high performance. For complex environments, such as multi-core clusters possibly with additional accelerators, these optimizations can be challenging when using a combination of several low-level libraries such as MPI, OpenMP, and CUDA.

One approach to overcome these issues is based on *Algorithmic Skeletons*. These are predefined typical, parallel-programming patterns, such as `map`, `fold` (a.k.a. `reduce`), and `zip`, which are implemented in parallel and can simply be composed by an application programmer without taking care of low-level programming aspects.

Support for algorithmic skeletons is typically provided as a library which contains the low-level parallel implementations. However, optimizations are hard to implement in this setting and programming might still be tedious for example because of required boiler plate code. Thus, we propose a domain-specific language (DSL) for algorithmic skeletons that simplifies the identification of relevant features for optimizations, which can now be incorporated into a generator to transform DSL code into C++ code, including calls to low-level libraries. Moreover, the DSL offers a set of necessary core features as well as meaningful validation and syntax highlighting, which facilitates efficient and robust code development.

We analyze four benchmark models, written in our DSL, as well as the execution time and speedup of the generated code. Our experimental results show that the models are significantly shorter and that the performance of the generated code can compete with equivalent implementations using the Muenster Skeleton Library (Muesli).

13.1 Introduction

Parallel programming for high-performance computing (HPC) is a difficult endeavor, which requires expertise in different frameworks and programming languages. For example, if the code targets a multi-core cluster environment, the programmer needs to know a framework for shared-memory architectures, such as OpenMP [CJv08], for distributed-memory architectures, such as MPI [GLS14], and possibly even for accelerators, such as CUDA [Nic+08] or OpenCL [SGS10] for GPUs. Especially if advanced performance tuning is required, basic skills might not be sufficient. Moreover, using different frameworks in combination can lead to errors, which are difficult to find and resolve.

One approach to solve this problem is based on predefined, typical parallel-programming patterns [Col91] such as `map`, `fold/reduce`, and `zip`. In addition to these data-parallel skeletons, there are task-parallel skeletons, such as `pipeline` or `farm`, and communication skeletons, such

as `gather` or `scatter`. In order to use a general-purpose skeleton, a function providing the application-specific details can be passed as an argument. This function is then executed in parallel on the different elements of a data structure or data stream.

Thus, by using algorithmic skeletons, low-level details become transparent to the programmer and he or she does not have to consider them or even does not need to know anything about the underlying frameworks. Moreover, algorithmic skeletons make sure that common parallel-programming errors, such as deadlocks and race conditions, do not occur.

There are several libraries, which provide algorithmic skeletons (see Section 13.2). A straightforward implementation of these skeletons introduces some overhead such as repeated calls to the function which is executed by a skeleton. On the level of C++ and using a combination of low-level frameworks, optimizations removing these overheads are difficult to implement, since relevant and irrelevant features are hard to distinguish. In the present paper, we show how to avoid these drawbacks by generating C++ code from a dedicated domain-specific language (DSL) model. Moreover, we demonstrate how such a DSL can facilitate the efficient development of parallel programs by reducing the language to the essential core features and offering useful validation for models.

Our paper is structured as follows: first, we give an overview about different libraries for algorithmic skeletons in Section 13.2. In Section 13.3, our DSL is described and Section 13.4 shows how the language constructs are transformed into C++ code. The results of four benchmark applications are presented in Section 13.5. In Section 13.6, we conclude and point out future work.

13.2 Related Work

Algorithmic skeletons for parallel programming are mostly provided as libraries. One instance is the Muenster Skeleton Library (Muesli), a C++ library which is used as a reference for the implementation of the presented approach. Muesli offers distributed data structures and the skeletons are member functions of them. Data-parallel skeletons implemented in Muesli are e.g. `map`, `fold`, `zip`, `mapStencil` and variants of these. Muesli works on multi-core and multi-GPU clusters [EK12; EK17].

Other well-known libraries are for example FastFlow, which focuses on task-parallel skeletons and stream parallelism for multi-core systems [Ald+10], and eSkel, which provides skeletons for C and MPI [Ben+05].

Two libraries we want to examine more closely here are SkePU2 [ELK17] and SkeTo [ME10], since they incorporate similar concepts as presented in this paper. SkePU2 includes a source-to-source compiler based on Clang and LLVM. A program written with SkePU can always be compiled into a sequential program. A precompiler transforms the program for parallel execution, e.g. adding the `__device__` keyword to functions. SkePU uses custom C++ attributes, which the precompiler recognizes and transforms accordingly.

SkeTo is a library for distributed-memory environments, which focuses on optimizations such as fusion transformations – i.e., combining two skeleton invocations into one and hence reducing the overhead for function calls and the amount of data which is passed between skeletons. The implementation is based on expression templates [Vel95], a metaprogramming technique. By using expression templates it is for example possible to a avoid temporary variables, which are required for complex expressions.

Lately, the concepts of model-driven and generative software development have gained attraction in academia and practice, mainly because of the expected benefits in development speed, software quality, and reduction of redundant code [SV06]. In addition, DSLs allow for better reuse and readability of models – targeted at both the modeling domain and user experience – while at the same time reducing the complexity through appropriate abstractions [MHS05]. In the domain of high-performance programming, few approaches have been presented in literature that adopt DSLs. Almorsy and Grundy [AG15] have presented a graphical notation to ease the shift from sequential to parallel implementations of existing software for CPU and GPU clusters. Anderson et al. [And+17] have extended the language Julia which is designed for scientific computing and partly aligned with the MATLAB notation. By applying optimization techniques such as parallelization, fusion, hoisting, and vectorization, the generated code significantly improves the computation. In contrast, our approach focuses on the parallelization on clusters of compute nodes.

There are also DSLs for parallel programming with skeletons or parallel patterns, which are embedded into other languages, or enable the creation of embedded DSLs, such as SPar [Gri+17] for C++ or Delite [Suj+14] for Scala. SPar uses C++ annotations and therefore, it is possible to parallelize an existing code base without much effort. Additionally, all features of the host language can easily be used. If a code base already exists, even a less complex standalone DSL requires more effort for a re-implementation. However, by reducing the language to core features in a standalone DSL the complexity can be easier handled by inexperienced programmers.

[DTK16] propose a DSL for designing parallel programs based on parallel patterns, which also allows for optimization and rewriting of the pattern composition. The DSL focuses on the management of non-functional properties such as performance, security, or power consumption. Based on the model and non-functional properties, a template providing an optimized pattern composition for the FastFlow library is generated, which the programmer can use to implement the application.

Since this work presents an own language for parallel programming, we want to highlight that some upcoming and established languages aim to benefit from parallel code execution as well. For example, Chapel has built-in concepts for parallel programming such as *forall loops* and the *cobegin statement*, which allows for starting independent tasks [CCZ07]. Also, the C++ standard includes extensions for parallel programming since version C++17 [Sta15]. For example, the algorithms `for_each` and `transform` are comparable to the presented skeletons `map(InPlace)`

. However, these capabilities are restricted to a single (potentially multi-core) node and do not support clusters.

13.3 A Domain-Specific Language for High-Level Parallel Programming

Many existing approaches to high-level parallel programming provide parallel constructs in the form of a library. As pointed out in the introduction, this causes some limitations such as the difficulty to implement optimizations and a higher entry barrier for inexperienced programmers. Thus, we propose a DSL named *Musket* (Muenster Skeleton Tool for High-Performance Code Generation) to tackle these limitations and generate optimized code.

13.3.1 Benefits of Generating High-Performance Code

The main drawback of using libraries for high-performance computing is the fact that library calls are included in arbitrarily complex code of a host language such as C++. Besides introducing some performance overhead, a library is always restricted by the host language's syntax. In contrast, important design decisions such a the syntax and structure of code can be selected purposefully when building a DSL. For example, different algorithmic skeletons as major domain concept for parallelization can be integrated as keywords in the designed language and recognized by the editing component. Consequently, the program specification is more readable for novice users who want to apply their domain knowledge.

A code generator can easily analyze the DSL features based on a formalized meta model and produce optimized code for the given hardware. Required transformations can be provided by the framework developers, for example when applying an algorithmic skeleton to a distributed data structure. In addition, this level of abstraction increases the readability for users who do not need to know the details of (potentially multiple) target platforms but can focus on the high-level sequence of activities.

With regard to framework developers who are concerned with efficient program execution, DSLs introduce additional flexibility. The abstract syntax of the parallel program can be analyzed and modified in order to optimize the generated high-performance code for the target hardware. In particular, recurring – and potentially inefficient – patterns of high-level user code can be transformed to hardware-specific low-level implementations by applying rewrite rules as described in [Ste+15]. For example, *map fusion* may be applied to combine multiple transformations on the same data structure instead of applying them consecutively.

Moreover, a DSL-based approach can be extended to additional platforms in the future by supplying new generator implementations – without changing the input programs. Compared to customizing compilers, DSL creation frameworks such as Xtext further support in creating usable

editing components with features such as syntax highlighting and meaningful model[1] validation [Bet13].

13.3.2 Language Overview

The Musket DSL targets rather inexperienced programmers who want to use algorithmic skeletons to quickly write high-performance programs that run on heterogeneous clusters. Therefore, a syntax similar to C++ was chosen to align with a familiar programming language that is common for high-performance scenarios such as simulating physical or biological systems. However, a Musket model is more structured than an arbitrary C++ program and provides four main sections which are described in more detail in the following[2]. The DSL was created using the Xtext language development framework which uses an EBNF-like grammar to specify the language syntax and derives a corresponding Ecore meta model [The18]. Furthermore, a parser as well as an editor component are generated which integrate with the Eclipse ecosystem for subsequent code generation. Consequently, common features of an integrated development environment (IDE) such as syntax highlighting, auto-completion, and validation are available and have been customized to provide contextual modeling support.

Meta-Information

The header of a Musket model consists of meta information that guides the generation process. On the one hand, target *platforms* and the compiler optimization *mode* can be chosen for convenient debugging of the program. More important, the configuration of *cores* and *processes* is used by the generator to optimize the code for a distributed execution on a high-performance cluster. For example, the setup of distributed data structures, the parallel execution of skeletons, and the intra-cluster communication of calculation results are then automatically managed. An exemplary model for a matrix multiplication according to the algorithm described in [EK17] is depicted in Listing **??**.

Data Structure Declaration

Because of the distributed execution of the program, all global data structures are declared upfront and distributed to the different compute nodes. Also, global constants can be defined in this block to easily parametrize the program (lines 6-10).

Musket currently supports several primitive data types (*float, double, integer*, and *boolean*). *Array* and *matrix* collection types also exist and are defined using the C++ template style, e.g. `matrix<double,512,512,dist> table;` . This definition contains the type and dimension of the collection, and also provides a keyword indicating whether the collection should be present

[1] The model-driven software development community prefers the notion DSL *model* rather than DSL *program*.

[2] Due to the lack of space, only the overall structure and the main concepts of the language are presented here and an excerpt of the DSL is given in Figure **??**. The full DSL specification can be found in our code repository [WR18].

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

const int dim = 16384;

matrix<float,dim,dim,dist> as = {1.0f};
matrix<float,dim,dim,dist> bs = {0.001f};
matrix<float,dim,dim,dist> cs = {0.0f};

float dotProduct(int i, int j, float Cij){
  float sum = Cij;

  for (int k = 0; k < cs.columnsLocal(); k++) {
    sum += as[[i,k]] * bs[[k,j]];
  }

  return sum;
}

main{
  as.shiftPartitionsHorizontally((int a) -> int {return -a;});
  bs.shiftPartitionsVertically((int a) -> int {return -a;});

  for (int i = 0; i < as.blocksInRow(); ++i) {
    cs.mapLocalIndexInPlace(dotProduct());
    as.shiftPartitionsHorizontally((int a)-> int {return -1;});
    bs.shiftPartitionsVertically((int a) -> int {return -1;});
  }

  as.shiftPartitionsHorizontally((int a) -> int {return a;});
  bs.shiftPartitionsVertically((int a) -> int {return a;});
}
```

Listing 13.1: Musket model for matrix multiplication.

on all nodes (*copy*), distributed across the nodes (*dist*, *rowDist*, or *columnDist*), or instantiated depending on the context (*loc*). The explicit distinction lets the user control the partitioning of a data structure by means of a user function (see subsection 13.3.2). To simplify the handling of distributed data structures, collections can be accessed either using their global index (e.g. `table[42]`) or the local index within the current partition (e.g. `table[[42]]`). Moreover, primitive and collection types can be composed into custom *struct* types.

User Function Declaration

The third section of a Musket program consists of custom user functions which specify behavior to be executed on each node within skeleton calls (such as the `dotProduct` function in lines 12-20). Therefore, a wide variety of calculations such as arithmetic and boolean expressions can be directly expressed in the DSL. In addition to assignments and skeleton applications, different control structures such as *sequential composition*, *if statements*, and *for loops* are available. Moreover, the modeler can use C++ functions from the standard library or call arbitrary external C++ functions.

Within functions, users can access globally available data structures (declared in the previous section) or create local variables to store temporary calculation results which are not available to other processes. The sophisticated validation capabilities allow for instant feedback to the user when errors are introduced in the model. For example, type inference aims to statically analyze the resulting data type of expressions or type casts, and thus warns the user before vainly starting the generation process.

Main Program Declaration

Finally, the overall sequence of activities in the program is described in the *main* block (lines 22-34 in Listing **??**). Besides the possible control structures and expressions described in the previous paragraphs, *skeleton functions* are the main features to write high-level parallel code. Currently, *map*, *fold*, *gather*, *scatter*, and *shift partition* skeletons are implemented in multiple variants. The different types of skeletons are described in more detail in Section 13.4. In general, they are applied to a distributed data structure and may take additional arguments such as the previously defined user functions. For convenience and code readability reasons, the user can instead specify a lambda abstraction for simple operations e.g. `(int a)-> int {return -a;}` . An excerpt of the Musket DSL concerning the main program declaration is depicted in Listing **??**.

Again, multiple validators have been implemented to ensure that the types and amount of parameters passed into skeletons match. Meaningful error messages such as depicted in Figure 13.1 can be instantly provided while writing the program instead of relying on cryptic failure descriptions when compiling the generated code.

To sum up, the Musket DSL represents a subset of the C++ language in order to handle the complexities of generating parallelism-aware and hardware-optimized code. With only few additions such as distribution modes, local/global collection access, and predefined skeleton

```
MainBlock:  // cf. Section 3.2.4
 'main' '{' content+=MainFunctionStatement+ '}';

MainFunctionStatement:
 MusketControlStructure |  // For loop and if clause variants
 MusketStatement ';';

MusketStatement:
 MusketVariable |          // Variable declarations
 MusketAssignment |        // Assigning values to variables
 SkeletonExpression |      // Arithmetic and boolean expressions
 FunctionCall;             // Function calls without assignment

SkeletonExpression: // apply skeleton to data structure (cf 3.2.2)
 obj=[CollectionObject] '.' skeleton=Skeleton;

Skeleton:           // available algorithmic skeletons
 {MapSkeleton} 'map'
     ('<' options+=MapOption (',' options+=MapOption)* '>')?
     '(' param=SkeletonParameterInput ')' |
 {FoldSkeleton} 'fold'
     ('<' options+=FoldOption (',' options+=FoldOption)* '>')? '(' identity=Expression ',' ↩
     param=SkeletonParameterInput ')'|
 {MapFoldSkeleton} 'mapFold'
     ('<' options+=FoldOption (',' options+=FoldOption)* '>')? '(' ↩
     mapFunction=SkeletonParameterInput ',' identity=Expression ',' ↩
     param=SkeletonParameterInput ')' |
 {ZipSkeleton} 'zip'
     ('<' options+=ZipOption (',' options+=ZipOption)* '>')?
     '(' zipWith=ObjectRef ',' param=SkeletonParameterInput ')' |
 {GatherSkeleton} 'gather'   '(' param=SkeletonParameterInput?')'|
 {ScatterSkeleton} 'scatter' '(' param=SkeletonParameterInput?')'|
 {ShiftPartitionsHorizontallySkeleton}
     'shiftHorizontally' '(' param=SkeletonParameterInput ')' |
 {ShiftPartitionsVerticallySkeleton}
     'shiftVertically' '(' param=SkeletonParameterInput ')'
; // alternative skeleton representations omitted

enum MapOption: index | localIndex | inPlace;
enum FoldOption: index;
enum ZipOption: index | localIndex | inPlace;

SkeletonParameterInput:
 InternalFunctionCall | // reference to user function (cf. 3.2.3)
 LambdaFunction;        // inline definition of functions
```

Listing 13.2: Excerpt of the Musket DSL

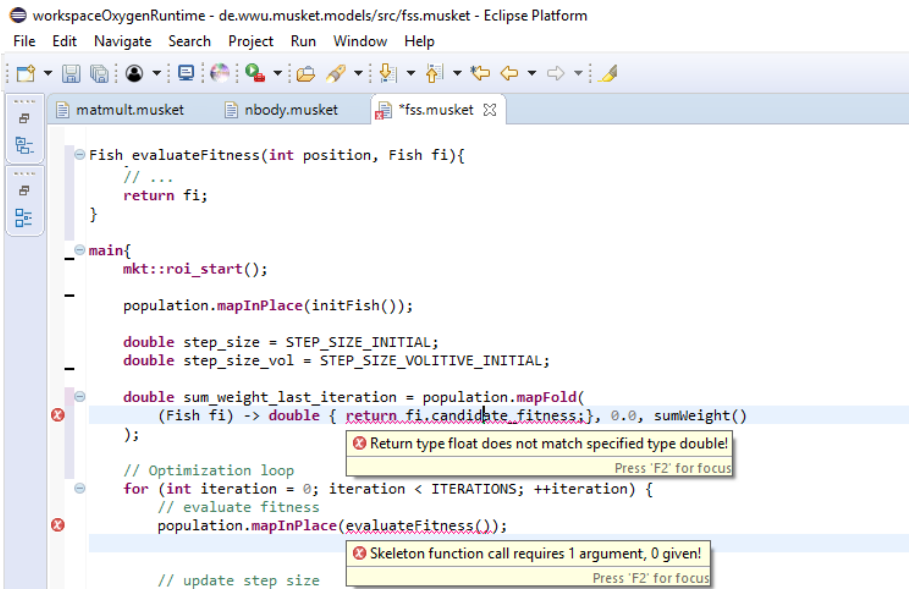


Figure 13.1: Integration of Custom Validation Errors in the Eclipse IDE

functions, a transformation of otherwise regular C++ code into distributed programs which are executable in a cluster environment can be achieved.

13.4 Code Generation for Multi-Core Clusters

In the following section, we demonstrate how certain language constructs are transformed into C++ code. We cover the data structures, data-parallel skeletons, as well as selected specific functions provided by the language. In general, we tried to generate code, which is still readable and makes use of modern features of C++11, 14, and 17.

Further, the way to generate code as described in the following is only one possibility. The approach of using a DSL allows for generating very different implementations to achieve the same behavior. It becomes also possible to consider cost models or descriptions of the target hardware and guide the generation accordingly. This also includes the generation of code for different architectures. By adding an additional generator, the same language and models can be reused to, for example, generate code for GPUs.

13.4.1 Data Structures

In general, all distributed data structures are represented as `std::vector` s. Based on the number of processes configured in the model and the distribution type, the size of the local vectors is calculated. Consequently, also for matrices the values are stored only in one vector. When an element in a matrix is accessed, the index is calculated accordingly.

Even though the size of the data structure is known when the code is generated, we decided to use `std::vector` over `std::array`. This is mostly because of the more efficient *move* operation for vectors: for some skeletons, intermediate buffers for sending and receiving data are required and we found vectors to be more efficient when data is moved from temporary buffers to the main vector.

Structs, which are defined in the model, are transformed into C++ structs. Additionally, a default constructor is generated, which initializes all members to default values. Moreover, the generation approach could be used to generate code for different data layouts. At the moment, the data is generated as array of structs, but it could be transformed to struct of arrays or any hybrid representation, which can increase the performance regarding data access and vectorization.

Moreover, there are collection functions, such as `show` and `size`, which can be invoked on data structures. Where possible, these function calls are already evaluated during the generation. For example, the global or local size is known for distributed data structures so that the function call can be replaced by the fixed value.

13.4.2 Model Transformation

The generation approach enables a re-writing step of skeleton calls by performing a model-to-model transformations before the actual generation. In this transformation, certain sequences of skeleton calls can be re-written in a more efficient way [Kuco4]. For example, one or more skeleton calls can be fused into one. This is the case for several calls of `map` on the same data structure. The sequence `a.mapInPlace(f); a.mapInPlace(g);` can be joined to `a.mapInPlace(g ∘f);`. For the generated code, this is one less parallel loop, which can save time for synchronization etc.

Also, different skeletons such as `map` and `fold` can be combined. In terms of the presented DSL, `a.mapInPlace(f); x = a.fold(0, g);` can be joined to `x = a.mapFold(f, 0, g);`. In the generated code this results in one parallel for loop with reduction and a call to `MPI_Allreduce` instead of two loops and the MPI call. Moreover, the intermediate result does not need to be stored in the result data structure. However, this transformation would only be valid if `a` was not used in any subsequent skeleton calls.

13.4.3 Custom Reduction

The implementation of the fold skeleton is based on a straightforward sequence of the OpenMP pragma `#pragmaomp parallel for simd reduction` for performing a local reduction in each thread, followed by an `MPI_Allreduce` for combining the local intermediate results. MPI requires a function with the following signature `void f(void *in, void *inout, int *len, MPI_Datatype *dptr)`, which can then be used in reduction operations. By generating this reduction function, it is possible to avoid the combination of a gather operation followed by a second local fold. Additionally, this allows for using complex types in the reduction.

13.4.4 User Functions

The current implementation makes heavy use of inlining to avoid overhead for function calls. This behaviour seems to be advantageous in certain situations, as observed for the Frobenius norm and matrix multiplication benchmarks, but it can also come with problems such as thrashing and cache misses, as observed in the Nbody benchmark (see Section 13.5). To this respect, the generation approach provides a rather convenient possibility to provide alternative code for the same model. However, this shifts the problem to selecting the best alternative.

13.4.5 Specific Musket Functions

There are some additional functions provided by Musket, which are not part of the standard library, such as `rand`. If the `rand` function is used in the model, random engines and distribution objects are generated in the beginning of the main function, so that they can be re-used without additional overhead. The actual call to `rand` is generated as `rand_dist[thread_id](random_engines[thread_id])`, thus it can be used as a part of an expression. Consequently, the DSL conveniently reduces the amount of boilerplate code, since the function can simply be used in the model without, for example, creating an object which creates the random engines on construction.

13.4.6 Build Files

The generation approach offers additional convenience for programmers. In addition to the source and header files, we also generate a CMake file and scripts to build and run the application as well as Slurm job files [YJG03]. Consequently, there is no effort required for the setup and build process, which lowers the entry threshold to parallel programming for inexperienced programmers.

13.5 Benchmarks

We used four benchmark applications to test our approach: calculation of the Frobenius norm, Nbody simulation, matrix multiplication, and Fish School Search (FSS). In the following subsections, we demonstrate the models, compare them to the C++ implementations with Muesli, and analyze the execution times for both. All execution times are presented in Table 13.2. The code has been compiled with g++7.1.0 and OpenMPI 3.0.0. Each node of the cluster we have used for the benchmark is equipped with two Intel Xeon E5-2680 v3 CPUs (12 cores each, 30MiB shared L3 cache per CPU and 256KiB L2 cache per core) and 7200MiB memory per node. Hyper-Threading has been disabled.

Table 13.2: Execution times of the benchmark applications (in seconds)

			Execution times (s)		
Benchmark	**Nodes**	**Cores**	**Muesli**	**Musket**	**Speedup**
Frobenius norm	1	1	9.8932	2.4589	4.0234
	1	6	2.1389	0.7561	2.8289
	1	12	1.4890	0.7177	2.0748
	1	18	1.5508	0.7282	2.1295
	1	24	1.6015	0.7279	2.2001
	4	1	2.4793	0.6810	3.6407
	4	6	0.5308	0.2020	2.6281
	4	12	0.3925	0.1904	2.0611
	4	18	0.3943	0.1908	2.0667
	4	24	0.3915	0.1859	2.1064
	16	1	0.6703	0.1723	3.8900
	16	6	0.1497	0.0524	2.8575
	16	12	0.1040	0.0503	2.0667
	16	18	0.1018	0.0500	2.0347
	16	24	0.1060	0.0515	2.0585
Nbody simulation	1	1	7422.7370	1482.3765	5.0073
	1	6	1234.0740	1236.6444	0.9979
	1	12	616.9001	617.8872	0.9984
	1	18	411.3290	412.0027	0.9984
	1	24	309.0764	322.9131	0.9572
	4	1	1850.1790	1673.9681	1.1053
	4	6	308.7439	309.0844	0.9989
	4	12	154.3643	154.5944	0.9985
	4	18	102.9115	103.0676	0.9985
	4	24	77.7512	87.0580	0.8931
	16	1	473.8626	469.9092	1.0084
	16	6	77.2279	79.9543	0.9659
	16	12	38.6245	42.9204	0.8999
	16	18	25.7638	25.8689	0.9959
	16	24	23.1896	23.2932	0.9956
Matrix Multiplication	1	1	83326.9252	14401.2013	5.7861

Benchmark	Nodes	Cores	Execution times (s)		Speedup
			Muesli	Musket	
	1	6	14726.9833	2832.8029	5.1987
	1	12	7700.4190	1634.6645	4.7107
	1	18	5185.8780	1166.2151	4.4468
	1	24	4758.7190	1045.0841	4.5534
	4	1	19245.4000	3641.2103	5.2854
	4	6	3578.7240	704.2898	5.0813
	4	12	2086.8870	369.6801	5.6451
	4	18	1549.4470	249.6089	6.2075
	4	24	1376.8620	197.2717	6.9795
	16	1	3655.1590	764.7999	4.7792
	16	6	729.3905	158.9674	4.5883
	16	12	413.3354	64.0172	6.4566
	16	18	252.2129	41.1361	6.1312
	16	24	224.8478	39.4737	5.6961
Fish School Search	1	1	916.3965	710.2205	1.2903
	1	6	158.2332	124.2070	1.2739
	1	12	81.0045	69.1641	1.1712
	1	18	54.8713	53.2463	1.0305
	1	24	45.5823	44.6268	1.0214
	4	1	211.6765	180.3886	1.1734
	4	6	40.1817	32.2177	1.2472
	4	12	20.9549	17.7964	1.1775
	4	18	15.2139	14.2155	1.0702
	4	24	13.5301	13.7475	0.9842
	16	1	54.9478	45.7327	1.2015
	16	6	11.7779	9.9477	1.1840
	16	12	7.1947	5.8437	1.2312
	16	18	5.9784	5.2521	1.1383
	16	24	5.7644	5.8257	0.9895

```
1  #config PLATFORM CPU
2  #config PROCESSES 4
3  #config CORES 24
4  #config MODE release
5
6  matrix<double,32768,32768,dist> as;
7
8  main{
9    // init
10   as.mapIndexInPlace((int x, int y, double a) -> double {return (double) x + y + 1.5;});
11
12   mkt::roi_start();
13
14   as.mapInPlace((double a) -> double {return a * a;});
15   double fn = as.fold(0.0, (double a, double b) -> double {return a + b;});
16   fn = std::sqrt(fn);
17
18   mkt::roi_end();
19
20   mkt::print("Frobenius norm is %.5f.\n", fn);
21 }
```

Listing 13.3: Model for Frobenius norm calculation

13.5.1 Frobenius Norm

The calculation of the Frobenius norm for matrices consists of three steps. First, all values are squared, then all values are summed up, and finally, the square root of the sum yields the result. We used a 32768×32768 matrix with double precision values. The model is presented in Listing ??.

There is one matrix defined as a distributed data structure. Since all user functions are written as lambda expressions, there is no need for separately defined user functions. Within the main block, the logic for the program is defined. First, in this case, the matrix is initialized with arbitrary values. The calculation of the Frobenius norm is modelled in lines 14-16. The functions `roi_start` and `roi_end` are merely for benchmark purposes and trigger the generation of timer functions. In line 20, the Musket function `print` is used, so that the result is only printed once by the main process.

The results for this benchmark already reveal some interesting insights, even though the complexity of the program is rather low, which is also the reason why the program does not scale very well, when increasing the number of cores per node.

Musket achieves a very high speedup for the sequential implementations, especially on one node. That is due to the fact that Muesli does not consider this configuration a special case, but relies on the fact that MPI routines also work for one process and that all used OpenMP pragmas are ignored for sequential programs. In contrast, Musket checks the configuration and generates the code accordingly, e.g. there are no MPI routines in the program, if there is only one process. Additionally, there are less operations required regarding the management of the

data structures. When a data structure is created in Muesli, the global and local sizes have to be calculated, the new memory has to be allocated etc. In Musket, the data structures are generated as global variables, and all required information, such as the size, can be generated and therefore have not to be calculated during program execution. The same behavior can be observed for the Nbody simulation and FSS benchmarks.

To give a perspective on the effort of writing a parallel program, we want to point to the lines of code for the Musket model, the Muesli implementation, and the generated source file. We did not use the lambda notation for Musket for counting the lines of code, since Muesli requires functors in certain situations and in this way the results become more comparable. While the Musket model consists of 20 lines, the Muesli implementation has 45 lines and the generated source file 94. Thus, we conclude that the DSL provides a concise way to express a parallel program.

Another aspect to mention here is the skeleton fusion optimization. The lines 14 and 15 could also be written as `double fn = as.mapFold(square(), 0.0, sum());`, if we assume that the lambda expressions correspond to the respective functions. In the generated code, this would reduce the two loops for the map and fold into a single loop. The execution times in Table 13.2 do not reflect this optimization to keep the results comparable, because Muesli does not offer a combined `mapFold` skeleton. As an example, for a configuration with 4 nodes and 24 cores the execution time has been 0.07s with skeleton fusion, which corresponds to a speedup of 2.68 compared to the Musket implementation without skeleton fusion.

13.5.2 Nbody Simulation

The Nbody simulation shows a case where inlining is not advantageous, since the results show that the benchmark runs faster with Muesli than with Musket for most configurations. We have used 500.000 particles over five time steps for the benchmark, so there is a lot of data required for the calculation, since a copy of all particles is needed to calculate the force acting upon one particle.

We have simulated the behavior of the application with Valgrind's cachegrind and callgrind tools [NS07]. The number of L3 cache misses as well as the amount of unnecessary data loaded to cache is higher than for the Muesli implementation. This results in the higher execution times. In order to overcome this, we could, of course, make inlining optional.

In terms of complexity, both implementations have about the same size. The Musket model consists of 77 lines of code, whereas the Muesli program consists of 84 lines.

13.5.3 Matrix Multiplication

The matrix multiplication benchmark shows a case, in which massive optimizations become possible due to the code generation approach. We have performed the matrix multiplication with matrices of 16.384×16.384 single precision values, but first, we have to emphasize again that Musket targets rather inexperienced programmers. The benchmark is set up in such a way that

the second matrix for the multiplication is not transposed. Hence, the data is stored row major, but the user function iterates column-wise through the matrix, which leads to inefficient memory accesses. The model is shown in Listing **??** in Section 13.3.

In the Muesli implementation, the compiler is not able to vectorize the calculation and to optimize the memory access. However, this is the case for the generated program, which leads to significant shorter execution times. The speedups for configurations with multiple nodes and cores range between 4.59 and 6.98.

The model has 42 lines of code, while the Muesli implementation has 74. Again, the effort for implementing the benchmark has been reduced. In comparison, the generated code has 662 lines of code. This is mainly because of the inlining done by the generator.

13.5.4 Fish School Search

The FSS benchmark showcases a complex and more real-world example of a parallel program. FSS is a swarm-intelligent meta-heuristic to solve hard optimization problems [Bas+08]. The model has 244 lines of code and the Muesli implementation even 623, which is a reduction of about 61%. The generated code consists of 390 lines of code. At first it might be surprising that the Muesli implementation is longer than the generated code, but this is due to the fact that Muesli has some limitations regarding struct types in distributed data structures and, therefore, multiple data structures have to be created. Moreover, the Muesli implementation requires the implementation of functors to be able to access other data structures, which leads to longer code. A detailed discussion of the implementation with Muesli can be found in [WMK18].

Listing **??** shows excerpts of the FSS model. Since Musket also supports distributed data structures of complex types, – which can include arrays – it becomes very convenient to work with. The struct for *Fish* is defined in lines 9–19 and the distributed array is defined in line 21. The fact that complex types are allowed in distributed data structures highlights the benefit of the fused `mapFold` skeleton. For one operator called *collective volitive movement*, it is necessary to calculate the sum of the weight of all fish. Line 26 shows how this can be conveniently done by invoking the `mapFold` skeleton on the `population` array. In the map part, the weight of each fish is taken and in the fold part the sum is calculated. In the generated, code this is efficiently performed in a single parallelized loop. The execution times show a slight improvement for most configurations with speedups up to 1.29 for the sequential program.

13.6 Conclusions and Future Work

In this paper, we have proposed a DSL for parallel programming, which is based on algorithmic skeletons. The benchmark applications have shown that the DSL offers a convenient and concise way to express applications. Regarding the execution times, the generated code can offer significant speedups (of up to 7) compared to the library-based approach Muesli for most benchmarks

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

const int NUMBER_OF_FISH = 2048;
const int DIMENSIONS = 512;

struct Fish{
  array<double,DIMENSIONS,loc> position;
  double fitness;
  array<double,DIMENSIONS,loc> candidate_position;
  double candidate_fitness;
  array<double,DIMENSIONS,loc> displacement;
  double fitness_variation;
  double weight;
  array<double,DIMENSIONS,loc> best_position;
  double best_fitness;
};

array<Fish,NUMBER_OF_FISH,dist> population;
// [...]

main{
  // [...]
  double sum_weight = population.mapFold(
      (Fish fi) -> double {return fi.weight;}, 0.0,
      (double a, double b) -> double {return a + b;});
  // [...]
}
```

Listing 13.4: Model for Fish School Search

and configurations. The generated code is very similar to the one which an experienced parallel programmer would write using MPI and OpenMP and it reaches the same speed and speedups. An interesting observation was that too extensive inlining may cause a small overhead. Thus, there should be an option to switch it off. Alternatively, this could be done based on a (very precise) cost model. Another possibility is to generate both versions of the program and compare them experimentally.

At the moment, there is only one generator for multi-core clusters and we intend to add additional generators for multi-GPU clusters. Moreover, the current generator for multi-core clusters does not yet fully support all optimizations described in this paper and we aim at implementing those in the near future. This includes the data layout as well as more extensive rewriting of skeleton calls.

References

[AG15] M. Almorsy and J. Grundy. "Supporting Scientists in Re-engineering Sequential Programs to Parallel Using Model-Driven Engineering". In: *2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*. IEEE, 2015, pp. 1–8. DOI: 10.1109/SE4HPCS.2015.8.

[Ald+10] M. Aldinucci et al. "Efficient Streaming Applications on Multi-core with FastFlow: The Biosequence Alignment Test-bed". In: *Parallel computing: from multicores and GPU's to petascale*. Ed. by B. Chapman et al. Vol. 19. Advances in Parallel Computing. Amsterdam: IOS Press, 2010. DOI: 10.3233/978-1-60750-530-3-273.

[And+17] T. A. Anderson et al. "Parallelizing Julia with a Non-Invasive DSL". In: *31st European Conference on Object-Oriented Programming (ECOOP '17)*. Ed. by P. Müller. Vol. 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 4:1–4:29. DOI: 10.4230/LIPIcs.ECOOP.2017.4. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7269.

[Bas+08] C. J. A. Bastos-Filho et al. "A Novel Search Algorithm Based on Fish School Behavior". In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC '08)*. IEEE, 2008, pp. 2646–2651. DOI: 10.1109/ICSMC.2008.4811695.

[Ben+05] A. Benoit et al. "Flexible Skeletal Programming with eSkel". In: *Proceedings of the 11th International Euro-Par Conference on Parallel Processing (Euro-Par '05)*. Ed. by J. C. Cunha and P. D. Medeiros. Vol. 3648. Lecture Notes in Computer Science. Springer, 2005, pp. 761–770. DOI: 10.1007/11549468{\textunderscore}83.

[Bet13] L. Bettini. *Implementing Domain-specific Languages with Xtext and Xtend*. Community experience distilled. Birmingham, UK: Packt Pub, 2013.

[CCZ07] B. L. Chamberlain, D. Callahan, and H. P. Zima. "Parallel Programmability and the Chapel Language". In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[CJv08] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Scientific and Engineering Computation. Cambridge, MA, USA: MIT Press, 2008.

[Col91] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[DTK16] M. Danelutto, M. Torquati, and P. Kilpatrick. "A DSL Based Toolchain for Design Space Exploration in Structured Parallel Programming". In: *Procedia Computer Science* 80 (2016). International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA, pp. 1519–1530. DOI: https://doi.org/10.1016/j.procs.2016.05.477. URL: http://www.sciencedirect.com/science/article/pii/S1877050916309620.

[EK12] S. Ernsting and H. Kuchen. "Algorithmic Skeletons for Multi-core, Multi-GPU Systems and Clusters". In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138. DOI: 10.1504/IJHPCN.2012.046370.

[EK17] S. Ernsting and H. Kuchen. "Data Parallel Algorithmic Skeletons with Accelerator Support". In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299. DOI: 10.1007/s10766--016--0416--7.

[ELK17] A. Ernstsson, L. Li, and C. Kessler. "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems". In: *International Journal of Parallel Programming* (2017). DOI: 10.1007/s10766-017-0490-5.

[GLS14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. 3rd ed. Scientific and Engineering Computation. Cambridge, MA, USA: MIT Press, 2014.

[Gri+17] D. Griebler et al. "SPar: A DSL for high-level and productive stream parallelism". In: *Parallel Processing Letters* 27.01 (2017), p. 1740005.

[Kuc04] H. Kuchen. "Optimizing Sequences of Skeleton Calls". In: *Domain-Specific Program Generation*. Ed. by C. Lengauer et al. Vol. 3016. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 254–274.

[ME10] K. Matsuzaki and K. Emoto. "Implementing Fusion-Equipped Parallel Skeletons by Expression Templates". In: *Implementation and Application of Functional Languages*. Ed. by M. T. Morazán and S. Scholz. Berlin, Heidelberg: Springer, 2010, pp. 72–89.

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[Nic+08] J. Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53. DOI: 10.1145/1365490.1365500.

[NS07] N. Nethercote and J. Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[SGS10] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science and Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[Sta15] ISO Standard. *Programming Languages – Technical Specification for C++ Extensions for Parallelism*. Standard ISO/IEC TS 19570:2015. Geneva, Switzerland: International Organization for Standardization, 2015. URL: https://www.iso.org/standard/65241.html.

[Ste+15] M. Steuwer et al. "Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP '15. Vancouver, BC, Canada: ACM, 2015, pp. 205–217. DOI: 10.1145/2784731.2784754. URL: http://doi.acm.org/10.1145/2784731.2784754.

[Suj+14] Arvind K Sujeeth et al. "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014), p. 134.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development*. Chichester: John Wiley & Sons, 2006.

[The18] The Eclipse Foundation. *Xtext Documentation*. 2018. URL: https://eclipse.org/Xtext/documentation/ (visited on 03/29/2018).

[Vel95] T. Veldhuizen. "Expression Templates". In: *C++ Report* 7.5 (1995), pp. 26–31.

[WMK18] F. Wrede, B. Menezes, and H. Kuchen. "Fish School Search with Algorithmic Skeletons". In: *International Journal of Parallel Programming* (Mar. 2018). DOI: 10.1007/s10766-018-0564-z. URL: %5Curl%7Bhttps://doi.org/10.1007/s10766-018-0564-z%7D.

[WR18] F. Wrede and C. Rieger. *Musket Material Respository*. https://github.com/wwu-pi/musket-material. 2018. URL: https://github.com/wwu-pi/musket-material.

[WRK18] Fabian Wrede, Christoph Rieger, and Herbert Kuchen. "Generation of High-Performance Code Based on a Domain-Specific Language for Algorithmic Skeletons". In: *High-Level Parallel Programming and Applications (HLPP)*. Orleans, France, 2018.

[YJG03] A. B. Yoo, M. A. Jette, and M. Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Berlin, Heidelberg: Springer, 2003, pp. 44–60.

14

Towards Model-Driven Business Apps for Wearables

Table 14.1: Fact sheet for publication P8

Title	Towards Model-Driven Business Apps for Wearables
Authors	Christoph Rieger[1] Herbert Kuchen[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2018
Conference	Mobile Web and Intelligent Information Systems
Copyright	Springer International Publishing
Full Citation	Christoph Rieger and Herbert Kuchen. "Towards Model-Driven Business Apps for Wearables". In: *Mobile Web and Intelligent Information Systems*. Ed. by Muhammad Younas, Irfan Awan, George Ghinea, and Marisa Catalan Cid. Cham: Springer International Publishing, 2018, pp. 3–17. DOI: 10.1007/978-3-319-97163-6_1 This version is not for redistribution. The final authenticated version is available online at https://doi.org/10.1007/978-3-319-97163-6_1.

Towards Model-Driven Business Apps for Wearables

Christoph Rieger Herbert Kuchen

Keywords: Cross-Platform, Model-Driven Software Development, Multi-Platform, Wearables, Mobile App, Business App

Abstract: With the rise of wearable devices expected to continue in the near future, traditional approaches of manually developing apps from scratch for each platform reach their limits. On the other hand, current cross-platform approaches are usually limited to platforms for smartphones and tablets. The model-driven paradigm seems well suited for developing apps for novel and heterogeneous devices. However, one of the main challenges for establishing a model-driven framework for wearables consists of bridging the variety of user interfaces and considering different capabilities of device input and output. This paper seeks to investigate the challenges of app development for wearable devices regarding user interfaces and discusses a possible mapping of typical application building blocks in the domain of business apps. Ultimately, apps modelled on a task-oriented level of abstraction using platform-independent notations such as MAML or CTT can then be transformed into code that adopts device class specific representations.

14.1 Introduction

Wearables have seen a drastic increase in popularity in the last years, with most major vendors providing software platforms and hardware products such as smartwatches and fitness devices for the mainstream consumer market. According to Gartner, the wearable device market is expected to grow significantly in the forthcoming years to more than 500 million devices sold in 2021 [vF17]. Many of these devices will be app-enabled and accessible to third-party developers.

For now, many available wearable applications – so-called apps – are of rather exploratory nature, providing only a limited set of companion functionality compared to a main app running on a smartphone. However, traditional approaches to app development are limited with regard to the plethora of input and output capabilities by current and announced wearable devices such as smartwatches and smart glasses. In addition, not all cross-platform approaches can be used for developing apps for wearables due to technical limitations. Many devices do not provide web views or support JavaScript execution, excluding hybrid frameworks such as the popular Apache Cordova.

Therefore, new methods are needed in order to extend app development to these different device classes. model-driven software development (MDSD) seems particularly suitable for targeting a large range of devices and ease the development as opposed to the repetitive manual implementation. When extending existing model-driven mobile development approaches to wearables, adapting business logic is probably not the most pressing issue and transpiling approaches such as ICPMD [El-+14] may be applied. One of the main challenges consists of bridging the heterogeneity

of User Interface (UI) regarding input capabilities and information visualisation according to the platform's User experience (UX).

In addition, wearable devices are not used in isolation but often considered as connected devices – although they might technically run standalone apps such as with Google's Wear OS (formerly Android Wear 2.0). Most likely, wearable apps in the near future will co-exist with other device counterparts or interact with their environment such as with smart personal agents or cyber-physical sensor networks. With regard to these scenarios, MDSD can play out its strengths even more when apps are jointly modelled for multiple devices using a single, abstract representation, and individually transforming the models to device-specific interfaces.

This paper seeks to investigate the challenges of app development for wearable devices, in particular related to the diversity issue of device interfaces. As a building block on the path towards extending model-driven development approaches to wearable devices, it contributes a conceptual mapping of UI representations across multiple device classes for typical operations in the domain of business apps. The eventual aim is to spark discussion on the long-standing issues of cross-platform UI modelling. The structure of this paper follows these contributions: after discussing related work in Section 14.2, we highlight the current challenges and our proposed mapping of user interfaces and suitable modelling approaches in Section 14.3. Section 14.4 discusses the applicability of model-driven development for wearable devices before we conclude in Section 14.5.

14.2 Related Work

In this work, we focus on the domain of business apps, i.e. form-based, data-driven applications interacting with back-end systems [MEK15]. Using this definition, business apps not only refer to smartphone applications but also apply to a broader scope of *app-enabled* devices. These can be described as being extensible with software that comes in small, interchangeable pieces that are usually provided by third parties unrelated to the hardware vendor or platform manufacturer and increase the versatility of the device after its introduction [RM17]. Although related to the term *mobile computing*, app-enablement also considers stationary devices such as smart TVs, smart personal assistants, or smart home devices.

Cross-platform overview papers such as [JM15] typically focus on a single category of devices and apply a very narrow notion of mobile devices. [RM17] provides the only classification that includes novel device classes. Furthermore, few papers provide a technical perspective on apps *spanning multiple device classes*. Singh and Buford [SB16] describe a cross-device team communication use case for desktop, smartphones, and wearables, and Esakia et al. [ENM15] performed research on Pebble smartwatch and smartphone interaction in computer science courses. In the context of Web-of-Things devices, Koren and Klamma [KK16] propose a middleware approach to integrate data and heterogeneous UI, and Alulema et al. [AIC17] propose a DSL for bridging the presentation layer of heterogeneous devices in combination with web services for incorporating business logic.

With regard to commercial cross-platform products, Xamarin[1] and CocoonJS[2] provide Android Wear support to some extent. Whereas several other frameworks claim to support wearables, this usually only refers to accessing its data by the main smartphone application or displaying notifications on coupled devices.

Together with the increase in devices, new software platforms have appeared, some of which are either related to established operating systems for other device classes or are newly designed to run on multiple heterogeneous devices. Examples include Wear OS, watchOS, and Tizen. Although these platforms ease the development of apps (e.g., reusing code and libraries), subtle differences exist in the available functionality and general cross-platform challenges remain.

14.3 Creating Business App UIs for Wearable Devices

To pave the way towards model-driven software development for wearable devices, we provide a possible mapping of typical tasks in this domain to different app-enabled devices and present a model-driven app development approach.

14.3.1 Challenges of Wearable UIs

From development and usage perspectives, two main categories of UI/UX challenges related to app development across different device classes can be identified.

Diversity of input and output capabilities

Traditionally, mobile apps are designed for rectangular screen sizes between 4” and 10” to cover smartphones and tablets with similar visual characteristics. However, wearables vary greatly in terms of screen size or provide completely different means of output such as audio or projection. In addition, current interface design considerations such as device orientation and pixel density are aggravated due to the introduction of different aspect ratios (e.g. ribbon-like fitness devices worn around the wrist), positions (e.g. objects at different angles and depths within the field of view for smart glasses) or form factors (e.g. round smartwatches) [RM17].

Correspondingly, novel app-enabled devices provide different possibilities for user interaction which span from hardware buttons to handling graphical UI elements on touch screens, using auxiliary devices (e.g., stylus pens), and voice inputs [RM17]. Moreover, multiple input alternatives may be available on one device and used depending on user preferences or usage context.

Multi-device usage patterns

Until now, cross-platform approaches were mainly designed to provide equivalent functionality for similar devices with different operating systems. However, novel app-enabled devices usually do

[1] https://www.xamarin.com
[2] https://docs.cocoon.io/article/canvas-engine/

not replace smartphone usage but represent complementary devices which are used contextually (e.g. location- or time-based) or depending on user preferences. This might occur *sequentially* when a user switches to a different device, e.g., using an app with smart glasses while walking and switching to the in-vehicle app when boarding a car. Alternatively, a *concurrent* usage of multiple devices for the same task is possible, for instance in second screening scenarios in which one device provides additional information or input/output capabilities for stationary devices in the room [NJE17]. Also, automated device-to-device communication (e.g. with sensor networks) might become more common in future mobile scenarios. Cross-platform development frameworks need to consider this additional complexity through device management as well as fast and reliable synchronisation which automatically updates other devices based on the current application state.

14.3.2 Conceptual UI Mapping

From the current use cases of business apps on smartphones and tablets, some types of user tasks are prevalent. These are mainly related to Create/Read/Update/Delete (CRUD) operations as well as mobile functionalities such as calling a person, receiving notifications, or accessing sensor information (e.g., GPS location). Based on the concepts of *abstract interaction objects* [VB93] and *presentation units* [EVP01b], a mapping is desirable to transform these typical operation components to concrete widget representations of novel smart devices.

In the following, we describe such a conceptual mapping for business app tasks. It is derived from an analysis of the publicly available design guidelines and best practices by several vendors (see Table 14.2) which provide app-enabled devices for ubiquitous usage (in contrast to specialised devices such as for cycling). Besides identifying suitable patterns complying to these guidelines, we also keep a generalisable appearance in mind (for inclusion in cross-platform frameworks).

Table 14.2: Analysed Design Guidelines per Device Class

Device class	**OS guidelines**	**Representative device**
Smartphones/Tablets	Android	Samsung Galaxy S8
	iOS	iPhone X
Smartwatches	Android Wear/Wear OS	LG Watch Sport
	watchOS	Apple Watch
	Fitbit	Fitbit Ionic
	Tizen	Samsung Gear
Smart glasses	Glass OS	Google Glass
	HoloLens	Microsoft HoloLens
Smart personal assistants	Amazon Alexa	Amazon Echo
	Actions on Google	Google Home

Figure 14.1: Possible Interface Representations for *Create* and *Update* Task Types

Therefore, possible representations are juxtaposed for traditional mobile devices (smartphones, tablets) and novel app-enabled devices (smartwatches, smart glasses, and smart personal agents). However, to allow for concrete visualisations, we chose representative devices of each class, in particular an iPhone smartphone, an Android-based tablet and smartwatch, and head-mounted Google glass.

Create and Update

Create and *update* interfaces usually share a similar design and are therefore considered together in Figure 14.1. On smartphones, create or update steps are usually represented as a list of input fields with corresponding captions and potentially provide contextual help such as placeholder texts or pop-up hints. To better make use of the available space, tablet apps support a multi-column layout, especially when the input fields can be grouped into multiple categories. Due to screen space limitations, smartwatch input can be either represented using a one-column scrolling layout or by a sequence of views per input field. Smart glasses can display the respective fields one by one and are updated using voice controls similar to voice-based smart personal assistants. To allow for a more flexible order of user inputs, advanced techniques might use the concept of frame-based dialogue managers known from chatbot applications [God+96]. Using so-called slots for each attribute to be set, the user can provide the required information in any order and the system can focus on asking for missing information to complete the given task.

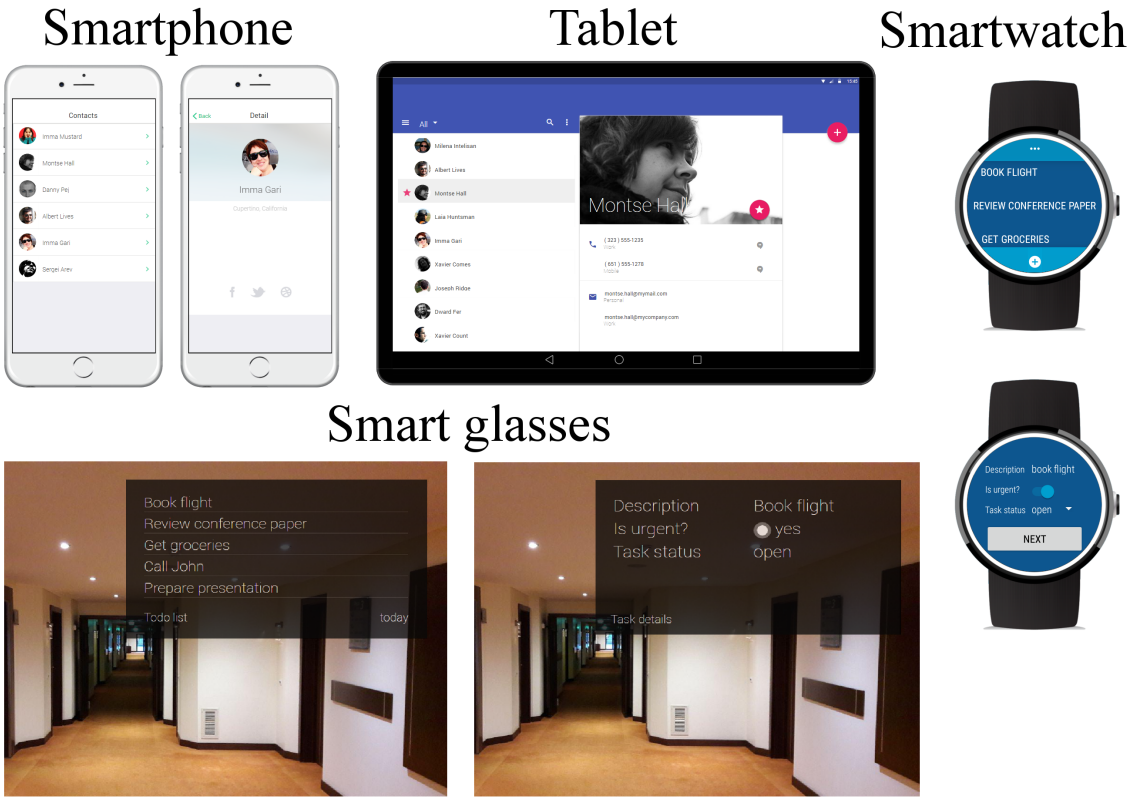


Figure 14.2: Possible Interface Representations for *Select* and *Read* Task Types

Select and Read

The *read* operation needs to be distinguished from another related task type: often the user first needs to *select* an item from a collection of objects, before seeing the object's detailed content as depicted in Figure 14.2. Therefore, smartphones often provide a scrollable list of items that can be filtered using search keywords and navigated using jump marks. Smartwatches also provide scrollable lists, for example using curved layouts to exploit the round layout of Android-based watches. After selecting an element, individual objects might further be split into different views (e.g., using logical groups of fields) to avoid scrolling behaviour. The same principles apply to smart glasses which also have a limited virtual screen size. Voice-based interfaces may read the list of potential answers or allow users to provide keywords in order to limit the set of results. However, tablets can combine select and read tasks in a single view using the so-called master-detail pattern: The left column of the view presents the list of objects and upon selection updates the right side displaying its content.

Delete and modal Pop-ups

When a *delete* operation is triggered, apps usually require a confirmation to avoid accidental information loss. This confirmation is represented as a modal pop-up view that provides the

options to confirm or cancel the current action (in case of deletion) or acknowledge the information prominently displayed on screen. The user must interact with this message in order to continue with his workflow. Therefore, modal views look similar on most screen-based devices and cover the majority of the screen. Voice-based interfaces can instead read out the message and wait for the user to react to it.

Mobile-specific tasks

Many mobile-specific tasks have similar representations across screen-based devices. Firstly, *notifications* are present on most devices as unobtrusive information about events when interacting with other apps, either as textual hint at the top or bottom of the screen. In contrast to pop-up messages, notifications disappear automatically or can be closed without blocking the user from performing his actions or waiting for a decision between different options. Voice-based devices can instead read out the message.

Secondly, *phone calls* can be performed not only on smartphones but also on tablets, smartwatches, smart glasses, and smart personal assistants as long as they are equipped with microphone, audio output capabilities, and cellular network connection (potentially indirectly through a connected smartphone). The communication target can usually be chosen by manually dialling a number or selecting someone from a list of contacts (cf. *select* task type) via touch or voice commands.

Finally, *sensor data* may be accessed by specific representations. For example, the GPS location can be visualised by a map on screen-oriented devices. However, this data is commonly contained in attributes of other data objects and therefore already considered in the *read* and *update* task types.

14.3.3 Modelling Apps Across Device Classes

The model-driven paradigm can provide strong benefits to app development both in terms of development effort regarding the plethora of novel devices as well as the integration and interaction with other applications. Arguably, many concepts from the more established domain of cross-platform development for smartphones can be reused and adapted. To achieve this, the challenges mentioned in Subsection 14.3.1 need to be tackled systematically.

Regarding the diversity of input and output characteristics, a high level of abstraction beyond screen-oriented UIs is mandated, for example using declarative notations for representing use tasks. Consequently, arbitrary platforms can be supported by developing respective generators which implement a suitable mapping from descriptive models to platform-specific implementations.

Ideally, a “one model fits all platforms” approach can therefore be achieved by applying two types of transformations that do not modify the content but adapt the task appearance to different supported devices:

1. On the one hand, information can be *layouted* according to the available screen sizes, for example by choosing an appropriate appearance – one could think of tabular vs. graphical representations – or, if necessary, leave out complementary details. In the inventory management example, the round design of a smartwatch can be exploited by the curved list layout such that content readability is improved and screen usage is maximised.

2. On the other hand, the content structure can be *re-formatted* by an adaptive UI according to usual platform interaction patterns, e.g., let the user scroll through large amounts of information, present it in multiple subsequent steps, or as a hierarchical structure providing more details on request [EVP01a; EVP01b].

A high degree of abstraction can be achieved for user interactions by modelling user inputs in terms of *intended actions* for completing a particular task. An intermediate mapping layer can then transform actual inputs to the respective actions. For instance, a "back" action can be linked to a hardware button, displayed in a navigation bar on screen, bound to the right-swipe gesture (as recommended by the Wear OS guidelines [Goo]), or recognized by a spoken keyword in smart personal assistants. Another example is the usage of *default actions* to navigate through the app. Whereas in iOS, one possible action can be displayed in the top-right corner of the navigation bar, Wear OS uses so-called *ActionDrawers*[3] which propose one or more possible actions from the bottom edge of the screen.

Business apps are usually designed to support specific goal-oriented workflows which can be decomposed into individual tasks. This perspective of a user task model aligns with the desired high level of abstraction [SCK07]. Subsequently, two task-oriented notations are presented that embody the task-oriented approach. The *ConcurTaskTree (CTT)* notation consists of three main elements [Pat00]:

- the abstract *task* descriptions which together form the use case's functionality,
- temporal *operator*s defining the allowed sequences of executed tasks, representing an abstract notion of navigation actions within a use case, and
- the user *role*s (per task) that are allowed to interact with the system

CTTs have been refined to suit user interface development through decomposition over multiple levels [Pri05]. An exemplary model suited for interface generation is presented in Figure 14.3. The example shows an excerpt of a simple inventory management task in which the user (either a warehouse clerk or product manager) first selects an item from a list (with the possibility to filter the list content by title) and gets presented the item details (with data on title, description, pictures, price, product category, and quantity on stock; collapsed in Figure 14.3). Depending on the user's role, different modes of editing the inventory are possible before concluding or

[3] https://designguidelines.withgoogle.com/wearos/components/action-drawer.html

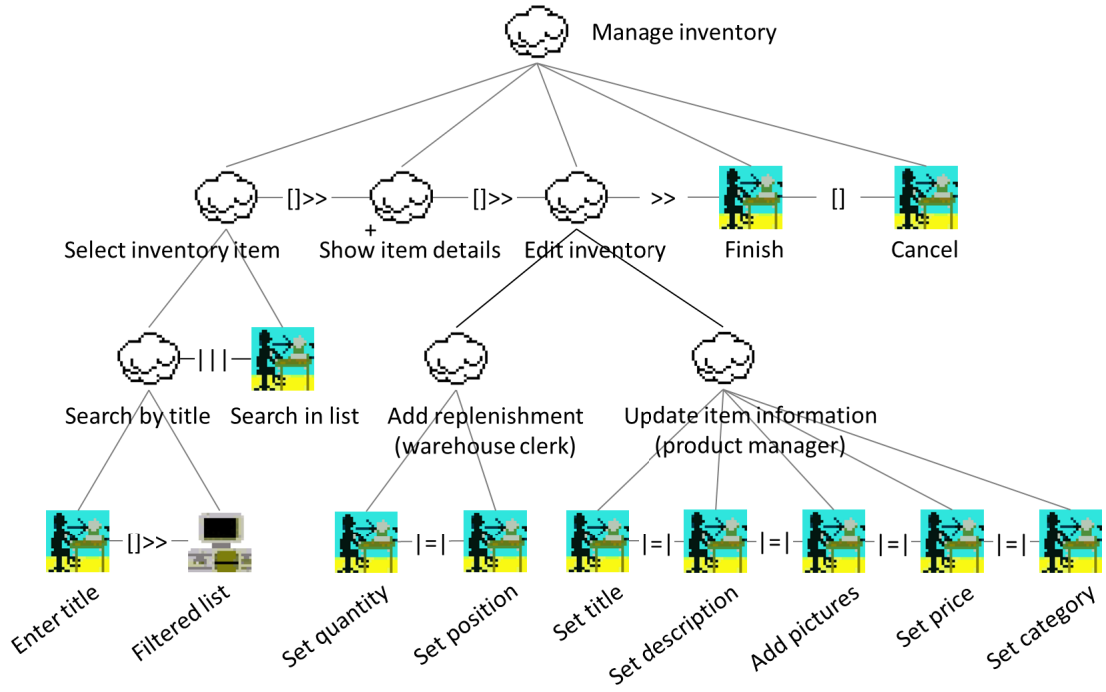


Figure 14.3: Sample CTT Model for an Inventory Management Use Case

aborting the process. Warehouse clerks can enter the quantity and position of newly arrived item replenishments. Product managers may instead update the item master data.

As can be seen, the detailed decomposition at operational level creates a tree with user tasks on multiple levels of abstraction which diminishes the overview about the high-level tasks and particular user interactions. In addition, still many aspects such as navigation, item groups, informative labels, or object structures are not explicitly contained in the notation and need to be provided in separate models (e.g., using UML class diagrams for data models) to enable a fully automated generation of user interfaces.

Previously, we have proposed the graphical Münster App Modeling Language (MAML) for specifying business apps based on five main design goals [RK18a]:

- *Automatic cross-platform app creation* by transforming a graphical model to fully functional source code for multiple platforms.
- *Domain expert focus* to allow non-technical stakeholders to create, alter, or communicate about an app using the actual models.
- *Data-driven process modelling* specifies the application domain but also sets a high level of abstraction by interpreting data manipulation as a process.
- *Modularisation* of activities in distinct use cases helps for maintenance, especially for domain experts.

- *Declarative description* of the complete app, including necessary specifications of data model, business logic, user interactions, and UI views.

Compared to CTTs, the same exemplary scenario is depicted in Figure 14.4 as MAML model. In MAML, this task is modelled in a *use case* as depicted in Figure 14.4. The model contains a sequence of activities, from a *start event* (labelled with (1) in Figure 14.4) towards one or several *end events* (2). In the beginning, a *data source* (3) specifies the data type of the manipulated objects and whether they are only saved locally on the device or managed by the remote backend system. Data can then be modified through a pre-defined set of (arrow-shaped) *interaction process elements* (4), for instance to *select/create/update/display/delete entities*, show *pop-up messages*, or access device functionalities such as the *camera* and starting a phone *call*. This describes the desired user actions on a high level and can be mapped to device-specific UI representations as described in Subsection 14.3.2. *Automated process elements* (5) represent invisible processing steps without user interaction, for example *including* other models for process reuse, or navigating through the object graph (*transform*). The navigation between connected process steps happens along the *process connectors* (6) which can be supplied with captions. To account for non-linear workflows, process flows can be branched out using *XOR* elements (7).

In contrast to CTTs, MAML models additionally contain the data objects which are displayed within a respective process step. *Attributes* (8) consist of a cardinality indicator, a name, and the respective data type. Besides pre-defined data types, custom types can be defined and further described through nested attributes. *Labels* (9) provide explanatory text, and *computed attributes* (10) are used to calculate derived values at runtime based on other attributes. In MAML, only those attributes are modelled which will be used in a particular process step, for instance for including it in the graphical user interface.

Two types of connectors exist for attaching data to process steps: Dotted arrows represent a *reading relationship* (11) whereas solid arrows signify a *modifying relationship* (12) with regard to the target element. Every connector which is connected to an interaction process element also specifies an order of appearance and a field description. For convenience, multiple connectors may point to the same UI element from different sources (given their data types match). Alternatively, to avoid confusing connections across larger models, UI elements may instead be duplicated to different positions in the model and will automatically be matched at runtime.

Finally, MAML supports a multi-role concept to define (13) and annotate (14) arbitrary role names to the respective interaction process elements because many business processes such as approval workflows involve different people or departments.

It can be observed that the domain-specific MAML notation allows for a more concise and clearly arranged representation of the app content, for example by offering different task types which encapsulate the respective semantics as opposed to the more general CTT models. At the same time, MAML models contain precise technical details on the resulting app, such that fully functional apps can be generated from the notation. Using the presented mapping approach, apps

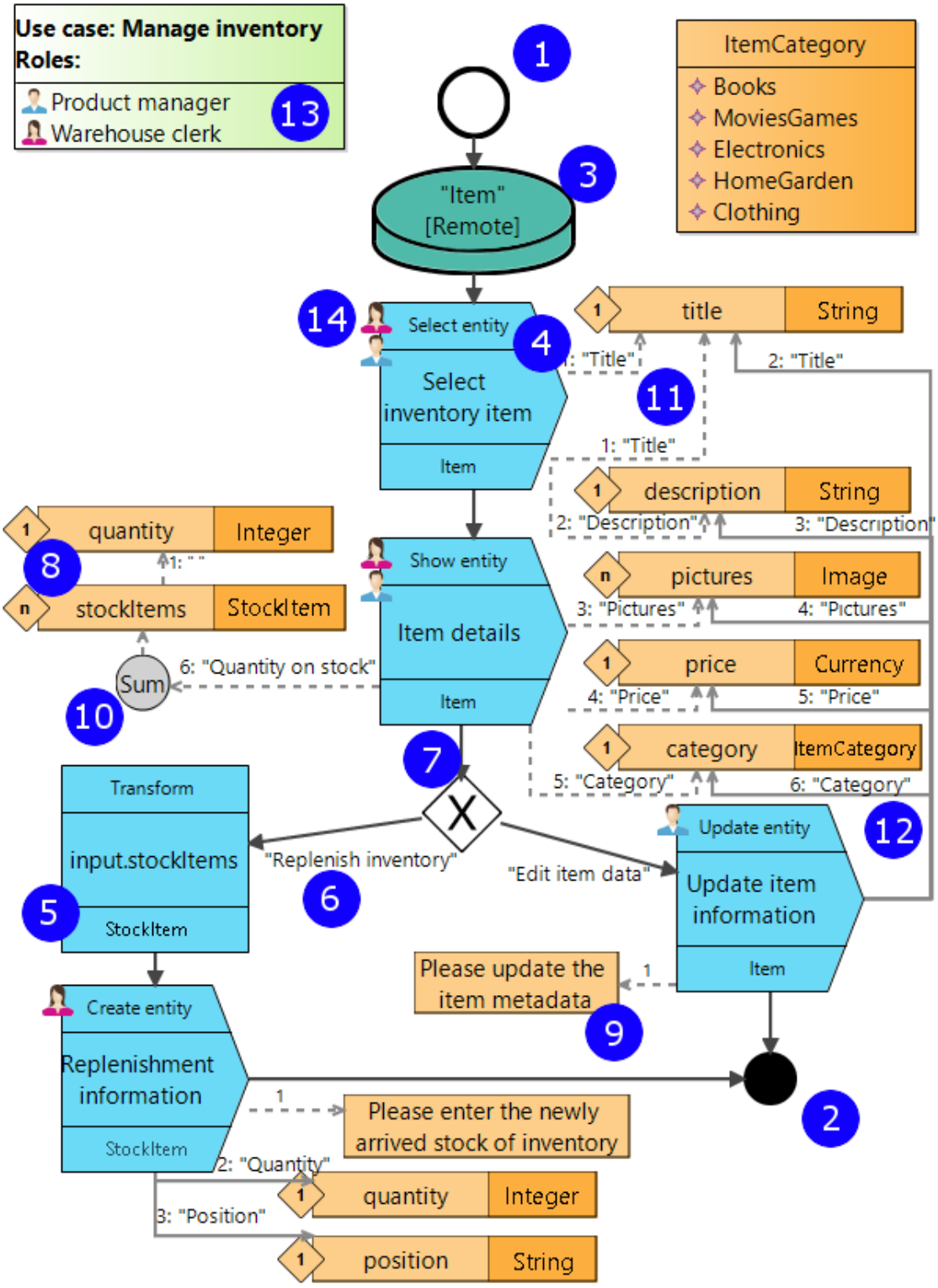


Figure 14.4: Sample MAML Model for an Inventory Management Use Case

can be flexibly generated from the same MAML models both for smartphones (Android and iOS) as well as Wear OS smartwatches.

14.4 Discussion

To put the presented approach in a broader context, we discuss two scenarios that underline the aforementioned issues and potential solution.

The first scenario in the *logistics* domain with an inventory management app was already depicted in Figure 14.3 and **??**. It represents a typical business process that can be performed by a single user when interacting with a centralised back-end system. However, it proves the potential for MDSD in generating apps for very different devices simultaneously from a common model: Firstly, a warehouse clerk might want to use smart glasses in order to have hands-free interaction with the inventory system when replenishing items. A product manager might instead use a smartphone to update erroneous information while inspecting shelves. Achieving business app development using model-driven techniques therefore seems particularly beneficial with regard to the flexibility it provides towards the future end user. This freedom to choose an appropriate device given the current usage context or personal preferences can not only propel technology acceptance but also allows for more experimentation with different modes of work.

As a second scenario, consider a *health app* which is integrated with a patient information system. Before making an appointment, patients can be asked to give more details about the health issue they are experiencing, for example through standardised questionnaires. This information is passed to the doctor to support his diagnosis. In order to follow-up on the medical treatment, sensors can regularly monitor vital parameters over a specified time interval and report the results back to the medical office such that doctor's assistants can take action when observing abnormal values.

This complex workflow involves multiple actors who need to share information over a longer period of time. In addition, different devices are combined according to their capabilities (e.g., wearable devices for sensing heart rate). However, tasks related to data manipulation might be limited on some devices, e.g., entering lots of data on a smartwatch is tedious. Using automated transformations of user interfaces can alleviate the problem only to a certain degree. Most probably, a co-existence of apps for multiple devices in parallel will be more useful than forcing all tasks to be performed on the same device. Again, the model-driven paradigm can provide significant benefits when offering a broad range of devices without linear increase of development efforts. However, to achieve the sketched multi-device usage, the potentially concurrent interaction between multiple devices requires a reliable and fast synchronisation of data among the respective devices. Different techniques such as Operational transformation (OT) and Conflict-Free Replicated Data Types (CRDT) have been proposed for mobile data synchronisation but further research is necessary to apply them to the context of business apps [EG89; Sha+11]. In both scenarios, the diversity of user interfaces is best tackled by transforming a high-level and platform-agnostic model that relies

on abstract *presentation units* which are later mapped to *concrete interaction objects*, i.e. suitable widget representations for platform-specific look-and-feel [EVP01b; VB93].

When putting the presented mapping into practice, the CTT representation as a general purpose notation for user tasks is not sufficient. Additional notations are required to specify the data model and possible interaction and navigation flows within an app. Other cross-platform approaches targeted to smartphones such as BiznessApps [Biz18] and Bubble [Bub18] provide graphical configurators for simplified app modelling, however, their screen-oriented approach falls short of covering wearable UIs. Similarly, user interaction standards such as the Interaction Flow Modeling Language (IFML) are implicitly tied to screen-based output [Obj15]. In contrast, domain-specific languages such as MAML can provide components on a high level of abstraction while at the same time integrating UI and interaction aspects. Apart from being useful for holistically modelling apps, the MAML framework also includes code generators that output fully functioning smartphone apps. To demonstrate the feasibility of deriving a suitable representation from the same input models, the framework was extended by implementing a generator for Android Wear 2.0 smartwatches (Figure 14.1 and **??** contain actual app screenshots).

The proposed high-level modelling approach for wearable app specifications also caters for the limitations of this paper regarding the future evolution of the field. Whereas differences in smartphone UIs have assimilated over time to a large degree (at least with regard the main input and output components), the chosen representations in Subsection 14.3.2 can only represent a small subset of the various UIs and interaction patterns which already exist or maybe emerge in the future for wearable devices. The novelty of this highly dynamic field of wearables will certainly entail several changes to typical platform characteristics regarding both hardware and software. Vendors continuously explore interfaces and interaction possibilities, heavily modifying their best practice guidelines when presenting new versions of their platform. At the same time, this paper only scratches the surface of cross-platform development complexities across multiple device classes. In-depth research needs to address each device class individually which still exhibits a high variability of devices capabilities – future consolidation across vendors is possible but not foreseeable anytime soon.

14.5 Conclusion and Outlook

In this paper, we have investigated the challenges and potential solutions for applying model-driven techniques to the development of business apps for novel app-enabled devices. In particular, user interface related challenges currently limit the extension of established cross-platform techniques to these new classes of heterogeneous devices. We exemplify how the typical building blocks of business apps might be generically mapped to smartphones, tablets, smartwatches, and smart glasses.

In order to incorporate these in a model-driven approach, task-oriented modelling – without actually specifying the concrete user interface – allows for a suitably high level of abstraction. We

deem this approach promising to bridge the heterogeneity of devices and enable fast development of apps that are flexibly targeted to a broader range of devices through adaptive layouts as well as device class specific reformatting of contents. Although both the MAML notation and CTT utilise a task-oriented approach, we argue that domain-specific languages such as MAML might be more suitable to modelling user interfaces than general purpose modelling notations due to the alignment with the target domain, thus condensing domain concepts more clearly to a high level of abstraction.

However, the current situation leaves room for further research in different directions such as observing emerging commonalities in user interfaces of the individual device classes that are subsumed by the umbrella term "wearables", and technical hurdles to synchronise content and application state in the dynamic interplay of mobile devices. Furthermore, the implementation of the presented concepts in an actual model-driven framework constitutes ongoing work of the authors by extending the MAML cross-platform framework with support for further novel device classes using the presented mapping approach.

References

[AIC17] D. Alulema, L. Iribarne, and J. Criado. "A DSL for the Development of Heterogeneous Applications". In: *FiCloudW*. 2017, pp. 251–257.

[Biz18] Bizness Apps. *Mobile App Maker – Bizness Apps*. 2018. URL: http://biznessapps.com/.

[Bub18] Bubble Group. *Bubble - Visual Programming*. 2018. URL: https://bubble.is/.

[EG89] C. A. Ellis and S. J. Gibbs. "Concurrency Control in Groupware Systems". In: *SIGMOD*. Portland, Oregon, USA: ACM, 1989, pp. 399–407.

[El-+14] W. S. El-Kassas et al. "ICPMD: Integrated cross-platform mobile development solution". In: *ICCES* (2014).

[ENM15] Andrey Esakia, Shuo Niu, and D. Scott McCrickard. "Augmenting Undergraduate Computer Science Education With Programmable Smartwatches". In: *SIGCSE*. Ed. by Adrienne Decker et al. ACM, 2015, pp. 66–71.

[EVP01a] J. Eisenstein, J. Vanderdonckt, and A. Puerta. "Applying model-based techniques to the development of UIs for mobile computers". In: *IUI* (2001).

[EVP01b] Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. "Applying Model-based Techniques to the Development of UIs for Mobile Computers". In: *Proceedings of the 6th International Conference on Intelligent User Interfaces*. Santa Fe, New Mexico, USA: ACM, 2001, pp. 69–76.

[God+96] D. Goddeau et al. "A form-based dialogue manager for spoken language applications". In: *ICSLP*. 1996, pp. 701–704.

[Goo] Google LLC. *Google Developers*. URL: https://developers.google.com/.

[JM15] Chakajkla Jesdabodi and Walid Maalej. "Understanding Usage States on Mobile Devices". In: *Int. Joint Conf. on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 1221–1225.

[KK16] István Koren and Ralf Klamma. "The Direwolf Inside You: End User Development for Heterogeneous Web of Things Appliances". In: *ICWE*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Springer, 2016, pp. 484–491.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14.

[NJE17] Timothy Neate, Matt Jones, and Michael Evans. "Cross-device Media: A Review of Second Screening and Multi-device Television". In: *Personal Ubiquitous Comput* 21.2 (2017), pp. 391–405.

[Obj15] Object Management Group. *Interaction Flow Modeling Language*. 2015. URL: http://www.omg.org/spec/IFML/1.0.

[Pat00] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. 2000.

[Pri05] C. Pribeanu. "An approach to task modeling for user interface design". In: *Proceedings of the 3rd World Enformatika Conference*. Vol. 5. 2005, pp. 5–8.

[RK18a] Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *Computer Languages, Systems & Structures* 53 (2018), pp. 43–58.

[RK18b] Christoph Rieger and Herbert Kuchen. "Towards Model-Driven Business Apps for Wearables". In: *Mobile Web and Intelligent Information Systems*. Ed. by Muhammad Younas et al. Cham: Springer International Publishing, 2018, pp. 3–17. DOI: 10.1007/978-3-319-97163-6_1.

[RM17] Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-Enabled Devices". In: *WEBIST*. 2017, pp. 332–339.

[SB16] K. Singh and J. Buford. "Developing WebRTC-based team apps with a cross-platform mobile framework". In: *Consumer Communications and Networking Conference* (2016).

[SCK07] Daniel Sinnig, Patrice Chalin, and Ferhat Khendek. "Common Semantics for Use Cases and Task Models". In: *Integrated Formal Methods*. Ed. by Jim Davies and Jeremy Gibbons. Vol. 4591. LNCS. Springer, 2007, pp. 579–598.

[Sha+11] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *SSS*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. 2011, pp. 386–400.

[VB93] Jean M. Vanderdonckt and François Bodart. “Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection”. In: *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems*. Amsterdam, The Netherlands: ACM, 1993, pp. 424–429.

[vF17] Rob van der Meulen and Amy Forni. *Gartner Says Worldwide Wearable Device Sales to Grow 17 Percent in 2017*. 2017. URL: https://www.gartner.com/newsroom/id/3790965.

15

A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle

Table 15.1: Fact sheet for publication P9

Title	A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle
Authors	Christoph Rieger[1] Tim A. Majchrzak[2] [1] *ERCIS, University of Münster, Münster, Germany* [2] *ERCIS, University of Agder, Kristiansand, Norway*
Publication Date	2018
Conference	Web Information Systems and Technologies
Copyright	Springer International Publishing
Full Citation	Christoph Rieger and Tim A. Majchrzak. "A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle". In: *Web Information Systems and Technologies*. Ed. by Tim A. Majchrzak, Paolo Traverso, Karl-Heinz Krempels, and Valérie Monfort. Cham: Springer International Publishing, 2018, pp. 202–220. DOI: 10.1007/978-3-319-93527-0_10

A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle

Christoph Rieger Tim A. Majchrzak

Keywords: App, Mobile App, Taxonomy, Categorization, Smart devices, Wearable, Smartphone, Tablet

Abstract: While the term application is known for a long time, what we now refer to as mobile apps has facilitated task-oriented, interoperable software. The term was initially only used for smartphones and tablets, but desktop software now is also referred to as apps. More important than the wording, however, is the trend towards app-enablement of many further kinds of devices such as smart TVs and wearables. App-enabled devices usually share some characteristics and developing apps is often similar. However, many complexities must be mastered: Device fragmentation and cross-platform app development already are challenging when only considering smartphones. When trying to grasp the field as a whole, app-enabled devices appear as a jungle: it becomes increasingly hard to get an overview. Devices might not be easy to categorize let alone to compare. Investigating similarities and differences is not straightforward, as the outer appearance might be deceiving, and technological peculiarities are often complex in nature. This article aims at mastering the jungle. For this purpose, we propose a taxonomy for app-enabled devices. It provides clear terms and facilitates precision when discussing devices. Besides presenting the taxonomy and the rationale behind it, this article invites for discussion.

15.1 Introduction

The continuous growth of the mobile device market [Sta16] and the recent emergence of devices such as smart watches [Chu+16] and connected vehicles [CM16] has attracted much attention from academia and industry. In the past decade, particularly the app ecosystem facilitated a trend towards task-oriented, interoperable software, arguably started with the advent of Apple's iPhone in 2007 [App07] and the App Store in 2008 [App08]. For *traditional* mobile devices (i.e. smartphones and tablets), the competition has yielded two major platforms (Android and iOS). However, whether these two will prevail can hardly be estimated, yet. Moreover, developing for such devices in a unified way is still not possible in all cases and with ease (cf. also [BMG17]). Several approaches for cross-platform development have been proposed to avoid the costly re-development of the same app for different platforms (cf., e.g., [HHM13; El-+15]).

Technological development has continued in the meantime and many new device types have emerged. Most of them fall under the umbrella term *mobile devices* and are concerned by the research field of *mobile computing*. Typically, they are more-or-less *app-enabled*. While app enablement is no fixed or even defined term (to the best of our knowledge), it can be understood as follows:

> An app-enabled device provides hardware that allows it to be used for multiple (typically many) purposes and in changing contexts – possibly even unforeseen

> by the device manufacturer – while the actual versatility of the device is achieved through means of extensible software that comes in small, interchangeable pieces which are usually provided by third parties.

Thus, it typically are apps that make such devices particularly useful and that extend the possibilities they offer. However, mobile devices that follow our rough definition differ greatly in intended use, capabilities, input possibilities, computational power, and versatility, to name just a few aspects. In early visions of a world connected by ubiquitous mobile devices, these were only thought of as tabs, pads, and boards [Wei91]. So-called "smart devices" such as smart watches and smart TVs are most prominent in the realm of consumer devices and exhibit double-digit sales growths over the past years [Sta17a; Sta17b], but plenty of possibilities exist with regard to the physical embodiment of virtual assistants. Furthermore, hardware in professionally contexts can be surprisingly similar to consumer-hardware; apps can make them seem even more akin. Lines towards sensor-driven devices for the Internet of Things (IoT) are often blurred and it is not always clear how to properly categorize a device [Iba+17]. This makes it hard to discuss, or, actually, to even correctly name them. The resulting blurriness makes it hard to delimit research and practical work. Much worse, when speaking and writing about mobile devices, the level of precision is often not as high as it is when well-known concepts are discussed. While this is normal for emerging fields, it is particularly noticeable for work on mobile computing. To our observation, there are only slow improvements.

We believe that more precision in speaking and writing will eventually also be beneficial for research on mobile devices and their app-enablement. These devices provide a plethora of new opportunities for intelligent and context-adaptive software. At the same time, they pose technical challenges regarding the development for new platforms and regarding heterogeneous hardware features. Interestingly, these challenges can be quite similar despite seemingly very different devices, as they can be completely different despite originating from the same kind of device. Moreover, app-enablement does not necessarily bring compatibility and portability. Naturally, running the same app on a variety of devices is normally desirable. If we still rely on cross-platform approaches and search for a development unifier merely for smartphones and tablets [BMG17], developing for heterogeneous mobile devices is an endeavour far greater in complexity. It would probably be ideal to reach something like a progression in functionality: the same app would function on many devices but respectively provide the highest level of functionality achievable on the given hardware and with the available other software. It must be doubted, however, that such an ideal can be reached as long as we do not even properly know *what* we are talking about.

While a plethora of case studies and contributions for individual device types – mainly focused an smartphones and tablets – can be found in the scientific literature (e.g., [HMK13; JJ14; CMK14; Bus+15; Dag+16]), a comprehensive study of the general field of app-enabled devices is missing. With our WEBIST position paper [RM17], we set out to close this gap by contributing a taxonomy

for app-enabled consumer devices. This contribution got favourable comments, encouraging us to provide an extension of our work with this article. The taxonomy aims at

- helping authors to clearly express what kind of device(s) they refer to,
- providing researchers and practitioners with more discriminatory power when referring to topics from modern mobile computing, and
- giving the general public a more straightforward understanding of similarities and differences between devices, both technically and tangibly.

Similar as in the prior paper, we have put much effort into literature work (cf. the next section), although the useful literature remains scarce. While the taxonomy has only been slightly updated to reflect the latest developments, we delve deeper into the theoretic dimension and also extend our discussion. Therefore, the work presented in this paper keeps a research-in-progress flavour, since it is impossible to suggest that our taxonomy is in its final state. However, it should be considered sufficiently stable for practical application. Any follow-up work from now on will honour this by either providing downward compatibility and (or alternatively) by explicating changes. We believe that more work will continue to be required; while we of course hope for this article to become a state-of-the reference, it should also stimulate further work. The mid-term goal remains to be a *de-facto* standard.

The remainder of this article is structured as follows. Section 15.2 takes an updated look at relevant literature in the narrow sense; related work to specific aspects is referenced throughout the paper. Then, our proposal for a taxonomy is presented in Section 15.3. Section 15.4 discusses the taxonomy with regard to its current and future applicability. Finally, we conclude and give an outlook in Section 15.5.

15.2 Related Work

If you consider the topic of our article broadly, a plethora of related work exists. Looking at it in more detail, hardly any closely-related approaches can be cited. This is not really surprising: all papers that deal with apps and app-enabled devices must (at least implicitly) explain *what* they actually deal with. However, no systematic work exists that defines kinds of devices, modes of app-enablement, notions of mobility of devices, and so on.

Particularly since Apple's iPhone founded the *smartphone* device class, which soon saw many devices follow, many papers have been published on the *modern* notion of mobile computing, centring around devices that are propelled by apps. However, even overview papers typically focus on *one* category of devices. For example, [JM15] classify apps by usage states but limit themselves to smartphones. Moreover, the scientific literature so far has only rudimentarily captured the latest developments in device development. [Cha+16], for instance, provide an overview of smart watch app markets with focus on the type of apps as well as privacy risks through third party trackers.

To make sure that we do not miss an existing taxonomy (or similar work), we conducted an extensive literature search. We focus on work from 2012 or later, where the first broader range of smart watches such as the Pebble had already been presented. Together with the increasing variety in devices, new operating systems have appeared since then. Examples are Android Wear and watchOS, which focus on wearable devices [Goo16; App16] as well as webOS and Tizen, which address a wider range of smart devices [LG 16; The16]. Additionally, also the app ecosystems have matured, with HTML5 gaining momentum and possible technological unifiers such as progressive web apps (PWAs) [BMG17] emerging.

In our search, we deliberately excluded the keywords *application* and *system*. The first yielded many results that were not applicable since the term was mostly used to mean *utilization* of something. The latter had originally been used to describe e.g. cyber-physical systems but now proved to be too generic. Also, the *medical* area was excluded as these papers focus on apps for therapeutic purposes and do not contribute to the question of app-enabled devices. We thus used the following search string in the Scopus database:

```
TITLE-ABS-KEY(
   (app-enabled OR app OR app-based)
AND
   (mobile OR smart OR intelligent OR portable)
AND
   (device OR vehicle OR "cyber-physical system" OR CPS OR gadget)
AND
   (classification OR categorization OR overview OR comparison OR review →
    OR survey OR framework OR model OR landscape OR "status quo" →
    OR taxonomy)
)
AND PUBYEAR AFT 2011
AND ( EXCLUDE ( SUBJAREA , "MEDI" ))
```

A search on 01-08-2017 yielded 1,268 results. Of these, not a single paper provided an approach for classification, let alone a complete taxonomy. To complicate matters, some papers mention that there are other *smart* devices than smartphones and tablets but do not go into detail. Only four papers went beyond a perspective on "classical" mobile devices: Some authors focus on specific combinations of devices, including Neate et al. [NJE17] who analyse the use case of second screening that combines smart TVs with additional mobile devices and Singh and Buford [SB16] who describe cross-device team communication apps for desktop, smartphones and wearables. Regarding more generalized approaches, Queirós et al. [QPM17] focus on context-aware apps also suitable for novel mobile devices using the example of an automotive app. Finally, Koren and Klamma [KK16] considered the integration of heterogeneous Web of Things device types by adopting a middleware approach.

In summary, the result set reveals no closely related work to which we can limit ourselves. However, we can draw from a myriad of sources that tackle *some* aspects that are relevant for a taxonomy of app-enabled devices. This finding aligns with the motivation for our paper. Obviously, other authors struggled with putting different device categories into context because no proper framing exists.

Despite not necessarily focussing on multiple device categories, work on *cross-platform app development* is conceptually related. Usually, cross-platform development exclusively targets traditional mobile devices such as smartphones and tablets, e.g. "the diversity in smart-devices (i.e. smartphones and tablets) and in their hardware features; such as screen-resolution, processing power, etc." [HEE13]. However, considering the differences in platforms, versions, and also at least partly in the hardware is similar to considering a different type of device. In fact, the difference in screen size between some wearables (such as some smart watches) and smartphones with small screens is less profound than between the same smartphones and tablets. Therefore, comparisons that target cross-platform app development have paved the way towards this article. This particularly applies to such works that include an in-depth discussion of criteria, such as by [HHM13], [SK13], [Dal+13], and [RM16].

A part of the difficulty with related work is the term *app-enabled* (or *app-enablement*) by itself. While it is often said that devices are enabled by apps, or that apps facilitate their functionality, it is usually not explained *what* this exactly means. But in the simplest devices that make use of computer hardware, software plays an important role; in consequence, merely being capable of running software that fulfils more than basic functionality is not enough to describe the term.

The typical usage that we also follow is to denote an app-enabled device as one that by its hardware and foundational software (such as the operation system or *platform*) alone provides far less versatility than it is able to offer in combination with additional applications. Such apps are not (all) pre-installed and predominantly provided by third party developers unrelated to the hardware vendor or platform manufacturer; moreover, the possibilities provided by apps typically increase over time *after* a device has been introduced. In addition, apps may expose use cases not originally intended or even imagined. While this still is no profound definition, it provides a demarcation for the time being. In particular, it rules out pure Internet-of-Things devices as well as computational equipment that only is occasionally firmware-updated or that is not built for regular interaction with human users.

15.3 Taxonomy of App-Enabled Devices

In the following, we describe the preconditions of a taxonomy before describing how a device categorization can be tailored. We deem three dimensions to be viable as the static structure for classification. Eventually, we propose a categorization that captures the status quo. It positions current and foreseeable future device classes according to this matrix.

15.3.1 Basic Considerations

Categorizing app-enabled devices is difficult: there is a wide variety of possible hardware features *across* all types of devices, which even further increases. For example, in the past fingerprint scanners were restricted to few notebooks but today also appear on smartphones because of simplified user authentication and changing security requirements with regard to the device purpose (consumer vs. commercial). If classes are set – such as the widely acknowledged distinction between smartphones and tablets – there still is a heterogeneity of device capabilities *within* each class. For instance, the first smartwatches offered only a few sensors. Current devices have many more sensors, and their characteristics can differ significantly depending on the target sector (such as low-end vs. high-end).

Any simple solution is prone to not sufficiently discriminate. For example, processing power does not differ a lot between smartphones and tablets anymore, and microphones are no distinguishing feature for voice-controlled devices. The fast-paced technological progress manifests as a constant stream of new devices, partly rendering previous devices obsolete. Moreover, device types converge, illustrated e.g. by the *phablet* phenomenon (devices that fall in between smartphones and tablets).

Mobility in the strict sense even is no exclusive feature; smart TVs for example are not really mobile. Cars with smart entertainment systems or even self-driving features might be app-enabled, but it can be disputed whether the whole car is the *device* and thereby the device is actually mobile by itself. As a result, a taxonomy of app-enabled devices mandates a more open categorization along several dimensions, allowing for partial overlaps and future additions. In the following, we present steps towards such a taxonomy.

15.3.2 Dimensions of the Taxonomy

We position app-enabled devices with regard to the three dimensions *media richness of inputs*, *media richness of outputs*, and the *degree of mobility*. Instead of enumerating concrete technologies that are available today or *may be* introduced in the future, each dimension should rather be regarded as continuously increasing intensity and variability of the particular capability, with several exemplary cornerstones depicted in the following. This approach not only provides the highest degree of objectivity but also should keep the taxonomy flexible enough to capture future developments without actually changing the dimensions.

Media richness of inputs describes the characteristic user input interface for the respective device class. Thereby, it captures how human users can interact with a device. Additional machine-to-machine communication through the same or distinctive interfaces is *not* considered.

None refers to fully automated data input through sensors.[1]

[1]Strictly, most if not all input is done via sensors, but *none* at this point denotes no manual activity by a user.

Pass-through represents the indirect manipulation through data exchange with an external device (which in turn might originate from user input) whose purpose is not solely to provide the user interface for the main device.[2]

Buttons including switches and dials, are (physically) located at the device and provide rather limited input capabilities.

Remote controls including also joysticks and gamepads, refer to dedicated devices that are tethered or wirelessly connected to the app-enabled device. Technically, they merely make use of buttons, switches, dials etc. but provide a richer experience due to being decoupled from the device.

Keyboards are also dedicated devices to control the target devices, but with more flexible input capabilities due to a variety of keys. Input still is discrete.

Pointing devices refer to all dedicated devices to freely navigate and manipulate the (mostly graphical) user interface, for example mouse, stylus, and graphic tablet. While these devices technically still provide discrete input, the perception of input is continuous.

Touch adds advanced input capabilities on the device itself, allowing for more complex interactions such as swipe and multi-touch gestures. Strictly speaking, within this category simple touch events and several forms of increasingly complex multi-touch gestures can be subdivided.

Voice -based devices are not bound to tangible input surfaces but can be controlled without haptic contact.

Gestures allow for a hands-free user interaction with the device, for example using gloves or motion sensing. Technologically, different solutions are possible, e.g. based on gyroscopes, cameras, and lidars [LKN17; Bho13].

Neural interfaces can be expected to become the richest form of user inputs by directly tapping into the brain or nervous system of the human operator.[3]

As the second dimension, *media richness of outputs* describes the main output mechanisms for the respective device class. Similarly to the input, human users are concerned as the receivers; possible machine-to-machine communication is not relevant for this dimension.

None refers to no user-oriented communication by the device itself. This applies to cyber-physical actuators with direct manipulation of real-world objects (e.g., switching on light).

[2]For example, an autonomous device with a companion smartphone app for remote handling can be subsumed under this category.

[3]Since the possibilities of neural interfaces are yet very limited and any work so far is experimental, future developments *might* mandate splitting up this category into different kinds of neural interfaces.

Pass-through includes mechanisms that in general or in some situations do not produce human-directed output of their own but pass it through to a connected managing device (e.g., a smartphone) which retrieves information and handles user output.

Screen output is the prevalent form of user communication found in app-enabled devices. Although a clear subdivision is not possible, several classes are typically observed, ranging from tiny screen displays (<3") to small screens such as for smartphones (<6"), medium screens for handheld devices (<11"), large screens (≤ 20"), and usually permanently installed huge screens >20".

Projection refers to the first type of disembodied device output to a device-external surface without physical contact.

Voice -based output extends the disembodiment with auditive output to communicate with the user without physical contact.

Augmented reality includes virtual reality applications and hologram representations, further increases the richness of device outputs by modifying or fully replacing the perceived reality around the user.

Neural interfaces connect directly to the user in order to a achieve a tightly coupled human-computer interaction.[4]

Finally, the combination of input and output characteristics ignores different application areas of the respective device class. For example, intelligent switches and drones for aerial photography can both be remotely controlled and have no direct output, but can hardly be grouped as being in the same device class. Whereas several studies deal with usage characteristic particular devices such as smartphones (e.g., [Hin+17]), to the best of our knowledge no closely related work exists on context-dependent device usage across different mobile devices. Therefore, the *degree of mobility* describes the usage characteristics as the third dimension on a high level. With regard to trends such as ubiquitous computing [Gub+13], this dimension also reflects the pervasiveness and integration of mobile devices in everyday activities – from on-demand usage of stationary devices to always connected autonomous assistants.

Stationary devices are permanently installed and have no mobile characteristics during use.

Moveable devices can be carried to the place of use. This includes an "on-the-go" utilization, such as a smartphone being used while walking.

Wearable devices are designed for a more extensive usage and availability through the physical contact with the user. In contrast to "mobile", transporting the device is implicit and often hands-free.

[4] Similarly to the considerations for the input, it will need to be seen whether neural interfaces for output require some form of subdivision. Technology so far is in an early experimental state.

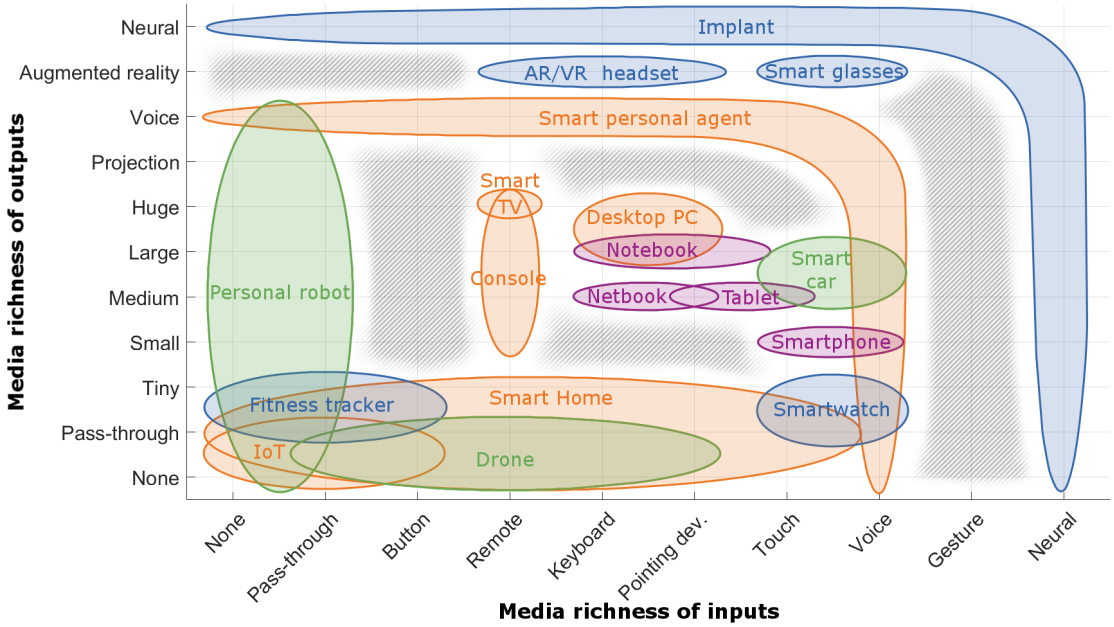


Figure 15.1: Matrix of Input and Output Dimensions (adapted from [RM17])

Self-moving devices provide the capability to move themselves (directly or indirectly controlled by the user). Ultimately, autonomous devices represent the richest form of mobility for app-enabled devices.

15.3.3 Categorizing the Device Landscape

The proposed dimensions allow for an initial categorization of the device landscape. Figures 15.1 to 15.3 (pages 364 to 366) visualize the three-dimensional categorization of different device classes using three two-dimensional projections for better readability. Also, Table 15.2 summarizes the device classes discussed in this paper.

As depicted in Figure 15.1, many devices classes can be assigned to distinct positions in the two-dimensional space of input/ output media richness. However, it should be noted that the ellipses represent (current) major interaction mechanisms within the device classes. For example, *smartphones* also have a few physical buttons but are mainly operated by touch input. Individual devices may also deviate from the presented position, for instance specialized or experimental devices that do not (yet?) constitute a distinct class of devices. Additionally, devices might be extended. For example, through special plugs computer mouses can typically be attached to smartphones. Since this is normally meant for debugging purposes, "pointing devices" would not normally be considered an input for smartphones. Similarly, so-called pico projectors allow image projection from smartphones and tablets. They are no typical mean for output, although this might change in the future.

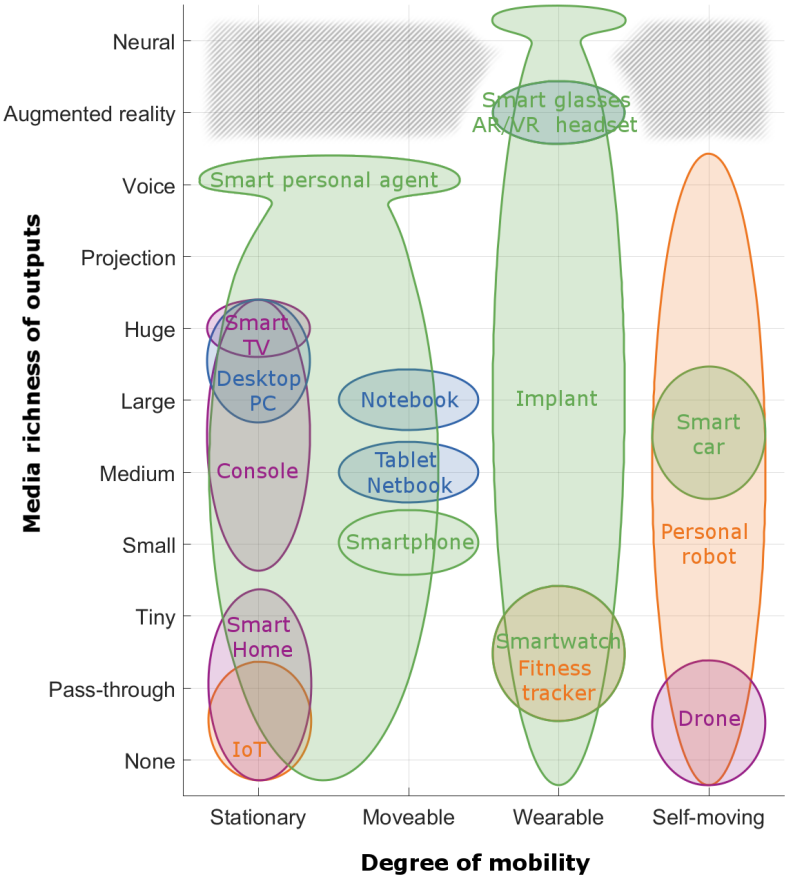


Figure 15.2: Matrix of Output and Mobility Dimensions (adapted from [RM17])

Not all devices falling into a device class must necessarily implement all possibilities of that class. Therefore, ellipses are a well-suited representation as opposed to, e.g., the maximum value for the respective devices. A good approximation would be to consider at least 80% of all devices to match a category, with the lowest and the highest decile being outliers. For example, *convertibles* as hybrid devices between keyboard-based notebooks and touch-oriented tablets are not considered as they still represent a small minority in both categories.

The chosen level of abstraction implies that the taxonomy *dimensions* are intended to be rather static. Instead of chasing the actual technological development to reflect the latest emergence of devices, only seldom and slow changes are necessary to keep them up to date. Nevertheless, the categorization of *classes* is more dynamic and will need to be regularly checked for continued relevance. Moreover, classes might need to be split or at least be adapted regarding their placement on the dimensions' continuum when new possibilities arise. Thus, we explain some noteworthy classes exemplarily and rely on the general understanding of the well-known classes (such as smartphones).

Figure 15.1 reveals differences in the specificity (i.e., represented size) of the device classes. Some of them fill specific spots in the diagram, either due to technical restrictions (*smart TVs*

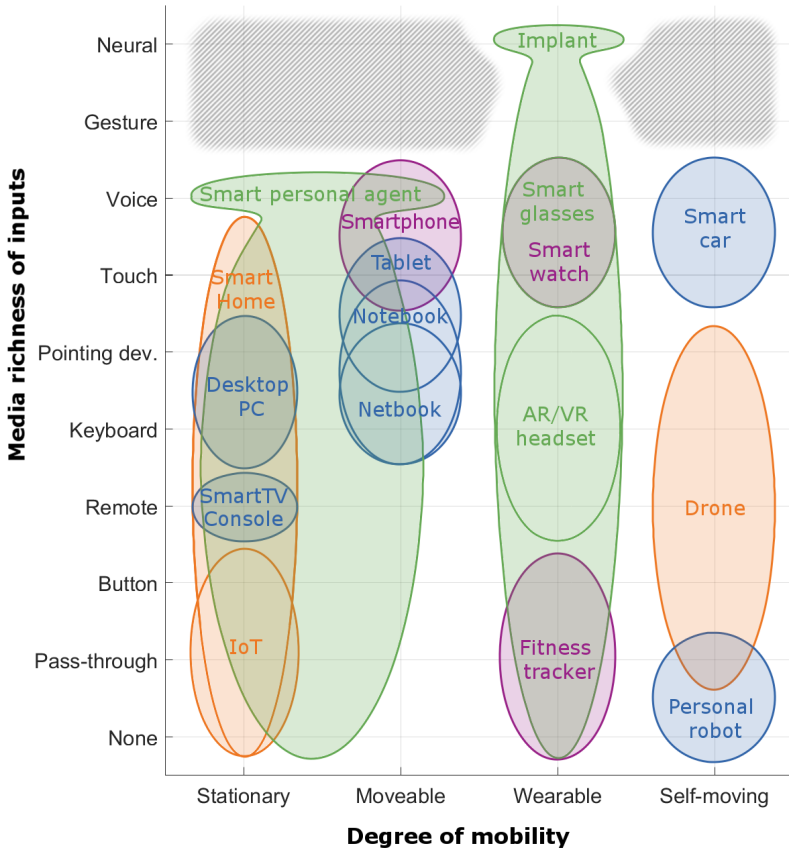


Figure 15.3: Matrix of Input and Mobility Dimensions (adapted from [RM17])

evolved from traditional remote-controlled TVs with large screens) or special purposes (*smart glasses* enable hands-free interaction and visualization). Less specific device classes exist for two reasons. On the one hand, terms such as *smart home* comprise every technology that relates to a specific domain, subsuming very heterogeneous devices – thereby such a class represents an excellent high-level overview yet a poor low-level discriminating power. On the other hand, *underspecified* device classes such as *implants* and *smart personal agents* are presented *as they are* due to their novelty; there are few devices on the market and a high level of uncertainty must be ascertained regarding future hardware characteristics and interaction patterns.

Differences in the device classes can also be explained with regard to *media richness theory* (MRT). MRT describes a corridor of effective communication with matching levels of message ambiguity and media richness [DLT87]. When applying this idea to the input and output characteristics of app-enabled devices, similar observations can be made. For example, *IoT* devices have only rudimentary possibilities for direct user input but also give not much feedback in return. Notebooks allow for medium levels of input richness through keyboard and mouse input, with large screens as more flexible output capabilities. Furthermore, *smart glasses* directly embed their output into the real world by projection. Consequently, their voice-based input is equally rich in order to handle complex user interactions.

Table 15.2: App-enabled device classes and their position in the continuum

Device class	Input richness	Output richness	Degree of mobility
AR/VR headset	Remote - Pointing	Augmented reality	Wearable
Console	Remote	Small - Huge	Stationary
Desktop PC	Keyboard - Pointing	Large - Huge	Stationary
Drone	Pass-through - Pointing	None - Pass-through	Self-moving
Fitness trackers	None - Button	Pass-through - Tiny	Wearable
Implant	Neural	*any*	Wearable
	any	Neural	Wearable
IoT	None - Button	None - Pass-through	Stationary
Netbook	Keyboard - Pointing	Medium	Moveable
Notebook	Keyboard - Touch	Large	Moveable
Personal robot	None - Pass-through	None - Voice	Self-moving
Smart car	Touch - Voice	Medium - Large	Self-moving
Smart glasses	Touch - Voice	Augmented reality	Wearable
Smart home	None - Touch	None - Tiny	Stationary
Smart personal	Voice	None - Voice	Stationary-Moveable
agent	None - Voice	Voice	Stationary-Moveable
Smartphone	Touch - Voice	Small	Moveable
Smart TV	Remote	Huge	Stationary
Smartwatch	Touch - Voice	Pass-through - Tiny	Wearable
Tablet	Pointing - Touch	Medium	Moveable

Figure 15.2 depicts the combination of output media richness and mobility. Unsurprisingly, a general tendency towards large screen output for stationary devices can be observed. With increasing mobility, output capabilities develop in two directions. On the one hand, screen sizes tend to diminish, from small screens on *smartphones* to very limited *fitness tracker* screens and screen-less *drones*. On the other hand, output capabilities become richer and overcome traditional screen-based approaches due to recent technological developments enabling intangible outputs, for instance *augmented / virtual reality (AR/VR) headsets*. It can also be observed that device classes with a high degree of mobility are more variable and occupy larger spaces of the continuum. This is potentially caused by a fragmentation into various domains of application, or their novelty of appearance with insufficient time to establish wide-spread interaction patterns. Especially autonomously moving devices such as *smart cars* and *personal robots*, are driven by the increased availability of sensor technology and not restricted to particular output capabilities.

Finally, Figure 15.3 visualizes the relationship between input media richness and mobility. Usually, an increasing degree of mobility entails less physical input mechanisms with dedicated buttons and keys. This might be attributed to practicability reasons, for example using voice commands is easier for wearable *smart glasses* than requiring dedicated input devices. In addition, smarter devices are usually more complex with regard to their output, and equally sophisticated input capabilities are necessary to match this level as explained by media richness theory. *Consoles*,

for instance, provide basic navigation functionalities. *Desktop personal computers* and *notebooks* can be equipped with intelligent software such that keyboard and mouse are helpful means for interaction, and *smart personal agents* integrate advanced interpretation mechanisms that allow for voice-based communication in everyday situations.

MRT also partly explains why there are areas in the continuum with no assigned device class. Rich forms of user input such as gestures overcomplicate interactions for devices that have just small screens and therefore are typically equipped with limited sensing and processing resources. On the other extreme, devices with barely a few buttons do not provide sufficiently flexible input capabilities to manipulate large screens (such as several fingers multi-touch on a small smart watch). Of course, empty spaces in the taxonomy might also be caused by a lack of technological progress or use cases so far. Thus, they might actually be filled by future devices, or existing classes might "stretch" into these areas. For example, voice interfaces just recently emerged as mainstream technology in various devices from smartphones to smart home applications but augmented reality devices are still an active field of research. In general, with the evolution and differentiation of input media, existing device classes might extend towards further areas or even converge. For example, consider convertibles, such as the Lenovo Yoga Book, which represent hybrid devices between keyboard-based *netbooks* and touch-optimized *tablets* utilizing docking or folding mechanisms. Also, the evolution of one device class might render another obsolete; this can currently be observed with smartwatches cannibalizing the market for fitness trackers with more advanced input and output capabilities.

15.4 Discussion

The field of modern mobile computing does not show signs of less rapid progress. It, thus, is likely that amendments will need to be made. Additionally, we will need to keep updating the taxonomy once it has been acknowledged by the scientific community. Moreover, a taxonomy should be appealing for the use by practitioners, particularly in a field where scientific research and technological progress go hand-in-hand. Therefore, this section presents ideas for discussion that go beyond the narrower focus of Section 15.3.

15.4.1 Alternative Categorization Schemes

Devices can be categorized according to other device features. Not all are compatible with our taxonomy, nevertheless we deem several of them noteworthy.

Simple schemes such as a categorization by hardware feature (e.g., camera resolution, raw computing power, touch screen availability) or usage (e.g., business, entertainment, sports, or communication) fail to provide clear criteria for a taxonomy. While they may even pose discriminatory power, they do not necessarily help with forming adequate classes of devices. In particular, a fast adaptation and convergence of available technologies could be observed in the past years.

Using simple hardware features for categorization would thus be prone to quickly becoming obsolete. To give some examples: so-called *phablets* blur the lines between smartphones and tablets; cameras with resolutions a few years ago only imaginable in professional photography equipment now are routinely built into many mobile devices; and gyroscope sensors have found wide-spread adoption in a variety much mobile hardware for a variety of purposes.

Matrix-based categorizations allow for a better juxtaposition on two dimensions, for instance regarding the input and output characteristics of app-enabled devices. However, the heterogeneity of devices within a device class provides insufficient discriminating power. For example, medium-sized, touch-based screens are usual interfaces both for tablets and for the infotainment systems within smart cars. Similarly, distinguishing between apps for embedded or stand-alone devices is not always possible due to different types of device integrations within a device category (cf. e.g. [CM16] for smart cars).

Therefore, the third dimension chosen for our taxonomy adds the degree of mobility to distinguish between similar device hardware in different usage contexts. Other potential approaches for categorizing devices include the degree of integration, automation, or intelligence attainable or provided by the device. This reaches from simple input / output devices with limited app interaction (such as fitness trackers), to interoperable software (such as smartphones), highly cross-linked and automated devices (in the IoT or smart home field), and finally to intelligent machines. While we deem it reasonable to discuss such an optional fourth dimension, we do not think the taxonomy would profoundly gain more discriminatory power. The added complexity would not be justified as the underlying assumption of increasing processing complexity is to some extent already encoded in the richness of inputs and outputs.

An alternative or possibly additional means for categorization is a *graph*, more specifically a *tree*. This way, categorization would get a hierarchical character that could for example honour development history and be subdivided. Additionally, this representation would be well suited to reveal similarities in particular features. If shown as a *polytree* such as sketched in Figure 15.4, even complex dependencies could be displayed (a smart watch, for example as a combination of wrist-worn fitness trackers and basic smartphone functionality). However, while a tree representation surely is charming for its depiction of dependencies and historical connections, it poses far less discriminatory power than the taxonomy we designed. Blending both possibilities, for instance using a multi-layered representation of several trees that show the connection between device classes from different perspectives or for various criteria, would become too complex to be practical.

15.4.2 Further Development

Firstly, future discussion needs to include the demarcation of devices to be included. As argued earlier, mobility is not necessarily the proper boundary. App-enablement has proven to be feasible, yet we will need to find (or provide) a profound definition for it. This is an ongoing task.

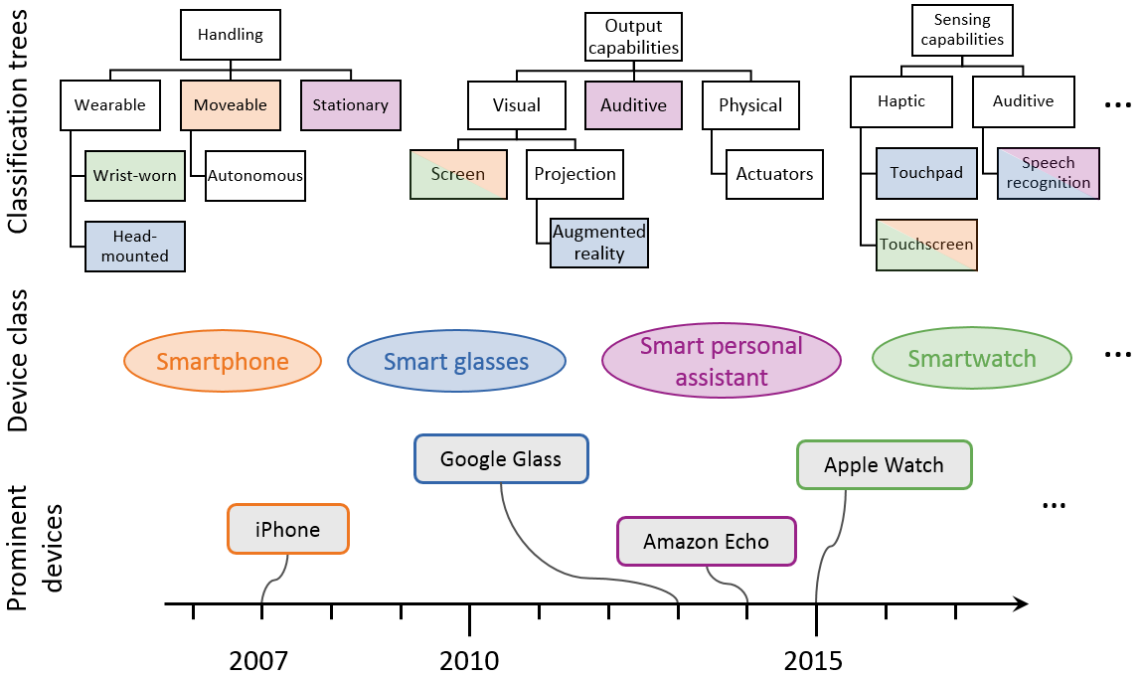


Figure 15.4: Exemplary Alternative Classification Approach

Secondly, it needs to be determined how the taxonomy can be kept up to date. In many other cases, taxonomies have proven to be either too detailed and thus requiring constant adjustments, or too little detailed and thus lacking discriminatory power. In any case, taxonomies that are used in any not entirely static field ought to evolve.

Due to a restriction to three orthogonal dimensions and clearly distinguishable values in each of it, we are optimistic that the taxonomy will be future-proof. Nevertheless, proper ways of deciding when adaptations are needed and what developments can be reflected without changes need to be defined. As part of this, we will need to scrutinize how to handle the differences in precision regarding categories. For example, it is very well understood what a smartphone is; smart homes, and to an even higher degree neural devices are (yet) diffuse with a lack of devices and applications to characterize them.

Thirdly, we so far have limited ourselves to consumer devices. This includes many devices that are *also* used for professional purposes, but arguably not all. Beyond that, some specialised devices are (so far) solely used for professional means. Examples can be found in industry, particularly in logistics. However, some of these might simply be subsumed by consumer devices. It could be said that, e.g., the devices used by parcel couriers are very similar to smartphones, despite the difference in form and the absence of a general purpose utilization. Moreover, commercial (and, similarly, military) devices might be derivatives of consumer hardware that has been "hardened" and more extensively tested. The same applies to special devices from areas such as healthcare or crisis prevention and response. While such devices typically have specific capabilities (such as

error-tolerance), on an abstract level they again are very similar to general purpose hardware. Thus, an updated taxonomy could try to include non-consumer devices. However, due to the complexity that arises particularly with devices that are so specialised that information regarding them is scarce, we deem the current limitation justified. Additionally, if kinds of devices are seldom addressed in writing, including them in a taxonomy arguably is superfluous anyway.

Fourthly, it should be scrutinized how the taxonomy can be provided in a form that is useful for researchers *and* for practitioners. Most scientists know taxonomies for research topics enforced by publication outlets.[5] Quite often these feel more like a "try to fit somewhere" game, particularly if a paper tackles a contemporary topic and the taxonomy provides little flexibility. If we want our taxonomy to be helpful for researchers, and – probably even harder to achieve – employed by practitioners, it needs to be easy to use yet powerful. Achieving this will be very valuable, as can e.g. be seen for cross-platform development, where new approaches can be clearly categorized by their characteristics. We think that our taxonomy should allow to put each device into exactly one class – choosing several applicable classes might be practical for the above named paper-theme categories, but we do not deem it practical for the purpose of our taxonomy.

The four discussion points have also illustrated the limitations of our work. Besides these issues that need to be worked on, an eventual verification of the taxonomy is mandated. Our planned work to further on this topic is sketched in the next section.

It would also be possible to develop the taxonomy towards an ontology (cf. [Cas+13]), possibly resulting in automated categorization aid which would also take ideas from the alternative polytree-based categorization as discussed in the previous section. For an unknown device, a decision tree could be traversed, leading to a prediction which kind of device is at hand. However, this would require rich semantic data (to allow inference), and it is currently not clear whether such an ontology would be considered to have much more value than the taxonomy already possesses.

15.5 Conclusion and Outlook

With this article, we have proposed a taxonomy for app-enabled devices. It builds on a position paper presented in April 2017 – and it remains the first such work. The taxonomy is based on three dimensions: the media richness of inputs and of outputs, and the degree of mobility. Examined separately, each dimension is relatively simple. In combination, they provide high discriminatory power. This becomes particularly evident when categorizing the current device landscape. We have provided figures that support this assessment throughout this paper. In general, it has proven to be much easier to use "flat" representations of two dimensions at a time than to render a 3D model – at least for publication.

[5] An example is be the ACM Computing Classification System, firstly presented in 1964 and revised in 1991, 1998, and 2012 [ACM15] (cf. also [Cas+13]).

The presented taxonomy can be considered as a milestone, and we deem it to be static for now. Undoubtedly, progress in the field will mandate future changes, but these will rather lead to a new *version* of the taxonomy than to a *new* taxonomy. Nonetheless, this article should still act as an invitation for discussion. After all, the taxonomy is but a step towards a more unified view of mobile computing and a solidified theoretic base in this field. Moreover, future work will need to continue with keeping a systematic overview of app-enabled devices.

A better theoretic understanding of mobile computing, producible advice for practice, and word towards unified development mark the pillars of our future work. As a part of this, we will use the taxonomy and also put it up for further discussion, e.g. as part of conference talks. Additionally, we will now reach out to our partners from practice and ask them for an assessment. If the taxonomy will become well adopted, empirical work should follow.

As we already wrote in the position paper, we do not hope for our work to become "yet another computer science taxonomy". Therefore, we hope that this article can illustrate the usefulness of a taxonomy and spark the interest for employing it.

References

[ACM15] ACM. *ACM Computing Classification System ToC*. 2015. URL: http://www.acm.org/about/class (visited on 07/28/2017).

[App07] Apple Inc. *Apple Reinvents the Phone with iPhone*. 2007. URL: http://%20www.apple.com/pr/library/2007/01/09Apple%20-Reinvents-the-Phone-with-iPhone.html (visited on 01/11/2017).

[App08] Apple Inc. *iPhone App Store Downloads Top 10 Million in First Weekend*. 2008. URL: http://%20www.apple.com/pr/library/%202008/07/14iPhone-App-Store-Downloads-Top%20-10-Million-in-First-Weekend.html (visited on 01/11/2017).

[App16] Apple Inc. *watchOS*. 2016. URL: www.apple.com/watchos/ (visited on 01/12/2017).

[Bho13] Achintya K. Bhowmik. "39.2: Invited Paper: Natural and Intuitive User Interfaces: Technologies and Applications". In: *SID Symposium Digest of Technical Papers* 44.1 (2013), pp. 544–546. DOI: 10.1002/j.2168-0159.2013.tb06266.x.

[BMG17] Andreas Biørn-Hansen, Tim A. Majchrzak, and Tor-Morten Grønli. "Progressive Web Apps: The Possible Web-Native Unifier for Mobile Development". In: *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST)*. SciTePress, 2017, pp. 344–351.

[Bus+15] Christoph Busold et al. "Smart and secure cross-device Apps for the Internet of advanced things". In: *Lecture Notes in Computer Science (LNCS)* 8975 (2015), pp. 272–290. DOI: 10.1007/978-3-662-47854-7{\textunderscore}17.

[Cas+13] Lillian N. Cassel et al. “The New ACM CCS and a Computing Ontology”. In: *Proc. 13th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)*. Indianapolis, Indiana, USA: ACM, 2013, pp. 427–428.

[Cha+16] J. Chauhan et al. “Characterization of early smartwatch apps”. In: *2016 IEEE Int. Conf. on PerCom Workshops 2016* (2016). DOI: 10.1109/PERCOMW.2016.7457170.

[Chu+16] Stephanie Hui-Wen Chuah et al. “Wearable technologies: The role of usefulness and visibility in smartwatch adoption”. In: *Computers in Human Behavior* 65 (2016), pp. 276–284. DOI: 10.1016/j.chb.2016.07.047.

[CM16] Riccardo Coppola and Maurizio Morisio. “Connected Car: Technologies, Issues, Future Trends”. In: *ACM Comput. Surv.* 49.3 (2016), 46:1. DOI: 10.1145/2971482.

[CMK14] Jagmohan Chauhan, Anirban Mahanti, and Mohamed Ali Kaafar. “Towards the Era of Wearable Computing?” In: *Pro. 2014 CoNEXT on Student Workshop*. CoNEXT Student Workshop ’14. ACM, 2014, pp. 24–25. DOI: 10.1145/2680821.2680833.

[Dag+16] Jan C. Dageförde et al. “Generating App Product Lines in a Model-Driven Cross-Platform Development Approach”. In: *49th Hawaii International Conference on System Sciences (HICSS)*. 2016, pp. 5803–5812. DOI: 10.1109/HICSS.2016.718.

[Dal+13] I. Dalmasso et al. “Survey, comparison and evaluation of cross platform mobile application development tools”. In: *2013 9th Int. Wireless Communications and Mobile Computing Conf. (IWCMC)*. 2013. DOI: 10.1109/IWCMC.2013.6583580.

[DLT87] Richard L. Daft, Robert H. Lengel, and Linda Klebe Trevino. “Message Equivocality, Media Selection, and Manager Performance: Implications for Information Systems”. In: *MIS quarterly* 11.3 (1987), p. 355. DOI: 10.2307/248682.

[El-+15] Wafaa S. El-Kassas et al. “Taxonomy of Cross-Platform Mobile Applications Development Approaches”. In: *Ain Shams Engineering Journal* (2015).

[Goo16] Google Inc. *Android Wear*. 2016. URL: https://android.com/wear (visited on 01/12/2017).

[Gub+13] Jayavardhana Gubbi et al. “Internet of Things (IoT): A vision, architectural elements, and future directions”. In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660. DOI: 10.1016/j.future.2013.01.010. URL: http://www.sciencedirect.com/science/article/pii/S0167739X13000241.

[HEE13] Shah Rukh Humayoun, Stefan Ehrhart, and Achim Ebert. “Developing Mobile Apps Using Cross-Platform Frameworks: A Case Study”. In: *Proc. 15th Int. Conf. HCI International, Part I*. Ed. by Masaaki Kurosu. Springer, 2013, pp. 371–380. DOI: 10.1007/978-3-642-39232-0{\textunderscore}41.

[HHM13] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. "Evaluating Cross-Platform Development Approaches for Mobile Applications". In: *Revised Selected Papers WEBIST 2012*. Ed. by José Cordeiro and Karl-Heinz Krempels. Vol. 140. Lecture Notes in Business Information Processing (LNBIP). Springer, 2013, pp. 120–138.

[Hin+17] Daniel Hintze et al. "A Large-Scale, Long-Term Analysis of Mobile Device Usage Characteristics". In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1.2 (2017), pp. 1–21. DOI: 10.1145/3090078.

[HMK13] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. "Cross-platform model-driven development of mobile applications with MD2". In: *Proc. of the 28th Annual ACM Symposium on Applied Computing (SAC)*. Ed. by Sung Y. Shin and José Carlos Maldonado. SAC '13. ACM, 2013, pp. 526–533. DOI: 10.1145/2480362.2480464.

[Iba+17] Jorge E. Ibarra-Esquer et al. "Tracking the Evolution of the Internet of Things Concept Across Different Application Domains". In: *Sensors (Basel, Switzerland)* 17.6 (2017). DOI: 10.3390/s17061379.

[JJ14] Chris Jones and Xiaoping Jia. "The AXIOM model framework: Transforming requirements to native code for cross-platform mobile applications". In: *2nd Int. Conf. on Model-Driven Engineering and Software Dev.* Ed. by Luis Ferreira Pires. IEEE, 2014.

[JM15] Chakajkla Jesdabodi and Walid Maalej. "Understanding Usage States on Mobile Devices". In: *Proc. 2015 ACM Int. Joint Conf. on Pervasive and Ubiquitous Computing*. UbiComp '15. ACM, 2015, pp. 1221–1225. DOI: 10.1145/2750858.2805837.

[KK16] István Koren and Ralf Klamma. "The Direwolf Inside You: End User Development for Heterogeneous Web of Things Appliances". In: *Proc. 16th Int, Conf, ICWE 2016*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Springer, 2016, pp. 484–491. DOI: 10.1007/978-3-319-38791-8{\textunderscore}35.

[LG 16] LG Electronics. *WebOS for LG Smart TVs*. 2016. URL: http://www.lg.com/uk/smarttv/webos (visited on 05/31/2016).

[LKN17] Houssem Lahiani, Monji Kherallah, and Mahmoud Neji. "Vision Based Hand Gesture Recognition for Mobile Devices: A Review". In: *Proceedings of the 16th International Conference on Hybrid Intelligent Systems (HIS 2016)*. Ed. by Ajith Abraham et al. Springer International Publishing, 2017, pp. 308–318. DOI: 10.1007/978-3-319-52941-7{\textunderscore}31.

[NJE17] Timothy Neate, Matt Jones, and Michael Evans. "Cross-device Media: A Review of Second Screening and Multi-device Television". In: *Personal Ubiquitous Comput* 21.2 (2017), pp. 391–405. DOI: 10.1007/s00779-017-1016-2.

[QPM17] Ricardo Queirós, Filipe Portela, and José Machado. "Magni - A Framework for Developing Context-Aware Mobile Applications". In: *Recent Advances in Information Systems and Technologies: Volume 3*. Ed. by Álvaro Rocha et al. Springer International Publishing, 2017, pp. 417–426. DOI: 10.1007/978-3-319-56541-5{\textunderscore}43.

[RM16] Christoph Rieger and Tim A. Majchrzak. "Weighted Evaluation Framework for Cross-Platform App Development Approaches". In: *Proc. 9th SIGSAND/PLAIS EuroSymposium*. Ed. by Stanislaw Wrycza. Springer, 2016, pp. 18–39. DOI: 10.1007/978-3-319-46642-2{\textunderscore}2.

[RM17] Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-Enabled Devices". In: *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST)*. SciTePress, 2017, pp. 332–339.

[RM18] Christoph Rieger and Tim A. Majchrzak. "A Taxonomy for App-Enabled Devices: Mastering the Mobile Device Jungle". In: *Web Information Systems and Technologies*. Ed. by Tim A. Majchrzak et al. Cham: Springer International Publishing, 2018, pp. 202–220. DOI: 10.1007/978-3-319-93527-0_10.

[SB16] K. Singh and J. Buford. "Developing WebRTC-based team apps with a cross-platform mobile framework". In: *2016 13th IEEE Annual Consumer Communications and Networking Conference, CCNC 2016* (2016). DOI: 10.1109/CCNC.2016.7444762.

[SK13] A. Sommer and S. Krusche. "Evaluation of cross-platform frameworks for mobile applications". In: *Lecture Notes in Informatics (LNI)* P-215 (2013), pp. 363–376.

[Sta16] Statista Inc. *Global smartphone shipments forecast from 2010 to 2020*. 2016. URL: https://www.statista.com/%20statistics/263441/ (visited on 08/08/2017).

[Sta17a] Statista Inc. *Global smart TV unit sales from 2014 to 2018 (in millions)*. 2017. URL: https://www.statista.com/statistics/540675/global-smart-tv-unit-sales/ (visited on 08/08/2017).

[Sta17b] Statista Inc. *Smartwatches*. 2017. URL: https://www.statista.com/study/36038/smartwatches-statista-dossier/ (visited on 08/08/2017).

[The16] The Linux Foundation. *Tizen*. 2016. URL: https://www.tizen.org (visited on 01/12/2017).

[Wei91] Mark Weiser. "The Computer for the 21st Century". In: *Scientific American* 265.3 (1991), pp. 94–104. DOI: 10.1038/scientificamerican0991-94.

16

Interoperability of BPMN and MAML for Model-Driven Development of Business Apps

Table 16.1: Fact sheet for publication P10

Title	Interoperability of BPMN and MAML for Model-Driven Development of Business Apps
Authors	Christoph Rieger[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2018
Conference	Business Modeling and Software Design (BMSD)
Copyright	Springer International Publishing
Full Citation	Christoph Rieger. "Interoperability of BPMN and MAML for Model-Driven Development of Business Apps". In: *Business Modeling and Software Design*. Ed. by Boris Shishkov. Cham: Springer International Publishing, 2018, pp. 149–166. DOI: 10.1007/978-3-319-94214-8_10 This version is not for redistribution. The final authenticated version is available online at https://doi.org/10.1007/978-3-319-94214-8_10.

Interoperability of BPMN and MAML for Model-Driven Development of Business Apps

Christoph Rieger

Keywords: BPMN, Business Process Modeling, Mobile App, Model Transformation, Business App

Abstract: With process models widely used as means for documentation and monitoring of business activities, the conversion into executable software often still remains a manual and time-consuming task. The MAML framework was developed to ease the creation of mobile business apps by jointly modeling process, data, and user interface perspectives in a graphical, process-oriented model for subsequent code generation. However, this domain-specific notation cannot benefit from existing process knowledge which is often encoded in BPMN models. The purpose of this paper is to analyze conceptual differences between both notations from a software development perspective and provide a solution for interoperability through a model-to-model transformation. Therefore, workflow patterns identified in previous research are used to compare both notations. A conceptual mapping of supported concepts is presented and technically implemented using a QVT-O transformation to demonstrate an automated mapping between BPMN and MAML. Consequently, it is possible to simplify the automatic generation of mobile apps by reusing processes specified in BPMN.

16.1 Introduction

Business process models have been used for communicating, documenting, monitoring, and optimizing business processes for many years [TK08]. A wide variety of notations has emerged among which the Business Process Model and Notation 2.0 (BPMN) is best known [Obj11]. However, process models need to be regularly synchronized with actual activities because most of them lack a sufficient level of detail to be directly executable using workflow engines. At the same time, companies are faced with global trends such as digitization and big data which require flexible business processes and new forms of organization in order to adapt to external influences.

At the intersection between process modeling and software specification, it is not possible to fully specify applications using solely BPMN models because of their limited focus on the sequence flow of activities. Essential software development perspectives such as data models, user interfaces, and user interactions are missing. In addition, user tasks are vaguely specified and thus hard to transform into adequate representations on screen.

Especially with the increased use of model-driven approaches, domain-specific modeling notations compete against traditional general-purpose modeling notations. Ideally, changes within models propagate automatically to the derived software artifacts, thus enabling a fast reaction to changing requirements without time-consuming development cycles by IT departments. One example is the Münster App Modeling Language (MAML) which provides a graphical notation understandable both for programmers and end users to specify mobile business apps on a high

level of abstraction using data-driven processes [Rie18a]. Without need for manual programming, the framework automatically generates functional cross-platform app source code for mobile devices.

MAML models integrate the full spectrum of app specification in contrast to the narrower scope of BPMN which only covers a process perspective. Nevertheless, with BPMN as the de facto standard for documenting business processes, the research question guiding this work arises: How can previous process documentation encoded in BPMN models be reused to support cross-platform app specification with MAML? An automated transformation between both notations would therefore be beneficial with regard to consistency and potential time savings: Existing process models could be enriched with app-specific information and app models representing the course of activities could be converted to BPMN for documentation and analysis.

The main contributions of this paper are three-fold and reflected by its structure: Firstly, the capabilities and shortcoming of the BPMN notation are analyzed with regard to the specification of mobile apps (Section 16.3, which also briefly introduces the MAML notation). Secondly, supported workflow patterns for BPMN and MAML are compared to highlight the conceptual differences (Section 16.4). Thirdly, transformations from BPMN to MAML and vice versa are proposed to demonstrate the practicability of the approach (Section 16.5).

16.2 Related work

Several approaches which deal with the representation and transformation from process models to executable artifacts have been discussed in scientific literature, e.g., using Petri nets [van94] or YAWL models [Dec+08]. Their focus lies on supporting the model interpretation by workflow engines. In the context of software development, applications are often conceptually described using layered architectures in order to distinguish the required constituents of data model, business logic, and presentation layer and, consequently, manage complexity. This can be achieved using a combination of generic notations such as UML [Gia+17] or domain-specific frameworks [Bar+15; BBF10] but not necessarily depicting the application contents as process sequence.

For reasons detailed in the following section, the BPMN notation is not sufficient to represent all aspects of the target application. With regard to process-aware information system modeling, three categories of notations can be distinguished, each exhibiting advantages and shortcomings. Firstly, BPMN is often used to describe the process perspective together with additional notations. For example, Brambilla et al. [Bra+08] use BPMN and the WebML language focused on user interaction in order to create rich internet applications, and Traettenberg and Krogstie [TK08] use the refinement by Diamodl diagrams for dialog specification.

Secondly, the BPMN 2.0 notation itself can be extended to annotate further data using *ExtensionDefinitions* and *ExtensionAttributeDefinitions*. This approach is used by workflow engines such as Camunda [Cam17] but does not allow for visualization of annotated data or new custom diagram elements.

Thirdly, custom notations have been proposed that provide integrated modeling for workflows in general or tailored to specific domains. For example, Kannengiesser et al. [Kan+16] propose an approach with a custom notation to combine UI representations with process flows through UI-related task types, and Kalnins et al. [Kal+14] combine a graphical notation for process flows with a textual expression syntax to manipulate data.

The MAML notation we use for modeling mobile business apps – i.e., form-based, data-driven apps interacting with back-end systems [MEK15] – falls into the latter category of a domain-specific graphical notation that combines control flow logic, UI, and data perspectives such that all application concerns are specified (see Subsection 16.3.2). However, the transformation approach does not aim to modify the BPMN meta model but instead extracts the maximum amount of information such that the modeler only needs to enrich missing elements. In this regard, our approach is most similar to the WebRatio BPM tool [BBF10] which transforms BPMN models to the more technical WebML notation.

16.3 Business Process Notations for Mobile App Development

In this section, the most commonly used notation for business process modeling, BPMN, is analyzed regarding the development of process-driven mobile applications, and the domain-specific language MAML is introduced.

16.3.1 Business Process Model and Notation (BPMN)

BPMN is a control flow-based notation to define a sequence of process steps. The main graphical elements can be subdivided into flow objects (activities, gateways, and events) and artifacts (data objects, groups, and annotations) which are interlinked by connecting objects (sequence flows, message flows, and associations) and organized in roles representing actors or organizations (pools and lanes) [Obj11]. A simplified model depicting the process of writing a thesis is shown in Figure 16.1 and contains several typical task types, control flow branches, and data transfer mechanisms. It is further used to demonstrate how tool support for supervision-related tasks can be provided by transforming it into a mobile app.

Problems of the notation have already been studied in general [Rec10] as well as for its applicability in software development projects [Tuo+07]. The main criticism arises from deficiencies in business rule specification, symbol overload with partially superfluous or fuzzily delimited concepts, and the abundance of event types. The following paragraphs highlight the issues with regard to model-driven mobile development beyond the use of models for informally sketching process sequences.

Data layer BPMN is control-flow oriented and mostly neglects data modeling. Even with the enhancements of BPMN 2.0, the user can only specify the input and output of tasks using

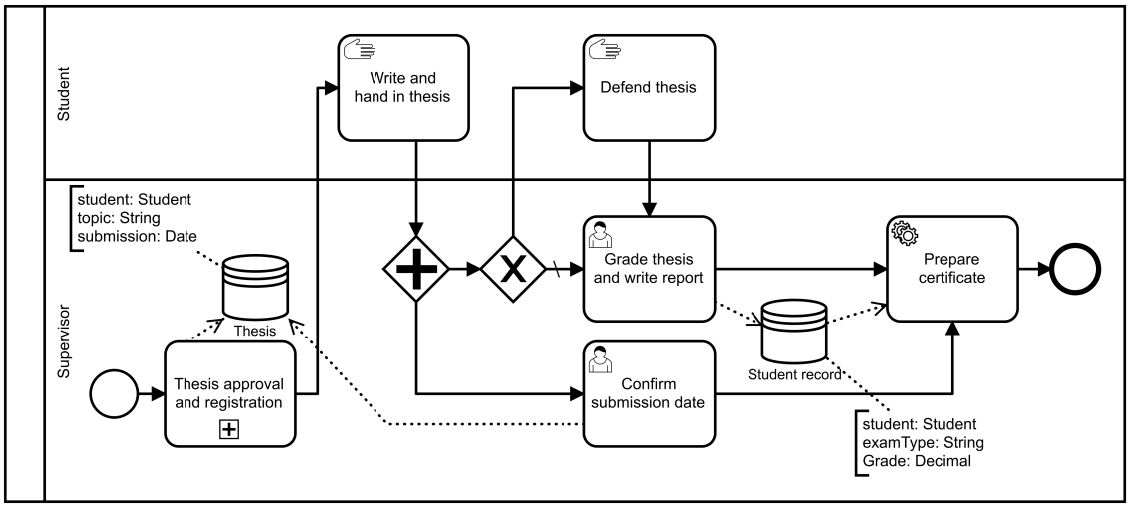


Figure 16.1: Sample BPMN Model Representing a (Simplified) Process for Thesis Writing

process-internal (*Data Input / Data Output / Data Object*) or persistent (*Data Store*) items and additionally distinguish between single items and collections as well as multiple states of the data. However, attributes, data types, and type interrelations – essential for the actual execution of data manipulation tasks – cannot be specified and can only be inferred on a generic level [CMS12].

Business logic layer BPMN models provide a wide range of elements to model the control-flow and conditional paths of execution. Whereas workflow engines support many such concepts, three characteristics of mobile devices complicate a direct utilization of generic BPMN elements.

Firstly, mobile apps focus on user-oriented tasks. In contrast to enterprise contexts in which many steps can be delegated to web services, apps focus on direct user interaction and require a specification of what actions to perform in a single step. Regarding data manipulation, this, e.g., includes whether an item is retrieved, updated, displayed, created, or deleted. Also, mobile-specific functionalities such as sensor / camera access and starting phone calls are not available.

Secondly, mobile devices may experience connectivity issues at any time and for unknown duration, causing unreliable response times if workflow instances are strictly allocated to one device. Similarly, multi-step transactions may lock the whole system.

Thirdly, mobile apps are usually small and inherently distributed systems. Parallel execution of tasks is hardly possible on small screens (and questionable as regards frequent context switches). Even considering a multi-role context in which activities are potentially performed by different users collaboratively, issues arise from the coordination of up-to-date distributed data and conflicting concurrent modifications.

Presentation layer Although default representations could be derived from the (unavailable) data attributes to display, a minimal specification of sensible user interfaces also requires the

addition of informative texts and means of controlling the user interactions to navigate between views. Neither exists in BPMN.

16.3.2 Muenster App Modeling Language (MAML)

MAML is a graphical domain-specific language with the purpose of modeling cross-platform business apps. Following the model-driven paradigm, the framework allows for the fully automated generation of app source code for multiple smartphone platforms (currently Android and iOS) as well as a Java-based back-end component for app coordination and as interface to other company systems.

The notation is built around five main design goals: (1) code-less cross-platform app creation, (2) a strong domain expert focus in contrast to complex technical notations, (3) a high level of abstraction by modeling data-driven processes, (4) task-oriented modularization using use cases, and (5) a declarative and platform-neutral description of logical steps in contrast to UI-centered configurators. Figure 16.2 depicts the same scenario as Figure 16.1, however, it contains all the information to generate the app.

Although not necessarily limited to the domain of mobile apps, the notation considers interoperability with mobile device hardware (e.g., using elements for camera usage or sensor access), mobile interaction patterns (such as starting phone calls), and current generators target the Android and iOS platforms. Also, the integrated modeling approach to process flows and data structures may not be well suited for large-scale applications. For a more detailed introduction to MAML the reader is referred to [Rie18a; RK18].

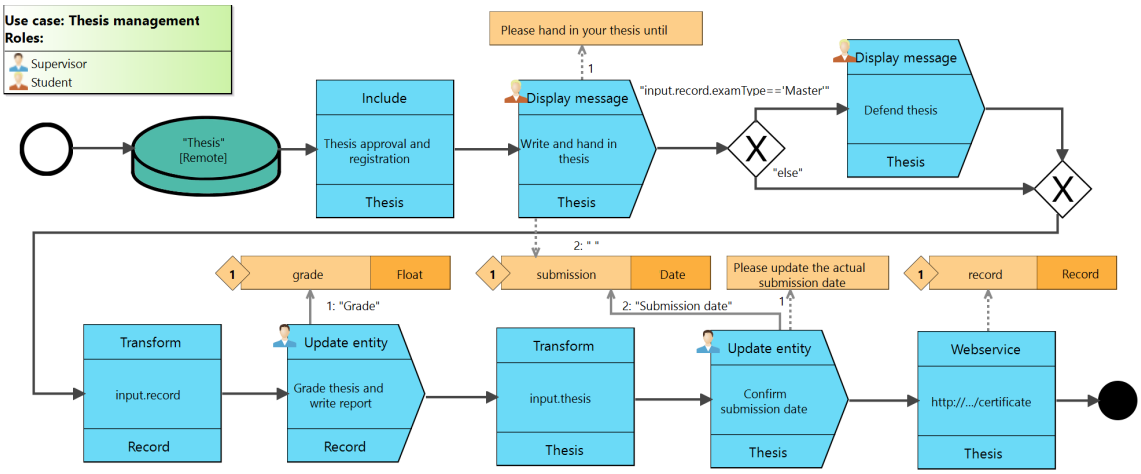


Figure 16.2: Sample MAML Model With Thesis Management Process Equivalent to Figure 16.1

16.4 Comparison of Workflow Patterns in MAML and BPMN

Russell et al. [Rus+06] have collected an extensive list of workflow patterns. As a comprehensive overview on workflow patterns also comprises a data access and resource perspective, respective patterns have been presented in [Rtv04; Rus+05]. To prepare a possible mapping of concepts, supported patterns need to be assessed first for both notations. An analysis of the BPMN notation is provided in [Woh+06]. Therefore, this section evaluates workflow concepts of MAML according to the pattern catalog and highlights the most important differences.

16.4.1 Control-flow Patterns

Considering Workflow Control Patterns (WCP), basic patterns include the possibility to depict a *Sequence* (WCP1) of activities, a *Parallel Split* (WCP2) from a single thread of control into multiple threads executed in parallel, the *Synchronization* (WCP3) of multiple parallel activities into a single thread of control, an *Exclusive Choice* (WCP4) to execute one of multiple alternatives according to given conditions, and a *Simple Merge* (WCP5) to consolidate multiple alternative branches. In MAML, WCP1 is given by the corresponding "process connector" construct. However, as explicated in Subsection 16.3.1, WCP2 and WCP3 as well as advanced branching and synchronization patterns including inter-workflow parallelism are not supported. Figure 16.3 depicts how branching the control flow (WCP4) in MAML can be based on a user decision (a) or evaluated based on the state of data objects (b), and can be merged accordingly (WCP5; (c)). Unlike BPMN, implicit branching and merging is disallowed to foster consistency and explicitly indicate control flow variations.

Regarding structural patterns, *Deferred Choice* (WCP16) represents a runtime choice between different branches of which the first executes. In MAML, this is possible to a certain extent: If multiple roles are assigned to one task, (only) the first user will execute the task (d). *Interleaved Parallel Routing* (WCP17) refers to a partial ordering of process step dependencies to be linearized and executed sequentially. *Critical Section* (WCP39) and *Interleaved Routing* (WCP40) are similar patterns that specify a set of subgraphs or individual activities to be executed once in arbitrary sequential order. To clarify the future app structure, MAML currently relies on an explicitly modeled, fixed sequence of activities.

A *Milestone* (WCP18) represents the ability to execute an activity until a state-dependent execution point is reached. An exclusive choice with state-based condition (b) can be reused to represent this in MAML. BPMN also implements the previous patterns but does not support the concept of workflow state at all.

Concerning cancellation patterns, MAML supports the *Cancel Case* (WCP20) pattern (e) to stop the current workflow instance. As tasks are allocated to roles – not individual users – the same process instance might also be started accidentally from multiple devices and automatically canceled for all but the first execution according to the *Cancel Activity* (WCP19) pattern (although not explicitly modeled that can occur in situations such as (f)). *Cancel Region* (WCP25) denotes

Table 16.2: Workflow Control Patterns According to [Rus+06] in BPMN [Woh+06; Wor17] and MAML

	Workflow Control Pattern (WCP)	**BPMN**	**MAML**
1	Sequence	+	+
2	Parallel Split	+	−
3	Synchronization	+	−
4	Exclusive Choice	+	+
5	Simple Merge	+	+
6	Multi-Choice	+	+/−
7	Structured Synchronizing Merge	+	−
8	Multi-Merge	+	+
9	Structured Discriminator	+/−	−
10	Arbitrary Cycles	+	+
11	Implicit Termination	+	+
12	Multiple Instances without Synchronization	+	−
13	Multiple Instances with a Priori Design-Time Knowledge	+	−
14	Multiple Instances with a Priori Run-Time Knowledge	+	−
15	Multiple Instances without a Priori Run-Time Knowledge	−	−
16	Deferred Choice	+	+
17	Interleaved Parallel Routing	−	−
18	Milestone	−	+/−
19	Cancel Activity	+	+/−
20	Cancel Case	+	+
21	Structured Loop	+	+/−
22	Recursion	−	+
23	Transient Trigger	−	−
24	Persistent Trigger	+	+/−
25	Cancel Region	+/−	+/−
26	Cancel Multiple Instance Activity	+	−
27	Complete Multiple Instance Activity	−	−
28	Blocking Discriminator	+/−	−
29	Canceling Discriminator	+	−
30	Structured Partial Join	+/−	−
31	Blocking Partial Join	+/−	−
32	Canceling Partial Join	+/−	−
33	Generalized AND-Join	+	−
34	Static Partial Join for Multiple Instances	+/−	−
35	Canceling Partial Join for Multiple Instances	+/−	−
36	Dynamic Partial Join for Multiple Instances	−	−
37	Local Synchronizing Merge	−	−
38	General Synchronizing Merge	−	−
39	Critical Section	−	−
40	Interleaved Routing	+/−	−
41	Thread Merge	+	−
42	Thread Split	+	−
43	Explicit Termination	+	+

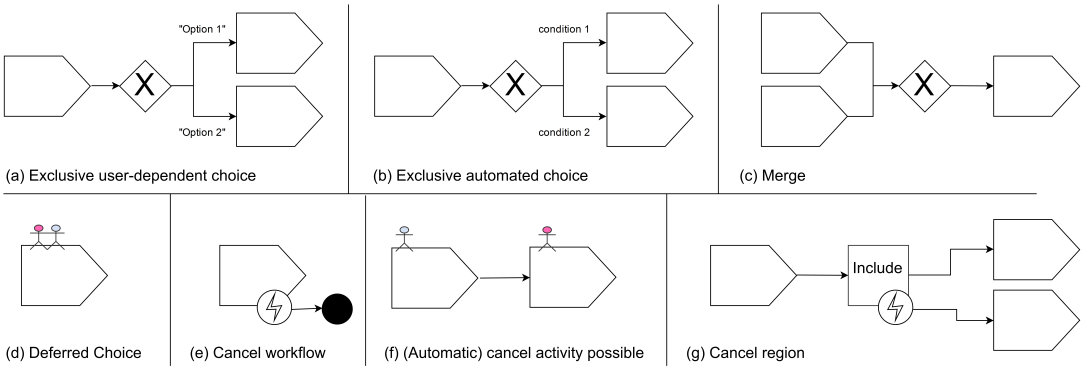


Figure 16.3: Basic control-flow and cancellation patterns in MAML

a subgraph being canceled and can be represented in MAML through a subprocess terminating unsuccessfully (g).

Within a process model, *Arbitrary Cycles* (WCP10) with multiple end points can occur as depicted in Figure 16.4 (a). Regarding *Structured Loops* (WCP21) to execute activities repeatedly, two representations are possible: A loop is explicitly defined using an XOR element with an expression to test as condition before (for loop; (b)) or after (do-while loop; (c)) the activities are executed. In addition, users may choose multiple elements in a "select entity" step such that subsequent activities are performed for each of the selected elements. In contrast to BPMN, *Recursion* (WCP22) is possible because of passing data to subsequent process elements (d) instead of BPMN's token-based approach to sequence flows. *Persistent Triggers* (WCP24) can be used to start tasks based on a data condition (e). Finally, MAML supports both *Explicit Termination* (WCP43; (f)) and *Implicit Termination* (WCP11; (g)) of workflow instances, although the latter is discouraged to avoid incomplete process branches. Section 16.4 summarizes the full (+), partial (+/−), or lacking support of the workflow control patterns for BPMN and MAML.

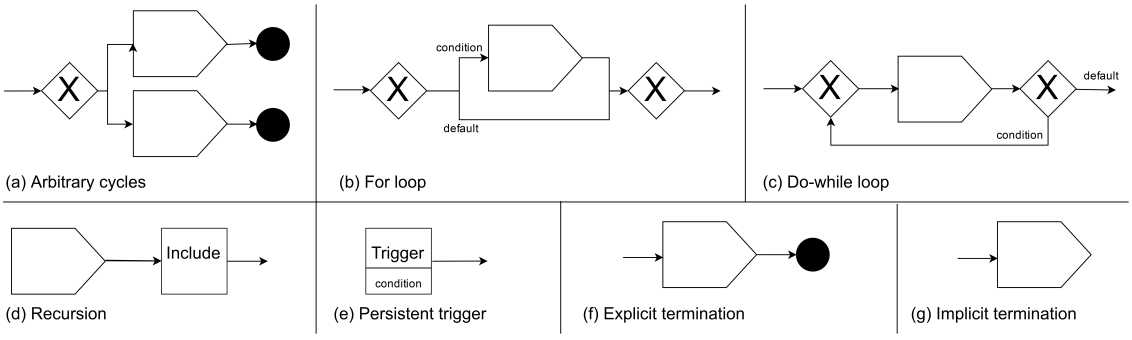


Figure 16.4: Iteration, Trigger, and Termination Patterns in MAML

Table 16.3: Workflow Data Patterns According to [Rus+05] in BPMN [Woh+06; Wor17] and MAML

Workflow Data Pattern (WDP)		BPMN	MAML
1	Task Data	+	+
2	Block Data	+	+
3	Scope Data	+	+/−
4	Multiple Instance Data	+	+/−
5	Case Data	+	+
6	Folder Data	+	−
7	Workflow Data	+	+
8	Environment Data	+	+
9	Task to Task	+/−	+
10	Block Task to SubWorkflow Decomposition	+	+
11	SubWorkflow Decomposition to Block Task	+	+
12	To Multiple Instance Task	+	−
13	From Multiple Instance Task	+	−
14	Case to Case	+	−
15	Task to Environment - Push-Oriented	−	+
16	Environment to Task - Pull-Oriented	+	+
17	Environment to Task - Push-Oriented	−	−
18	Task to Environment - Pull-Oriented	−	−
19	Case to Environment - Push-Oriented	+	+
20	Environment to Case - Pull-Oriented	+	+
21	Environment to Case - Push-Oriented	+	−
22	Case to Environment - Pull-Oriented	−	−
23	Workflow to Environment - Push-Oriented	−	+
24	Environment to Workflow - Pull-Oriented	+	+
25	Environment to Workflow - Push-Oriented	+/−	−
26	Workflow to Environment - Pull-Oriented	+	−
27	Data Transfer by Value - Incoming	−	−
28	Data Transfer by Value - Outgoing	+/−	−
29	Data Transfer - Copy In/Copy Out	+	−
30	Data Transfer by Reference - Unlocked	+/−	+
31	Data Transfer by Reference - With Lock	+/−	−
32	Data Transformation - Input	+/−	−
33	Data Transformation - Output	+	−
34	Task Precondition - Data Existence	+/−	+/−
35	Task Precondition - Data Value	+/−	+/−
36	Task Postcondition - Data Existence	−	+/−
37	Task Postcondition - Data Value	−	+/−
38	Event-Based Task Trigger	−	−
39	Data-Based Task Trigger	−	+
40	Data-Based Routing	+/−	+

16.4.2 Data Patterns

A variety of data patterns (WDP) complements the process view of a workflow (cf. Table 16.3). In MAML, a global perspective is adopted to define the flow of data objects through the process. This represents the *Workflow Data* (WDP7) concept as depicted in Figure 16.5 (a). Only data required in a certain step is modeled and the state is passed to subsequent process steps, thus effectively also implementing the *Task Data* (WDP1) and *Block Data* (WDP2) patterns. *Case Data* (WDP5) representing objects specific to a process instance, and *Scope Data* (WDP3) denoting custom regions of data visibility can be achieved with local data storage which is transferred to the global data store at a later point of process execution (c). *Environment Data* (WDP 8) from the operating system can also be incorporated, for example by accessing the camera (d) or GPS location (e). Again, patterns related to concurrency are not supported, but the same task may be executed with different data objects in loops (*Multiple Instance Data*; WDP4; (f)).

This broad set of data visibility patterns reaches beyond the capabilities of BPMN which is limited to task, block, and case data through the use of parameters.

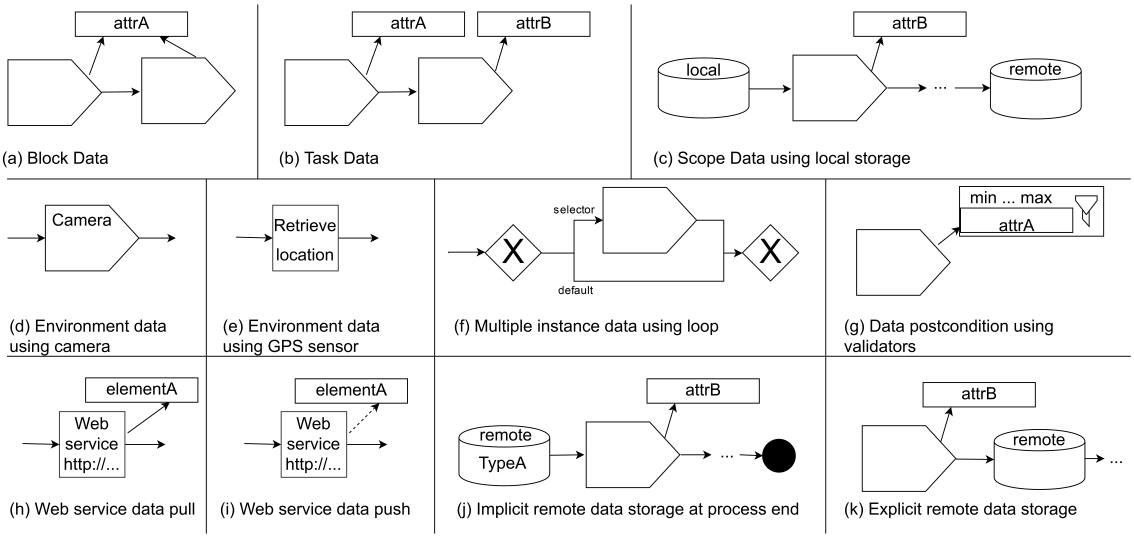


Figure 16.5: Data Visibility, Interaction, and Routing Patterns in MAML

Regarding internal data interaction patterns, MAML and BPMN have similar capabilities. Data can be passed from *Task to Task* (WDP9; e.g. (a) in Figure 16.5), from Block Task to *SubWorkflow Decomposition* (WDP10; e.g. (d) in Figure 16.4), and *SubWorkflow Decomposition to Block Task* (WDP11) by simply connecting the respective process elements. *Data interaction To/From Multiple Instance Tasks* (WDP12-13) are unavailable for lack of concurrency. Also, *Case to Case* (WDP14) data interaction is not intended because of distributed execution on mobile devices with varying connectivity. External data interaction in MAML relies on the global data view and the user being in control of data interactions. Therefore, data can be transferred between task and environment in pull fashion using web services (WDP16; (h) in Figure 16.5) as well as push-oriented (WDP15)

using web services (i) or implicit/explicit remote data storage (j/k). As task data is available to the subsequent tasks of the case, this also applies to push- and pull-oriented data transfer between the environment and the case/workflow (WDP19-20, WDP23-24). All data is passed by reference (WDP30) without locking mechanisms to support low-connectivity scenarios. In contrast, BPMN supports value- and reference-based data transfer with locks as well as additional in-/ output transformations, but only regarding task interactions with the environment (WDP15-18, WDP27-29, WDP32-33).

Finally, *Task Preconditions* (WDP34-35) are supported using the exclusive choice construct in MAML (cf. (b) in Figure 16.3) but can flexibly validate the value of the data item in contrast to BPMN's more limited check for data existence. *Task Postconditions* (WDP36-37) can be applied equivalently or alternatively using data validators ((g) in Figure 16.5). *Data-Based Task Triggers* (WDP39) (cf. (e) in Figure 16.4) and *Data-Based Routing* (WDP40) possibilities (cf. (b) in Figure 16.3) exist both in MAML and BPMN.

16.4.3 Resource Patterns

Concerning workflow resource patterns (WRP; cf. Table 16.4), both notations share the concept of roles to distribute tasks. A role can therefore represent a single actor (WRP1) or a group of resources (WRP2) as depicted in Figure 16.6 (a/b). The role concepts also serves as *Authorization* (WRP4) because distinct apps are created which allow only the execution of tasks related to that role.

Whereas work items in BPMN are allocated to resources (WRP14), the mobile context of MAML fosters an on-demand, pull-oriented execution of work items. Work items are *Distributed by Offer to Multiple Resources* (WRP13) and each worker can initiate the immediate execution of work items (WRP23) while having the *Selection Autonomy* (WRP26) to choose tasks from a list of open work items. To limit the amount of context switches in the distributed execution of workflows, MAML implements the *Retain Familiar* (WRP7) pattern which by default starts the next workflow step unless the current user does not comply with any of the assigned roles (c). If all tasks are assigned to the same role or only one role is defined for the use case without explicit assignment (d), this also allows for the *Case Handling* (WRP6) pattern in which all activities are performed by the same individual. Otherwise, subsequent roles are automatically notified about the process instance for *Chained Execution* (WRP39). Also, complying with the *Commencement on Creation* (WRP36) pattern, new use case instances can be initiated for any process whose first task matches a user's role.

Automatic Execution (WRP11), i.e. trigger-based workflow execution without explicit resource allocation, is possible both in BPMN and in MAML (see (e) in Figure 16.4). Also, there are no theoretical constraints on how many instances of a task are executed simultaneously (WRP42) by different users with the same role.

Table 16.4: Workflow Resource Patterns According to [Rtv04] in BPMN [Woh+06; Wor17] and MAML

Workflow Resource Pattern (WRP)		BPMN	MAML
1	Direct Allocation	+	+
2	Role-Based Allocation	+	+
3	Deferred Allocation	−	−
4	Authorization	−	+/−
5	Separation of Duties	−	−
6	Case Handling	−	+/−
7	Retain Familiar	−	+
8	Capability Based Allocation	−	−
9	History Based Allocation	−	−
10	Organizational Allocation	−	−
11	Automatic Execution	+	+
12	Distribution by Offer - Single Resource	−	−
13	Distribution by Offer - Multiple Resources	−	+
14	Distribution by Allocation - Single Resource	+	−
15	Random Allocation	−	−
16	Round Robin Allocation	−	−
17	Shortest Queue	−	−
18	Early Distribution	−	−
19	Distribution on Enablement	+	+
20	Late Distribution	−	−
21	Resource-Initiated Allocation	−	−
22	Resource-Initiated Execution - Allocated Work Item	−	−
23	Resource-Initiated Execution - Offered Work Item	−	+
24	System Determined Work Queue Content	−	−
25	Resource-Determined Work Queue Content	−	−
26	Selection Autonomy	−	+
27	Delegation	−	−
28	Escalation	−	−
29	Deallocation	−	−
30	Stateful Reallocation	−	−
31	Stateless Reallocation	−	−
32	Suspension/Resumption	−	−
33	Skip	−	−
34	Redo	−	−
35	Pre-Do	−	−
36	Commencement on Creation	+	+
37	Commencement on Allocation	−	−
38	Piled Execution	−	−
39	Chained Execution	+	+
40	Configurable Unallocated Work Item Visibility	−	−
41	Configurable Allocated Work Item Visibility	−	−
42	Simultaneous Execution	+	+
43	Additional Resources	−	−

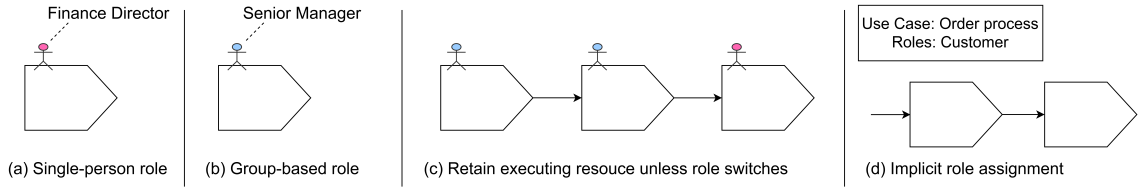


Figure 16.6: Resource Patterns in MAML

16.5 Model-to-Model Transformation

MAML models provide integrated representations for process-driven apps but need to be created from scratch. On the other hand, business processes in many organization are documented in BPMN models, although they lack the ability to fully describe mobile applications. To exploit the benefits of interoperability, a model-to-model transformation is presented based on the QVT-O language [The16].

16.5.1 Mapping of Equivalent Language Constructs

Many core elements of the notations have direct correspondences, for instance in the thesis organization system depicted in Figure 16.1 and Figure 16.2. Firstly, both notations have the concepts of roles that can represent individuals or groups of resources. A MAML *use case* is represented as *pool* with a *lane* for each role in BPMN. Conversely, a process element is annotated with the respective *participant*.

Secondly, *user tasks* in BPMN correspond to *process elements* in MAML. A keyword-based matching strategy tries to classify the more specific MAML task type (e.g., *Create entity*) according to its description. Further mappings include:

- *Sub-process tasks* and *service tasks* have equivalent MAML elements named *Include* and *Webservice*
- *Manual tasks*, i.e. application-external actions, can be regarded as *Display message* steps which only output information and allow to proceed
- *Script tasks* indicate automated actions by the process engine. They need to be converted to web services as no arbitrary code can be executed.
- *Call activities* are created only for MAML's location retrieval, camera, and phone calls steps to reflect these global tasks. Other BPMN call tasks are transformed like regular sub-process tasks.

Thirdly, *start*, *end*, *timer*, and *error events* have the same semantics in both notations. Regarding data flows, this also holds true for inputs and outputs using *data associations* in BPMN and *parameter connectors* in MAML.

16.5.2 Mapping of Related Language Constructs

MAML applications are data-driven and therefore consider data flows together with the process flow such that each process element operates on one or multiple data items which are persisted and globally available after the process completes. Within the transformation, the respective MAML *data sources* need to be retrieved for each activity and annotated as BPMN *data store*. In addition, corresponding *InputSets* and *OutputSets* for an activity's *InputOutputSpecification* need to be created. However, nested attributes cannot be represented in BPMN and are lost during the transformation.

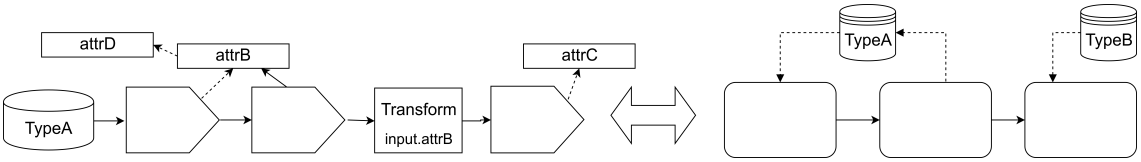


Figure 16.7: Data Transformation Between MAML (left) and BPMN (right)

Conversely, data stores associated with BPMN tasks need to be integrated in the MAML process flow. In case the data item to process differs between tasks, additional *Transform* elements are needed to establish the link (see Figure 16.7).

Without direct support for concurrency, several workflow patterns can be rewritten to concepts available in MAML. *Parallel gateways* and (equivalent) *implicit parallel splits* are for example transformed to a sequence of activities as depicted in Figure 16.8 (a). With exclusive gateways to branch process flows, inclusive gateways are mapped to a series of optional steps (b). Similarly, loops (WRP21; (b/c) in Figure 16.4) and milestones can be transformed using automatically evaluated, state-based decisions.

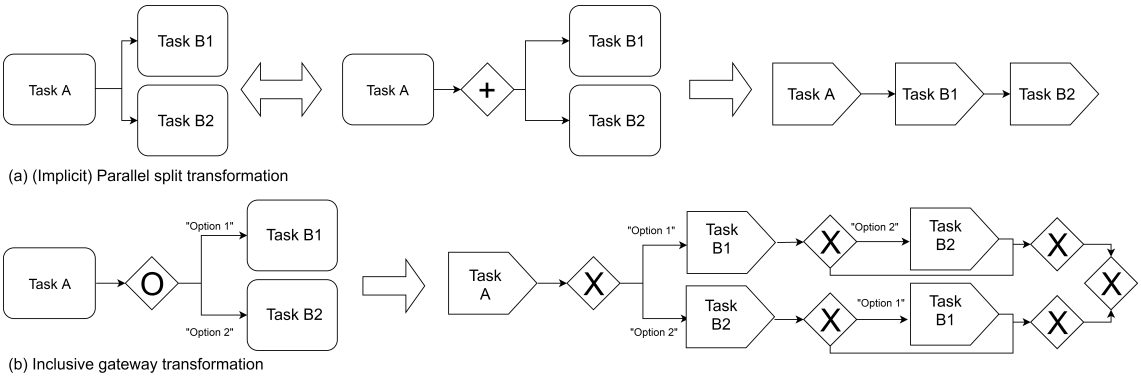


Figure 16.8: Gateway Transformation from BPMN to MAML

16.5.3 Unmapped Language Constructs

BPMN contains several elements without equivalent representation in MAML. Because unstable mobile connectivity precludes concurrency, BPMN's *conversations* and *choreographies* are un-

supported in MAML. In addition, *multiple instance tasks*, *(parallel) multiple events*, and *message flows* are not available in MAML and cannot be mapped to equivalent concepts because of the inherently sequential design of smartphone apps. Similarly, protocols for BPMN's *transactional tasks*, *compensation*, and *cancel events* contradict the distributed characteristics of mobile systems.

At the current state of development, MAML also does not contain complex control flows for two reasons. Firstly, the model-driven approach aims to actually create executable source code. Features equivalent to BPMN's *signal*, *escalation*, or *termination events* and respective *event-based gateways* need to have intuitive user interactions in the app. Secondly, some elements in BPMN such as *complex gateways*, *business rule tasks*, and *compensation* are less structured or delegate processing to a workflow engine. However, each element requires an adequate in-app representation and a sufficient level of detail in the model to allow for an automated transformation.

On the other hand, some elements in MAML have no adequate representation in BPMN. Most important, custom data types and data objects with nested attribute structures reach far beyond BPMN's capability of specifying primitive key-value pairs as properties of activities. For the same reason, data-driven features such as *computed attributes* and data-dependent variations unknown at design time (*SingleResultEvent*/ *MultiResultEvent*) have no equivalent representation in BPMN. Also, interface-related information such as labels, buttons, and field captions is not in the scope of BPMN.

16.6 Discussion

Section 16.4 explained the capabilities of MAML and contrasts them to the established BPMN notation. The transformation result of the BPMN model in Figure 16.1 is displayed in Figure 16.2 (enriched with label texts). As can be seen, many concepts are shared by both notations, although MAML does not strive for being a feature-complete workflow engine. Especially the data layer is currently not well represented in BPMN and has no direct interrelation with data modeling languages such as UML. Workflow engines such as Camunda bypass some of the limitations with custom extension attributes, but data-centric actions still need to be extracted into programmed web services. In addition, the granularity of modeled processes varies strongly and even with a detailed representation, the modeler might often apply mobile-specific adaptations to the representation and enrich the model with UI elements. We therefore refrain from extending BPMN for our needs. Instead, our transformation eases the import of potentially already existing process documentation and mobile app design. Conversely, small enterprises which have only process documentation on a higher level of abstraction can reuse models used for app generation and export a detailed BPMN representation. As the model-driven paradigm focuses on this primary artifact, the process documentation always remains synchronized with the actual implementation, both regarding process flows and data models which can be derived together with access restrictions directly from the MAML representation (cf. [RK18] for more details on the data model

inference). Particularly for small and medium enterprises, MAML can therefore be seen as a tool supporting digitized business models and increasing quality of operations through custom business apps – eventually even designed by end users within the company. A transformation from and to BPMN thus unburdens departments by simplifying the specification of software and documentation of activities.

Limitations of our work include both improvements to the transformation between BPMN snd MAML as well as its practical application. On the one hand, the evolution of the domain-specific MAML notation may introduce workflow patterns already found in BPMN. Also, previously unmapped language constructs with no adequate representation might be substituted by app-related concepts in future (e.g., establishing conventions for utilizing event-based gateways in the context of notifications). On the other hand, a real-world use in companies with different engagement in process documentation is required to further validate the applicability of the approach and constitutes future work. By analyzing existing models of different granularities, recurring patterns may be uncovered that can also be used for improving the transformation.

16.7 Conclusion and Outlook

In this paper, the interoperability between the established process modeling notation BPMN and the domain-specific graphical modeling language MAML for mobile-app creation have been assessed. Therefore, MAML was analyzed based on 43 control flow, 40 data, and 43 resource patterns identified for workflows by Russell et al. [Rtv04; Rus+05; Rus+06]. Although many correspondences exist with regard to control flow, data, and resource handling, differences relating to concurrency, complex control flows, and transactional behavior remain due to different application domains of BPMN using synchronous workflow engines and MAML in distributed mobile environments. Nonetheless, a mapping between many concepts can be found and an implemented model-to-model transformation using QVT-O demonstrates the transferability of concepts. From MAML to BPMN, fully specified models are created. In the opposite direction, additional data- and UI-related elements need to be added due to their absence in the specification of BPMN. Thus, the development process of mobile business apps using MAML can indeed be supported by reusing process knowledge documented in BPMN models. Regarding future work, a real-world introduction of the approach is pending in order to validate the transformation rules, especially considering the detailedness of input models encountered in practice.

References

[Bar+15] Scott Barnett et al. “A Multi-view Framework for Generating Mobile Apps”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 305–306. DOI: 10.1109/VLHCC.2015.7357239.

[BBF10] Marco Brambilla, Stefano Butti, and Piero Fraternali. "WebRatio BPM: A Tool for Designing and Deploying Business Processes on the Web". In: *ICWE*. Springer, 2010, pp. 415–429. DOI: 10.1007/978-3-642-13911-6_28.

[Bra+08] M. Brambilla et al. "Business Process-Based Conceptual Design of Rich Internet Applications". In: *ICWE*. 2008, pp. 155–161. DOI: 10.1109/ICWE.2008.22.

[Cam17] Camunda Services GmbH. *BPMN Workflow Engine*. 2017. URL: https://camunda.org/ (visited on 03/08/2018).

[CMS12] E. F. Cruz, R. J. Machado, and M. Y. Santos. "From Business Process Modeling to Data Model: A Systematic Approach". In: *QUATIC*. 2012, pp. 205–210. DOI: 10.1109/QUATIC.2012.31.

[Dec+08] Gero Decker et al. "Transforming BPMN Diagrams into YAWL Nets". In: *Business Process Management*. Ed. by Marlon Dumas, Manfred Reichert, and Ming-Chien Shan. Springer, 2008, pp. 386–389.

[Gia+17] Giuseppe de Giacomo et al. "Linking Data and BPMN Processes to Achieve Executable Models". In: *CAiSE*. Ed. by Eric Dubois and Klaus Pohl. Springer, 2017, pp. 612–628. DOI: 10.1007/978-3-319-59536-8_38.

[Kal+14] Audris Kalnins et al. "DSL Based Platform for Business Process Management". In: *SOFSEM*. Springer, 2014, pp. 351–362. DOI: 10.1007/978-3-319-04298-5_31.

[Kan+16] Udo Kannengiesser et al. "Modelling the Process of Process Execution: A Process Model-Driven Approach to Customising User Interfaces for Business Process Support Systems". In: *BPMDS*. Springer, 2016, pp. 34–48. DOI: 10.1007/978-3-319-39429-9_3.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14. DOI: 10.19210/OJIS_2015v2i2n02_Majchrzak.

[Obj11] Object Management Group. *BPMN*. 2011. URL: http://www.omg.org/spec/BPMN/2.0.

[Rec10] Jan Recker. "Opportunities and constraints: The current struggle with BPMN". In: *Business Process Management Journal* 16.1 (2010), pp. 181–201. DOI: 10.1108/14637151011018001.

[Rie18a] Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *HICSS*. 2018, pp. 5725–5734.

[Rie18b] Christoph Rieger. "Interoperability of BPMN and MAML for Model-Driven Development of Business Apps". In: *Business Modeling and Software Design*. Ed. by Boris Shishkov. Cham: Springer International Publishing, 2018, pp. 149–166. DOI: 10.1007/978-3-319-94214-8_10.

[RK18] Christoph Rieger and Herbert Kuchen. "A process-oriented modeling approach for graphical development of mobile business apps". In: *COMLAN* 53 (2018), pp. 43–58. DOI: 10.1016/j.cl.2018.01.001.

[Rtv04] N. Russell, A.H.M. ter Hofstede, and W.M.P. van der Aalst. *Workflow Resource Patterns. BETA Working Paper Series, WP 127*. Ed. by Eindhoven University of Technology. Eindhoven, 2004.

[Rus+05] Nick Russell et al. "Workflow Data Patterns: Identification, Representation and Tool Support". In: *Conceptual Modeling – ER*. Springer, 2005, pp. 353–368. DOI: 10.1007/11568322_23.

[Rus+06] N. Russell et al. *Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22*. 2006.

[The16] The Eclipse Foundation. *QVT Operational*. 2016. URL: http://www.eclipse.org/mmt/qvto (visited on 12/03/2016).

[TK08] Hallvard Trætteberg and John Krogstie. "Enhancing the Usability of BPM-Solutions by Combining Process and User-Interface Modelling". In: *PoEM*. Springer, 2008, pp. 86–97. DOI: 10.1007/978-3-540-89218-2_7.

[Tuo+07] M. Tuomainen et al. "Model-centric approaches for the development of health information systems". In: *Studies in Health Technology and Informatics* 129 (2007).

[van94] W. M. P. van der Aalst. "Putting high-level Petri nets to work in industry". In: *Computers in Industry* 25.1 (1994), pp. 45–54. DOI: 10.1016/0166-3615(94)90031-0. URL: http://www.sciencedirect.com/science/article/pii/0166361594900310.

[Woh+06] P. Wohed et al. "On the Suitability of BPMN for Business Process Modelling". In: *BPM*. Springer, 2006, pp. 161–176. DOI: 10.1007/11841760_12.

[Wor17] Workflow Patterns Initiative. 2017. URL: http://www.workflowpatterns.com/.

17

Challenges and Opportunities of Modularizing Textual Domain-Specific Languages

Table 17.1: Fact sheet for publication P11

Title	Challenges and Opportunities of Modularizing Textual Domain-Specific Languages
Authors	Christoph Rieger[1] Martin Westerkamp[1] Herbert Kuchen[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2018
Conference	6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)
Copyright	SciTePress CC BY-NC-ND 4.0
Full Citation	Christoph Rieger, Martin Westerkamp, and Herbert Kuchen. "Challenges and Opportunities of Modularizing Textual Domain-Specific Languages". In: *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. Scitepress, 2018, pp. 387–395. DOI: 10.5220/0006601903870395

Challenges and Opportunities of Modularizing Textual Domain-Specific Languages

Christoph Rieger Martin Westerkamp Herbert Kuchen

Keywords: Domain-Specific Language, Modularization, Xtext, Language Composition

Abstract: Over time, domain-specific la nguages (DSL) tend to grow beyond the initial scope in order to provide new features. In addition, many fundamental language concepts are reimplemented over and over again. This raises questions regarding opportunities of DSL modularization for improving software quality and fostering language reuse – similar to challenges traditional programming languages face but further complicated by the surrounding editing infrastructure and model transformations. Mature frameworks for developing textual DSLs such as Xtext provide a wealth of features but have only recently considered support for language composition. We therefore perform a case study on a large-scale DSL for model-driven development of mobile applications called MD², and review the current state of DSL composition techniques. Subsequently, challenges and advantages of modularizing MD² are discussed and generalized recommendations are provided.

17.1 Introduction

Domain-specific languages (DSL) have emerged for various purposes [MHS05]. Despite their unique capabilities, shared fundamental concepts are rarely reused but are re-implemented for every new language. In the context of Model-Driven Software Development (MDSD) which tries to automate the process of software creation, this causes a frequent re-implementation of similar functionality. In recent years, modularization of DSLs has become a topic of increasing interest in academia due to the expected improvements on software quality [Pes+15; CV16]. Tightly coupled to the concepts of language evolution, the composition of languages introduces a variety of opportunities regarding maintainability, reusability, and extensibility [CP10]. For example, changes to language features can be performed in isolation, requiring one to update and rebuild only parts of the language [Vac+14].

Whereas a variety of DSL development frameworks have evolved in the past, support for language composition varies in practice [Erd+15]. On the one hand, development frameworks specifically tailored to language modularization often emerged from niche projects and lack features such as sophisticated Integrated Development Environment (IDE) support [Vac+14; EH07]. On the other hand, language workbenches used in practice often provide a broad set of features for developers and modellers but consider DSL composition as a negligible feature.

For example, the mature Xtext framework is widely used for developing textual DSLs and provides seamless IDE integration. However, modularization capabilities such as grammar inheri-

tance are still limited. One objective of this work is to analyse capabilities concerning language modularization in Xtext and the resulting implications for DSL developers.

The analysis is based on a prototypical implementation of a modularized DSL. With Model-Driven Mobile Development (MD²), Heitkötter and Majchrzak (2013) presented a cross-platform development approach to create data-driven business apps for smartphones and tablets. MD² generates native apps for multiple platforms, providing a native look and feel, access to device sensors, and a high level of abstraction for modellers. Whereas the textual DSL facilitates modularization by utilizing the Model-View-Controller (MVC) pattern [Ern+16] for its models, the framework itself is not aligned with this structure but implemented in a monolithic project. In order to improve maintainability and extensibility, an enhanced framework architecture is proposed for MD² with respect to its DSL design and tooling.

The contributions of this work are threefold. Firstly, the paper reviews language modularization approaches in the field of external domain-specific languages. Secondly, modularization capabilities and limitations of the Xtext framework are further investigated, using a case study of the large-scale MD² DSL. Thirdly, we generalize the observed benefits and drawbacks and propose recommendations for modularizing existing DSLs. The structure of this paper follows these contributions. Based on related work presented in Section 17.2 and general modularization concepts in Section 17.3, Section 17.4 provides the case study. The findings are then generalized and discussed in Section 17.5 before concluding in Section 17.6.

17.2 Related Work

Introducing formal models as a higher level of abstraction permits domain experts to express their requirements using semantics close to the notation known within the domain, usually using either a textual or graphical syntax [Völ13].

External DSLs are independently developed languages and separate to any host-language, therefore often providing a custom syntax specifically crafted according to domain experts' requirements [Fow05]. Since they are independent, appropriate tools such as linkers, parsers, compilers, or interpreters need to be provided [Völ13]. *Internal DSLs* are encapsulated into a General Purpose Language (GPL) and consequently use the same syntax. However, they utilize only a subset of its features to create domain abstractions [Fow05]. *Language Workbenches* offer a custom IDE, specifically designed to the development and usage of DSLs. The IDE becomes an integral part of model-processing, blurring the lines between programming and modelling [Fow05].

In contrast to GPLs, DSLs are designed in line with a (potentially changing) domain scope and software systems need to cope with changes in their environment [CP10]. Moreover, DSLs are often iteratively developed, e.g., when deficiencies in the language design are exposed or a DSL is intentionally designed to cover only the most important domain concepts in the beginning, with further components to be developed in the future [Völ13]. Additional complexities arise when

downward compatibility to previous versions should be provided through techniques such as deprecation markers [Völ13].

Consequently, the attempt to enhance maintainability during the ongoing evolution of a language is a major driver for DSL modularization. Internal DSLs use the extensibility of a host language, e.g., macros in Lisp or metaprogramming in C++ . They, however, lack DSL-specific tooling support as well as a customizable syntax [Fow05]. Therefore, we focus on external DSLs as well as Language Workbenches in the following.

In recent years, the implementation of modular external DSLs has become a subject of increasing research interest [EH07; KRV10; CV16]. Subsequently, a variety of development frameworks have been presented that support the creation of necessary tooling such as parser generation, implementing generators, and supplying IDE integration [Erd+15]. With *Neverlang*, Cazzola and Poletti have introduced a language development framework that specifically focuses on creating reusable DSLs [CP10]. In Neverlang, modularization is organised along two dimensions. Language features are defined individually in modules, containing the syntax definition in Backus-Naur-Form (BNF) notation and an arbitrary amount of so-called roles describing the semantics [Vac+14]. Whereas Neverlang provides sophisticated modularization techniques such as traits [CV16], it does not provide IDE integration for generated DSLs, hampering its adoption by domain experts.

A DSL development framework that supports the entire MDSD process including language definition, generator implementation, and IDE integration, is *MontiCore*. The framework generates Eclipse plug-ins for DSLs which support syntax highlighting, foldable code regions, and error messages. Flexible language composition is enabled through interfaces and multiple inheritance. MontiCore includes external fragments at runtime so that modellers are capable of utilizing arbitrary DSLs when modelling. While a variety of template-engines are available for MontiCore, the framework provides a native engine that facilitates agile development by introducing tags within the source code which are implementable in later iterations [KRV10]. Although the framework is under active development, no supporting community has established yet, leaving its future maintenance uncertain.

Being part of the Eclipse Modeling Project, the *Xtext*[1] framework's evolution and support is backed by a comparably large community. It provides sophisticated tooling support such as IDE integration but is not specifically tailored towards language composition (cf. Section 17.4). For code generation, Xtext advocates Xtend[2], a Java-like programming language that additionally supplies developers with template expressions to ease the implementation of generators.

[1]Xtext – http://www.eclipse.org/Xtext/documentation/
[2]Xtend – https://www.eclipse.org/xtend/

17.3 DSL Modularization Concepts

A large amount of design patterns for domain-specific languages have been presented in literature [Spio1; KRV10]. As this work focuses on modularizing external DSLs, six applicable modularization techniques are visualized in Figure 17.1.

Language extension allows new features to be added to an existing language [Spio1]. The novel DSL inherits from the base language, including its semantics and syntax. Being closely related to object-oriented forms of inheritance, language extension is generally not limited to single inheritance, but also allows obtaining features from multiple DSLs. However, due to the threat of possible conflicts, mainly single inheritance is used in practice [Duc+06].

Mixins represent a special form of language extension. Unlike multiple inheritance, mixins are not tied to a particular super class in the type hierarchy. Instead, a mixin provides a self-contained increment of functionality and specifies its dependencies (to classes or other mixins). Concepts defined in a mixin can therefore be used by multiple classes, thus enabling a more flexible class composition. During language compilation, the type hierarchy is linearised such that all language dependencies are resolved using single inheritance [BC90].

Language specialization represents the counterpart to language extension. Rather than extending a DSL, unnecessary parts of a language are removed, creating a novel DSL. It thereby comprises a subset of the former language elements [Spio1].

Pipeline pattern In contrast to language extension and specialization, the language modules in this pattern are on the same hierarchical level. Each language handles a set of language elements and passes the rest to the next one, so that a language's output is the input for the next language in the pipeline. Pipelining thus encourages the separation of tasks and discourages the use of too feature-rich languages [MHS05].

Trait Only recently, the use of traits has been proposed for DSL composition [CV16]. Similar to an interface with method implementations, a trait is a collection of methods and attributes. Nonetheless, a trait is stateless by definition and does not enforce an order of composition, enabling a more flexible reuse across classes compared to inheritance. In language composition, traits provide or extend language constructs, and conflicts are explicitly disambiguated by the target language. In contrast to mixins, a trait only provides reusable functionality without affecting the type hierarchy and respective semantic implications [Duc+06].

Aspect Similar to traits, aspects provide composable units of functionality which are independent of their order and enable a flexible architecture design. However, in contrast to traits, aspects are not invoked within the target class, but an aspect itself declares *pointcuts*, i.e., the set of events during program execution for which the aspect's action should be executed [WKD04]. This composition mechanism requires a language processor, the so-called *aspect weaver*, which is used to resolve the composition of components and aspects. With respect to language composition, there are two general approaches to implement aspects. Firstly, aspects can be defined in correspondence with generated GPL code as long as an aspect weaver for the target language

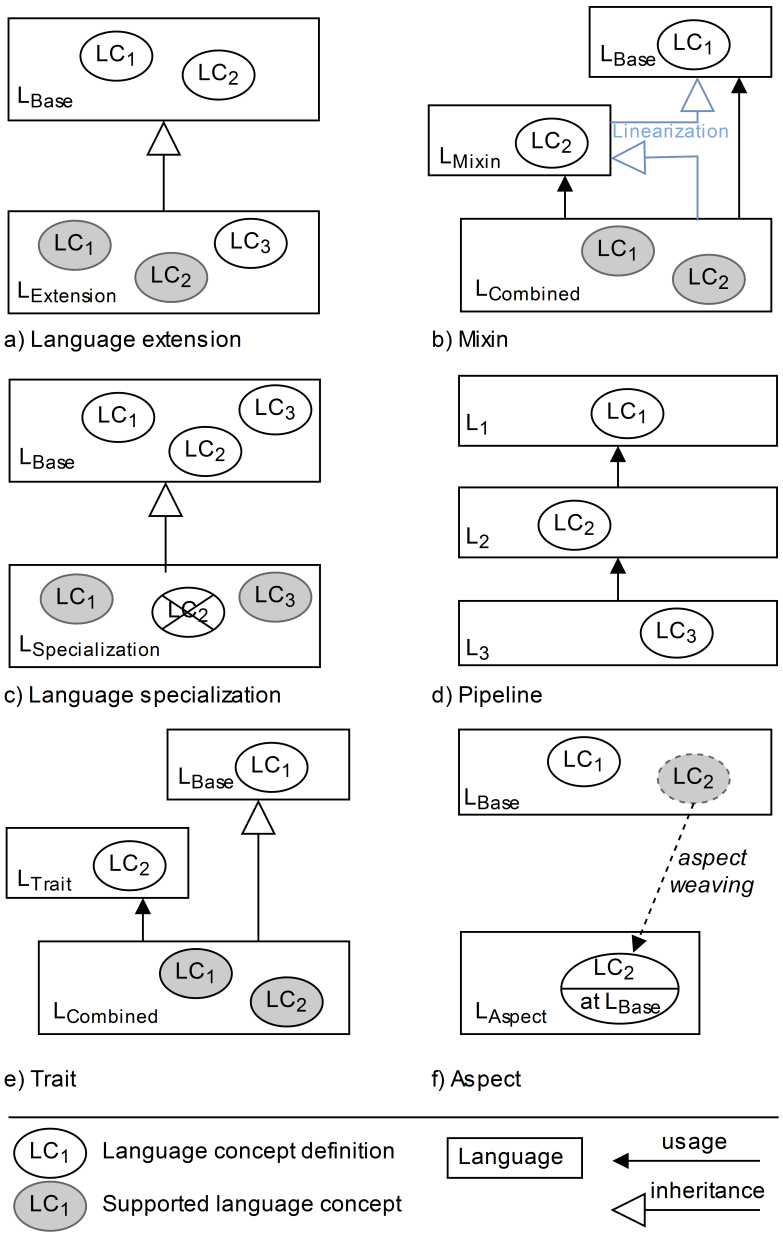


Figure 17.1: Language Modularization Concepts

exists. Moreover, the developer needs detailed knowledge about the generator to define pointcuts. Secondly, defining aspects on a grammar-level provides the possibility to declare pointcuts according to a DSL's structure [Wu+05]. Yet, such an approach requires a transformation engine which operates as aspect weaver.

17.4 Modularization in Xtext

In this section, we present a case study on a large DSL called MD² in order to demonstrate the practical implications of current modularization capabilities of DSLs created using the language workbench Xtext.

17.4.1 Modularization Concepts in Xtext

Whereas a large variety of modularization concepts exists for DSLs in general (cf. Section 17.3), Xtext's support for language extension is limited. As depicted in **??**, grammars can extend other grammars using the `with` keyword (line 1). The inheriting grammar may extend existing language concepts, replace rule definitions, or introduce new ones. However, Xtext only supports single inheritance.

```
1 grammar de.md2.View with de.md2.Md2Basics
2 import "http://md2.de/Model" as model
3
4 OptionInput: 'Option' name=ID
5         widgetInfo
6         'options' values = [model::Enum]
7 fragment widgetInfo:
8         'label' labelText = STRING
9         'tooltip' tooltipText = STRING
```

Listing 17.1: Language Inheritance and Grammar Mixins

In addition, Xtext provides a feature called *grammar mixins*. In contrast to the modularization concept with the same name presented in the previous section, any metamodel can be used as Xtext mixin using the `import` keyword (line 2). When importing a metamodel, elements may then be *referenced* across models. However, Xtext does not actually employ the referenced grammar, but its metamodel. Therefore, it is not possible to directly access rules from these referenced grammars. Consequently, its syntax is not available within the importing DSL to *define* new elements of the imported class in the including language. For example, a modeller adding an `OptionInput` element can reference a list of values provided in a distinct model file (according to line 6) but cannot create an Enum object directly in the View model.

Finally, the concept of *fragments* is another instrument for enhanced reusability which has been added recently[3]. Instead of repetitively declaring a similar syntax in multiple parser rules, common parts can be extracted into a fragment and integrated by multiple rules. For example, the fragment `widgetInfo` (line 7) specifies the syntax of two attributes and is included in the

[3]http://zarnekow.blogspot.de/2015/10/the-xtext-grammar-learned-new-tricks.html

`OptionInput` rule (line 5). Although this concept introduces multiple inheritance on the level of model elements, it is only available within a single grammar and cannot be used across languages.

17.4.2 Case Study on MD²

Cross-platform development frameworks aim to mitigate the redundancy of developing applications for multiple platforms. However, hybrid apps based on web technologies lack a native look & feel and do not provide an additional level of abstraction beyond a common Application Programming Interface (API), e.g., regarding platform-dependent design guidelines [HMK13]. The Model-Driven Mobile Development (MD²) framework abstracts from the low-level implementation of business apps – i.e. form-based, data-driven apps interacting with back-end systems [MEK15] – and allows for modelling the desired result in a platform-agnostic fashion [HMK13]. From this model, the framework currently generates native source code for the Android and iOS platform as well as a Java-based back-end application.

Models designed using the MD² DSL follow the Model-View-Controller (MVC) pattern, extended by an additional workflow layer [Ern+16]. Workflows can trigger workflow elements defined in the controller, while conversely the controller can fire a workflow's events as illustrated in Figure 17.2.

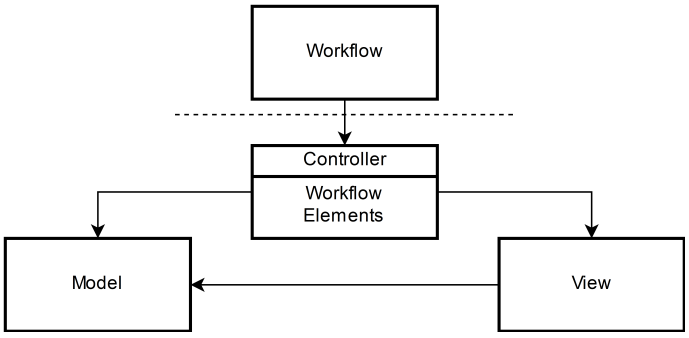


Figure 17.2: Architecture of MD² Models

In contrast to the subdivided design of MD² models, the DSL itself is defined in a single Xtext grammar, roughly grouped into Model, View, Controller, and Workflow components which reference each other. However, some cases exist in which this hierarchy is bypassed. For example, the *AutoGenerator* feature is used to automatically derive a generic view representation from a given data structure. This feedback between view layer and content provider (located in the controller component) causes a circular dependency which needs to be considered while restructuring the language.

The purpose of MD²'s modularization is therefore twofold. First and foremost, it should result in a framework that provides enhanced maintainability and increased software quality. Resulting from a series of modifications and extensions, the DSL was not well structured and interrelations were hard to grasp for developers due to the sheer size of the language. Before modularizing,

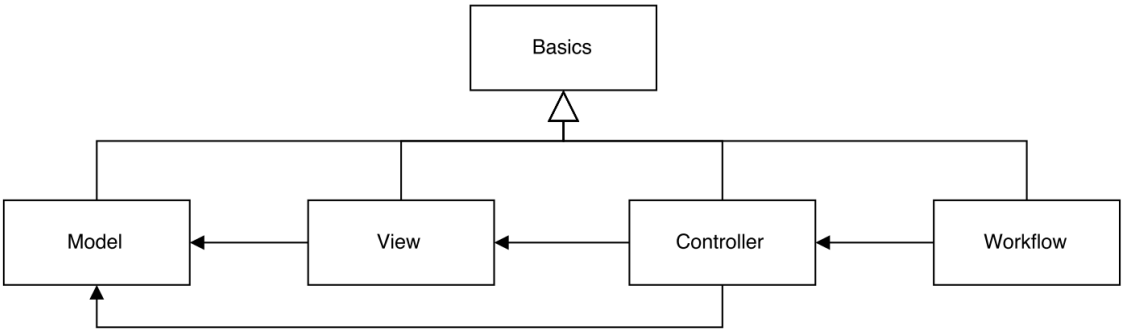


Figure 17.3: Proposed Module Structure for MD² Models

the Xtext file comprised 924 lines of code organized in 188 grammar rule definitions. This makes MD² one of the five largest Xtext-based DSLs (in lines of code and file size) published on Github. However, the initial structure contradicted the MVC approach enforced in MD² models, and DSL evolution was seriously hampered by this complexity. For example, evaluations of the framework have unveiled a lack of abstraction in the DSL's View layer [Vau+14] which could be tackled more efficiently when dependencies to other components of MD² are clear.

Second, MD² is tailored to the domain of data-driven business apps [HMK13]. Future users may wish to create DSLs tailored to their subdomains or even organisations. This can be achieved by extending or specializing reusable modules. In addition, the creation of multiple (technical) sub-DSLs can remedy current deficiencies and foster a more agile and targeted evolution of language features.

To achieve these aims, a clear separation of concerns is mandated on DSL level and further components dealing with code generation to trace concepts along the processing chain, indicate dependencies, and clarify interrelations. In order to focus this work on the modularization of the DSL itself, the existing generators were adapted to the new project structure without formally subdividing them into submodules.

17.4.3 Modularizing MD²

Complying with the structure of the MD²-DSL and corresponding reference architecture [Ern+16], the MVC+Workflow pattern is adopted to decompose the monolithic MD² grammar into four sub-DSLs. In addition, a fifth *Basics* DSL is introduced as common infrastructure.

Inheritance-Based Modularization

Each module is a domain-specific language on its own but does not exist in isolation as visualized in Figure 17.3. The *Model* module only relies on *Basics* and provides the MD² type system, rules for modelling custom entity structures, and typed parameter definitions to be used by other modules. While the *View* module depends on Model in order to reference data types, the *Controller* relies

```
grammar de.md2.Basics

MD2Model:
      package = PackageDefinition &
      model = LanguageElement?;
LanguageElement:
      {LanguageElement};
PackageDefinition:
      'package' pkgName = QUALIFIED_NAME;
```

Listing 17.2: Basics Grammar Defining Interfaces

```
grammar de.md2.Model with de.md2.Basics
import "http://md2.de/Basics" as basics

MD2Model returns basics::MD2Model:
      super
      model = Model;
Model returns basics::LanguageElement:
      {Model} modelElements += ModelElement+;
```

Listing 17.3: Model Grammar Implementing Interfaces

on both Model and View such that the modeller can define custom actions linking data objects with the desired representation. The *Workflow* module only depends on Controller, since it builds workflow paths from low-level process steps.

Sophisticated modularization techniques such as multiple inheritance are not available in Xtext. However, it permits reusing grammars to a certain extent through single inheritance (see Subsection 17.4.1). In theory, unidirectional dependencies between sub-DSLs could be recompiled into a chain of DSLs only using single inheritance, similar to the concept of mixins (see Section 17.3). Nevertheless, automation would be required to build a (temporary) inheritance structure from the dependency graph each time any of the modules changes, and adapt the IDE and generator code accordingly. We therefore decided to avoid this approach and interrelate multiple DSLs differently.

Interface-Based Modularization

In contrast to single inheritance, more than one DSL can be imported in Xtext. As explained in Subsection 17.4.1, the flexibility of importing multiple sub-DSLs comes at the cost of modelling components in multiple files according to the module in which the rules are located, and reference the respective objects.

To achieve a coherent structure across multiple DSLs, Voelter and Solomatov (2010) suggest the utilization of interfaces. The modularized MD² DSL creatively combines standard Xtext features to create such extensible interfaces as depicted in **??** and Listing 17.3. Firstly, the concept of *unassigned rule calls* (line 7 in **??**) forces the instantiation of rules. As no further attributes are specified, the rule is effectively transformed into an empty interface. Secondly, the `returns` keyword influences the meta model by explicitly merging the inferred class with the given type. A common use case for this feature is the definition of expressions such that the actual subtype is transparent to referencing elements.

For example, the `Model` rule (line 7 in **??**) creates valid *Basics::LanguageElement* objects, effectively implementing the imported interface. Thirdly, the `super` keyword (line 5 in **??**) provides all contents of the inherited rule to the implementing rule. Therefore, *Basic::MD2Model*'s attributes are available in the corresponding rule of the *Model* grammar and can be overwritten. Together, the concept of interfaces is emulated in Xtext not only within a single language but also across DSLs.

Modularizing Bidirectional Dependencies

Beyond the presented linear dependency structure, intertwined components can also be extracted into separate sub-DSLs. As mentioned before, the *AutoGenerator* feature is such an example that provides a view element but relies both on the Model and Controller module. During the language generation process, Xtext is not able to resolve bidirectional language references. Extracting the problematic feature is possible by introducing a separate module that implements a new *LanguageElement* subtype complying with the aforementioned interface principles. The new structure as visualized in Figure 17.4 inevitably creates circular dependencies between those DSLs. However, the metamodel of each DSL can be built independently by avoiding bidirectional dependencies. Using cross-language references via `imports`, other grammars can access the respective target normally and the resulting cycle is resolved soundly by Xtext.

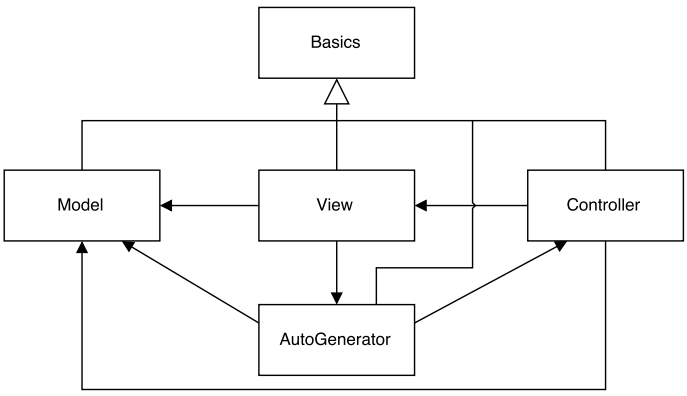


Figure 17.4: Resolving Bidirectional Dependencies

Domain Extension

Domain extension is a second type of extension to the presented modular structure. The modularization of MD², as described to this point, mainly provides benefits from a DSL developer's perspective. The separated modules retain clear responsibilities and dependencies, easing future development of the framework. Nevertheless, it could be extended to specialized domains or applied to specific organizations which bring along new requirements by adapting MD² to the target domain, thus achieving a variable scope of the language [VS10]. Instead of creating new modules that inherit from the Basics grammar, existing modules can be reused through inheritance. For example, a newly introduced *ViewAddon* module inherits from the View grammar to add a new type of view element as depicted in Figure 17.5.

The add-on's grammar needs to implement the common root of MD², i.e. *MD2Model*. Note that in this case, the `super` call (line 6 in **??**) does not refer to the rule in Basics but to the inherited definition within the View module. In order to extend a specific rule, e.g., adding a specific view element, a new rule alternative can be provided for the definition of a *ContentElement* (lines 7-8). Nevertheless, the original implementation is maintained by delegating other model inputs to the super language implementation. Conversely, rules may limit which super language rules can be referenced, or place inherited elements arbitrarily in the derived language's structure without affecting the inherited grammar.

17.4.4 Advantages and Disadvantages of Modularization

In this section, the results of the modularization are discussed with regard to the implications for MD² modellers and DSL developers.

Modelling experience DSL modularization should minimize changes to the language's scope and its usage for modellers. It can be observed that the scope of the modularized MD² DSL has not changed and all language concepts could be transferred. Because the language syntax

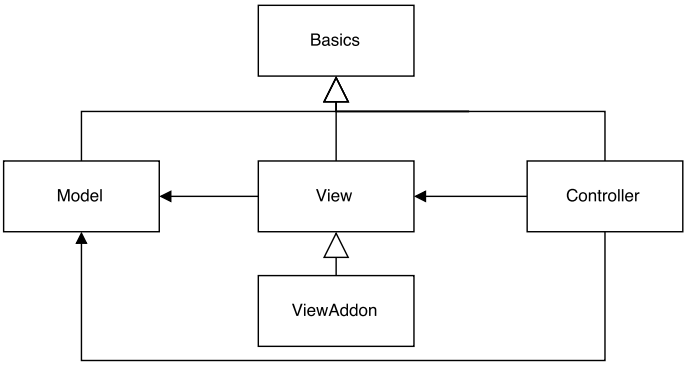


Figure 17.5: Domain Extension

```
grammar de.md2.ViewAddon with de.md2.View
import "http://md2.de/View" as view
import "http://md2.de/Basics" as basics

MD2Model returns basics::MD2Model:
      super;
ContentElement returns view::ContentElement:
      super | AddedButton;
AddedButton: ...
```

Listing 17.4: ViewAddon Grammar

was not modified, three structural metrics were chosen from a plethora of grammar-related metrics in order to asses the overhead of the modularization [Čre+10]. Table 17.2 compares the different DSL sizes using the number of grammar rules, non-comment and non-blank lines of code (NCLOC), and the amount of (non-comment) characters in each DSL grammar. As can be seen, the modularization process incurs a slight overhead of about 5% lines of code due to the declaration of imports and the specification of interfaces. However, complexity is greatly reduced compared to the original DSL specification with 2087 lines (including comments). Admittedly, the resulting six DSLs represent only a first high-level separation of concerns. However, the effects of the modularization will amplify when complex constructs of the large *Controller* and *View* modules are further broken down.

Table 17.2: DSL Comparison Metrics

(Sub-)DSL	Rules	NCLOC	Characters
Original MD²	188	924	30.208
Basics	26	93	1.893
Model	20	97	3.498
View	62	301	8.319
Controller	85	421	16.198
Workflow	8	31	1.050
AutoGenerator	5	27	955
Modular MD²	206	970	31.913
Difference	+9.6 %	+5.0 %	+5.6 %

However, as drawback of the extension approach, new model files (and file types) are required for each type of extension such as *AutoGenerator* model elements. This restriction is acceptable for large modules (such as enforcing MVC separation on file level) but gets inconvenient in case of multiple small additions.

Language development With regard to DSL development, several advantages can be observed. Firstly, splitting the large-scale DSL makes the framework more maintainable. Both MD² and Xtext evolve over time, therefore some parts of the implementation became outdated but could not be replaced because of the unknown implications of such actions. Also, the underlying functionalities for validation, preprocessing, and source code generation are untangled. Therefore, deprecated and irrelevant code can be better spotted and safely deleted without fearing side-effects on other parts of the framework.

Secondly, the separation of concerns will likely improve development speed and quality of sub-DSLs. Instead of managing the whole language scope, particular sub-DSLs can evolve separately. This does not only apply to language features but also includes tool support for validation, formatting, code completion, etc. Also, developing generators does not require comprehensive knowledge about MD² anymore but can focus on the transformation of specific domain concepts to the respective target platform.

Finally, the modularization prepares the language for future developments. New sub-DSLs can introduce new or extend existing language concepts. Also, language reuse in different DSLs is possible. However, the complete replacement of a language with an existing DSL in the same domain is currently not easily possible because of the interface-based language structure. Yet, considering the complex interrelations within generated code it is questionable whether a replacement is actually desired.

Modularizing MD² also introduces some drawbacks for language developers: The grammar of each sub-DSL is simplified at the cost of fragmentation in the overall framework structure. In particular, each new DSL in Xtext is based on five Eclipse projects for grammar definition, editor integration, and unit tests. The current module structure of six DSLs already results in a set of 30 projects and will quickly rise if new extensions and add-ons are introduced. Managing the configuration – including dependencies, DSL versions, and overall language bundling – needs to be automated using build tools such as Gradle[4].

From a conceptual perspective, a balance between core language features and extensions needs to be found. Language specializations may be tailored to the specific environment but such adjustments ideally do not require any change to the core language to avoid compromising on the reusability of the host language. Furthermore, the composition of domain extensions is based on grammar inheritance and therefore prone to the same problems of manually maintaining inheritance chains. As a consequence, introducing a multitude of minor DSL extensions is currently not advisable.

[4]Gradle Build Tool – https://gradle.org/

17.5 Discussion

Beyond MD²-related advantages and disadvantages derived from the case study, the generalized reflections on the results are presented as recommendations for the modularization of DSLs in a broader context.

Recommendation 1 *Inheritance-based modularization should be limited to closely related language extensions and language specialization.*

Language workbenches often do not support powerful language composition techniques. For example in Xtext, the limitation to *single-inheritance* and *language imports* as main approaches to modularization is not flexible enough to cope with the extensive (de-)composition of real-world DSLs. Because inheritance introduces tight coupling between two languages, this should be used sparingly. Common base languages and language specialization, for instance regarding sub-domains or company-/project-specific adaptations, are well-suited use cases for grammar inheritance. On the other hand, inheritance only allows for the addition or modification of grammar rules but cannot remove existing language concepts. Also, long inheritance chains of otherwise unrelated sub-languages provide maintainability benefits compared to one large-scale definition.

Recommendation 2 *Interface-based modularization should be used for a flexible combination of independent languages as large-scale layers of the DSL.*

A multi-layer structure of the resulting DSL can be achieved by emulating interfaces between different languages as described in Subsection 17.4.3. With the concept of language imports, references between elements from different DSLs cannot only be performed for hierarchical relationships but can also be used in settings with bidirectional or circular dependencies. Although it was shown how issues can be overcome using existing features, the workarounds are based on implicit conventions that need to be shared by all involved DSL developers.

Recommendation 3 *Language reuse works best if a common root language and infrastructure exists.*

Unfortunately, there is no simple way to "mix & match" arbitrary DSL specifications in Xtext. Reusing a distinct set of rules in different languages is complicated for reasons detailed in Subsection 17.4.1. In general, a common infrastructure is required to effectively integrate language concepts across multiple languages. A set of fundamental interfaces shared by all related sub-DSLs eases the integration process also in development frameworks which are not targeted to language modularization. However, the dependency on such a base language limits wide-spread language reuse as no standard set of primitives exists that can be applied generically to a broad set of languages.

Recommendation 4 *The granularity of modules should match the designed DSL's structure.*

Due to the limited flexibility of referencing constructs by importing the target metamodel, the module structure ideally matches the structure of the resulting DSL. For example, the effect of requiring separate model files for language add-ons is potentially acceptable for larger chunks of functionality that form intrinsic sub-units of the designed DSL. Otherwise, numerous small language extensions require content to be modelled in many different files, potentially causing confusion for the modeller.

Obviously, these pieces of advice are based on the current features of the Xtext framework. If better concepts for modularization are introduced, these recommendations might change over time. Possible options include native interfaces that can be implemented by any grammar rule [KRV10] or techniques such as aspects and traits (general multi-inheritance has its own shortcomings) for embedding individual language concepts in different DSLs such that the resulting integration is transparent to the user. However, currently no development efforts beyond the *fragment* concept for non-extensible and DSL-internal interfaces are known.

17.6 Conclusion and Outlook

Until now, basic language constructs in DSLs need to be implemented from scratch again and again, thus limiting the degree of true domain-specificity. Consequently, interoperability, maintainability, and reuse of DSLs do not reach their full potential. In this work, modularization techniques for language (de-) composition using a state-of-the-art framework for DSL development called Xtext were investigated. A case study on the large Xtext-based DSL MD² for modelling business apps is presented in order to achieve a high degree of modularity using available modularization techniques.

Several challenges limit the flexible applicability of language modules. Most importantly, the constraints of single-inheritance and Xtext's inability to embed external grammars negatively affect the possibilities of modular languages and the resulting modelling experience for users. Beyond the particular use case, general opportunities of DSL modularization in Xtext include the reduction of legacy code and improved maintainability of the current code base. The applied practices are distilled into four recommendations for exploiting the current features.

This work reveals future research need concerning suitable modularization techniques for textual DSLs which necessitate more flexibility regarding the reuse of small-scale languages and the extraction of – often technical and not domain-specific – concepts into sub-DSLs. Also, fully modularizing the model-driven process including editing component, model processor, and code generators constitutes future work.

References

[BC90] Gilad Bracha and William Cook. "Mixin-based inheritance". In: *ACM SIGPLAN Notices* 25.10 (Oct. 1990), pp. 303–311. DOI: 10.1145/97946.97982. URL: http://portal.acm.org/citation.cfm?doid=97946.97982.

[CP10] Walter Cazzola and Davide Poletti. "DSL Evolution through Composition". In: *7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*. 2010, pp. 1–6. DOI: 10.1145/1890683.1890689.

[Čre+10] M. Črepinšek et al. "On automata and language based grammar metrics". In: *Computer Science and Information Systems* 7.2 (2010), pp. 309–330. DOI: 10.2298/CSIS1002309C.

[CV16] Walter Cazzola and Edoardo Vacchi. "Language components for modular DSLs using traits". In: *Computer Languages, Systems & Structures* 45 (Apr. 2016), pp. 16–34. DOI: 10.1016/j.cl.2015.12.001. URL: http://dx.doi.org/10.1016/j.cl.2015.12.001%20http://linkinghub.elsevier.com/retrieve/pii/S1477842415300208.

[Duc+06] Stéphane Ducasse et al. "Traits: A Mechanism for Fine-Grained Reuse". In: *ACM Transactions on Programming Languages and Systems* 28.2 (2006), pp. 331–388. DOI: 10.1145/1119479.1119483.

[EH07] Torbjörn Ekman and Görel Hedin. "The JastAdd system — modular extensible compiler construction". In: *Science of Computer Programming* 69.1–3 (2007), pp. 14–26. DOI: http://dx.doi.org/10.1016/j.scico.2007.02.003. URL: http://www.sciencedirect.com/science/article/pii/S0167642307001591.

[Erd+15] Sebastian Erdweg et al. "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer Languages, Systems & Structures* 44, Part A (2015), pp. 24–47. DOI: http://dx.doi.org/10.1016/j.cl.2015.08.007. URL: http://dx.doi.org/10.1016/j.cl.2015.08.007%20http://www.sciencedirect.com/science/article/pii/S1477842415000573.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *Intl. Conference on Web Information Systems and Technologies*. 2016, pp. 307–316. DOI: 10.5220/0005862103070316. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005862103070316.

[Fow05] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* http://martinfowler.com/articles/languageWorkbench.html. 2005. URL: http://martinfowler.com/articles/languageWorkbench.html (visited on 12/15/2016).

[HMK13] Henning Heitkötter, Tim A Majchrzak, and Herbert Kuchen. "Cross-platform model-driven development of mobile applications with MD2". In: *28th Annual ACM Symposium on Applied Computing*. 2013, pp. 526–533. DOI: 10.1145/2480362.2480464. URL: http://dl.acm.org/citation.cfm?doid=2480362.2480464.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "MontiCore: a framework for compositional development of domain specific languages". In: *International Journal on Software Tools for Technology Transfer* 12.5 (Sept. 2010), pp. 353–372. DOI: 10.1007/s10009-010-0142-1. URL: http://link.springer.com/10.1007/s10009-010-0142-1.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344. DOI: 10.1145/1118890.1118892. URL: http://portal.acm.org/citation.cfm?doid=1118890.1118892%20http://doi.acm.org/10.1145/1118890.1118892.

[Pes+15] Ana Pescador et al. "Pattern-based development of Domain-Specific Modelling Languages". In: *International Conference on Model Driven Engineering Languages and Systems*. Sept. 2015, pp. 166–175. DOI: 10.1109/MODELS.2015.7338247. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7338247.

[RWK18] Christoph Rieger, Martin Westerkamp, and Herbert Kuchen. "Challenges and Opportunities of Modularizing Textual Domain-Specific Languages". In: *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. Scitepress, 2018, pp. 387–395. DOI: 10.5220/0006601903870395.

[Spi01] Diomidis Spinellis. "Notable design patterns for domain-specific languages". In: *Journal of Systems and Software* 56.1 (2001), pp. 91–99. DOI: http://dx.doi.org/10.1016/S0164-1212(00)00089-3. URL: http://www.sciencedirect.com/science/article/pii/S0164121200000893.

[Vac+14] Edoardo Vacchi et al. "Neverlang 2: A Framework for Modular Language Implementation". In: *13th International Conference on Modularity*. ACM, 2014, pp. 29–32. DOI: 10.1145/2584469.2584478. URL: http://doi.acm.org/10.1145/2584469.2584478.

[Vau+14] Steffen Vaupel et al. "Model-Driven Development of Mobile Applications Allowing Role-Driven Variants". In: *MODELS* 45.355 (2014). Ed. by Juergen Dingel et al., pp. 1–17. DOI: 10.1007/978-3-319-11653-2_1. URL: http://dx.doi.org/10.1007/978-3-319-11653-2%7B%5C_%7D1.

[Völ13] Markus Völter. *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013, p. 558.

[VS10] Markus Völter and Konstantin Solomatov. "Language modularization and composition with projectional language workbenches illustrated with MPS". In: *3rd Intl. Conference on Software Language Engineering, LNCS* (2010).

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. "A Semantics for Advice and Dynamic Join Points in Aspect-oriented Programming". In: *TOPLAS* 26.5 (Sept. 2004), pp. 890–910. DOI: 10.1145/1018203.1018208. URL: http://doi.acm.org/10.1145/1018203.1018208.

[Wu+05] Hui Wu et al. "Weaving a debugging aspect into domain-specific language grammars". In: *Proceedings of the 2005 ACM Symposium on Applied computing*. 2005, pp. 1370–1374. DOI: 10.1145/1066677.1066986. URL: http://portal.acm.org/citation.cfm?doid=1066677.1066986.

18

Evaluating a Graphical Model-Driven Approach to Codeless Business App Development

Table 18.1: Fact sheet for publication P12

Title	Evaluating a Graphical Model-Driven Approach to Codeless Business App Development
Authors	Christoph Rieger[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2018
Conference	51st Hawaii International Conference on System Sciences (HICSS)
Copyright	CC BY-NC-ND 4.0
Full Citation	Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *51st Hawaii International Conference on System Sciences (HICSS)*. Waikoloa, Hawaii, USA, 2018, pp. 5725–5734

Evaluating a Graphical Model-Driven Approach to Codeless Business App Development

Christoph Rieger

Abstract: Despite the growing interest in mobile app development, the creation of apps still follows traditional software development practices. Business apps are used by non-technical users in everyday work routines. However, their development is exclusively performed by software developers that need to centrally collect requirements and domain knowledge. Recent advances such as textual domain-specific languages (DSL) for cross-platform app generation reduce development efforts, but still focus on technical users. To alleviate these problems, the Münster App Modeling Language (MAML) is proposed as novel graphical DSL for specifying business apps. For each task to be accomplished within the app, the abstract process flows are modelled together with the respective data elements and view specifications in a combined model. Consequently, also non-technical users can express their domain knowledge without dealing with software engineering specifics. In contrast to existing process modelling notations, the MAML framework then allows for a codeless generation of apps for multiple platforms through model transformations and code generators. In order to automatically generate apps, the notation has to balance technical specificity and graphical simplicity. To assess the comprehensibility and usability of MAML's DSL, a qualitative usability evaluation was performed with software developers, process modellers, and domain experts.

18.1 Introduction

Model-Driven Software Development has sparked significant interest in the past years. Through the use of models as primary software artefacts, increased efficiency and flexibility are only two of the expected benefits [SV06]. In the domain of mobile business apps, several approaches have been researched in academic literature such as MD^2[ME15], Mobl [HV11], and AXIOM [JJ14]. Those approaches provide cross-platform development functionalities using one common model for multiple target platforms, supporting current trends such as Bring your own device [Tho12]. However, most of them focus on a textual Domain-Specific Language (DSL) to specify apps, often simplifying app development significantly but still restricting app creation to users with programming skills [MHS05]. From a software engineering perspective, this predominantly top-down approach aligns well with traditional software development practices but may not be ideal in the context of small-scale mobile apps. Business apps focus on specific tasks to be accomplished by employees. A centralized definition of such processes may therefore deviate from the individual user's needs and then imposes additional burdens on the workforce instead of improving efficiency. In addition, operating employees have valuable insights into the actual process execution as well as unobvious process exceptions. Giving them a means to shape the software they use in their everyday work routines offers not only the possibility to explicate their tacit knowledge for the development of best practices, but also actively involves them in the

evolution of the enterprise. Instead of participating only in early requirements engineering phases of software development, continuously co-designing such systems may increase the adoption of the result and strengthen their identification with the company [Ful+06].

The research company Gartner predicts that more than half of all company-internal mobile apps will be built using codeless tools by 2018 [Rv14]. To tap into the full potential, mobile app development can benefit from the incorporation of people from all levels of the organization. Development tools should therefore be understandable to both programmers and domain experts. Regarding business apps, graphical notations are particularly suitable to represent the concepts of a data-driven and process-focused domain. However, current approaches often lack the capacity for holistic app modelling, often operating on a low level of abstraction with visual user interface builders or templates (e.g., [Xam17][Goo17]).

In order to advance research in the domain of cross-platform development of mobile apps and investigate opportunities for organizations in a digitized world, this paper proposes and evaluates the Münster App Modeling Language (MAML) framework. Rooted in the Eclipse ecosystem, the DSL grammar is defined as an Ecore metamodel, and technologies such as QVT Operational and Xtend are used for the creation of Android and iOS apps.

The contributions of this paper are twofold: First, a graphical DSL is presented that allows for the visual definition of business apps. The codeless app creation capabilities are demonstrated using the MAML editor with advanced modelling support and a fully automatic generation of native app source code through a two-step model transformation process. Second, a usability study is performed to demonstrate the potential and intricacies of an integrated app modelling approach for a wide audience of process modellers, domain experts, and programmers.

The structure of the paper follows these contributions. After presenting related work in Section 18.2, the proposed framework is presented in Section 18.3. Section 18.4 discusses the setup and results of the usability evaluation. In Section 18.5, the findings and implications of MAML are discussed before concluding with a summary and outlook in Section 18.6.

18.2 Related Work

Different approaches to cross-platform mobile app development have been researched. In general, five approaches can be distinguished [MEK15]. Concerning runtime approaches, *mobile webapps* are browser-run web pages optimized for mobile devices but without native elements, *hybrid approaches* such as Apache Cordova [Apa16] provide a wrapper to web-based apps that allow for accessing device-specific features through interfaces, and *self-contained runtimes* provide separate engines that mimic core system interfaces in which the app runs. In addition, two generative approaches produce native apps, either by *transpiling* apps between programming languages such as J2ObjC [Goo16] to transform Android-based business logic to Apple's language Objective-C, or *model-driven software development* for transforming a common model to code.

With regard to model-driven development, DSLs are used to model mobile apps on a platform-independent level. According to Langlois et al. [LJJ07], DSLs can be classified in textual, graphical, tabular, wizard-based, or domain-specific representations as well as combinations of those. Several frameworks for mobile app development have been developed in the past years, both for scientific and commercial purposes. In the particular domain of business apps – i.e., form-based, data-driven apps interacting with back-end systems [MEK15] – the graphical approach JUSE4Android [dB14] uses annotated UML diagrams to generate the appearance of and navigation within object graphs, and Vaupel et al. [Vau+14] presented an approach focusing around role-driven variants of apps using a visual model representation. Other approaches such as AXIOM [JJ14] and Mobl [HV11] provide textual DSLs to define business logic, user interaction, and user interface in a common model. A more extensive overview of current model-driven frameworks is provided by Umuhoza and Brambilla [UB16]. However, current approaches mostly rely on a textual specification which limits the active participation of non-technical users without prior training [Żył15], and graphical approaches are incapable of covering all structural and behavioural aspects of a mobile app. For generating source code, the work in this paper is based on the Model-Driven Mobile Development (MD^2) framework which also uses a textual DSL for specifying all constituents of a mobile app in a platform-independent manner. After preprocessing the models, native source code is generated for each target platform as described by Majchrzak and Ernsting [ME15]. This intermediate step is however automated and requires no intervention by the user (see Subsection 18.3.4).

In contrast to DSLs, several general-purpose modelling notations exist for graphically depicting applications and processes, such as the Unified Modeling Language (UML) with a collection of interrelated standards for software development. The Interaction Flow Modeling Language (IFML) can be used to model user interactions in mobile apps, especially in combination with the mobile-specific elements introduced as extension by Breu et al. [BKF14]. Process workflows can for example be modelled using BPMN [Obj11], Event-Driven Process Chains [Aal99], or flowcharts [Int85]. However, such notations are often either suitable for generic modelling tasks and remain on a superficial level of detail, or represent rather complex technical notations designed for a target group of programmers [Fra+06]. However, a trade-off is necessary to balance the ease of use for modellers with the richness of technical details for creating functioning apps. Moody [Moo09] has pointed out principles for the cognitive effectiveness of visual notations and subsequent studies have revealed comprehensibility issues through effects such as symbol overload, e.g., for the WebML notation preceding IFML [Gra+15]. Examples of technical notations in the domain of mobile applications include a UML extension for distributed systems [SW07] and a BPMN extension to orchestrate web services [BDF09]. Nevertheless, the approach presented in this work goes beyond pure process modelling. While IFML is closest to the work in this paper regarding the purpose of modelling user interactions, MAML is domain-specific to mobile apps and covers both structural (data model, views) and behavioural (business logic, user interaction) aspects.

Lastly, visual programming languages have been created for several domains such as data integration [Pen17] but few approaches focus specifically on mobile apps. RAPPT combines

a graphical notation for specifying processes with a textual DSL [Bar+15], and AppInventor provides a language of graphical building blocks for programming apps [Wol11]. However, non-technical users are usually ignored in the actual development process. Hence, those visual notations do not exploit the potential of including people with additional domain knowledge. Considering commercial frameworks, support for visual development of mobile apps varies significantly. In practice, most upcoming tools [Pro16] focus only on individual components such as back-end systems or content management, or support particular development phases such as prototyping. Start-ups such as Bizness Apps [Biz16] and Bubble Group [Bub16] aim for more holistic development approaches using configurators and web-based editors. Similarly, development environments have started to provide graphical tools for UI development, enhancing the programmatic specification of views by complementary drag and drop editors [Xam17]. The WebRatio Mobile Platform also supports codeless generation of mobile apps through a combination of IFML, other UML standards, and custom notations [Web17]. In contrast, this work focuses on a significantly more abstract and process-centric modelling level as presented in the next section.

18.3 Münster App Modeling Language

At its core, the MAML framework consists of a graphical modelling notation that is described in the following subsections. Contrary to existing notations, its models have sufficient information to transform them into fully functional mobile apps. Therefore, the framework also comprises the necessary development tools to design MAML models in a graphical editor and generate apps without requiring manual programming. The generation process is described in more detail in Subsection 18.3.4.

18.3.1 Language Design Principles

The graphical DSL for MAML is based on five design goals:

Automatic cross-platform app creation: Most important, the whole approach is built around the key concept of codeless app creation. To achieve this, individual models need to be recombined and split according to different roles (see Subsection 18.3.4) and transformed into platform-specific source code. As a consequence, models need to encode technical information such as data fields and interrelations between workflow elements in a machine-interpretable way as opposed to an unstructured composition of shapes filled with text.

Domain expert focus: MAML is explicitly designed with a non-technical user in mind. Process modellers as well as domain experts are encouraged to read, modify and create new models by themselves. The language should therefore not resemble technical specification languages drawn from the software engineering domain but instead provide generally understandable visualizations and meaningful abstractions for app-related concepts.

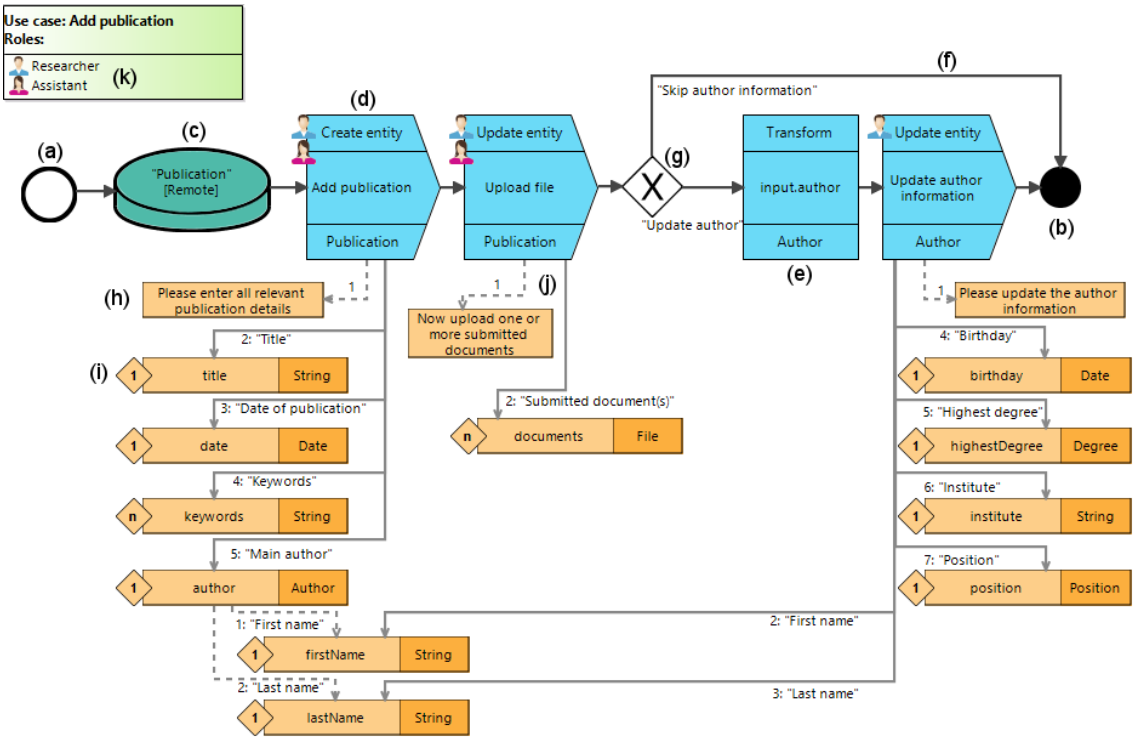


Figure 18.1: MAML Sample Use Case for Adding a Publication to a Review Management System [Rie17]

Data-driven process modelling: The basic idea of business apps to focus on data-driven processes determines the level of abstraction chosen for MAML. In contrast to merely providing editors for visual screen composition as replacement for manually programming user interfaces, MAML models represent a substantially higher level of abstraction. Users of the language concentrate on visualizing the sequence of data processing steps and the concrete representation of affected data items is automatically generated using adequate input/output user interface elements.

Modularization: To engage in modelling activities without advanced knowledge of software architectures, appropriate modularization is important to handle the complexity of apps. MAML embraces the aforementioned process-oriented approach by modelling use cases, i.e., a unit of functionality containing a self-contained set of behaviours and interactions performed by the app user [Obj15]. Combining data model, business logic, and visualization in a single model deviates from traditional software engineering practices which for instance often rely on the Model-View-Controller pattern [Gam+95]. In accordance with the domain expert focus, the end user is however unburdened from this technical implementation issue.

Declarative description: MAML models consist of platform-agnostic elements, declaratively describing *what* activities need to be performed with the data. The concrete representation in the resulting app is deliberately unspecified to account for different capabilities and usage patterns of

each targeted mobile platform. The respective code generator can provide sensible defaults for such platform specifics.

18.3.2 Language Overview

In the following, the key concepts of the MAML DSL are highlighted using the fictitious scenario of a publication management app. A sample process to add a new publication to the system consists of three logical steps: First, the researcher enters some data on the new publication. Then, he can upload the full-text document and optionally revise the corresponding author information. This self-contained set of activities is represented as one model in MAML, the so-called use case, as depicted in Figure 18.1.

A model consists of a *start event* (labelled with (a) in Figure 18.1) and a sequence of process flow elements towards an *end event* (b). A *data source* (c) specifies what type of entity is first used in the process, and whether it is only saved locally on the mobile device or managed by the remote back-end system. Then, the modeller can choose from predefined *interaction process elements* (d), for example to *create/show/update/delete* an entity, but also to *display messages*, access device sensors such as the *camera*, or *call* a telephone number. Because of the declarative description, no device-specific assumptions can be made on the appearance of such a step. The generator instead provides default representations and functionalities, e.g., display a *select entity* step using a list of all available objects as well as possibilities for searching or filtering. In addition, *automated process elements* (e) represent steps to be performed without user interaction. Those elements provide the minimum amount of technical specificity in order to navigate between the model objects (*transform*), request information from *web services*, or *include* other models to reuse existing use cases.

The order of process steps is established using *process connectors* (f), represented by a default "Continue" button unless specified differently along the connector element. *XOR* (g) elements branch out the process flow based on a manual user action by rendering multiple buttons (see differently labelled connectors in Figure 18.1), or automatically by evaluating expressions referring to a property of the considered object.

The lower section of Figure 18.1 contains the data linked to each process step. *Labels* (h) provide explanatory text on screen. *Attributes* (i) are modelled as combination of a name, the data type, and the respective cardinality. Data types such as *String, Integer, Float, PhoneNumber, Location* etc. are already provided but the user can define additional custom types. To further describe complex types, attributes may be nested over multiple levels (e.g., the "author" type in Figure 18.1 specifies a first name and last name). In addition, *computed attributes* (not depicted in the example) allow for runtime calculations such as counting or summing of other attribute values.

A suitable UI representation is automatically chosen based on the type of *parameter connector* (j): Dotted arrows signify a reading relationship whereas solid arrows represent a modifying relationship. This refers not only to the manifest representation of attributes displayed either as

read-only text or editable input field. The interpretation also applies in a wider sense, e.g., regarding web service calls in which the server "reads" an input parameter and "modifies" information through its response. Each connector also specifies an order of appearance and, for attributes, a human-readable caption derived from the attribute name unless manually specified.

Finally, annotating freely definable *roles* (k) to all interactive process elements allows for the visualization of coherent processes that are performed by more than one person, for example in scenarios such as approval workflows. When a role change occurs, the app automatically saves modified data and users with the subsequent role are informed about in their app.

18.3.3 App Modelling

In contrast to other notations, all of the modelling work is performed in a single type of model, mainly by dragging elements from a palette and arranging them on a large canvas. The modelling environment was developed using the Eclipse Sirius framework [The17b] that was extended to provide advanced modelling support for MAML.

Modelling only the information displayed in each process step effectively creates a multitude of partial data models for each process step and for each use case as a whole. Also, attributes may be connected to multiple process elements simultaneously, or can be duplicated to different positions to avoid wide-spread connections across the model. An inference mechanism [Rie17] aggregates and validates the complete data model while modelling. During generation, app-internal and back-end data stores are automatically created. As a result, the user does not need to specify a distinct global data model and consistency is automatically checked when models change.

Apart from validation rules to prevent users from modelling syntactically incorrect MAML use cases in the first place, additional validity checks have been implemented in order to detect inconsistencies across use cases (based on the inferred data model) as well as potentially unwanted behaviour (e.g., missing role annotations). Moreover, advanced modelling support attempts to provide guidance and overview to the user. For example, the current data type of a process element (lower label of (d) in Figure 18.1) is derived from the preceding elements to improve the user's imagination within the process. Also, suggestions of probable values are provided when adding elements (e.g., known attributes of the originating type when adding UI elements).

18.3.4 App Generation

Technically, MAML is built using the Eclipse Modeling Framework (EMF), for example specifying the DSL's metamodel as an Ecore model. In order to generate apps, the proposed approach reuses previous work on MD^2(see Section 18.2). The complete generation process is depicted in Figure 18.2. First, a set of model transformations described in QVT Operational notation [The17a] is applied to transform graphical MAML models to the textual MD^{2}representation. Amongst other activities, all relevant use cases are recombined, a complete data model across all use cases is inferred and explicated, and processes are broken down according to the specified roles. In the

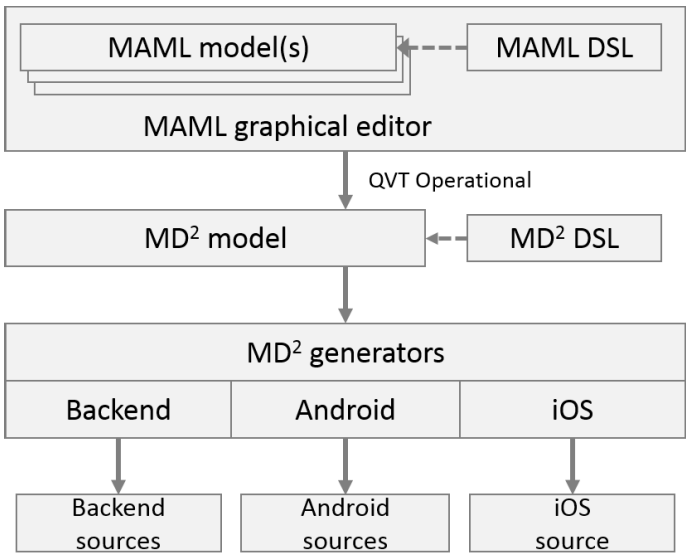


Figure 18.2: MAML App Generation Process

subsequent code generation step, previously existing generators in MD²create the actual source code for all supported target platforms.

This is however not an inherent limitation of the framework. Newly created generators might just as well generate code directly from the MAML model or use interpreted approaches instead of code generation.

Do note that this proceeding differs from approaches such as UML's Model Driven Architecture [Arco1] in that the intermediate representation is still a platform-independent representation but with a more technical focus that adds the possibility to modify default representations and configure parts of the application in more detail. Although the tooling around MAML is still in a prototypical state, it currently supports the generation of Android and iOS apps as well as a Java-based server back-end component. For example, the screenshots in Figure 18.3 depict the generated Android app views for the first process steps of the MAML model in Figure 18.1.

18.4 Evaluation

As demonstrated, MAML aligns with the goals (cf. Subsection 18.3.1) of automated cross-platform app creation from modular and platform-agnostic app models. However, the suitability of data-driven process models with regard to the target audience needed to be evaluated in more detail. Therefore, an observational study was performed to assess the utility of the newly developed language. After describing the general setup in Subsection 18.4.1, the results on comprehensibility and usability of the graphical DSL are presented.

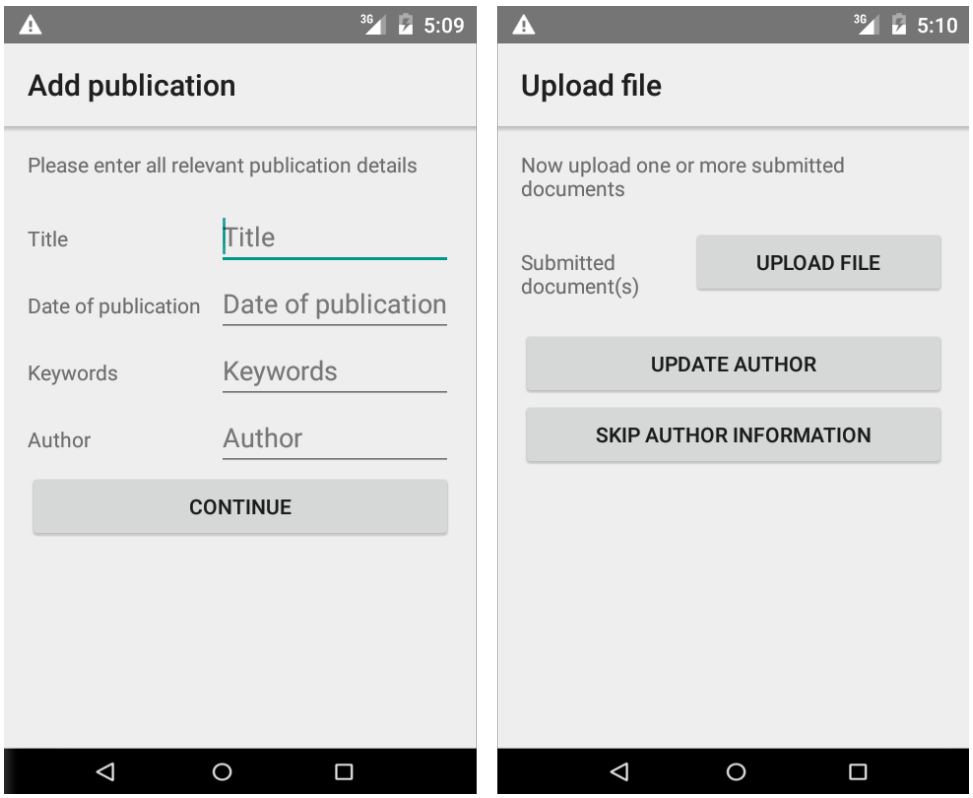


Figure 18.3: Exemplary Screenshots of Generated Android App Views

18.4.1 Study Setup

The purpose of the study was to assess MAML's claim to be understandable and applicable by users with different backgrounds, in particular including non-technical users. From the variety of methodologies for usability evaluation, observational interviews according to the think-aloud method were selected as empirical approach [GH02]. Participants were requested to perform realistic tasks with the system under test and urged to verbalize their actions, questions, problems and general thoughts while executing these tasks. Due to the novelty of MAML which excludes the possibility of comparative tests, this setup focused on obtaining detailed qualitative feedback on usability issues from a group of potential users.

Therefore, 26 individual interviews of around 90 minutes duration were conducted. An interview consisted of three main parts: First, an online questionnaire had to be filled out in order to collect demographic data, previous knowledge in the domains of programming or modelling, and personal usage of mobile devices. Second, a MAML model and an equivalent IFML model were presented to the participants (in random order to avoid bias) to assess the comprehensibility of such models without prior knowledge or introduction. In addition to the verbal explanations, a short 10-question usability questionnaire was presented to calculate a System Usability Score (SUS) for each notation (cf. Subsection 18.4.2) [Bro96]. Third, the main part of the interview

consisted of four modelling tasks to accomplish using the MAML editor. Finally, the standardized ISONORM questionnaire was used to collect more quantitative feedback, aligned with the seven key requirements of usability according to DIN 9241/110-S [Int] (cf. Subsection 18.4.3).

To capture the variety of possible usability issues, 71 observational features were identified a priori and structured in six categories of interest: comprehensibility, applying the notation, integration of elements, tool support, effectivity, and efficiency. In total, over 1500 positive or negative observations were recorded as well as additional usability feedback and proposals for improvement.

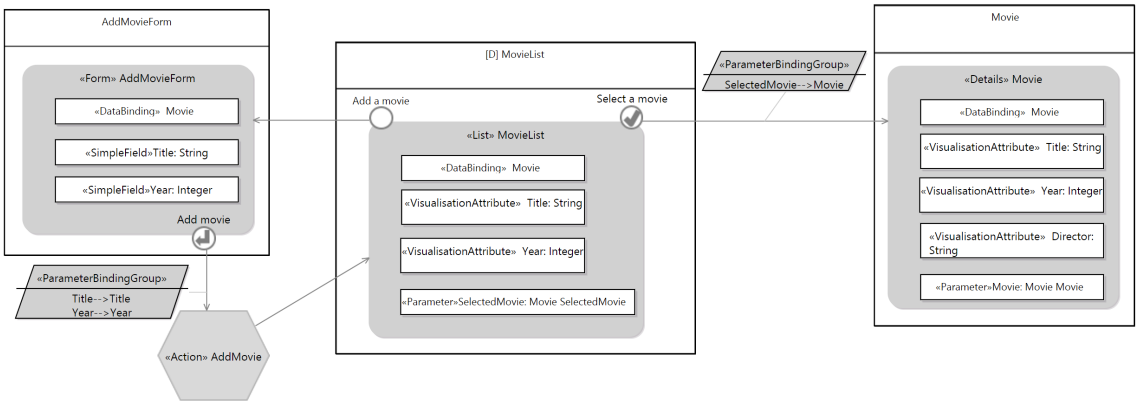


Figure 18.4: IFML Model to Assess the a Priori Comprehensibility of the Notation

Regarding participant selection, 26 potential users in the age range of 20 to 57 years took part in the evaluation. Although they mostly have a university background, technical experience varied widely and none had previous knowledge of IFML or MAML. To further analyse and interpret the results, the participants were categorized in three distinct groups according to their personal background stated in the online questionnaire: 11 software developers have at least medium knowledge in traditional/web/app programming or data modelling, 9 process modellers have at least medium knowledge in process modelling (exceeding their development skills), and 6 domain experts are experienced in the modelling domain but have no significant technical or process knowledge. Although it is debated whether Virzi's [Vir92] statement of five participants being sufficient to uncover 80% of usability problems in a particular software holds true [SS01], arguably the selected amount of participants in this study is reasonable with regard to finding the majority of grave usability defects for MAML and generally evaluating the design goals.

For their private use, participants stated an average smartphone usage of 19.2 hours per week, out of which 16.3 hours are spent on apps. In contrast, tablet use is rather low with 3.5 hours (3.2 hours for apps), and notebook usage is generally high with 27.5 hours but only 4.7 hours are spent on apps. For business uses, similar patterns can be observed on total / app-only usage per week on smartphones (5.5h / 4.3h), tablets (0.7h / 0.2h), and notebooks (18.2h / 3.7h). Although this sample is too low for generalizable insights, the figures indicate a generally high share of app usage on smartphones and tablets compared to the total usage duration, both for personal and

business tasks. In addition, with mean values of 1.81 / 2.12 on a scale between 0 (strongly reduce) and 4 (strongly increase), the participants stated to have no desire of significantly changing their usage volumes of private / business apps.

18.4.2 Comprehensibility Results

Before actively introducing MAML as modelling tool, the participants should explicate their understanding of a given model without prior knowledge. Comprehensibility is an important characteristic in order to easily communicate app-related concepts via models without need for extensive training. To compare the results with an existing modelling notation, equivalent IFML (see Figure 18.4) and MAML models [Rie16] of a fictitious movie database app were provided with the task to describe the purpose of the overall model and the particular elements. The monochrome models were shown to the participants on paper in randomized order, excluding anchor effects and potential influences from a software environment.

After each model, participants were asked to answer the SUS questionnaire for the particular notation. This questionnaire has been applied in many contexts since its development in 1986 and can be seen as easy, yet effective, test to determine usability characteristics. Each participant answers ten questions using a five-point Likert-type scale between strong disagreement and strong agreement, which is later converted and scaled to a [0;100] interval according to Brooke [Bro96]. The participants' scores for both languages and the respective standard deviations are depicted in Table 18.2.

Table 18.2: System Usability Scores for IFML and MAML

SUS ratings	IFML	MAML
All participants	52.79 (σ = 23.0)	66.83 (σ = 15.6)
Software developers	45.91 (σ = 23.6)	64.09 (σ = 17.3)
Process modellers	64.17 (σ = 19.0)	69.44 (σ = 12.0)
Domain experts	48.33 (σ = 24.5)	67.92 (σ = 18.7)

However, it should be noted that the results do not represent percentage values. Instead, an adjective rating scale was proposed by Bangor et al. [BKM09] to interpret the results as depicted in Figure 18.5. The results show that MAML's scores are superior overall as well as for all three groups of participants. In addition, the consistency of scores across all groups supports the design goal of creating a notation which is well understandable for users with different backgrounds. Particularly, domain experts without technical experience expressed a drastic difference in comprehensibility of almost 20 points.

Considering also the qualitative observations, some interesting insights can be gained. According to the questionnaire results, most of the criticism is related to the categories "easy to understand" and "confidence in the notation". IFML's approach of visually hinting at the outcome through the order of elements and their composition in screen-like boxes was often noted as

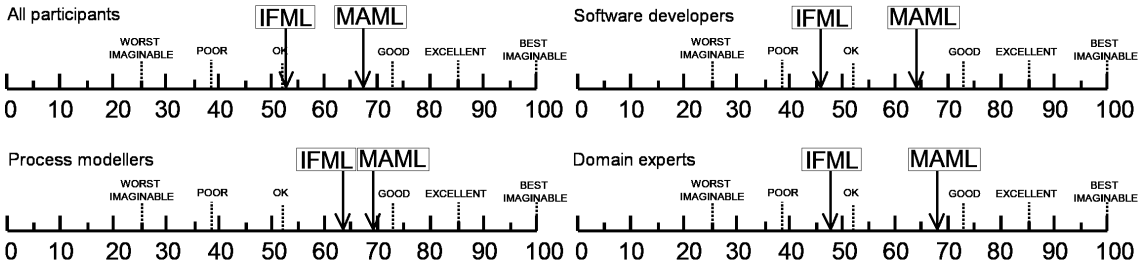


Figure 18.5: SUS Ratings for IFML and MAML

positive and slightly more intuitive compared to MAML. This argument is not unexpected as the level of abstraction was designed to be higher than a pure visual equivalent of programming activities. Also, the notation is not limited to the few types of mobile device known by a participant, e.g., smartwatches and smartphones exhibit very different interface and interaction characteristics. Therefore, a fully screen-oriented approach generally contradicts the desired platform-independent design of MAML. However, this is valuable feedback for the future, e.g., improving modelling support by using an additional simulator component to preview the outcome while modelling.

Surprisingly, IFML scores were worst for the group of software developers, although they have knowledge of other UML concepts and diagrams. Despite this apparent familiarity, reasons for the negative assessment of IFML can be found in the amount of "technical clutter", e.g., regarding parameter and data bindings, as well as perceived redundancies and inconsistencies. In contrast, 86% of these participants highlight the clarity of MAML regarding the composition of individual models and 88% are able to sketch a possible appearance of the final app result based on the abstract process specification.

Overall, three in four participants can also transfer knowledge from other modelling notations, e.g., to interpret elements such as data sources. All participants within the process modeller group immediately recognize analogies from other process modelling languages such as BPMN, and understand the process-related concepts of MAML. Whereas elements such as data sources (understood by 75% of all participants) and nested attribute structures (83%) are interpreted correctly on an abstract level, comprehensibility drops with regard to technical aspects, e.g., data types (57%) or connector types (43%).

Finally, domain experts also have difficulties to understand the technical aspects of MAML without previous introduction. Although concepts such as cardinalities (0%), data types (25%), and nested object structures (67%) are not initially understood and ignored, all participants in this group are still able to visualize the process steps and main actions of the model. As described in Subsection 18.3.1, further reducing these technical aspects constrains the possibilities to generate code from the model. Some suggestions exist to improve readability, e.g., replacing the textual data type names with visualizations. Nevertheless, MAML is comparatively well understandable.

Curiously enough, the sample IFML model is often perceived as being a more detailed technical representation of MAML instead of a notation with equivalent expressiveness.

To sum up, MAML models are favoured by participants from all groups, despite differences in personal background and technical experience. This part of the study is not supposed to discredit IFML but emphasizes their different foci: Whereas IFML covers an extensive set of features and integrates into the UML ecosystem, it is originally designed as generic notation for modelling user interactions and targeted at technical users. In contrast, the study confirms MAML's design principle of an understandable DSL for the purpose of mobile app modelling.

18.4.3 Usability Results

In addition to the language's comprehensibility, a major part of the study evaluated the actual creation of models by the participants using the developed graphical editor. After a brief ten-minute introduction of the language concepts and the editor environment, four tasks were presented that cover most of MAML's features and concepts. In the hands-on context of a library app (cf. supplementary online material [Rie16]), a first simple model to add a new book to the library requires the combination of core features such as process elements and attributes. Second, participants should model how to borrow a book based on screenshots of the resulting app. This requires more interaction element types, a case distinction, and complex attributes. Third, modelling a summary of charges includes a web service call, exception handling, and calculations. Fourth, a partial model in a multi-role context needed to be altered.

The final evaluation was performed using the ISONORM questionnaire in order to assess the usability following the ISO 9241-110 standard [Int]. 35 questions with a scale between -3 and 3 cover the seven criteria of usability as presented in Table 18.3. Again, MAML achieves positive results for every criterion, both for the participant subgroups and in total. Taking the interview observations into account for qualitative feedback, these figures can be evaluated in more detail.

Regarding the *suitability for the task*, observations on the effectiveness and efficiency of the notation show that handling models in the editor is achieved without major problems. 94% of the participants themselves noticed a fast familiarization with the notation, although domain experts are generally more wary when using the software. The deliberately chosen high level of abstraction manifests in 37% of participants describing this approach as uncommon or astonishing (see also Section 18.5). Nevertheless, 67% of the participants state to have an understanding of the resulting app while modelling.

Self-descriptiveness refers to comprehension issues but additionally deals with the correct integration of different elements while modelling. For example, the concept of assigning roles was introduced to the participants but not the concrete usage. Still, 86% of them intuitively drag and drop role icons on process elements. Furthermore, process exceptions were not explained at all in the introduction but 71% of the participants applied the "error event" element correctly without help. Self-descriptiveness is, however, more limited when dealing with technical issues.

Table 18.3: ISONORM usability questionnaire results for MAML.

Criterion	All participants	Software developers	Process modellers	Domain experts
Suitability for the task	1.63 (σ = 1.04)	1.36 (σ = 1.13)	1.62 (σ = 1.12)	2.13 (σ = 0.62)
Self-descriptiveness	0.51 (σ = 0.73)	0.62 (σ = 0.62)	0.38 (σ = 1.02)	0.50 (σ = 0.41)
Controllability	2.10 (σ = 0.83)	2.20 (σ = 0.63)	2.02 (σ = 0.63)	2.03 (σ = 1.41)
Conformity with user expectations	1.78 (σ = 0.52)	1.85 (σ = 0.47)	1.64 (σ = 0.47)	1.87 (σ = 0.70)
Error tolerance	0.92 (σ = 0.96)	0.89 (σ = 0.63)	1.11 (σ = 0.81)	0.70 (σ = 1.63)
Suitability for individualisation	1.20 (σ = 0.90)	1.04 (σ = 1.05)	1.42 (σ = 1.02)	1.17 (σ = 0.27)
Suitability for learning	1.83 (σ = 0.67)	2.02 (σ = 0.54)	1.69 (σ = 0.66)	1.70 (σ = 0.90)
Overall score	1.43 (σ = 0.49)	1.43 (σ = 0.46)	1.41 (σ = 0.53)	1.44 (σ = 0.59)

Side effects of transitive attributes are only recognized by 43% of process modellers and 25% of domain experts. Model validation or additional modelling support is needed in order to guide the users towards semantically correct models. Similarly, the complexity of modelling web service responses within the use case's data flow poses challenges to 44% of the participants.

The very positive responses for the *controllability* criterion can be explained by the simplistic design of MAML and its tools, performing all activities in a single view instead of switching between multiple models. Many participants utter remarks such as "the editor does not evoke the impression of a complex tool". Parts of this impression can be attributed to sophisticated modelling support, including live data model inference when connecting elements in the model, validation rules, and suggestions for available types.

Related to the clarity of possible user actions, the *conformity with user expectations* is also clearly positive. Despite occasional performance issues caused by the prototypical nature of the tools, a consistent handling of the program is confirmed by the participants. Although aspects such as the direction of parameter connections may be interpreted differently (e.g., either a sum refers to attributes or attributes are incoming arguments to the sum function), the consistent use of concepts throughout the notation is easily internalized by the participants.

Regarding *error tolerance* and *suitability for individualisation*, scores are moderate but the prototype was not yet particularly optimized for performance or production-ready stability. Also, an individual appearance was not intended, thus providing only basic capabilities such as resizing and repositioning components. Whereas the editor is very permissive with regard to the order of

modelling activities, adding invalid model elements is mostly avoided by automatic validity checks, e.g. which elements are valid end points of a connector. Participants appreciate the support of not being able to model invalid constellations, however criticism arises from disallowing actions without further feedback on why a specific action is invalid. The modelling environment Sirius is currently not able to provide such details, yet users might benefit more from such dynamic explanations than from traditional help pages.

Finally, *suitability for learning* can be demonstrated best using quotes such as MAML being judged as "a really practical approach", and participants having "fun to experiment with different elements" or being "surprised about what I am actually able to achieve". Using the graphical approach, users can express their ideas and apply concepts consistently to different elements. As mentioned above, many unknown features such as roles or web service interactions can be learned using known drag and drop or read/modify relationship patterns.

18.5 Discussion

In this section, key findings of the proposed MAML framework and subsequent evaluation are discussed with regard to the design objectives and general implications on model-driven software development for mobile applications. Regarding the principle of data-driven process modelling, using process flows in a graphical notation has shown to be a suitable approach for declaratively designing business apps. Graphical DSLs can also simplify modelling activities for the users of other domains, especially those that benefit from a visual composition of elements such as graph structures. Particularly for MAML, the chosen level of abstraction allows for a much wider usage compared to low-level graphical screen design: Besides the actual app product, models can be used to discuss and communicate small-scale business processes in a more comprehensive way than BPMN or similar process notations through combined modelling of process flows and data structures. In contrast to alternative codeless app development approaches focused on the graphical configuration of UI elements, users do not get distracted by the eventual position of elements on screen but can focus on the task to be accomplished. Moreover, the DSL is platform-agnostic and can thus be used to describe apps for a large variety of mobile devices. Apart from smartphones and tablets, generators for novel device types such as smartwatches or smart glasses may be created in the future based on the same input models.

Second, the challenge of developing a machine-interpretable notation that is understandable both for technical and non-technical users is a balancing act, but the interview observations and consistent scores in the evaluation indicate this design goal was reached. The most significant differences in the participants' modelling results are related to technical accuracy, mostly because of (missing) knowledge about programming or process abstractions. As such issues not always manifest as modelling errors but often happen through oversights, preventing them while keeping a certain joy of use is only achievable using a combined approach: The notation itself should be permissive instead of overly formal. Moreover, clarity (e.g., wording of UI elements) and simplicity

of the DSL contribute to manageable models. Most important, however, is the extensive use of modelling support for different levels of experience. Novice users learn from hints (e.g., hover texts and error explanations) whereas advanced users can benefit from domain-specific validation rules and optional perspectives to preview results of model changes. Particularly for MAML, advanced modelling support is achieved by interpreting the models and inferring a global object structure from a variety of partial data models as described in [Rie17]. Consequently, this feature allows for dynamically generated suggestions such as available data types, implicit reactions such as forbidding illegitimate element connections, and validation of conflicting data types and cardinalities. In general, a model-driven approach with advanced modelling support enables the active involvement of business experts in software development processes and can be regarded as major influencing factor for a successful integration of non-programmers.

Finally, the choice of mixing data model, business logic, and view details in a single model deviates from traditional software engineering practices in order to ease the modelling process for non-technical users. This does not mean that we recommend MAML for all process-oriented modelling tasks. Large business processes are just too complex to be jointly expressed with all data objects in a single model. However, mobile apps with small-scale tasks and processes are well suited to this kind of integrated modelling approach. The evaluation has shown that users appreciate the simplicity of the editor without switching between multiple interrelated models, a major distinction from related approaches to graphical mobile app development. Possibly related to the aforementioned modelling support, not even programmers miss the two-step approach of first specifying a global data model and then separately defining the respective processes. Nevertheless, as potential future extension, an optional view of the inferred data model may be interesting for them to check the modelling result before generation. Similarly, two non-technical users stated the wish for a preview of the resulting screens. However, both suggestions are neither meant to be editable nor mandatory for the app creation process and rather serve as reassuring validation while modelling the use case. It can therefore be said that modelling activities should suit the users' previous experience, potentially ignoring established concepts of (technical) domains for the greater good of a more comprehensible and seamless modelling environment.

As a result, bringing mobile app modelling to this new level of abstraction not only bridges the gap to the field of business process modelling but can also impact organizations. On the one hand, new technical possibilities arise from process-centric app models. For example, already documented business processes can be used as input for cross-platform development targeting a variety of heterogeneous mobile devices. On the other hand, codeless app generation creates the opportunity for different development methodologies. Instead of involving domain experts only in requirements phases before the actual development, an equitable relationship with fast development cycles is possible because changes to the model can be deployed instantly. Furthermore, future non-technical users may themselves develop applications according to their needs, extending the idea of self-service IT to its actual development. All of these ideas, however,

rely on the modelling support provided by the environment, as begun with MAML's data model inference mechanism. Smart software to guide and validate the created models is required instead of simply representing the digital equivalent of a sheet of paper. In the future, graphical editors may evolve beyond just organizing and linking different models, towards tools enabling novel digital ecosystems through supportive technology.

18.6 Conclusion

In this work, a model-driven approach to mobile app development called MAML was presented which focuses around a declarative and platform-agnostic DSL to graphically create mobile business apps. The visual editor component provides advanced modelling support, and transformations allow for a codeless generation of app source code for multiple platforms. To evaluate the notation with regard to comprehensibility and usability, an extensive observational study with 26 participants confirms the design goals of achieving a wide-spread comprehensibility of MAML models for different audiences of software developers, process modellers, and domain experts. In comparison to the IFML notation, an equivalent MAML model is perceived as much less complex – in particular by non-technical users – and participants felt a high level of control, thus confidently solving their tasks. As a result, MAML's approach of describing a mobile app as process-oriented set of use cases reaches a suitable balance between the technical intricacies of cross-platform app development and the simplicity of usage through the high level of abstraction.

In case of the presented study results, some limitations may threaten their validity. Although a reasonable amount of participants was chosen for the observational interviews, additional evaluations may be carried out after the next iteration of MAML's development. Our participants were mostly students, yet their generation of app-experienced adults already participates in the general workforce and can be seen as realistic (albeit not representative) sample. The synthetic examples within the case study were designed to test a wide range of MAML's capabilities. Therefore, a real-world application would strengthen the validity of the approach and at the same time represents future work.

Regarding limitations of the approach itself, the chosen level of abstraction requires assumptions on the generic representation of data in the prototype. Possibilities to customize low-level details such as UI styling for different device classes need to be addressed in future, for example on the level of the intermediate MD^{2}representation. Also, complex control flow logic and parallelism are so far not considered.

The presented process-oriented DSL offers the opportunity for research on transformations between MAML and process modelling notation such as BPMN in order to further integrate mobile app development with traditional business process management. Technically, further iterations on the framework's development are planned in order to provide additional user support, improve performance, and incorporate feedback based on the observed usability issues. Finally, with the recent popularity of novel mobile devices such as smartwatches, their applicability to business

apps through model-driven transformations of MAML's platform-agnostic models also present exciting possibilities for future research.

References

[Aal99] W.M.P. van der Aalst. "Formalization and verification of event-driven process chains". In: *Information and Software Technology* 41.10 (1999), pp. 639–650. DOI: 10.1016/S0950-5849(99)00016-6.

[Apa16] Apache Software Foundation. *Apache Cordova Documentation*. 2016. URL: https://cordova.apache.org/docs/en/latest/ (visited on 12/03/2016).

[Arc01] Architecture Board ORMSC. *Model Driven Architecture (MDA): Document number ormsc/2001-07-01*. 2001. URL: http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01 (visited on 12/03/2016).

[Bar+15] Scott Barnett et al. "A Multi-view Framework for Generating Mobile Apps". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 305–306. DOI: 10.1109/VLHCC.2015.7357239.

[BDF09] M. Brambilla, M. Dosmi, and P. Fraternali. "Model-driven engineering of service orchestrations". In: *5th World Congress on Services* (2009). DOI: 10.1109/SERVICES-I.2009.94.

[Biz16] Bizness Apps. *Mobile App Maker | Bizness Apps*. 2016. URL: http://biznessapps.com/ (visited on 12/03/2016).

[BKF14] R. Breu, A. Kuntzmann-Combelles, and M. Felderer. "New Perspectives on Software Quality [Guest editors' introduction]". In: *IEEE Software* 31.1 (2014), pp. 32–38. DOI: 10.1109/MS.2014.9.

[BKM09] Aaron Bangor, Philip Kortum, and James Miller. "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale". In: *J. Usability Studies* 4.3 (2009), pp. 114–123.

[Bro96] John Brooke. "SUS-A quick and dirty usability scale". In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7.

[Bub16] Bubble Group. *Bubble - Visual Programming*. 2016. URL: https://bubble.is/ (visited on 12/03/2016).

[dB14] L. P. da Silva and F. Brito e Abreu. "Model-driven GUI generation and navigation for Android BIS apps". In: *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2014, pp. 400–407.

[Fra+06] R. B. France et al. "Model-Driven Development Using UML 2.0: Promises and Pitfalls". In: *Computer* 39.2 (2006), pp. 59–66. DOI: 10.1109/MC.2006.65.

[Ful+06] J. B. Fuller et al. “Perceived external prestige and internal respect: New insights into the organizational identification process”. In: *Human Relations* 59.6 (2006), pp. 815–846. DOI: 10.1177/0018726706067148.

[Gam+95] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995.

[GH02] Günther Gediga and Kai-Christoph Hamborg. “Evaluation in der Software-Ergonomie”. In: *Journal of Psychology* 210.1 (2002), pp. 40–57. DOI: 10.1026//0044-3409.210.1.40.

[Goo16] Google Inc. *J2ObjC*. 2016. URL: http://j2objc.org/ (visited on 11/23/2016).

[Goo17] GoodBarber. *GoodBarber: Make an app*. 2017. URL: https://www.goodbarber.com/ (visited on 06/15/2017).

[Gra+15] David Granada et al. “Analysing the cognitive effectiveness of the WebML visual notation”. In: *Software & Systems Modeling* (2015). DOI: 10.1007/s10270-014-0447-8.

[HV11] Zef Hemel and Eelco Visser. “Declaratively Programming the Mobile Web with Mobl”. In: *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2011, pp. 695–712. DOI: 10.1145/2048066.2048121.

[Int] International Organization for Standardization. *ISO 9241-110:2006*.

[Int85] International Organization for Standardization. *ISO 5807:1985*. 1985.

[JJ14] Chris Jones and Xiaoping Jia. “The AXIOM Model Framework: Transforming Requirements to Native Code for Cross-platform Mobile Applications”. In: *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, 2014.

[LJJ07] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. “DSL classification”. In: *The 7th OOPSLA Workshop on Domain-Specific Modeling*. 2007.

[ME15] T. A. Majchrzak and J. Ernsting. “Reengineering an Approach to Model-Driven Development of Business Apps”. In: *8th SIGSAND/PLAIS EuroSymposium 2015, Gdansk, Poland*. 2015, pp. 15–31. DOI: 10.1007/978-3-319-24366-5_2.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. “Achieving Business Practicability of Model-Driven Cross-Platform Apps”. In: *OJIS* 2.2 (2015), pp. 3–14.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and How to Develop Domain-specific Languages”. In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[Moo09] Daniel Moody. “The “Physics” of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering”. In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 756–778.

[Obj11] Object Management Group. *Business Process Model and Notation*. 2011. URL: http://www.omg.org/spec/BPMN/2.0.

[Obj15] Object Management Group. *Unified Modeling Language*. 2015. URL: http://www.omg.org/spec/UML/2.5.

[Pen17] Pentaho Corp. *Data Integration - Kettle*. 2017. URL: http://community.pentaho.com/projects/data-integration/ (visited on 06/15/2017).

[Pro16] Product Hunt. *7 Tools to Help You Build an App Without Writing Code*. 2016. URL: https://medium.com/product-hunt/7-tools-to-help-you-build-an-app-without-writing-code-cb4eb8cfe394 (visited on 06/02/2017).

[Rie16] Christoph Rieger. *MAML Code Respository*. 2016. URL: https://github.com/wwu-pi/maml.

[Rie17] Christoph Rieger. "Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development". In: *Proceedings of the 32nd Annual ACM Symposium on Applied Computing*. 2017, pp. 1599–1606.

[Rie18] Christoph Rieger. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". In: *51st Hawaii International Conference on System Sciences (HICSS)*. Waikoloa, Hawaii, USA, 2018, pp. 5725–5734.

[Rv14] Janessa Rivera and Rob van der Meulen. *Gartner Says By 2018, More Than 50 Percent of Users Will Use a Tablet or Smartphone First for All Online Activities*. 2014. URL: http://www.gartner.com/newsroom/id/2939217 (visited on 12/03/2016).

[SS01] Jared Spool and Will Schroeder. "Testing Web Sites: Five Users is Nowhere Near Enough". In: *CHI '01 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2001, pp. 285–286. DOI: 10.1145/634067.634236.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Chichester: John Wiley & Sons, 2006.

[SW07] C. Simons and G. Wirtz. "Modeling context in mobile distributed systems with the UML". In: *Journal of Visual Languages and Computing* 18.4 (2007), pp. 420–439. DOI: 10.1016/j.jvlc.2007.07.001.

[The17a] The Eclipse Foundation. *QVT Operational*. 2017. URL: http://www.eclipse.org/mmt/qvto (visited on 06/15/2017).

[The17b] The Eclipse Foundation. *Sirius*. 2017. URL: https://eclipse.org/sirius/ (visited on 06/15/2017).

[Tho12] Gordon Thomson. "BYOD: enabling the chaos". In: *Network Security* 2012.2 (2012), pp. 5–8. DOI: 10.1016/S1353-4858(12)70013-2.

[UB16] Eric Umuhoza and Marco Brambilla. "Model Driven Development Approaches for Mobile Applications: A Survey". In: *Mobile Web and Intelligent Information Systems: 13th International Conference*. 2016, pp. 93–107. DOI: 10.1007/978-3-319-44215-0_8.

[Vau+14] S. Vaupel et al. "Model-driven development of mobile applications allowing role-driven variants". In: *Lecture Notes in Computer Science* 8767 (2014), pp. 1–17.

[Vir92] Robert A. Virzi. "Refining the Test Phase of Usability Evaluation: How Many Subjects is Enough?" In: *Hum. Factors* 34.4 (1992), pp. 457–468.

[Web17] WebRatio. *WebRatio*. 2017. URL: http://www.webratio.com/site/content/en/home.

[Wol11] D. Wolber. "App inventor and real-world motivation". In: *42nd ACM Technical Symposium on Computer Science Education (SIGCSE)* (2011). DOI: 10.1145/1953163.1953329.

[Xam17] Xamarin Inc. *Developer Center - Xamarin*. 2017. URL: https://developer.xamarin.com (visited on 06/15/2017).

[Żył15] Kamil Żyła. "Perspectives of Simplified Graphical Domain-Specific Languages as Communication Tools in Developing Mobile Systems for Reporting Life-Threatening Situations". In: *Studies in Logic, Grammar and Rhetoric* 43.1 (2015). DOI: 10.1515/slgr-2015-0048.

19

A Model-Driven Approach for Evaluating Traceability Information

Table 19.1: Fact sheet for publication P13

Title	A Model-Driven Approach for Evaluating Traceability Information
Authors	Hendrik Bünder[1] Christoph Rieger[2] Herbert Kuchen[2] [1] *itemis AG, Bonn, Germany* [2] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2017
Conference	Third International Conference on Advances and Trends in Software Engineering (SOFTENG)
Copyright	ARIA
Full Citation	Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Model-Driven Approach for Evaluating Traceability Information". In: *Third International Conference on Advances and Trends in Software Engineering (SOFTENG).* ed. by Mira Kajko-Mattsson, Pål Ellingsen, and Paolo Maresca. 2017, pp. 59–65

A Model-Driven Approach for Evaluating Traceability Information

Hendrik Bünder Christoph Rieger Herbert Kuchen

Keywords: Traceability, Domain-Specific Language, Software Metrics, Model-driven Software Development, Xtext

Abstract: A traceability information model (TIM), in terms of requirement traceability, describes the relation of all artifacts that specify, implement, test, or document a software system. Creating and maintaining these models takes a lot of effort, but the inherent information on project progress and quality is seldom utilized. This paper introduces a domain-specific language (DSL) based approach to leverage this information by specifying and evaluating company- or project-specific analyses. The capabilities of the Traceability Analysis Language (TAL) are shown by defining coverage, impact and consistency analysis for a model according to the Automotive Software Process Improvement and Capability Determination (A-SPICE) standard. Every analysis is defined as a rule expression that compares a customizable metric's value (aggregated from the TIM) against an individual threshold. The focus of the Traceability Analysis Language is to make the definition and execution of information aggregation and evaluation from a TIM configurable and thereby allow users to define their own analyses based on their regulatory, project-specific, or individual needs. The paper elaborates analysis use cases within the automotive industry and reports on first experiences from using it.

19.1 Introduction

Traceability is the ability to describe and follow an artifact and all its linked artifacts through its whole life in forward and backward direction [OC94]. Although many companies create traceability information models for their software development activities either because they are obligated by regulations [Cle+14] or because it is prescribed by process maturity models, there is a lack of support for the analysis of such models [BMP13].

On the one hand, recent research describes how to define and query traceability information models [JM09][MC13]. This is an essential prerequisite for retrieving specific trace information from a Traceability Information Model (TIM). However, far too little attention has been paid to taking advantage of further processing the gathered trace information. In particular, information retrieved from a TIM can be aggregated in order to support software development and project management activities with a real-time overview of the state of development.

On the other hand, research has been done on defining relevant metrics for TIMs [RM15], but the data collection process is non-configurable. As a result, potential analyses are limited to predefined questions and cannot provide comprehensive answers to ad hoc or recurring information needs. For example, projects using an iterative software development approach might be interested in the achievement of objectives within each development phase, whereas other projects might focus on a comprehensive documentation along the process of creating and modifying software artifacts.

The approach presented in this paper fills the gap between both areas by introducing the Traceability Analysis Language. By defining coverage, impact and consistency analyses for a model based on the Automotive Software Process Improvement and Capability Determination (A-SPICE) standard use cases for the Traceability Analysis Language (TAL) features are exemplified. Analyses are specified as rule expressions that compare individual metrics to specified thresholds. The underlying metrics values are computed by evaluating metrics expressions that offer functionalities to aggregate results of a query statement. The TAL comes with an interpreter implementation for each part of the language, so that rule, metric, and query expressions cannot only be defined, but can also be executed against a traceability information model. More specifically, the analysis language is based on a traceability meta model defining the abstract artifact types that are relevant within the development process. All TAL expressions therefore target the structural characteristics of the TIM.

The contributions of this paper are threefold: first, we provide a domain-specific Traceability Analysis Language to define rules, metrics, and queries in a fully configurable and integrated way. Second, we demonstrate the feasibility of our work with a prototypical interpreter implementation for real-time evaluation of those trace analyses. In addition, we illustrate the TAL's capabilities in the context of the A-SPICE standard and report on first experiences from real-world projects in the automotive sector.

Having discussed related work in Section 19.2, Section 19.3 presents the capabilities of the TAL by exemplifying impact, coverage, and consistency analyses, as well as the respective rule, metrics, and query features for retrieving information from the TIM in an automotive context. In Section 19.4, the language, our prototypical implementation, and first usage experiences are discussed before the paper concludes in Section 19.5.

19.2 Related Work

Requirements traceability is essential for the verification of the progress and completeness of a software implementation [Völ13]. While, e.g., in the aviation or medical industry traceability is prescribed by law [Cle+14], there are also process maturity models requesting a certain level of traceability [Cle+12].

Traceable artifacts such as *Software Requirement*, *Software Unit*, or *Test Specification*, and the links between those such as *details*, *implements*, and *tests* constitute the TIM [MGP13]. Retrieving traceability information and establishing a TIM is beyond the scope of this paper and approaches for standardization such as [GSG12] have already been researched.

In contrast to the high effort that is made to create and maintain a TIM, only a fraction of practitioners takes advantage of the inherent information [Cle+14]. However, Rempel and Mäder (2015) have shown that the number of related requirements or the average distance between related requirements have a positive correlation with the number of defects associated with this requirement. Traceability models not only ease maintenance tasks and the evolution of software

systems [ME15] but can also support analyses in diverse fields of software engineering such as development practices, product quality, or productivity [BZ14]. In addition, other model-driven domains, such as variability management in software product lines, benefit from traceability information [Anq+10].

Due to the lack of sophisticated tool support, these opportunities are often missed [BMP13]. On the one hand, query languages for TIMs have been researched extensively, including Traceability Query Language (TQL) [JM09], Visual Trace Modeling Language (VTML) [MC13], and Traceability Representation Language (TRL) [MRA15]. On the other hand, traceability tools mostly offer a predefined set of evaluations, often with simple tree or matrix views, e.g., [Sch12]. Hence, especially company- or project-specific information regarding software quality and project progress cannot be retrieved and remains unused.

Our approach integrates both fields of research using a textual DSL [MHS05] that is focused on describing customized rule, metric and query expressions. In contrast to the Traceability Metamodelling Language [Dri+09] defining a domain-specific configuration of traceable artifacts, our work builds on a model regarding the specification of type-safe expressions and for deriving the scope of available elements from concrete TIM instances.

19.3 An Integrated Traceability Analysis Language

19.3.1 Scenarios for Traceability Analyses

The capabilities of the TAL will be demonstrated by defining analyses from the categories of coverage, impact and consistency analysis as introduced by the A-SPICE standard [Aut15]. In addition to these rather static analyses, there are also traceability analyses focusing on data mining techniques as introduced by [BZ14]. Even though some of these could be defined using the introduced domain-specific language, they remain out of scope of this paper.

The first scenario focuses on measuring the impact of the alteration of one or more artifacts on the whole system [AB93]. Recent research has shown that artifacts with a high number of trace links are more likely to cause bugs when they are changed [RM15]. Moreover, the impact analysis can be a good basis for the estimation of the costs of changing a certain part of the software. This estimation then not only includes the costs of implementing the change itself, but also the effort needed to adjust and test the dependent components [IR12].

The second scenario appears to be the most common, since many TIM analyses are concerned with verifying that a certain path and subsequently a particular coverage is given, e.g., "*are all requirements covered by a test case*" or "*have all test cases a link to a positive test result*" [BMP13]. In addition to verifying that certain paths are available within a TIM, coverage metrics are mostly concerned with the identification of missing paths [MGP13].

The third use case describes the consistency between traceable artifacts. Besides ensuring that all requirements are implemented, consistency analyses should also ensure that there are no

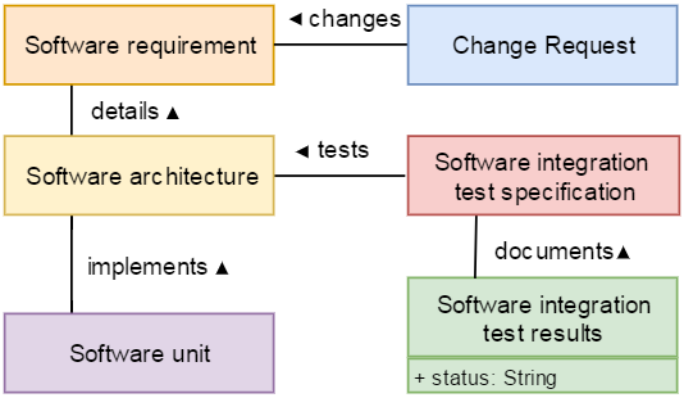


Figure 19.1: Traceability Information Configuration Model

unrequested changes to the implementation [KGB06]. Consistency is generally required between all artifacts within a TIM in accordance to the Traceability Information Configuration Model (TICM), so that all required trace links for the traced artifacts are available [Aut15].

Figure 19.1 shows a simplified TICM based on the A-SPICE standard [Aut15] that defines the traceable artifact types *Change Request, Software Requirement, Software Architecture, Software Unit, Software Integration Test Specification*, and *Software Integration Test Result*. Also, the link types *changes, details, implements, tests*, and *documents* are specified by the configuration model. The arrowheads in Figure 19.1 represent the primary trace link direction, however, trace links can be traversed in both directions [CGZ12]. The traceable artifact *Software Integration Test Result* also defines a customizable attribute called "status" that holds the actual result.

Considering the triad of economic, technical, and social problem space, the flexibility to adapt to existing work practices increases the productivity of a traceability solution [AFT07]. Therefore, configuration models provide the abstract description of traced artifact types in a company context.

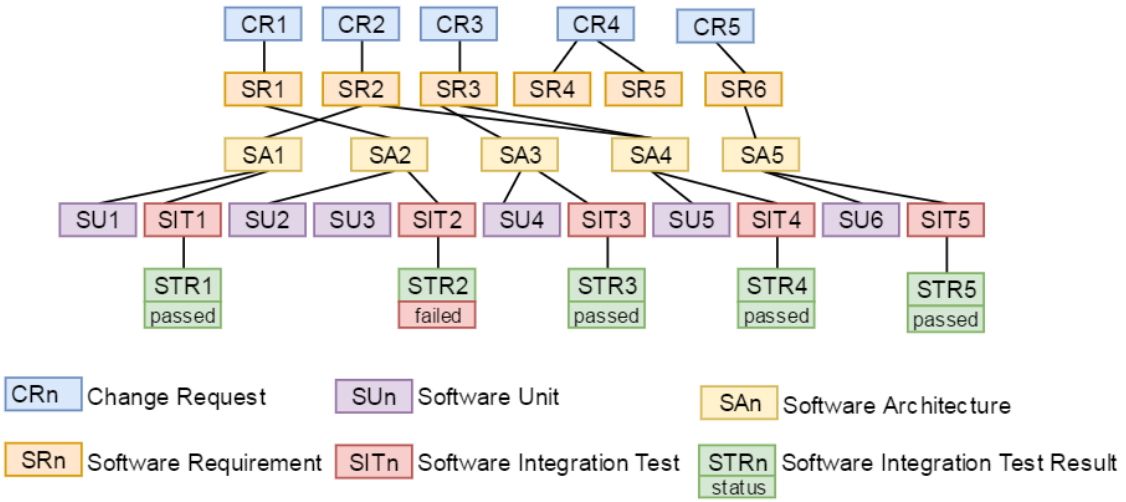


Figure 19.2: Sample Traceability Information Model

A TIM captures the concrete artifact representations and their relationships according to such a TICM and constitutes the basis for the analyses (cf. Subsection 19.3.2).

Figure 19.2 shows a traceability information model based on the sample TICM described above. The TIM contains multiple instances of the classes defined in the TICM that can be understood as proxies of the original artifacts. Those artifacts may be of different format, e.g., Word, Excel or Class files. Within the traceability software, adapters can be configured to parse an artifact's content and create a traceable proxy object in accordance to the TICM. In addition, the underlying traceability software product offers the possibility to enhance the proxy objects with customizable attributes. The *Software Integration Test Result* from Figure 19.1, for example, holds the actual result of the test case in the customizable attribute "status".

Impact Analysis

The impact analysis shown in Figure 19.3 checks the number of related requirements (NRR) [RM15] starting from every *Change Request* by using the aggregated results of a metric expression which is based on a query. The analysis begins after the *rule* keyword that is followed by an arbitrary name. The right hand side of the equation specifies the severity of breaking the rule stated in the parentheses. In this case, a rule breach will lead to a warning message with the text in quotation marks. The most important part of the analysis is the comparison part that specifies the threshold which in this case, is a number of related requirements greater than 2. If the metrics' value is greater, the warning message will be returned as a result of the analysis.

```
result relatedReqs =
  tracesFrom Software Requirement to Software Requirement
  collect(start.name ->srcRequirement,
          end.name   -> trgtRequirement)
metric NRR = count(relatedReqs.srcRequirement)
rule NRRWarning = warnIf(NRR>2, "A high number of related
     software requirements could provoke errors.")
```

Figure 19.3: Metric: Number of Related Requirements (NRR)

The second component of the TAL expression is the metric expression that in this case, counts the related requirements. Each metric is introduced by the keyword *metric*, again followed by an arbitrary name which is used to reference a metric either from another metric or from a rule as shown in Figure 19.3. The expression uses the *count* function to compute the number of related requirements. The *count* function takes a column reference to count all rows that have the same value in the given column. In the metric expression shown above, all traces from one *Software Requirement* to a *Software Requirement* have the name of the source *Software Requirement* in their first column, so that the *count* function will count all traces per *Software Requirement*. As shown in Table 19.2, the result of the metric evaluation is a tabular data structure with always two columns. The first holds the source artifact and the second column holds the evaluated metric value. For the given example, the first column holds the name of each *Software Requirement* and

the second column contains the evaluated number of directly and indirectly referenced *Software Requirements*.

Table 19.2: NRR Metric: Tabular Result Structure

Software Requirement	NRR
SR1	1
SR2	2
SR3	2
SR4	2
SR5	1

Finally, the metric is based on a query expression that is used to retrieve information from the underlying TIM. The *tracesFrom... to...* function returns all paths between source and target artifact passed into the function as parameters. In comparison to expressing this statement in other query languages such as Structured Query Language (SQL), no knowledge about the potential paths between the source and target artifacts in the TIM is needed.

Figure 19.3 shows that the columns of the tabular result structure are defined in the brackets after the keyword *collect*. In the first column the name of the *Software Requirement* of each path is given and in the second column the name of each target *Software Requirement* is given. Both columns can contain the same artifacts multiple times, but the combination of each target with each source artifact is only contained once.

Coverage Analysis

Figure 19.4 shows a coverage analysis that is concerned with the number of related test case results per software requirement. In contrast to the analysis shown in Figure 19.3, it introduces two new concepts. First, the analysis is not dependent on a metric expression, but directly bound

```
result tracesSwReqToTestResult =
  tracesFrom Software Requirement
  to         Software Integration Test Result
  collect(start.name-> name, count(1)-> tcrs)
  where(end.status = "passed")
  groupBy(name)
rule lowTC = warnIf(tracesSwReqToTestResult.tcrs < 2,
"Low number of test results!")
rule noTC = errorIf(tracesSwReqToTestResult.tcrs < 1,
  "No test results found!")
```

Figure 19.4: Software Requirement Test Result Coverage Analysis

to a query result. Since metric and query expression results are returned in the same tabular structure, rules can be applied to both. Second, the analysis shown in Figure 19.4 demonstrates the concept of a staggered analysis, i.e., one column or metric is referenced once from a warning

and error rule, respectively. The rule interpreter will recognize this construct and will return the analysis result with the highest severity, e.g., when the error rule applies, the warning rule message is omitted. The rules shown above ensure that the test of each *Software Requirement* is documented by at least one test result. However, to fulfill the rule completely, each *Software Requirement* should be covered by two *Software Integration Tests* and subsequently two *Software Integration Test Results*.

Table 19.3: Coverage Analysis: Tabular Result Structure

Software Requirement	**Analysis Result**
SR1	No test results found!
SR2	Ok
SR3	Ok
SR4	No test results found!
SR5	No test results found!
SR6	Low number of test results!

Table 19.3 shows the result of the staggered analysis. The test coverage analysis returns an "Ok" message for two of the six *Software Requirements*, while one is marked with a warning message and the remaining three caused an error message.

The query expressions result is limited to *Software Integration Test Results* with status "passed" by evaluating the customizable attribute "status" using a *where* clause. Since the query language offers some functions to do basic aggregation, it is possible to bypass metric expressions in this case. In Figure 19.4 the aggregation is done by the *groupBy* and the *count* function. The second column specifies an aggregation function that counts all entries in a given column per row based on the column name passed as parameter. In general, the result of this function will be 1 per row since there is only one value per row and column but in combination with the "groupBy" function the number of aggregated values per cell is computed. The resulting tabular structure contains one row per *Software Requirement* with the respective name and the cumulated number of traces to different Software Integration Test Results as columns.

Consistency Analysis

The following will show two consistency analysis samples to verify that all *Software Requirements* are linked to at least one *Software Unit* and vice versa. Figure 19.5 shows a consistency analysis composed of a rule and a query expression. The rule *notCoveredError* returns an error message if the number of traces between *Software Requirements* and *Software Units* is smaller than one which means that the particular *Software Requirements* is not implemented.

Table 19.4 shows the result of the analysis as defined in Figure 19.5. For "SR4" and "SR5" there is no trace to a *Software Unit* so that the analysis marks these two with an error message. To verify that all implemented *Software Units* are requested by a *Software Requirement*, the query can easily

```
result consistetSrcTrgt =
  tracesFrom Software Requirement to Software Unit
  collect(start.name ->name, count(1)-> targets)
  groupBy(name)
rule notCoveredError = errorIf(swUnits<1, "The software
     requirement is not implemented.")
```

Figure 19.5: Consistency Analysis

Table 19.4: Consistency Analysis: Software Requirement Implementation

Name	Analysis Result
SR1	Ok
SR2	Ok
SR3	Ok
SR4	The Software Requirement is not implemented!
SR5	The Software Requirement is not implemented!
SR6	Ok

be altered by switching the parameters of the "tracesFrom... to..." function and by changing the error message. Table 19.5 shows the result of the altered analysis revealing that "SU3" despite all others has not been requested.

Table 19.5: Consistency Analysis: Software Unit Requested

Name	Analysis Result
SU1	Ok
SU2	Ok
SU3	The Software Requirement has not been requested!
SU4	Ok
SU5	Ok
SU6	Ok

These examples show that the language offers extensive support for retrieving and aggregating information in TIMs. The following sections will demonstrate how the TAL integrates with the traceability solution it is build upon, and how the different parts of the language are defined.

19.3.2 Composition of the Traceability Analysis Language

Modeling Layers

Figure 19.6 shows the integration between the different model layers referred to in this paper, starting from the *Eclipse Ecore Model* as shared meta meta model [Gro09]. The Xtext framework which is used to define the analysis language generates an instance of this model [The17c] to represent the *Analysis Language Meta Model* (ALMM). Individual queries, metrics, and rules are

specified within a concrete instance, the *Analysis Language Model* (ALM), using the created domain-specific language. An interpreter was implemented using Xtend, a Java extension developed as part of the Xtext framework and specially designed to navigate and interact with the analysis language's Eclipse Ecore models [The17b].

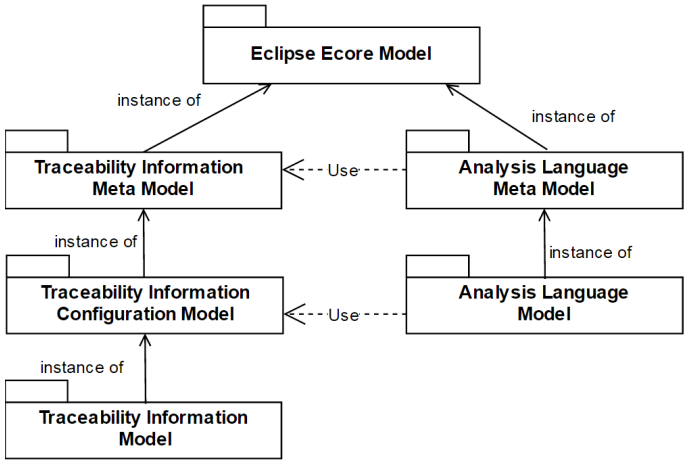


Figure 19.6: Conceptual Integration of Model Layers

Likewise, the *Traceability Information Model* used in this paper contains the actual traceability information, for example the concrete software requirement *SR1*. It is again an instance of a formal abstract description, the so called TICM. The TICM describes traceable artifact types, e.g., *Software Requirement* or *Software Architecture*, and the available link types, e.g., *details*. This model itself is based on a proprietary *Traceability Information Meta Model* (TIMM) defining the basic traceability constructs such as an artifact type and link type. To structure the DSL, the TAL itself is hierarchically subdivided into three components, namely rule, metric, and query expressions.

Rule Grammar

Since a query result or a metric value alone delivers few insights into the quality or the progress of a project, rule expressions are the main part of the TAL. Only by comparing the metric value to a pre-defined threshold or another metrics' value information is exposed. The grammar contains rules for standard comparison operations which are *equal, not equal, greater than, smaller than, greater or equals*, and *smaller or equals*. A rule expression can either return a warning or an error result after executing the comparison including an individual message. Since query and metrics result descriptions implement the same tabular result interface, rules can be applied to both. Accordingly, the result of an evaluated rule expression is also stored using the same tabular interface.

The *RuleBody* rule shown in Figure 19.7 is the central part of the rule grammar. On the left side of the *Operator* a metric expression or a column from a query expression result can be referenced.

```
WarnIf = ID '=' 'warnIf(' RuleBody ')';
RuleBody = (MetricDefinition | ResultDeclaration '.' Column)
      Operator RuleAtomic ',' MESSAGE;
```

Figure 19.7: Rule Grammar

The next part of the rule is the comparison *Operator* followed by a *RuleAtomic* value to compare the expression to. The *RuleAtomic* value is either a constant number or a reference to another metrics expression.

Metrics Grammar

Complimentary to recent research that focuses on specific traceability metrics and their meaningfulness [RM15], the approach described in this paper allows for the definition of individual metrics. An extended Backus-Naur form (EBNF)-like Xtext grammar defines the available features including arithmetic operations, operator precedence using parentheses, and the integration of query expressions. The metrics grammar of the TAL itself has two main components. One is the *ResultDeclaration* that encapsulates the result of a previously specified query. The other is an arbitrary number of metrics definitions that may aggregate query results or other metrics recursively.

```
MetricDefinition = 'metric' ID '=' MetricExpression;

MetricExpression = Term  { ('+' | '-') Term};
Term = Factor { ('*' | '/') Factor};
Factor = SumFunction | CountFunction | LengthFunction |
     DOUBLE | ColumnSelection | MetricDefinition | '('
     MetricExpression ')' ;
```

Figure 19.8: Grammar Rules for Metrics Expressions

Figure 19.8 shows a part of the metric grammar defining the support for the basic four arithmetic operations as well as the correct use of parentheses. Since the corresponding parser generated by Another Tool for Language Recognition (ANTLR) works top-down, the grammar must not be left recursive [Bet13]. First, the rule *Factor* allows for the usage of constant double values. Second, metric expressions can contain pre-defined functions such as sum, length, or count to be applied to the results of a query. Due to a lack of space, their grammar rules are not elaborated further. Third, columns from the result of a query can be referenced so that metric expressions per query expression result row can be computed. Finally, metric expressions can refer to other metric expressions to further aggregate already accumulated values. Thereby, interpreting metric expressions can be modularized to reuse intermediate metrics and to ensure maintainability.

The metrics grammar as part of the TAL defines arithmetic operations that aggregate the results of an interpreted query expression. The combination of a configurable query expressions with configurable metric definitions allows users to define their individual metrics.

Query Grammar

The analyses defined using metric and rule expressions depend on the result of a query that retrieves the raw data from the underlying TIM. Although there are many existing query languages available, a proprietary implementation is currently used because of three reasons.

First, the query language should reuse the types from TICM to enable live validation of analyses even before they are executed. The Xtext-based implementation offers easy mechanisms to satisfy this requirement, while others such as SQL are evaluated only at runtime. Second, some of the existing query languages such as SQL or Language Integrated Query (LINQ) are too verbose (cf. Figure 19.9) or do not offer predefined functions to query graphs. Finally, other languages such as SEMMLE QL [VHM07] or RASCAL [KvV09] are focused on source code analyses and do not interact well with Eclipse Modeling Framework (EMF) models.

The formal description of the syntax of a query is quite lengthy and out of scope of this paper, where we focus on the metrics and rules language. From the example in Section 19.3, the reader gets an idea, how a query looks like. The query expressions offer a powerful and well-integrated mechanism to retrieve information from a given TIM. Especially, the integration with the traceability information configuration model enables the reuse of already known terms such as the trace artifact type names. Furthermore, complex graph traversals are completely hidden from the user who only specifies the traceable source and target artifact based on the TICM. For example, the concise query of Figure 19.4 already requires a complex statement when expressed in SQL syntax (cf. Figure 19.9).

```
SELECT r.name, count(u.id) AS tcrs
FROM SwRequirement r
INNER JOIN SwRequirement_SwArchitecture ra ON r.id=ra.r_id
INNER JOIN SwArchitecture a ON ra.a_id=a.id
INNER JOIN SwArchitecture_SwIntegrationTest ai
  ON a.id=ai.a_id
INNER JOIN SwIntegrationTest i ON ai.i_id=i.id
INNER JOIN SwIntegrationTest_SwIntegrationTestResult it
  ON i.id=it.i_id
INNER JOIN SwIntegrationTestResult t ON it.t_id=t.id
WHERE t.status='passed'
GROUP BY r.name;
```

Figure 19.9: SQL Equivalent to Query of Figure 19.4

19.4 Discussion

19.4.1 Eclipse Integration and Performance

To demonstrate the feasibility of the designed TAL and perform flexible evaluations of traceability information models, a prototype was developed. The analysis language is based on the aforementioned Xtext framework and integrated in the integrated development environment Eclipse using

its plug-in mechanism [The17a]. The introduced interpreter evaluates rule, metric, and query expressions whenever the respective expression is modified and saved in the editor.

Currently, both components are tentatively integrated in a software solution that envisages a commercial application. Therefore, the analysis language is configured to utilize a proprietary TIMM from which traceability information configuration models and concrete TIMs are defined. At runtime, the expression editor triggers the interpreter to request the current TIM from the underlying software solution and subsequently perform the given analysis. Within our implementation, traceable artifacts from custom traceability information configuration models as shown in Figure 19.1 can be used for query, metrics, and rule definitions. Due to an efficient implementation used by the *tracesFrom... to...* function, analysis are re-executed immediately when an analysis is saved or can be triggered from a menu entry. The efficiency of the depth-first algorithm implementation was verified by interpreting expressions using TIMs ranging from 1,000 to 50,000 artificially created traceable artifacts. The underlying TICM was build according to the traceable artifact definitions of the A-SPICE standard [Aut15].

Table 19.6: Duration of TAL Evaluation

Artifacts	Start Artifacts	Duration (in s)
1,000	300	0.012
8,000	1,500	0.1
50,000	8,500	2.2

Table 19.6 shows the duration for interpreting the analysis expression from Figure 19.4 against TIMs of different sizes. The first column shows the overall number of traceable artifacts and links in the TIM. The second column gives the number of start artifacts for the depth-first algorithm implementation, i.e., the number of *Software Requirement*s for the exemplary analysis expression. The third column contains the execution time on an Intel Core i7-4700MQ processor at 2.4 GHz and 16 GB RAM. As shown, executing expressions can be done efficiently even for large size models, sufficient for real-world applications to regular reporting and ad hoc analysis purposes.

19.4.2 Applying the Analysis Language

Defining and evaluating analysis statements with the prototypical implementation has shown that the approach is feasible to collect metrics for different kinds of traceability projects. The most basic metric expression reads like *the proportion of artifacts of type A that have no trace to artifacts of type B*. Some generic scenarios focused on impact, coverage, and consistency analyses have been exemplified in Subsection 19.3.1. However, there are more specific metrics that are applicable and reasonable for a particular industry sector, a specific project organization, or a certain development process.

Industry-specific metrics, e.g., in the banking sector, could focus on the impact of a certain change request regarding coordination and test effort estimation. Project-specific management rules may for instance highlight components causing a high number of reported defects to indicate where to perform quality measures, e.g., code reviews. Moreover, the current progress of a software development project can be exposed by defining a staggered analysis relating design phase artifacts (e.g., *Software Requirement*s that are not linked to a *Software Architecture*) and implementation artifacts (e.g., *Software Architecture*s without trace to a *Software Unit*) in relation to the overall number of *Software Requirements*. Analysis expressions could also be specific to the software development process. In agile projects for example the velocity of an iteration could be combined with the number of bugs related to the delivered functionality. Thereby, it could be determined whether the number of bugs correlates with the scope of delivered functionality. These use cases emphasize the flexibility of the analysis language — in combination with an adaptable configuration model — for applying traceability analyses to a variety of domains, not necessarily bound to programming or software development in general. For example, a TIM for an academic paper may define traceable artifacts such as *authors*, *chapters*, and *references*. An analysis on such a paper could find all papers that cite a certain author or the average number of citations per chapter. It is therefore possible to execute analyses on other domains with graph-based structures that can benefit from traceability information.

Besides theoretical usage scenarios for the TAL, first experiences in real-world projects were gained with an automotive company. The Traceability Analysis Language was used in five projects with TIMs ranging from 30,000 to 80,000 traceable artifacts defined in accordance to the Automotive SPICE standard. For all five projects, a predefined analysis was created to compute the test coverage of each system requirement. A system requirement is considered fully tested when all linked system and software requirements have a test case with a positive test result linked (cf. Subsection 19.3.1). The execution time of the analysis in the real world projects confirmed the results from the artificial sample explained in Table 19.6. The predefined analysis has replaced a complex SQL statement that included seven joins to follow the links trough the traceability information model. Because the *tracesFrom... to...* function encapsulates the graph traversal, the TAL analysis is also more resilient to changes of the traceability configuration model.

19.4.3 Limitations

The approach presented in this paper is bound to limitations regarding both technical and organizational aspects. Regarding the impact of the developed DSL on software quality management practices, first investigations have taken place, however, more are needed to draw sustainable conclusions.

Using the TAL in industry projects has shown the need for additional analysis capabilities. One main requirement is to evaluate how much of an expected trace path is available in a certain TIM. If there is no complete path from a *System Requirement* to a *Software Integration Test Result*, it

would be beneficial to show partial matches, for instance if there is no *Software Integration Test Result* or if there is no *Software Integration Test Specification* at all. Extending the result of an analysis in accordance to this requirement would enhance the information about the progress of a project.

From a language user perspective, the big advantage of being free to configure any query, metric or rule expression is also a challenge. A language user has to be aware of the traceable artifacts and links in the TIM and how this trace information could be connected to extract reasonable measures. Moreover, the context-dependent choice of suitable metrics in terms of type, number, and thresholds is subject to further research. These limitations do not impede the value of our work, though. In fact, in combination with the discussed application scenarios they provide the foundation for our future work.

19.5 Conclusion

This work describes a textual domain-specific language to analyze existing traceability information models. The TAL is divided into query, metric, and rule parts that are all implemented with the state-of-the-art framework Xtext. The introduced approach goes beyond existing tool support for querying traceability information models. By closing the gap between information retrieval, metric definition, and result evaluation, the analysis capabilities are solid ground for project- or company-specific metrics. Since the proposed analysis language reuses the artifact type names from the traceability information configuration model, the expressions are defined using well known terms. In addition to reusing such terms, the editor proposes possible language statements at the current cursor position while writing analysis expressions. Utilizing this feature could lower the initial effort for defining analysis expressions and could result in faster evolving traceability information models.

On the one hand, the introduced approach is based on an Eclipse Ecore model and is thereby completely independent of the specific type of traced artifacts. On the other hand, it is well integrated into an existing TICM and IDE using Xtext and the Eclipse platform. All parts of the TAL are fully configurable regarding analysis expression, limit thresholds, and query statements in an integrated approach to close the gap between *querying* and *analyzing* traceability information models. Subsequently, measures for traceability information models can be specific to a certain industry sector, a company, a project or even a role within a project. The scenarios described in Subsection 19.3.1 propose areas in which configurable analyses provide benefits for project managers, quality managers, and developers. Using the implemented interpreter for real-time execution of expressions, first project experiences within the automotive industry have shown that the TAL analyses are evaluated efficiently and are more resilient than other approaches, e.g., SQL-based analyses.

Future work could focus on further assessing the applicability in real world projects and defining a structured process to identify reasonable metrics for a specific setting. Such a process

might not only support sophisticated traceability analyses but could also propose industry-proven metrics and thresholds. Some advanced features such as metrics comparisons over time using TIM snapshots to further enhance the analysis are yet to be implemented. In addition to evaluating the metrics against static values, future work might also focus on utilizing statistical methods from the data mining field. Classification algorithms or association rules for example could be used to find patterns in traceability information models and thus gain additional insights from large-scale TIMs.

References

[AB93] Robert S. Arnold and Shawn A. Bohner. "Impact Analysis-Towards a Framework for Comparison". In: *ICSM*. Vol. 93. 1993, pp. 292–301.

[AFT07] Hazeline U. Asuncion, Frédéric François, and Richard N. Taylor. "An End-to-end Industrial Software Traceability Tool". In: *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 2007, pp. 115–124. DOI: 10.1145/1287624.1287642.

[Anq+10] Nicolas Anquetil et al. "A model-driven traceability framework for software product lines". In: *Software & Systems Modeling* 9.4 (2010), pp. 427–451. DOI: 10.1007/s10270-009-0120-9.

[Aut15] Automotive Special Interest Group. *Automotive SPICE Process Reference Model*. URL: http://automotivespice.com/fileadmin/software-download/Automotive_SPICE_PAM_30.pdf [retrieved: 14.8.2017]. 2015.

[Bet13] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Pub, 2013.

[BMP13] Elke Bouillon, Patrick Mäder, and Ilka Philippow. "A Survey on Usage Scenarios for Requirements Traceability in Practice". In: *Requirements Engineering: Foundation for Software Quality*. Springer, 2013, pp. 158–173. DOI: 10.1007/978-3-642-37422-7{\textunderscore}12.

[BRK17] Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Model-Driven Approach for Evaluating Traceability Information". In: *Third International Conference on Advances and Trends in Software Engineering (SOFTENG)*. Ed. by Mira Kajko-Mattsson, Pål Ellingsen, and Paolo Maresca. 2017, pp. 59–65.

[BZ14] Andrew Begel and Thomas Zimmermann. "Analyze This! 145 Questions for Data Scientists in Software Engineering". In: *36th International Conference on Software Engineering*. ACM, 2014, pp. 12–23. DOI: 10.1145/2568225.2568233.

[CGZ12] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, eds. *Software and Systems Traceability*. Springer London, 2012.

[Cle+12] J. Cleland-Huang et al. “Trace queries for safety requirements in high assurance systems”. In: *LNCS* 7195 (2012), pp. 179–193. DOI: 10.1007/978-3-642-28714-5{\textunderscore}16.

[Cle+14] Jane Cleland-Huang et al. “Software Traceability: Trends and Future Directions”. In: *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 55–69. DOI: 10.1145/2593882.2593891.

[Dri+09] Nikolaos Drivalos et al. “Engineering a DSL for Software Traceability”. In: *Software Language Engineering*. Vol. 5452. Springer, 2009, pp. 151–167. DOI: 10.1007/978-3-642-00434-6{\textunderscore}10.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1st. Addison-Wesley Professional, 2009.

[GSG12] Andreas Graf, Nirmal Sasidharan, and Ömer Gürsoy. “Requirements, Traceability and DSLs in Eclipse with the Requirements Interchange Format (ReqIF)”. In: *Second International Conference on Complex Systems Design & Management*. Springer, 2012, pp. 187–199. DOI: 10.1007/978-3-642-25203-7{\textunderscore}13.

[IR12] Claire Ingram and Steve Riddle. “Cost-Benefits of Traceability”. In: *Software and Systems Traceability*. Ed. by Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman. Springer London, 2012, pp. 23–42. DOI: 10.1007/978-1-4471-2239-5{\textunderscore}2.

[JM09] J. I. Maletic and M. L. Collard. “TQL: A query language to support traceability”. In: *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*. 2009, pp. 16–20. DOI: 10.1109/TEFSE.2009.5069577.

[KGB06] Nihal Kececi, Juan Garbajosa, and Pierre Bourque. “Modeling functional requirements to support traceability analysis”. In: *2006 IEEE International Symposium on Industrial Electronics*. Vol. 4. 2006, pp. 3305–3310.

[KvV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. “RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation”. In: *9th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2009, pp. 168–177. DOI: 10.1109/SCAM.2009.28.

[MC13] P. Mäder and J. Cleland-Huang. “A visual language for modeling and executing traceability queries”. In: *Software and Systems Modeling* 12.3 (2013), pp. 537–553. DOI: 10.1007/s10270-012-0237-0.

[ME15] Patrick Mäder and Alexander Egyed. “Do Developers Benefit from Requirements Traceability when Evolving and Maintaining a Software System?” In: *Empirical Softw. Eng.* 20.2 (2015), pp. 413–441. DOI: 10.1007/s10664-014-9314-z.

[MGP13] Patrick Mader, Orlena Gotel, and Ilka Philippow. "Getting back to basics: Promoting the use of a traceability information model in practice". In: *7th Intl. Workshop on Traceability in Emerging Forms of Software Engineering* (2013), pp. 21–25. DOI: 10.1109/TEFSE.2009.5069578.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[MRA15] Arthur Marques, Franklin Ramalho, and Wilkerson L. Andrade. "TRL: A Traceability Representation Language". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 1358–1363. DOI: 10.1145/2695664.2695745.

[OC94] O. C. Z. Gotel and C. W. Finkelstein. "An analysis of the requirements traceability problem". In: *Proceedings of IEEE International Conference on Requirements Engineering*. 1994, pp. 94–101. DOI: 10.1109/ICRE.1994.292398.

[RM15] Patrick Rempel and Patrick Mäder. "Estimating the Implementation Risk of Requirements in Agile Software Development Projects with Traceability Metrics". In: *Requirements Engineering: Foundation for Software Quality*. Springer, 2015, pp. 81–97. DOI: 10.1007/978-3-319-16101-3_6.

[Sch12] Hannes Schwarz. *Universal traceability*. Logos Verlag Berlin, 2012.

[The17a] The Eclipse Foundation. *PDE/User Guide*. URL: http://wiki.eclipse.org/PDE/User_Guide [retrieved: 14.8.2017]. 2017.

[The17b] The Eclipse Foundation. *Xtend Modernized Java*. URL: http://eclipse.org/xtend/ [retrieved: 14.8.2017]. 2017. (Visited on 08/13/2017).

[The17c] The Eclipse Foundation. *Xtext Documentation*. URL: https://eclipse.org/Xtext/documentation/ [retrieved: 14.8.2017]. 2017.

[VHM07] Mathieu Verbaere, Elnar Hajiyev, and Oege de Moor. "Improve Software Quality with SemmleCode: An Eclipse Plugin for Semantic Code Search". In: *22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. ACM, 2007, pp. 880–881. DOI: 10.1145/1297846.1297936.

[Völ13] Markus Völter. *DSL engineering: Designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform, 2013.

20

A Domain-specific Language for Configurable Traceability Analysis

Table 20.1: Fact sheet for publication P14

Title	A Domain-specific Language for Configurable Traceability Analysis
Authors	Hendrik Bünder[1] Christoph Rieger[2] Herbert Kuchen[2] [1] *itemis AG, Bonn, Germany* [2] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2017
Conference	5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)
Copyright	SciTePress CC BY-NC-ND 4.0
Full Citation	Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Domain-specific Language for Configurable Traceability Analysis". In: *5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. ed. by Luis Ferreira Pires, Slimane Hammoudi, and Bran Selic. 2017, pp. 374–381

A Domain-specific Language for Configurable Traceability Analysis

Hendrik Bünder Christoph Rieger Herbert Kuchen

Keywords: Traceability, Domain-Specific Language, Software Metrics

Abstract: In safety-critical industries such as the aviation industry or the medical industry traceability is required by law and specific regulations. In addition, process models such as CMMI require traceability information for documentation purposes. Although creating and maintaining so-called traceability information models (TIM) takes a lot of effort, its potential for reporting development progress, supporting project management, and measuring software quality often remains untapped. The domain-specific language presented in this paper builds on an existing traceability solution and allows to define queries, metrics, and rules for company- or project-specific usage. The basis for such an analysis is a query expression to retrieve information from a TIM. Customizable metrics are then defined to compute aggregated values, which are evaluated against company- or project-specific thresholds using the rules part of the domain-specific language. The focus of this paper is to show how the combination of query, metric, and rule expressions is used to define and compute customizable analyses based on individual requirements.

20.1 Introduction

Traceability is the ability to describe and follow an artifact and all its linked artifacts through its whole life in forward and backward direction [GF94]. Many companies create traceability information models for their software development activities either because they are obligated by regulations [Cle+14] or because it is prescribed by process maturity models. However, there is a lack of support for the analysis of these models [BMP13].

Recent research describes how to define and query traceability information models, [MC09; MC13]. This is an essential prerequisite for retrieving specific trace information from a TIM. However, far too little attention has been paid to taking advantage of further processing of the gathered trace information. In particular, information retrieved from a TIM can be aggregated in order to support software development and project management activities with a "real-time" overview of the current state of development.

On the other hand, research has been done on defining relevant metrics for TIMs [RM15], but the data collection process is non-configurable. As a result, potential analyses are limited to predefined questions and cannot provide comprehensive answers to ad hoc or recurring information demands. For example, projects using an iterative software development approach might be interested in the accomplishments of objectives within each development phase, whereas other projects might focus on a comprehensive documentation during the process of creating and modifying software artifacts.

The approach presented in this paper fills the gap between those two areas by introducing a sophisticated analysis language. As a foundation, query expressions can be used to retrieve information from TIMs and subsequent metric statements aggregate the results of an executed query. In addition, rule expressions can be specified so that metric values can be checked against individually configured thresholds. All three parts come with an interpreter implementation so that they cannot only be defined but also executed against a traceability information model. The analysis language builds on a traceability meta model that is an instance of the Eclipse Ecore model [Ste+08].

This paper contributes a domain-specific traceability analysis language to define queries, metrics and rules in a fully configurable and integrated way. Further, the feasibility of the elaborated analysis language will be demonstrated by a prototypical interpreter implementation for real-time evaluation of those trace analyses.

Having discussed related work in Section 20.2, Section 20.3 introduces the query and metric expressions that are used to retrieve information from TIMs. Afterwards, the definition of rules is presented, which completes the approach with a mechanism to compare metric values with predefined thresholds. In Section 20.4, DSL and our prototypical implementation are discussed before the paper concludes in Section 20.5.

20.2 Related Work

Requirement traceability is essential for verifying the progress and completeness of a software implementation [Völ13]. While in the aviation or medical industry traceability is prescribed by law [Cle+14], there are also process maturity models requesting a certain level of traceability [Cle+12]. Traceable artifacts such as *requirement*, *unit of code*, or *test case*, and the links between those - such as *specifies*, *implements*, and *verifies* - constitute the TIM [MGP13].

In contrast to the efforts made to create and maintain a TIM, only a fraction of practitioners takes advantage of the inherent information according to recent research [Cle+14]. However, Rempel and Mäder [RM15] showed that the number of related requirements or the average distance between related requirements have a positive correlation with the number of defects associated with these requirements. In addition, empirical data shows that traceability models also facilitate maintenance tasks and the evolution of software systems [ME15].

Due to the lack of sophisticated tool support, the opportunities discussed above are often missed [BMP13]. In contrast to the highly configurable traceability information models, traceability tools such as IBM Rational DOORS [IBM16] just offer a predefined set of evaluations, often with simple tree or matrix views [Sch12]. As a consequence, especially company- or project-specific information regarding software quality and project progress cannot be retrieved and thus remains unused.

This paper introduces a textual domain-specific language [MHS05] that is focused on describing customized query, metric, and rule expressions in the domain of software traceability. The

language is implemented using the Xtext framework that is part of the Eclipse ecosystem. Based on a grammar in EBNF-like format, a parser and an Eclipse Ecore Model, representing the meta model of the language, are generated [Bet13; Völ13].

The IDE generated by the Xtext language workbench provides extensible features such as syntax highlighting, live validation, code completion, and automatic formatting [The16c]. Additionally, Xtext validates references between concrete model elements that are available in the so-called scope of the current language element. The syntactically and semantically valid elements are determined by the configurable scope provider as part of the Xtext framework and may contain elements of different Ecore models [The16c].

20.3 Defining and Integrating the Domain-Specific Language

20.3.1 Composition of Modeling Layers

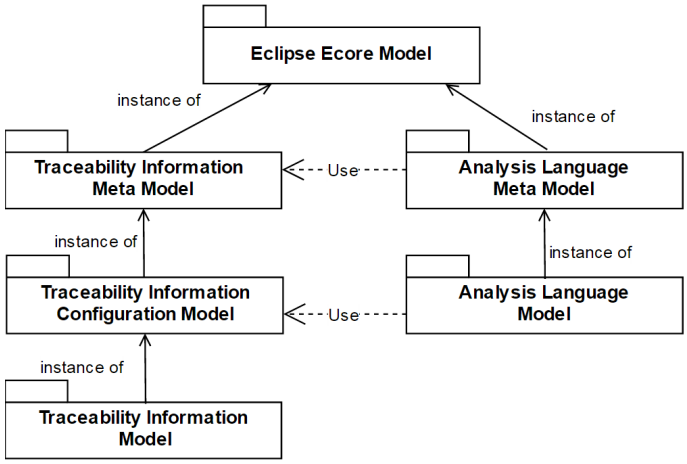


Figure 20.1: Conceptual Integration of Model Layers

Figure 20.1 shows the integration between the different model layers referred to in this paper, starting with the *Eclipse Ecore Model* as a shared meta meta model. The Xtext framework used to define the analysis language generates an instance of this model [The16c], representing the *Analysis Language Meta Model* (ALMM). Individual queries, metrics, and rules are specified within a concrete instance, the *Analysis Language Model* (ALM), by using the developed syntax.

Likewise, the *Traceability Information Model* used in this paper contains the actual traceability information, for example the concrete *Requirement "RQ1"*. It is again an instance of a formal abstract description, the so called *Traceability Information Configuration Model* (TICM). The TICM describes traceable artifact types, e.g. *Requirement* or *Java file*, and the available link types, e.g. *implements*. This model itself is based on a proprietary *Traceability Information Meta Model* (TIMM) that defines the basic constructs such as a traceable artifact type and a traceable link type by inheriting basic EClass and EReference elements of the Eclipse Ecore Model [Gro09].

Since the analysis language is related to the Eclipse Ecore Model, concepts such as EClass definitions can be referenced. Further references between concepts on the meta model layer (ALMM using TIMM) are the prerequisite for subsequently referencing instances on the concrete model layer. For instance, a query definition of the ALM could at some point reference the traceable artifact type "Requirement" in the TICM. On the model layer this reference is established by referring to a concrete instance of the traceable type, e.g. the result of the specific query "JavaClassesForRequirement" references "RQ1".

To structure the DSL, the analysis language itself is hierarchically subdivided into three components, namely the query, metric, and rule expressions. In order to establish clear interfaces, only query expressions may reference elements of the traceability information configuration model. Metric and rule expressions are built on top of query expressions and are thus independent from TICM and TIM. Live evaluation is performed by the query interpreter accessing the query expression's AST and applying the respective computation to the concrete TIM. The result of such a query execution (as elaborated in the following subsections) is then used by the interpreter to evaluate the metric and rule expressions.

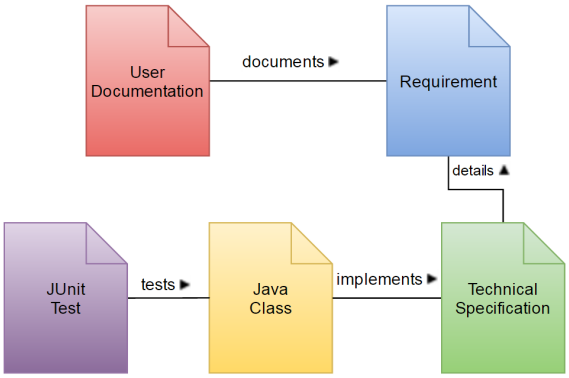


Figure 20.2: Traceability Information Configuration Model

Figure 20.2 shows an example traceability information configuration model that defines the traceable artifacts and link types. The arrowheads in Figure 20.2 represent the primary trace link direction. However, trace links can be traversed in both directions [CGZ12]. Besides being the abstract description for the TIM instances, the configuration model artifacts can be referenced within query language expressions.

20.3.2 Querying the Traceability Information Model

The query expressions are part of the analysis language which is defined as a textual domain-specific language using the Xtext framework.

```
1 query "tracesFromRequirementToJUnitTest"
2 traceFromTo(Requirement,JUnitTest) as paths
3     .collect(paths.getStart.getName as name,
```

```
4             count(1) as testCases)
5     .groupBy(paths.getStart.getName)
```

Listing 20.1: Sample Query Definition

?? shows an example query that retrieves the shortest path between each instance of *Requirement* and *JUnitTest* artifacts from a given TIM. The query definition starts with the keyword *query* that is followed by the actual name as string literal. The center of the query is the function *traceFromTo* that takes a source and a target traceable artifact as parameters. The available traceable artifacts are determined by the evaluation of the traceability information configuration model of the TIM. The function itself encapsulates an algorithm to find the shortest path between all instances of the two specified configuration model artifacts in the TIM. All resulting paths are assigned to the variable *paths* that is subsequently used to access these within the query. Each path object in the list offers several convenience functions such as `getStart` that returns the start element of the trace or `getEnd` that returns the last artifact of the trace.

The result of an executed query expression as well as the result of a metric or rule expression is returned as a tabular structure. For computing the query expression's result, the query interpreter iterates through all list entries in the variable *paths* that contain the results of the *traceFromTo* function called before. For every entry in the variable *paths* a new row in the tabular structure is created. The column definition is introduced by the keyword *collect*. For example in **??**, two column headings are defined called *name* and *testCases*, each introduced by the keyword *as*. The first part of the column definition is the reference to the variable *paths* that represent a list. The next part is introduced by "." which is followed by the functions available on a single entry of the path list. After entering ".", the Xtext proposal provider [The16c] will propose all functions available on each list element. Introduced by another ".", further method calls will be proposed based on the return type of the former function call. The second column specifies an aggregation function that counts all entries in a given column per row. Based on the column index passed as a parameter to the *count* function, the number of entries in each row of that column is counted. In general, the result of this function will be 1 per row since there is only one value per row and column but in combination with the *groupBy* function the number of aggregated values per cell is computed. The *groupBy* function of **??** aggregates all start artifacts ("Requirement") with the same name.

While queries can be expressed without an instance of the configuration model, the execution is done in a traceability information model. The query expression of **??** is applied to the example TIM shown in Figure 20.3 by the query interpreter. The result is a tabular structure with three rows and two columns where the first row contains the name "RQ1" in column one and the number "5" as count of the available traces in column two, for example. Even though there are five *Requirement* artifacts in the given TIM, only three of them are linked to a *JUnitTest* artifact so that there are three rows in the tabular result description.

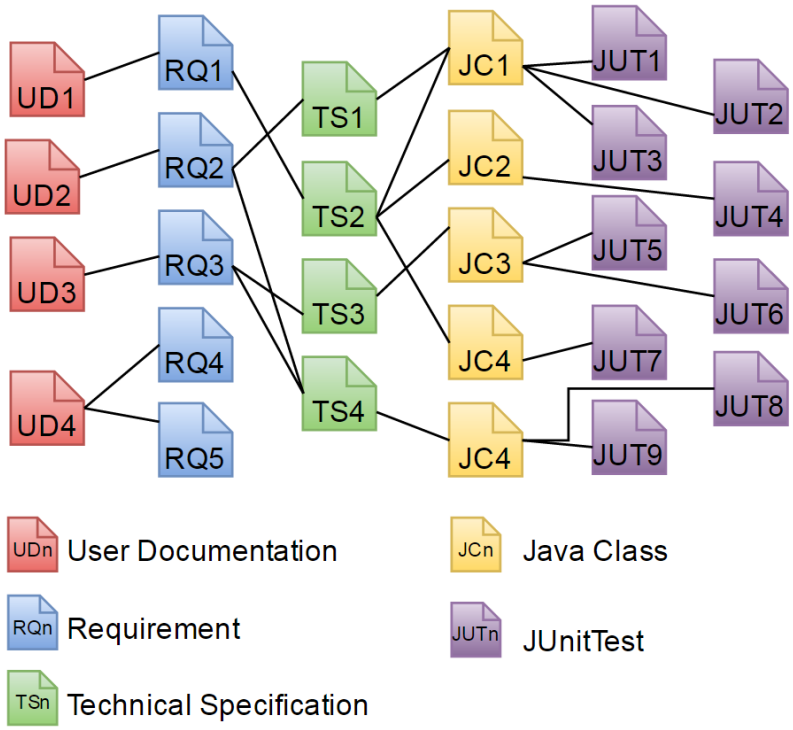


Figure 20.3: Traceability Information Model

The query expressions offer a powerful and well-integrated mechanism to retrieve information from a given TIM. At this point the data retrieved from the TIM can be understood as raw data that needs to be further processed to make use of it. In the following, it is explained how query expression results can be aggregated to metrics that represent information on the quality and progress of a project.

20.3.3 Defining Individual Metrics

A software quality metric can be defined as a "function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality" [Soco4]. A well-known and simple software metric is *lines of code* [Rig96], but there are also specific metrics for traceability models. These include, e.g., the average distance between requirements, the coverage between two levels of specification or full-depth from requirement to the lowest level, and linkage statistics concerning the count of related higher/lower level trace artifacts [RD95; RHH98]. Complimentary to recent research that focuses on specific traceability metrics and their relevance [RM15], the approach described in this paper introduces an analysis language to define individual metrics. By utilizing this approach, a project team or a quality department can establish and evolve custom metrics until they meet their specific requirements.

An Xtext grammar in EBNF-like format defines the available features including referencing queries, arithmetic operations, and operator precedence using parentheses. The metrics grammar of the analysis language itself has two main components. One is the result declaration that encapsulates the result of a previously specified query[1]. The other is an arbitrary number of metrics definitions that may aggregate query results or other metrics recursively.

```
MetricDefinition:
  'metric' name=ID '=' expression=MetricsExpression;

PlusOrMinus returns MetricsExpression:
  MulOrDiv(({Plus.left=current} '+' |
       {Minus.left=current} '-') right=MulOrDiv)*;

MulOrDiv returns MetricsExpression:
  MetricAtomic(({MulOrDiv.left=current} op=('*'|'/')) right=MetricAtomic)*;

MetricAtomic returns MetricsExpression:
  '('MetricsExpression')' |
  {MetricsRef} metric=[MetricsDefinition]|
  {ColumnSelection} col=ColumnSelection
  {DoubleConstant} value=DOUBLE |
  {SumFunction} sum= SumFunction |
  {CountFunction} count=CountFunction;
```

Listing 20.2: Grammar Rules for Metric Expressions

?? shows the most important rules that describe the possible arithmetic expressions used in metrics. Since the corresponding parser generated by ANTLR works top-down, the grammar must not be left recursive [Bet13]. ?? shows the grammar to support the four basic arithmetic operations as well as the correct use of parentheses. While the *PlusOrMinus* and *MulOrDiv* rules to enable ANTLR to handle a left recursive grammar are well described by Bettini (2013), the *MetricAtomic* rule contains the essential computations of the metrics grammar. First, the rule allows for the usage of constant double values. Second, metric expressions can contain pre-defined functions to sum up or count the results of a query. Third, columns from the result of a query can be referenced so that metric expressions per query expression result row can be computed. Finally, metric expressions can refer to other metric expressions to further aggregate already accumulated values as shown in the last line of ??.

```
SumFunction:
```

[1]The formal description of the syntax of a query is quite lengthy and extends beyond the scope of this paper, in which we focus on the metrics and rules language. From the example in subsection 20.3.2, the reader gets an impression of what a query looks like.

```
  'sum' '(' columns+=ColumnSelection
      (',' columns+=ColumnSelection)* ')';
```

Listing 20.3: Rule for Sum Aggregation

Exemplary for the predefined aggregation functions, the *SumFunction* rule shown in **??** describes the syntax for summing up the results of a query. After the keyword *sum*, any number of *ColumnSelections* can be stated as parameters. To compute the result, the interpreter will iterate over all the rows of the query expression's result and sum up the values of the referenced columns. **??** shows the actual *ColumnSelection* that is also used by the *CountFunction* and refers to the query expression's result columns. While a concrete metric is specified, the available columns are proposed in the editor based on the result declarations in the current scope. In addition, there is a live validation of the metric expressions that displays error markers as soon as a referenced result declaration or a column within the query expression result is no longer in scope.

```
ColumnSelection:
  resultDeclaration=[ResultDeclaration|ID] '.' column= [analysis::Column];
```

Listing 20.4: Rule for Column Selection

Using the given grammar, reusable metric expressions can be defined to compute the number of related requirements (NRR) as described by Rempel and Mäder (2015), i.e., the number of directly and indirectly referenced requirements from one requirement to another for all requirement artifacts in the TIM. The basis for this metric are all trace links from the TICM shown in Figure 20.2 that could be instantiated to describe a trace between two requirements.

```
result relatedRequirements from
traceFromTo(Requirement, Requirement) as paths
.collect(paths.getStart.getName as srcRequirement,
         paths.getEnd.getName as trgtRequirement)

metric NRR=cnt(relatedRequirements.srcRequirement)
```

Listing 20.5: Metric: Number of Related Requirements

The inlined query expression in **??** returns a tabular result that contains the shortest path between every pair of requirement artifacts from the given TIM. The path between each *Requirement* artifact can contain multiple other artifacts since the traceable links are bidirectionally navigable. In the example TIM from Figure 20.3, the trace from "RQ1" to "RQ2" contains the following artifacts "RQ1" to "TS2" to "JC1" to "TS2" to "RQ2". The problem of circular dependencies causing infinite loops in the computation of the available paths is solved by the depth-first algorithm implementation.

The first column of the tabular result structure contains the name of the source requirement in the path while the second column holds the name of the target requirement. The following NRR metric expression uses the *cnt* function to compute the number of related requirements.

Table 20.2: NRR Metric: Tabular Result Structure

Requirement	NRR
RQ1	2
RQ2	2
RQ3	2
RQ4	1
RQ5	1

Table 20.2 shows the tabular result structure of the metric expression executed against the sample TIM from Figure 20.3. The tabular result structure is made of two columns derived from the query and metric expression as described above.

```
result pathLengthPerRequirement from
traceFromTo(Requirement, Requirement) as paths
.collect(paths.getStart.getName as srcRequirement,
         sum(paths.getLength) as pathLength)
.groupBy(paths.getStart.getName)

metric ADRR= pathLengthPerRequirement.pathLength/ NRR
```

Listing 20.6: Composition of Metric Expressions

?? shows the reuse of an existing metric to compute the average distance between two requirements as explained by Rempel and Mäder (2015). The query expression shown in the example returns the shortest path between two requirements already aggregated per source requirement. The subsequent metric takes the *pathLength* column from the query expression result and divides every entry by the already computed metric NRR shown in **??**.

The combination of configurable query expressions with configurable metric definitions allows users to define their individual metrics. The analysis language is complemented by the rules grammar, which is described in the following section.

20.3.4 Evaluating Metrics

A metric value itself delivers few insights to the quality or the progress of a project. However, comparing a metric value to a pre-defined threshold or another metric value exposes information. The grammar contains rules for standard comparison operations which are *equal, not equal, greater than, smaller than, greater or equals*, and *smaller or equals*. A rule expression can either return a warning or an error result that may both be detailed by an individual message. Since query and metrics result descriptions implement the same tabular result interface as described above, rules can be applied to both. Finally, the result of an evaluated rule expression is also stored using the same tabular interface.

```
WarnIf returns RuleSpecification:
'rule' name=ID '=' 'warnIf(' ruleBody=RuleBody')';

RuleBody:
('m:' metric=[metrics::MetricsDefinition]|
 'c:' column=[ResultDeclaration|ID] '.' column=
      [analysis::Column]) compareOperator=Operator
      compareTo=RuleAtomic ',' msg=STRING;
```

Listing 20.7: Syntax for Rule Expressions

The *RuleBody* rule shown in **??** is the central part of the rules grammar. On the left side of the *compareOperator*, a metric expression or a column from a query expression result can be referenced. During definition of the rule, the metric expressions and query result columns available in the current scope are proposed. The next part of the rule is the *comparisonOperator* followed by a *RuleAtomic* value to compare the expression to. The *RuleAtomic* value is either a constant number or a reference to another metric expression. The evaluation of rule, metric and query expressions during run-time is implemented using Xtend, a Java extension developed as part of the Xtext framework and especially designed to navigate and interact with the analysis languages Eclipse Ecore models [The16b].

```
rule checkADRR=warnIf(m: ADRR >4.0, "High average distance between related requirements")
```

Listing 20.8: Sample Rule Definition

The metric specified in **??** is referenced by an instance of the warning grammar rule shown in **??** and it is compared to the value 4.0. In case that the metric contains a greater value, a warning message is produced and stored in the result object created by the rule interpreter. In other cases there will be an automatically produced message stating that the metric value is as expected. To create a staggered analysis, a warning and an error message for the same metric expression can be defined, thus classifying the result in two categories with varying severity. For example, an *ADRR* value greater than 4.0 causes a warning while an *ADRR* value higher than 6.0 causes an error message. The rule interpreter will recognize that there are two rule expressions based on the same metric and will only return one result.

The result of an interpreted rule expression contains not only information about the compared metric values but can also provide a meaningful warning or error message. All in all, the rule grammar finalizes the analysis language capabilities by providing mechanisms to compare aggregated information from a TIM against custom thresholds.

20.4 Discussion

To demonstrate the feasibility of the designed analysis language and perform flexible evaluations of traceability information models, a prototype was developed. The analysis language is based on the aforementioned Xtext framework and integrated into Eclipse using its plug-in infrastructure [The16a]. In addition, an interpreter was implemented that evaluates query, metric, and rule expressions ad hoc whenever the respective expression is modified and saved. Currently, both components are tentatively integrated in a software solution that envisages a commercial application. Therefore, the analysis language is configured to utilize a proprietary TIMM from which traceability information configuration models and concrete TIMs are defined.

Within our implementation, traceable artifacts from custom traceability information configuration models as shown in Figure 20.2 can be used for query, metric, and rule definitions. Due to an efficient implementation of the depth-first algorithm used by the *traceFromTo* function, queries are (re-)executed immediately when a query is saved. The efficiency of the depth-first algorithm implementation was verified by interpreting expressions using TIMs ranging from 1,000 to 50,000 traceable artifacts.

Table 20.3: Duration of Analysis

Total Artifacts	Start Artifacts	Duration (in s)
1,000	300	0.012
8,000	1,500	0.1
50,000	8,500	2.2

Table 20.3 shows the duration for interpreting the analysis expression from **??** against generated TIMs of different sizes. The first column shows the overall number of traceable artifacts and links in the TIM. The second column displays the number of start artifacts for the depth-first algorithm implementation, i.e. the number of "Requirement" artifacts for the exemplary analysis expression. The third column contains the execution time on a computer with Intel Core i7-4700MQ processor at 2.4 GHz and 16 GB RAM. As shown, executing expressions can be done efficiently even for large size models.

Defining and evaluating analysis statements with the prototypical implementation has shown that the approach is feasible to collect metrics for different kinds of traceability projects. The analysis language can for example be utilized to create company-specific metrics. Within the same industry sector some companies use a model-driven approach, others apply test-driven development, or directly start coding from a textual requirement. When a company uses an entity DSL to describe the data model of the application, it could be valuable to compute the average number of attributes per entity, assuming that a high number indicates bad design. In case of a company deciding to directly code from a textual requirement, a metric to calculate the number of classes per requirement in relation to the number of words in the requirement definition might

be reasonable to assess its specificity. If there is meaningful information in such company-specific metrics, the company initially needs to discover them. However, the more important finding is that company-specific metrics can be created, prototyped, and evolved easily by employing the analysis language.

Since the analysis language is based on a highly configurable TIMM, it allows for a large variety of traceable artifacts, including formerly unused documents such as documentation, test results or even tickets from collaboration tools [DP13]. Including artifacts from different systems, metrics can also be used to indicate the quality or the progress of a certain software product. A metric to measure the quality of a software component could compute the number of defects related to a specific unit of code by traversing the TIM to find all links between those two artifacts. Using a project-specific rule to highlight code units causing a high number of defects gives an indication on where to perform quality measures, e.g. code reviews. The current progress of a software development project can be exposed by defining a staggered analysis. Taking the exemplary TICM from Figure 20.2, a first query could find all *Requirement* artifacts that are not linked to a *Technical Specification* artifact. From a project management perspective the design of these requirements could be understood as not started. The next part of the staggered analysis could retrieve the *Requirement* artifacts that are linked to a *Technical Specification* artifact, but have no trace to a *Java Class* artifact, therefore indicating that the implementation has not yet begun. Relating the described query expression results to the overall number of *Requirement* artifacts measures the project's progress with regard to different phases of development.

The approach presented in this paper is bound to limitations. In particular, an investigation of real world projects is pending in order to assess the impact of the developed DSL on software quality management practices.

In addition, any analysis is only as good as the underlying traceability information model, thus requiring wary treatment of metrics results. For example, we discovered missing trace links through our queries in a preliminary analysis of a real world TIM, as those were configured in the TICM but never instantiated in the TIM. On the positive side, analysis statements are usually defined and executed by domain experts, so that problems with the underlying data can often be identified and resolved quickly.

These are, however, no inherent limitations of the approach but rather constitute future work in deploying the DSL in real-world scenarios that benefit from traceability metrics.

20.5 Conclusion

The DSL described in this paper offers functions and expressions to analyse existing TIMs, structured in query, metric, and rule component. For retrieving traceable artifacts and trace links, query expressions can be defined. In a subsequent metric expression, the results of an interpreted query are condensed to a comparable value using arithmetic operations and aggregate functions.

In order to assess this value in context, rule expressions are defined to compare metrics' values among each other or against a threshold value.

The introduced approach closes the gap between information retrieval, metrics definition, and result evaluation, thus forming a solid foundation for project- or company-specific metrics. Regarding flexibility, a configuration model makes it completely independent from the specific type of traced artifacts. Further, it is well integrated into an established workbench and development environment using Xtext and the Eclipse Modeling Framework. Features such as live validation and error markers detect broken or outdated expressions early and ensure a rich user experience.

The focus of future work should be on identifying new industry-relevant metrics by applying the proposed approach to real-world projects. Also, the data mining field offers statistical methods through association rules or regression algorithms to find patterns and gain insights from large data sources such as traceability models.

To sum up, the analysis language proposed in this paper offers an integrated approach to close the gap between querying traceability information models and defining configurable metric expressions. The concept of ad hoc evaluation of expressions was demonstrated in a prototypical implementation.

References

[Bet13] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Community experience distilled. Birmingham, UK: Packt Pub, 2013.

[BMP13] E. Bouillon, P. Mäder, and I. Philippow. "A survey on usage scenarios for requirements traceability in practice". In: *Lecture Notes in Computer Science* 7830 LNCS (2013), pp. 158–173. DOI: 10.1007/978-3-642-37422-7{\textunderscore}12. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84875822055&partnerID=40&md5=3d5caa608a983edb8ac1d29809bf345e.

[BRK17] Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. "A Domain-specific Language for Configurable Traceability Analysis". In: *5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. Ed. by Luis Ferreira Pires, Slimane Hammoudi, and Bran Selic. 2017, pp. 374–381.

[CGZ12] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, eds. *Software and Systems Traceability*. London: Springer London, 2012.

[Cle+12] J. Cleland-Huang et al. "Trace queries for safety requirements in high assurance systems". In: *Lecture Notes in Computer Science* 7195 LNCS (2012), pp. 179–193. DOI: 10.1007/978-3-642-28714-5{\textunderscore}16. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84858313806&partnerID=40&md5=3b7c58911e01ff98d0e6457b54d91620.

[Cle+14] Jane Cleland-Huang et al. “Software Traceability: Trends and Future Directions”. In: *Proceedings of the on Future of Software Engineering*. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 55–69. DOI: 10.1145/2593882.2593891. URL: http://doi.acm.org/10.1145/2593882.2593891.

[DP13] Alexander Delater and Barbara Paech. “Analyzing the Tracing of Requirements and Source Code during Software Development”. In: *Requirements Engineering: Foundation for Software Quality*. Springer Berlin Heidelberg, 2013, pp. 308–314. DOI: 10.1007/978-3-642-37422-7{\textunderscore}22. URL: http://dx.doi.org/10.1007/978-3-642-37422-7_22.

[GF94] Orlena Gotel and Anthony Finkelstein. “Analysis of the requirements traceability problem”. In: *International Conference on Requirements Engineering* (1994). URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0028012990&partnerID=40&md5=848b8948ed6e01ee735b3399fb81d813.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1st. Addison-Wesley Professional, 2009.

[IBM16] IBM. *Rational DOORS*. www.ibm.com/software/products/en/ratidoor. 2016. URL: www.ibm.com/software/products/en/ratidoor (visited on 12/13/2016).

[MC09] J. I. Maletic and M. L. Collard. “TQL: A query language to support traceability”. In: *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE 2009* (2009). DOI: 10.1109/TEFSE.2009.5069577. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-70349912061&partnerID=40&md5=bd12be8c28a1bec30cc429c157117fce.

[MC13] P. Mäder and J. Cleland-Huang. “A visual language for modeling and executing traceability queries”. In: *Software and Systems Modeling* 12.3 (2013), pp. 537–553. DOI: 10.1007/s10270-012-0237-0. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84879788036&partnerID=40&md5=3a579838a265be634c63b0e7eeacd8d1.

[ME15] Patrick Mäder and Alexander Egyed. “Do Developers Benefit from Requirements Traceability when Evolving and Maintaining a Software System?” In: *Empirical Softw. Eng.* 20.2 (2015), pp. 413–441. DOI: 10.1007/s10664-014-9314-z. URL: http://dx.doi.org/10.1007/s10664-014-9314-z.

[MGP13] Patrick Mader, Orlena Gotel, and Ilka Philippow. “Getting back to basics: Promoting the use of a traceability information model in practice”. In: *7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)* (2013), pp. 21–25. DOI: 10.1109/TEFSE.2009.5069578.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and How to Develop Domain-specific Languages”. In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892. URL: http://doi.acm.org/10.1145/1118890.1118892.

[RD95] Rita J. Costello and Dar-Biau Liu. "Metrics for requirements engineering". In: *Journal of Systems and Software* 29.1 (1995), pp. 39–63. DOI: 10.1016/0164-1212(94)00127-9. URL: http://www.sciencedirect.com/science/article/pii/0164121294001279.

[RHH98] Linda Rosenberg, Theodore F. Hammer, and Lenore L. Huffman. "Requirements, testing and metrics". In: *15th Annual Pacific Northwest Software Quality Conference*. 1998.

[Rig96] Fabrizio Riguzzi. *A Survey of Software Metrics*. 1996.

[RM15] Patrick Rempel and Patrick Mäder. "Estimating the Implementation Risk of Requirements in Agile Software Development Projects with Traceability Metrics". In: *Requirements Engineering: Foundation for Software Quality*. Springer International Publishing, 2015, pp. 81–97. DOI: 10.1007/978-3-319-16101-3{\textunderscore}6. URL: http://dx.doi.org/10.1007/978-3-319-16101-3_6.

[Sch12] Hannes Schwarz. *Universal traceability*. [Place of publication not identified]: Logos Verlag Berlin, 2012.

[Soc04] IEEE Computer Society. *IEEE Standard for a Software Quality Metrics Methodology: IEEE std 1061-1998 (R2004)*. Tech. rep. IEEE Computer Society, 2004.

[Ste+08] Dave Steinberg et al. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[The16a] The Eclipse Foundation. *PDE/User Guide*. http://wiki.eclipse.org/PDE/User_Guide. 2016. URL: http://wiki.eclipse.org/PDE/User%5Ctextunderscore%20Guide.

[The16b] The Eclipse Foundation. *Xtend Modernized Java*. http://www.eclipse.org/xtend/. 2016. URL: http://www.eclipse.org/xtend/ (visited on 09/19/2016).

[The16c] The Eclipse Foundation. *Xtext Documentation*. https://eclipse.org/Xtext/documentation/. 2016. URL: https://eclipse.org/Xtext/documentation/ (visited on 09/19/2016).

[Völ13] Markus Völter. *DSL engineering: Designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform, 2013.

21

Conquering the Mobile Device Jungle: Towards a Taxonomy for App-Enabled Devices

Table 21.1: Fact sheet for publication P15

Title	Conquering the Mobile Device Jungle: Towards a Taxonomy for App-Enabled Devices
Authors	Christoph Rieger[1] Tim A. Majchrzak[2] [1] *ERCIS, University of Münster, Münster, Germany* [2] *ERCIS, University of Agder, Kristiansand, Norway*
Publication Date	2017
Conference	13th International Conference on Web Information Systems and Technologies (WEBIST)
Copyright	SciTePress CC BY-NC-ND 4.0
Full Citation	Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-enabled Devices". In: *13th International Conference on Web Information Systems and Technologies (WEBIST)*. Porto, Portugal, 2017, pp. 332–339. DOI: 10.5220/0006353003320339

Conquering the Mobile Device Jungle: Towards a Taxonomy for App-Enabled Devices

Christoph Rieger Tim A. Majchrzak

Keywords: App, Mobile App, Taxonomy, Categorization, Smart devices, Wearable, Smartphone, Tablet

Abstract: Applications for mobile devices (apps) have created an ecosystem that facilitated a trend towards task-oriented, interoperable software. Following smartphones and tablets, many further kinds of devices became (and still become) app-enabled. Examples for this trend are smart TVs and cars. Additionally, new types of devices have appeared, such as Wearables. App-enabled devices typically share some characteristics, and many ways exist to develop for them. So far for smartphones and tablets alone, issues such as device fragmentation are discussed and technology for cross-platform development is scrutinized. Increasingly, app-enabled devices appear to be a jungle: It becomes harder to keep the overview, to distinguish and categorize devices, and to investigate similarities and differences. We, thus, set out with this position paper to close this gap. In our view, a taxonomy for app-enabled devices is required. This paper presents the first steps towards this taxonomy and thereby invites for discussion.

21.1 Introduction

The continuous growth of the mobile device market [Sta16] and the recent emergence of devices such as smartwatches [Chu+16] and connected vehicles [CM16] has attracted much attention from academia and industry. In the past decade, particularly the app ecosystem facilitated a trend towards task-oriented, interoperable software, arguably started with the advent of Apple's iPhone in 2007 [App07] and the App Store in 2008 [App08]. For *traditional* mobile devices (i.e. smartphones and tablets), the competition has yielded two major platforms (Android and iOS). Several approaches for cross-platform development have been proposed to avoid the costly re-development of the same app for different platforms (cf., e.g., [HHM13; El-+15]).

Technological development has continued and many new device types have emerged. Most of them fall under the umbrella term *mobile devices* and are more-or-less *app-enabled*. The latter denotes that it typically are apps that make such devices particularly useful and that extend the possibilities they offer. However, they differ greatly in intended use, capabilities, input possibilities, computational power, and versatility, to name just a few aspects. Smart devices such as smart watches and smart TVs are most prominent in the realm of consumer devices but plenty of possibilities exist. Lines towards sensor-driven devices for the Internet of Things (IoT) are often blurred and it is not always clear how to properly categorize a device. This makes it hard to discuss, or, actually, to even correctly name them. Being usually app-enabled, these devices provide new opportunities for intelligent and context-adaptive software but at the same time pose technical challenges regarding the development for new platforms and regarding heterogeneous hardware

features. Moreover, app-enablement does not necessarily bring compatibility and portability whereas running the same app on a variety of devices is normally desirable.

While a plethora of case studies and contributions for individual device types such as smartphones and tablets can be found in the scientific literature (e.g., [HMK13; JJ14; CMK14; Bus+15; Dag+16]), a comprehensive study of the general field of app-enabled devices is missing. This paper sets out to close this gap by contributing a taxonomy for app-enabled consumer devices. While we have put much effort into literature work (cf. the next section), the useful literature base is small. Therefore, we propose research-in-progress, arguing for our ideas to stimulate work on this topic. The eventual aim is the discussion on a conference scale for extension into a state-of-the art paper, possibly further leading to a kind of standard.

The structure of this paper is as follows: After discussing related work in Section 21.2, our proposal for a taxonomy is presented in Section 21.3. Section 21.4 discusses the taxonomy with regard to its current and future applicability before we conclude in Section 21.5.

21.2 Related Work

For the topic of our paper on the one hand a plethora of related work exists, on the other hand hardly any closely-related approaches can be cited. Particularly since Apple's iPhone initiated the growth of the smartphone device class, many papers have been published on the *modern* notion of mobile computing, centring around devices that are propelled by apps. However, even overview papers typically focus on one category of devices. For example, Jesdabodiand Maalej [JM15] classify apps by usage states but limit themselves to smartphones. Moreover, the scientific literature so far has only rudimentarily captured the latest developments in device development. Chauhan et al. [Cha+16], for instance, provide an overview of smartwatch app markets with focus on the type of apps as well as privacy risks through third party trackers.

To make sure that we do not miss an existing taxonomy (or similar work), we did an extensive literature search. We focus on work from 2012 or later, where the first broader range of smart watches such as the Pebble had already been presented. Together with the increasing variety in devices, new operating systems have appeared since then. Examples are Android Wear and watchOS, which focus on wearable devices [Goo16; App16] as well as webOS and Tizen, which address a wider range of smart devices [LG 16; The16].

We deliberately excluded the keywords *application* and *system*. The first yielded many results that were not applicable since the term was mostly used to mean *utilization* of something. The latter had originally been used to describe e.g. cyber-physical systems but now proved to be too generic. Also, the medical area was excluded as these papers focus on apps for therapeutic purposes and do not contribute to the question of app-enabled devices. We thus used the following search string in the Scopus database:

```
TITLE-ABS-KEY(
(app-enabled OR app OR app-based)
AND
(mobile OR smart OR intelligent OR portable)
AND
(device OR vehicle OR "cyber-physical system" OR CPS OR gadget)
AND
(classification OR categorization OR overview OR comparison OR review OR survey OR
framework OR model OR landscape OR "status quo" OR taxonomy))
AND PUBYEAR AFT 2011
AND ( EXCLUDE ( SUBJAREA , "MEDI" ) )
```

A search on 07-12-2016 yielded 998 results. Of these, not a single paper provided an approach for classification, let alone a complete taxonomy. Only one paper [KK16] went beyond a perspective on "classical" mobile devices and considered heterogeneous device types. To complicate matters, papers mention that there are other *smart* devices than smartphones but do not go into detail.

In summary, the result set reveals no closely related work to which we can limit ourselves to. However, we can draw from a myriad of sources that tackle *some* aspects that are relevant for a taxonomy of app-enabled devices. This finding aligns with the motivation for our paper. Obviously, other authors struggled with putting different device categories into context because no proper framing exists.

Although not necessarily focused on multiple device categories, work on cross-platform app development is conceptually related. Usually, cross-platform development exclusively targets traditional mobile devices such as smartphones and tablets, e.g. "the diversity in smart-devices (i.e. smartphones and tablets) and in their hardware features; such as screen-resolution, processing power, etc." [HEE13]. However, considering the differences in platforms, versions, and also at least partly in the hardware is similar to considering a different type of device. In fact, the difference in screen size between some Wearables and smartphones with small screens is less profound than between the same smartphones and tablets. Therefore, comparisons that target cross-platform app development have paved the way towards this paper. This particularly applies to such works that include an in-depth discussion of criteria, such as by [HHM13], [SK13], [Dal+13], and [RM16].

A part of the difficulty with related work is the term *app-enabled* by itself. While it is often said that devices are enabled by apps, or that apps facilitate their functionality, it is usually not explained what this exactly means. The typical usage that we also follow is to denote an app-enabled device as one that by its hardware and basic software (such as the operation system or *platform*) alone provides far less versatility than it is able to offer in combination with additional applications. Such apps are not (all) pre-installed and predominantly provided by third party developers unrelated to the hardware vendor or platform manufacturer; moreover, the possibilities provided by apps typically increase over time *after* a device has been introduced. While this still

is no profound definition, it provides a demarcation for the time being. In particular, it rules out pure Internet-of-Things devices as well as computational equipment that only is occasionally firmware-updated or that is not built for regular interaction with human users.

21.3 Taxonomy of App-Enabled Devices

Categorizing app-enabled devices is difficult due to the variety of possible hardware features *across* all types of devices and the heterogeneity of device capabilities *within* each class. Any simple solution is prone to not sufficiently discriminate. For example, processing power does not differ a lot between smartphones and tablets anymore, and microphones are no distinguishing feature for voice-controlled devices. In addition, the fast-paced technological progress manifests as a constant stream of new devices, partly rendering previous devices obsolete. Moreover, device types converge, illustrated e.g. by the *phablet* phenomenon (devices that fall in between smartphones and tablets). Mobility in the strict sense even is no exclusive feature; smart TVs for example are not really mobile. Cars with smart entertainment systems or even self-driving features might be app-enabled, but it can be disputed whether the whole car is the *device* and thereby the device is actually mobile by itself. As a result, a taxonomy of app-enabled devices mandates a more open categorization along several dimensions, allowing for partial overlaps and future additions. In the following, we present steps towards such a taxonomy. First, the three chosen dimensions are discussed as static structure for classification. Subsequently, current and foreseeable future device classes are positioned according to this matrix.

21.3.1 Dimensions of the Taxonomy

This work positions app-enabled devices with regard to the three dimensions *media richness of inputs*, *media richness of outputs*, and the *degree of mobility*. Instead of enumerating concrete technologies that are available today or in future, each dimension should rather be regarded as continuously increasing intensity and flexibility of the particular capability, with several exemplary cornerstones depicted in the following. This approach not only provides the highest degree of objectivity but also should keep the taxonomy flexible enough to capture future developments without actually changing the dimensions.

Media richness of inputs describes the characteristic user input interface for the respective device class.

- *None* refers to fully automated data input through sensors.[1]
- *Buttons*, including switches and dials, are (physically) located at the device itself and provide limited input capabilities.
- *Remote controls*, including also joysticks and gamepads, refer to dedicated devices that are tethered or wirelessly connected to the app-enabled device.

[1] Strictly, most if not all input is done via sensors, but *none* at this point denotes no manual activity by a user.

- *Keyboards* are also dedicated devices to control the target devices, but with more flexible input capabilities due to a variety of keys. Input still is discrete.
- *Pointing devices* refer to all dedicated devices to freely navigate and manipulate the (mostly graphical) user interface, for example mouse, stylus, and graphic tablet. While these devices technically still provide discrete input, the perception of input is continuous.
- *Touch* adds advanced input capabilities on the device itself, allowing for more complex interactions such as swipe and multi-touch gestures.
- *Voice*-based devices are not bound to tangible input units but can be controlled without haptic contact.
- *Gestures* allow for a hands-free visual user interaction with the device, for example using gloves or motion sensing.
- *Neural interfaces* are the richest form of user inputs in the future by directly tapping into the brain or nervous system of the human operator.[2]

As second dimension, *media richness of outputs* describes the main output mechanisms for the respective device class.

- *None* refers to no user-oriented communication by the device itself. This applies to cyber-physical actuators with direct manipulation of real-world objects (e.g. switching on light). This also includes pass-through mechanisms that in general or in some situations do not produce a tangible output of their own but pass it through to a connected managing device (e.g., a smartphone) which retrieves information and handles user output.
- *Screen* output is a major form of user communication found in app-enabled devices. Although a clear subdivision is not possible, several classes are typical, ranging from tiny screen displays (<3”) to small screens such as for smartphones (<6”), medium screens for handheld devices (<11”), large screens (≤ 20”), and usually permanently installed huge screens >20”.
- *Voice*-based output refers to the first type of disembodied device output that communicates with the user without physical contact.
- *Projection* extends the disembodiment with visual output to a device-external location, including augmented reality applications and hologram representations.
- *Neural interfaces* connect directly to the user in order to a achieve a tightly coupled human-computer interaction.

Finally, the combination of input and output characteristics ignores different application areas of the respective device class. For example, intelligent switches and drones for aerial photography can both be remotely controlled and have no direct output, but can hardly be grouped as being in the same device class. Therefore, the *degree of mobility* describes the usage characteristics as third dimension.

[2] Since the possibilities of neural interfaces are yet very limited, future developments *might* mandate splitting up this category into different kinds of neural interfaces.

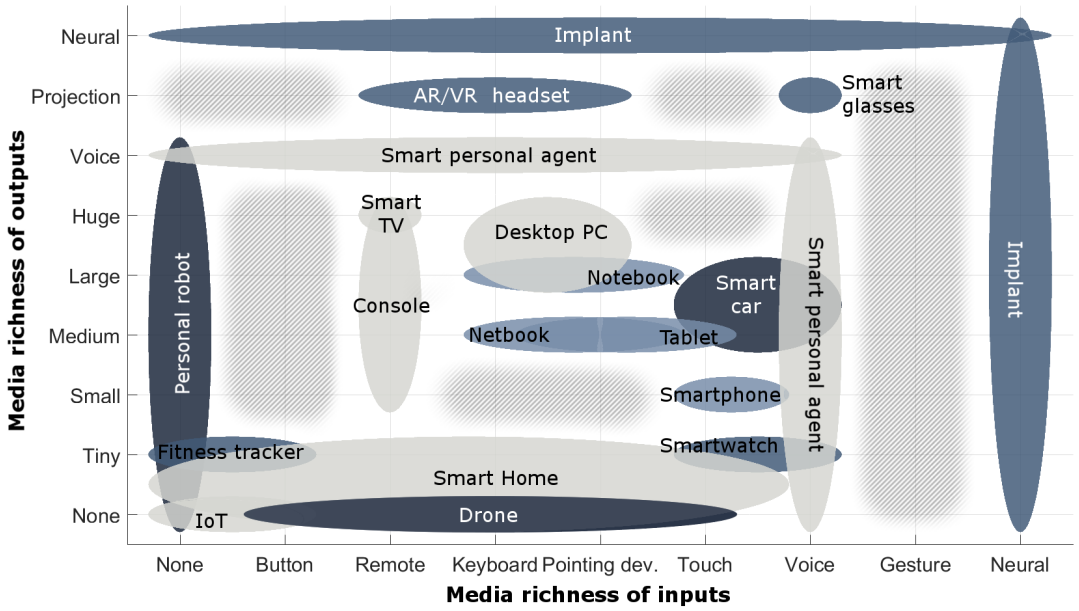


Figure 21.1: Matrix of Input and Output Dimensions

- *Stationary* devices are permanently installed and have no mobile characteristics during use.
- *Mobile* devices can be carried to the place of use.
- *Wearable* devices are designed for a more extensive usage and availability through the physical contact with the user. In contrast to "mobile", transporting the device is implicit and usually hands-free.
- *Self-moving* devices provide the capability to move themselves (controlled by the user). Ultimately, autonomous devices represent the richest form of mobility for app-enabled devices.

21.3.2 Categorizing the Device Landscape

The proposed dimensions allow for an initial categorization of the device landscape. Figures 21.1 to 21.3 (page 476) visualize the three-dimensional categorization of different device classes using the respective two-dimensional projections for better readability.

As depicted in Figure 21.1, many devices classes can be assigned to distinct positions in the two-dimensional space of input/ output media richness. However, it should be noted that the ellipses represent (current) major interaction mechanisms within the device class. For example, *smartphones* also have a few physical buttons but are usually operated by touch input. Individual devices may also deviate from the presented position, for instance specialized or experimental devices that do not (yet?) constitute a distinct class of devices. Additionally, not all devices falling into a device class must necessarily implement all possibilities of that class. Therefore, ellipses are a well-suited representation as opposed to, e.g., the maximum value for the respective devices.

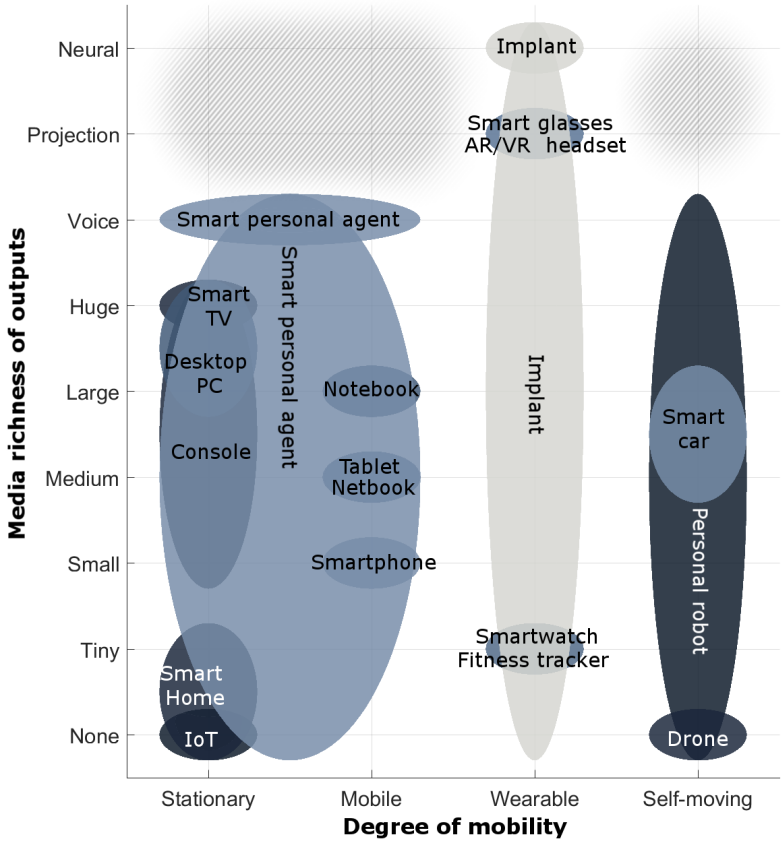


Figure 21.2: Matrix of Output and Mobility Dimensions

The chosen level of abstraction implies that the taxonomy *dimensions* are intended to be rather static. Instead of chasing the actual technological development to reflect the latest emergence of devices, only seldom and slow changes are necessary to keep them up to date. Nevertheless, the categorization of *classes* is more dynamic and will need to be regularly checked for continued relevance. Moreover, classes might need to be split or at least be adapted regarding their placement on the dimensions' continuum when new possibilities arise. For this reason, and also for the brevity of a position paper such as this, we do not provide detailed definitions for each class. Rather, we explain them exemplarily and rely on the general understanding of the well-known classes (such as smartphones).

Figure 21.1 reveals differences in the specificity (i.e., represented size) of the device classes. Some of them fill specific spots in the diagram, either due to technical restrictions (*smart TVs* evolve traditional remote-controlled TVs with large screens) or special purposes (*smart glasses* enable hands-free interaction and visualization). Less specific device classes exist for two reasons. On the one hand, terms such as *smart home* comprise every technology that relates to a specific domain, subsuming very heterogeneous devices. On the other hand, *underspecified* device classes such as *implants* and *smart personal agents* are presented as they are due to their novelty; there

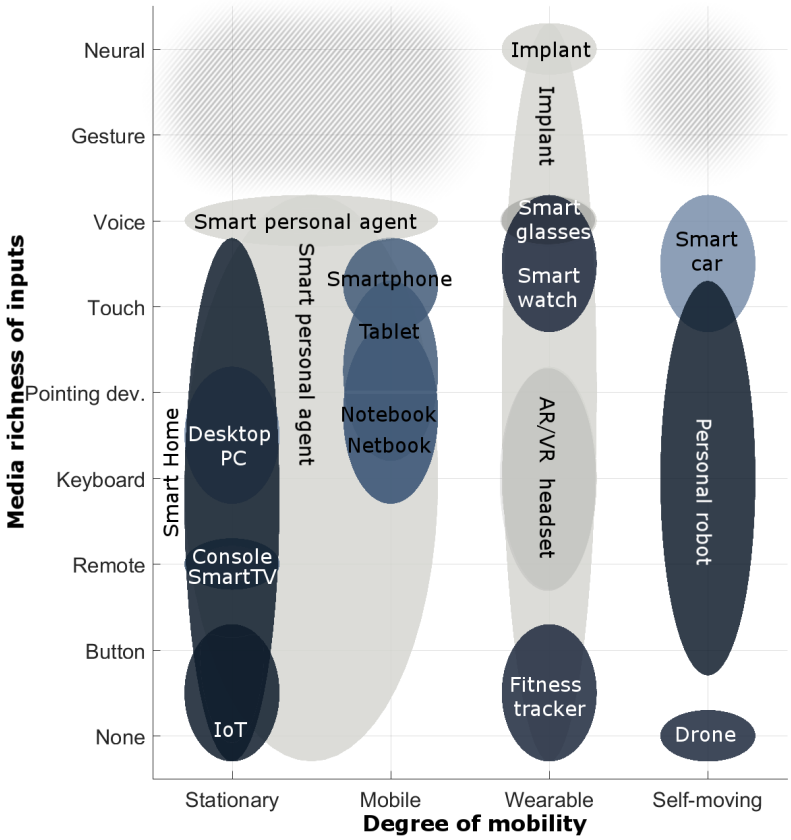


Figure 21.3: Matrix of Input and Mobility Dimensions

are few devices on the market and a high level of uncertainty must be ascertained regarding future hardware characteristics and interaction patterns. We expect to be able to draw a more concrete picture with a future version of the taxonomy, both based on the discussion of this paper and the meanwhile ongoing technological progress.

Differences in the device classes can also be explained with regard to *media richness theory* (MRT). MRT describes a corridor of effective communication with matching levels of message ambiguity and media richness [DLT87]. When applying this idea to the input and output characteristics of app-enabled devices, similar observations can be made. For example, *IoT* devices have only rudimentary direct user input possibilities but also give not much feedback in return. Notebooks allow for medium levels of input richness through keyboard and mouse input, with large screens as more flexible output capabilities. Furthermore, *smart glasses* directly embed their output into the real world by projection. Consequently, their voice-based input is equally rich in order to handle complex interactions with the user.

This correlation also partly explains why there are areas in Figure 21.1 with no assigned device class. Rich forms of user input such as gestures overcomplicate interactions for devices that have just small screens. On the other extreme, devices with barely a few buttons do not provide

sufficiently flexible input capabilities to manipulate large screens. Of course, empty spaces in the taxonomy might also be caused by a lack of use cases so far. Thus, they might actually be filled by future devices, or existing classes might "stretch" into these areas. In general, with the evolution and differentiation of input media, existing device classes might extend towards further areas or even converge. E.g., consider convertibles as hybrid devices between keyboard-based *netbooks* and touch-optimized *tablets*.

Figure 21.2 depicts the combination of output media richness and mobility. Unsurprisingly, a general tendency towards large screen output for stationary devices can be observed. With increasing mobility, screen sizes tend to diminish, from small screens on *smartphones* to very limited *fitness tracker* screens and screen-less *drones*. Beyond mobile devices, output capabilities become more flexible, potentially caused by specific domains of application, or their novelty with insufficient time to establish wide-spread interaction patterns. Some recently emerged device classes explore intangible output capabilities, for instance *augmented / virtual reality (AR/VR) headsets*. Others, especially autonomously moving devices such as *smart cars* and *personal robots*, are driven by the increased availability of sensor technology and not restricted to particular output capabilities.

Finally, Figure 21.3 visualizes the relationship between input media richness and mobility. Usually, an increasing degree of mobility entails less physical input mechanisms with dedicated buttons and keys. This might be attributed to practicability reasons, for example using voice commands is easier for wearable *smart glasses* than requiring dedicated input devices. In addition, smarter devices are usually more complex with regard to their output, and equally sophisticated input capabilities are necessary to match this level as explained by media richness theory. *Consoles*, for instance, provide basic navigation functionalities. *Desktop personal computers* and *notebooks* can be equipped with intelligent software such that keyboard and mouse are helpful means for interaction, and *smart personal agents* integrate advanced interpretation mechanisms that allow for voice-based communication in everyday situations.

21.4 Discussion

Due to the rapid proliferation of the field and the initial character of this work, it is likely that amendments will need to be made. Moreover, we will need to keep updating the taxonomy once it has been acknowledged by the scientific community. In addition, a taxonomy should also be appealing for the use by practitioners, particularly in a field like Mobile Computing where scientific research and technological progress go hand-in-hand. Therefore, this section presents ideas for discussion that go beyond Section 21.3.

21.4.1 Alternative Categorization Schemes

Devices can be categorized according to other device features. Not all are compatible with our taxonomy, nevertheless we deem several of them noteworthy.

Simple schemes such as a categorization by hardware feature (e.g., camera, computing power, touch screen) or usage (e.g., business, entertainment, sports, or communication use) fail to provide clear criteria for a taxonomy. In particular, a fast adaptation and convergence of available technologies could be observed in the past years. For example, so-called *phablets* blur the lines between smartphones and tablets, and gyroscope sensors have found wide-spread adoption in a variety of devices.

Matrix-based categorizations allow for a better juxtaposition on two dimensions, for instance regarding the input and output characteristics of app-enabled devices. However, the heterogeneity of devices within a device class provides insufficient discriminating power. For example, medium-sized and touch-based screens are usual interfaces both for tablets and smart cars. Similarly, distinguishing between apps for embedded or stand-alone devices is not always possible due to different types of device integrations within a device category (cf. e.g. [CM16] for smart cars).

Therefore, the third dimension chosen for our taxonomy adds the degree of mobility to distinguish between similar device hardware in different usage contexts. Other potential approaches for categorizing devices include the degree of integration, automation, or intelligence attainable or provided by the device. This reaches from simple input / output devices with limited app interaction (such as fitness trackers), to interoperable software (such as smartphones), highly cross-linked and automated devices (in the IoT or smart home field), and finally to intelligent machines. While we deem it reasonable to discuss such an optional fourth dimension, we do not think the taxonomy would gain more discriminatory power.

21.4.2 Further Development

Firstly, future discussion needs to include the demarcation of devices to be included. As argued earlier, mobility is not necessarily the proper boundary. App-enablement has proven to be feasible, yet we will need to find (or provide) a profound definition for it.

Secondly, it needs to be determined how the taxonomy can be kept up to date. In many other cases, taxonomies have proven to be either too detailed and thus requiring constant adjustments, or too little detailed and thus lacking discriminatory power. Due to a restriction to three orthogonal dimensions and clearly distinguishable values in each of it, we are optimistic that the taxonomy will be future-proof. Nevertheless, proper ways of deciding when adaptations are needed and what developments can be reflected without changes need to be defined. As part of this, we will need to scrutinize how to handle the differences in precision regarding categories. For example, it is very well understood what a smartphone is; smart homes, and to an even higher degree neural devices are (yet) diffuse with a lack of devices and/or applications to characterize them.

Thirdly, we so far have limited ourselves to consumer devices. This includes many devices that are *also* used for professional purposes, but arguably not all. Beyond that, some specialised devices are (so far) solely used for professional means. Examples can be found in industry, particularly in logistics. However, some of these might simply be subsumed by consumer devices. It could be said that e.g. the devices used by parcel couriers are very similar to smartphones, despite the difference in form and the absence of a general purpose utilization. The same applies to special devices from areas such as healthcare or crisis prevention and response. While such devices typically have specific capabilities (such as error-tolerance), on an abstract level they again are very similar to general purpose hardware. Thus, an updated taxonomy could try to include non-consumer devices. However, due to the complexity that arises particularly with devices that are so specialised that information on them is scarce, we deem the current limitation justified.

Fourthly, it should be scrutinized how the taxonomy can be provided in a form that is useful both for researchers and for practitioners. Most scientists know taxonomies for research topics enforced by publication outlets. Quite often these feel more like a "try to fit somewhere" game, particularly if a paper tackles a contemporary topic and the taxonomy provides little flexibility. If we want our taxonomy to be helpful for researchers, and – probably even hard to achieve – employed by practitioners, it needs to be easy to use yet powerful. Achieving this will be very valuable, as can e.g. be seen for cross-platform development, where new approaches can be clearly categorized by their characteristics.

The above discussion points have also shown the limitations of our work. Besides the issues that need to be worked on, an eventual verification of the taxonomy is mandated. For now, many tasks remain but a first – we deem noteworthy – step has been done. Further steps are sketched in the next Section.

21.5 Conclusion and Outlook

In this paper we have presented a taxonomy for app-enabled devices – it is the first such work. Based on three dimensions that take into account the input and output characteristics, we have built the taxonomy. Categorizing the device landscape and plotting the results into figures illustrates the discriminatory power of our taxonomy. However, as the first comprehensive work on the topic, we seek to amend and refine it. For now, it is put up for discussion. Nonetheless, due to the soundness of the dimensions and their alignment with availably theory, we are optimistic that this position paper provides a profound step towards a systematic overview of app-enabled devices.

Our future work instantaneously follows this precondition. We seek to discuss the taxonomy as part of a conference presentation and, subsequently, with interested colleagues. Moreover, we will also seek to have practitioners scrutinize it. The next step will be to provide an amended version of the taxonomy, along with an extended discussion of device classes. Eventually, empirical verification will be necessary.

The outlook is determined by an aspiration for our work. It should not become "yet another computer science taxonomy". Rather, it should prove useful in allowing

- authors to more clearly express what kind of device(s) they are referring to,
- researchers and practitioners to gain more discriminatory power when speaking about modern mobile computing devices, and
- the general public a more straightforward way of understanding similarities and differences between devices, both technically and tangibly.

References

[App07] Apple Inc. *Apple Reinvents the Phone with iPhone*. 2007. URL: http://%20www.apple.com/pr/library/2007/01/09Apple%20-Reinvents-the-Phone-with-iPhone.html (visited on 01/11/2017).

[App08] Apple Inc. *iPhone App Store Downloads Top 10 Million in First Weekend*. 2008. URL: http://%20www.apple.com/pr/library/%202008/07/14iPhone-App-Store-Downloads-Top%20-10-Million-in-First-Weekend.html (visited on 01/11/2017).

[App16] Apple Inc. *watchOS*. 2016. URL: www.apple.com/watchos/ (visited on 01/12/2017).

[Bus+15] Christoph Busold et al. "Smart and secure cross-device Apps for the Internet of advanced things". In: *LNCS* 8975 (2015), pp. 272–290. DOI: 10.1007/978-3-662-47854-7_17.

[Cha+16] J. Chauhan et al. "Characterization of early smartwatch apps". In: *2016 IEEE Int. Conf. on PerCom Workshops 2016* (2016). DOI: 10.1109/PERCOMW.2016.7457170.

[Chu+16] Stephanie Hui-Wen Chuah et al. "Wearable technologies: The role of usefulness and visibility in smartwatch adoption". In: *Computers in Human Behavior* 65 (2016), pp. 276–284. DOI: 10.1016/j.chb.2016.07.047.

[CM16] Riccardo Coppola and Maurizio Morisio. "Connected Car: Technologies, Issues, Future Trends". In: *ACM Comput. Surv.* 49.3 (2016), 46:1. DOI: 10.1145/2971482.

[CMK14] Jagmohan Chauhan, Anirban Mahanti, and Mohamed Ali Kaafar. "Towards the Era of Wearable Computing?" In: *Pro. 2014 CoNEXT on Student Workshop*. CoNEXT Student Workshop '14. ACM, 2014, pp. 24–25. DOI: 10.1145/2680821.2680833.

[Dag+16] Jan C. Dageförde et al. "Generating App Product Lines in a Model-Driven Cross-Platform Development Approach". In: *49th HICSS*. 2016, pp. 5803–5812. DOI: 10.1109/HICSS.2016.718.

[Dal+13] I. Dalmasso et al. "Survey, comparison and evaluation of cross platform mobile application development tools". In: *Proc. 9th IWCMC*. 2013. DOI: 10.1109/IWCMC.2013.6583580.

[DLT87] Richard L. Daft, Robert H. Lengel, and Linda Klebe Trevino. "Message Equivocality, Media Selection, and Manager Performance: Implications for Information Systems". In: *MIS quarterly* 11.3 (1987), p. 355. DOI: 10.2307/248682.

[El-+15] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* (2015).

[Goo16] Google Inc. *Android Wear*. 2016. URL: https://android.com/wear (visited on 01/12/2017).

[HEE13] Shah Rukh Humayoun, Stefan Ehrhart, and Achim Ebert. "Developing Mobile Apps Using Cross-Platform Frameworks: A Case Study". In: *Proc. 15th Int. Conf. HCI International, Part I*. Ed. by Masaaki Kurosu. Springer, 2013, pp. 371–380. DOI: 10.1007/978-3-642-39232-0_41.

[HHM13] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. "Evaluating Cross-Platform Development Approaches for Mobile Applications". In: *Revised Selected Papers WEBIST 2012*. Ed. by José Cordeiro and Karl-Heinz Krempels. Vol. 140. Lecture Notes in Business Information Processing (LNBIP). Springer, 2013, pp. 120–138.

[HMK13] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. "Cross-platform model-driven development of mobile applications with MD2". In: *Proc. 28th ACM SAC*. Ed. by Sung Y. Shin and José Carlos Maldonado. SAC '13. ACM, 2013, pp. 526–533. DOI: 10.1145/2480362.2480464.

[JJ14] Chris Jones and Xiaoping Jia. "The AXIOM model framework: Transforming requirements to native code for cross-platform mobile applications". In: *2nd Int. Conf. on Model-Driven Engineering and Software Dev.* Ed. by Luis Ferreira Pires. IEEE, 2014.

[JM15] Chakajkla Jesdabodi and Walid Maalej. "Understanding Usage States on Mobile Devices". In: *Proc. 2015 ACM Int. Joint Conf. on Pervasive and Ubiquitous Computing*. UbiComp '15. ACM, 2015, pp. 1221–1225. DOI: 10.1145/2750858.2805837.

[KK16] István Koren and Ralf Klamma. "The Direwolf Inside You: End User Development for Heterogeneous Web of Things Appliances". In: *Proc. 16th Int, Conf, ICWE 2016*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Springer, 2016, pp. 484–491. DOI: 10.1007/978-3-319-38791-8_35.

[LG 16] LG Electronics. *WebOS for LG Smart TVs*. 2016. URL: http://www.lg.com/uk/smarttv/webos (visited on 05/31/2016).

[RM16] Christoph Rieger and Tim A. Majchrzak. "Weighted Evaluation Framework for Cross-Platform App Development Approaches". In: *Proc. 9th SIGSAND/PLAIS EuroSymposium*. Ed. by Stanislaw Wrycza. Springer, 2016, pp. 18–39. DOI: 10.1007/978-3-319-46642-2_2.

[RM17] Christoph Rieger and Tim A. Majchrzak. "Conquering the Mobile Device Jungle: Towards a Taxonomy for App-enabled Devices". In: *13th International Conference on Web Information Systems and Technologies (WEBIST)*. Porto, Portugal, 2017, pp. 332–339. DOI: 10.5220/0006353003320339.

[SK13] A. Sommer and S. Krusche. "Evaluation of cross-platform frameworks for mobile applications". In: *Lecture Notes in Informatics* P-215 (2013), pp. 363–376.

[Sta16] Statista Inc. *Global smartphone shipments forecast from 2010 to 2020*. 2016. URL: https://www.statista.com/%20statistics/263441/ (visited on 01/11/2017).

[The16] The Linux Foundation. *Tizen*. 2016. URL: https://www.tizen.org (visited on 01/12/2017).

22

Business Apps with MAML

Table 22.1: Fact sheet for publication P16

Title	Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development
Authors	Christoph Rieger[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2017
Conference	32nd Annual ACM Symposium on Applied Computing
Copyright	ACM
Full Citation	Christoph Rieger. "Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development". In: *32nd Annual ACM Symposium on Applied Computing (SAC)*. Marrakech, Morocco: ACM, 2017, pp. 1599–1606. DOI: 10.1145/3019612.3019746 This version is not for redistribution. The final authenticated version is available online at https://doi.org/10.1145/3019612.3019746.

Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development

Christoph Rieger

Keywords: Graphical DSL, Mobile Application, Business App, Model-driven software development, Data model inference

Abstract: Business apps support the digitalization of business operations by utilizing the potential of ubiquitous mobile devices. Whereas many frameworks for *programming* cross-platform apps exist, few *modeling* approaches focus on platform-agnostic representations of mobile apps. In addition, development is mainly executed by software developers, while domain experts are rarely involved in the actual app creation. The MAML framework is proposed as model-driven approach also targeting non-technical users. Data, views, business logic, and user interactions are jointly modeled from a process perspective using a graphical domain-specific language. An inference mechanism is presented to merge partial data models into a global specification. Through model transformations, apps are then automatically generated for multiple platforms without writing code manually.

22.1 Introduction

The current trend toward developing apps for specific small-scale use cases originates from the advent of Apple's first iPhone in 2007. Almost a decade later, app development is still a task exclusively executed by programmers, often considering other stakeholders and future users primarily in requirements engineering phases upfront implementation. However, Gartner predicts that in the next two years, more than half of all internal business apps will be created using codeless tools [Rv14]. Modeling such apps can be achieved using two kinds of notations: On the one hand, a wide variety of general purpose process modeling notations such as Business Process Model and Notation (BPMN) exist [Obj11]. Usually, those are not detailed enough to cover mobile-specific aspects and, from a technical point of view, just represent connected, non-interpretable boxes with text. On the other hand, technical notations such as the Interaction Flow Modeling Language (IFML) are too complex to understand for domain experts and require software engineering knowledge [Obj15a; Żył15]. To adequately model mobile apps, a suitable notation needs to be *platform-independent*, ideally also considering emerging app-enabled devices such as wearables, smart home applications, and in-vehicle apps.

Concerning the implementation of mobile apps, existing commercial platforms provide cross-platform capabilities, but usually only regarding source code transformations or partly supported by graphical editors for designing individual views (e.g. [Esp16]). Model-driven software development sets out to take advantage of an already defined app model as input for a semi-/automatic creation of apps. Various textual domain-specific languages (DSL) [MHS05] follow this approach

[HV11; JJ15; HM13]. However, while DSLs are suited to cover a well-defined scope with sensible abstractions for inherent domain concepts, textual DSLs provide only minor benefits to non-technical users, i.e. they still feel like programming [ZS09]. Input from those stakeholders with strong domain knowledge is essential to ensure the developed software matches their tacit requirements [BKF14].

The work in this paper aims to alleviate the aforementioned problems by proposing the Münster App Modeling Language (MAML; pronounced 'mammal') framework. Its contributions are threefold: First, a *graphical DSL* to describe business apps is proposed that aims to be understandable not only for programmers but also for domain experts and process modelers. Second, to achieve this balancing act, a *data model inference mechanism* is presented to disburden the modeler from explicit data model specifications. Third, *model transformations* allow for a fully automatic generation of native apps from the specified graphical model.

The structure of the paper follows these contributions. After presenting related work in Section 22.2, the proposed framework is presented in Section 22.3. Section 22.4 discusses the approach and presents evaluation results from a usability study before giving an outlook in Section 22.5.

22.2 Related Work

The work presented in this paper draws on the scientific fields of domain-specific visual languages, model matching, and cross-platform mobile app generation.

Graphical DSLs and visual programming languages have been developed for several purposes, including process-related domains such as business process compliance [Knu+13] and data integration [Pen16]. Regarding mobile applications, only few approaches exist. For example, RAPPT represents a model-driven approach that mixes a graphical DSL for process flows with a textual DSL for programming [Bar+15], AppInventor encourages novices to create apps by combining building blocks of a visual programming language [Wol11], and Puzzle enables visual development of mobile applications from within a mobile environment [DP12]. Though all of them aim at simplifying the actual programming tasks for (potentially junior) developers, they disregard non-technical stakeholders.

In contrast, general purpose modeling notations exist to describe applications and process flows. The Unified Modeling Language (UML) is the most prominent example for software developers, providing a set of standards to define the structure and runtime behavior of an application [Obj15b]. In particular, the IFML (succeeding WebML) specifies user interactions within a software system [Obj15a]. To visualize process flows, a variety of modeling notations exist, e.g. BPMN, Event-driven Process Chains, or flowcharts [Obj11; van99; Int85]. However, such notations often remain too superficial for technical interpretation and mobile specifics are omitted.

Modeling approaches specific to the mobile domain rarely reach beyond user interface modeling. Some approaches explicitly try to incorporate non-technical users, e.g. through collaborative multi-

viewpoint modeling [FMM14]. Others use existing modeling notations such as statecharts [Fra+15] or extend them for mobile purposes, e.g. UML to model context in mobile distributed systems [SW07], IFML with mobile-specific elements [BMU14], or BPMN to orchestrate web services [BDF09]. Yet, technical modeling notations are often considered as complex to understand for domain experts [Fra+06]. Cognitive studies have been conducted, e.g. for WebML, and problems such as symbol overload are aggravated by the combination of multiple notations to represent the full range of data, business logic, user interface, and user interaction modeling [Gra+15].

Schema matching and model differentiation are further relevant fields of research with various approaches regarding the identification of common structures in models [Sut+16]. According to the classification by Rahm [RB01], a schema-only and constraint-based approach on element level is required for the inference of a data model from multiple input models. For the given problem of partial models, inference can be additive and name-based, although more sophisticated strategies such as ontology-based approaches may also identify modeling inconsistencies [LGJ07]. An application related to mobile devices, but focused on the visualization of data instead of its manipulation, is MobiMash for graphically creating mashup apps by configuring the representation and orchestration of data services [Cap+12].

In the context of meta modeling, reverse engineering approaches to track meta model evolution deal with similar problems of inferring object structures [Liu+12], and López-Fernández et al. [Lóp+15] presented a related idea in order to derive a common meta model from exemplary model frements.

Cross-platform mobile apps can be created with different approaches. El-Kassas [El-+15] identified three major categories of development approaches: *compiling* existing source code from a legacy application or different platform, *interpreting* a single code base through a runtime or virtual machine, and *model-driven* generation of app sources from a model representation. With regard to model-driven approaches, various academic and commercial frameworks exist [UB16]. Only few of them, such as Mobl [HV11] and AXIOM [JJ15], set out to cover the full spectrum of runtime behavior and structure of an app; often providing a custom textual DSL for this means. The work in this paper is based on MD²which focuses on the generation of business apps (i.e. form-based, data-driven apps interacting with back-end systems [MEK15]). The input model is specified using a platform-independent, textual DSL [HM13]. After preprocessing, code generators transform it into native platform source code [ME15; Ern+16]. Despite several improvements to support developers, textual DSLs often feel like programming to non-technical users [Żył15], motivating the need for further abstraction and visual modeling.

The commercial WebRatio Mobile Platform is closest to the work presented in this paper by generating apps from a graphically edited model [Ace+15], though it raises the entry barrier for potential users by relying on the combination of IFML and other notations. In addition, companies such as BiznessApps [Biz16] and Bubble [Bub16] promise codeless creation of apps using detailed configurators and web-based user interface editors. Our work in contrast focuses on a process-centric and platform-agnostic approach as described next.

22.3 MAML Framework

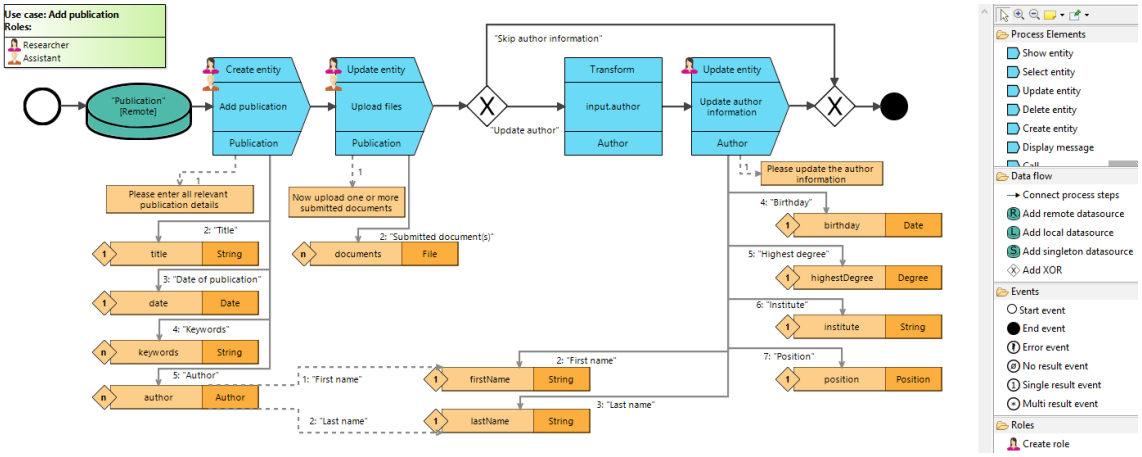


Figure 22.1: MAML Editor With Sample Use Case "Add publication"

The following subsections describe the main characteristics of the open-source MAML framework [Rie16b]. After highlighting the overall design goals, the proposed visual DSL as well as the data model inference mechanism, the implications on modeling support, and the app generation process are presented.

22.3.1 Language Design Principles

The MAML DSL is built around five main principles:

Domain expert focus: In contrast to technical specification languages, MAML is designed with a non-technical user in mind, e.g. a process modeler or expert knowledgeable about the domain but without software development experience.

Data-driven process: MAML models represent a process perspective on business apps, visualizing the sequence of processing steps performed on one or several data objects.

Modularization: The scope of a model is one *Use Case*, a unit of useful functionality containing a self-contained set of behaviors and interactions performed by the app user [Obj15b]. To support the domain expert focus, MAML combines data model, process steps and app visualization in a single model as opposed to software engineering patterns such as the Model-View-Controller (MVC) separation [Gam+95].

Declarative description: Use cases contain abstract, platform-agnostic elements describing *what* processing activities are possible on the data objects, without specifying their concrete representation on a mobile device. During generation, sensible defaults for platform specifics are provided.

Automatic cross-platform app generation: One major design goal of MAML is the capability to create fully-functional apps for multiple platforms in order to reach a large amount of

users. The graphical model is therefore designed to be interpretable by different code generators without further need for manual programming.

22.3.2 Language Overview

Figure 22.1 depicts a sample *use case* for adding an item to a publication management system. The model contains a sequence of activities, from a *start event* towards one or several *end events*. In the beginning, a *remote* or *local data source* specifies the data type of the manipulated objects. The modeler can then choose from a variety of (arrow-shaped) *interaction process elements*, for example to *select/create/update/display/delete entities*, show *popup messages*, or access device functionalities such as the *camera* and starting a phone *call*. Due to its declarative nature, the DSL does not indicate the concrete appearance but typically each logical step may be rendered as one view of the app. Furthermore, *automated process elements* represent invisible processing steps without user interaction, e.g. calling RESTful *web services*, *including* other models for process reuse, or navigating through the object structure (*transform*).

The navigation between connected process steps happens using an automatically created "Continue" button or an alternative denomination specified along the *process connectors*. To allow for conditional actions, the process flow can be branched out using an *XOR* element. The condition can either be triggered automatically by attaching an attribute to the XOR element and evaluating specified expressions, or requires a user decision by providing two button captions along the process connectors (such as in the example).

The rectangular elements (in the bottom half of Figure 22.1) represent the data displayed on the screen within the respective process step. *Attributes* consist of a cardinality indicator, a name and the respective data type. Apart from pre-defined data types such as *String, Integer, Float, PhoneNumber, Location* etc., custom types can be defined. Consequently, attributes may be nested over multiple levels in order to further describe the content of such a custom data type. *Labels* (depicted as rectangle without cardinality and type information) can be added to display explanatory text on screen, and *computed attributes* (not illustrated) may be used to output calculations on other attributes at runtime. To assign these UI elements to a process step, two types of connectors exist: Dotted arrows represent a *reading relationship* whereas solid arrows signify a *modifying relationship* regarding the target element. This refers not only to the manifest representation of attribute content displayed either as read-only text or editable input field. The interpretation also applies in a wider sense, e.g. concerning web service calls in which the server "reads" an input parameter and "modifies" information through its response.

Every connector which is connected to an interaction process element also specifies an order of appearance. Additionally, a human-readable representation of the field description is derived from the attribute name unless specified manually. To reduce the amount of elements to be modeled, multiple connectors may point to the same UI element from different sources (given their data types match). Alternatively, to avoid wide-spread connections across larger models, UI elements

may instead be duplicated to different positions in the model and will automatically be recognized as being the same element (see Subsection 22.3.3).

Finally, the MAML DSL supports a multi-role concept. The modeler can specify role names (displayed in the upper-left corner of Figure 22.1) and annotate them to the respective interaction process elements. This is particularly useful to describe scenarios in which parts of the process are performed by different people, e.g. approval workflows. If the assigned role changes, the process automatically terminates for the first app user, modified data objects are saved, and the subsequent user is informed about an open workflow instance in his app.

22.3.3 Data-Model Inference

One key principle of the MAML approach is the absence of a separately modeled, global data schema. Instead, each process step within a model refers only to the attributes needed (i.e. displayed or edited) within this particular step. Nevertheless, a global data model is required for code generation and is inferred from partial models on multiple levels: for each process element individually, for the whole use case, and across multiple use cases of the overall app. Semantic reasoning such as inferring generalization relationships, is however not in the scope of this work.

Partial Data-Model Inference

To infer separate data models for each process element, a graph data structure is set up in which a node represents a data type and an edge defines an attribute relationship between two types. It should be noted that the implemented algorithm [Rie16a] is more complex than sketched here. For clarification, *attribute* is further used to refer to the UI element in a MAML model whereas *property* signifies the corresponding member of the UML class diagram.

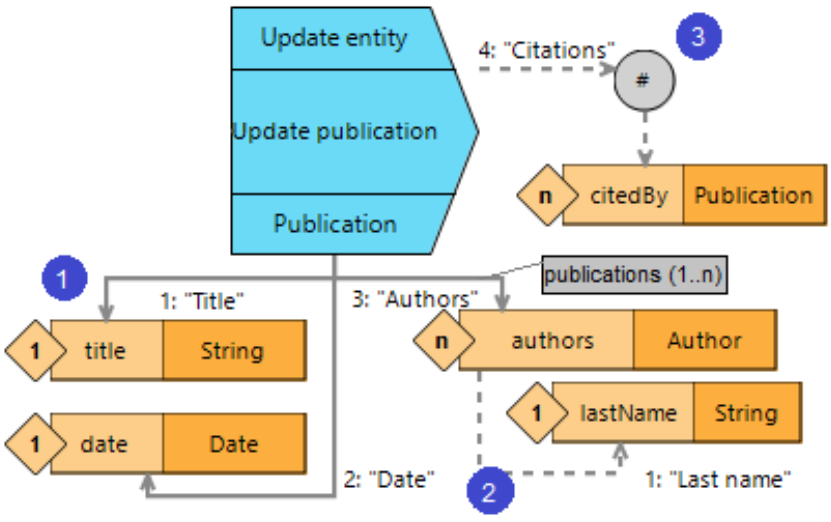


Figure 22.2: Exemplary Partial MAML Model

Initially, the data types contained in each process element or attribute are collected. For the example depicted in Figure 22.2, this creates the set of data types {*Publication, String, Date, Author*}. For each non-primitive data type in this set, a class is created in the partial data model. Subsequently, relationships between those data types are identified. Three origins need to be considered (cf. numbered circles in Figure 22.2): First, a relationship may exist between a process element and an attached attribute. Second, a nested attribute adds a relationship to the nesting attribute's data type. Third, an attribute may be transitively connected to the process element through one or more computed attributes, but still refers to the process element's data type. Relationships are transformed to properties and associations by distinguishing the following four cases.

Figure 22.3: Class Diagram of Inferred Primitive Types

Primitive: An attribute with primitive data type is converted to a single- or multi-valued property of the source data type. For the above example, *title*, *date*, and *lastName* are added as properties to the respective classes (see Figure 22.3).

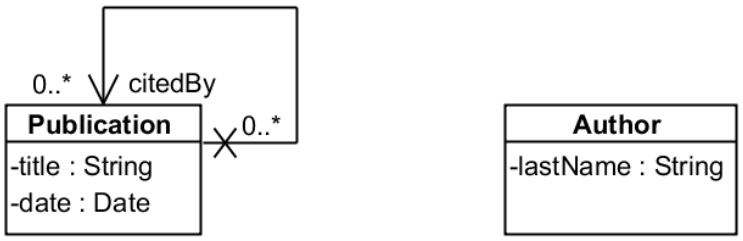


Figure 22.4: Class Diagram of Inferred Unidirectional Relationships

Unidirectional: Regular attribute connections can be represented as unidirectional relationships. Because multiple source elements are possible, the unknown opposite cardinality must be interpreted as o..*. For example, the *citedBy* property in Figure 22.4 holds a collection of *Publication*s but has no restrictions itself on being referenced by multiple *Publication*s.

Bidirectional: In contrast to the previous case, explicit bidirectional relationships in the model are fully specified and transformed to named properties in both classes. The *authors* attribute in Figure 22.2 specifies a multi-valued property and is converted to the association shown in Figure 22.5. In addition to access methods for consistent updates and deletions, potential cardinality restrictions need to be enforced on an implementation level (e.g. restrict the removal of the last *publication* of an *author* in the example to comply with the minimum cardinality).

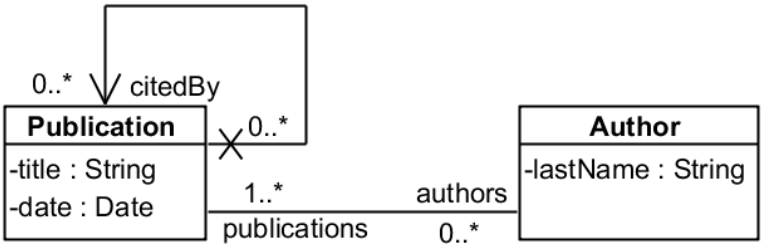


Figure 22.5: Class Diagram of Inferred Bidirectional Relationships

Singleton: An attribute of a singleton data type (not depicted in the example) is a variant of the unidirectional scenario in which the unknown opposite cardinality can be restricted to 0..1 (either the property is set in the single instance of the referencing class or not) [Gam+95].

Merging Partial Data Models

To consolidate the set of partial data models, they need to be merged within and across use cases (the order of iteratively merged partial models being irrelevant).

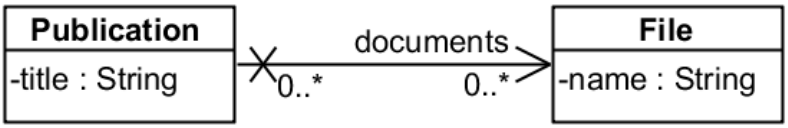


Figure 22.6: Class Diagram of a Second MAML Model

In a first step, the union of two sets of data types is created. As an example, when merging the partial data model of Figure 22.5 with another partial model depicted in Figure 22.6, the global data model contains the set of types {*Publication, String, Date, Author, File*}.

Next, each edge in both partial models is added to the global data model if it introduces a new name or cardinality to the respective combination of source and target data type.

With this name-based matching strategy, specifying multiple associations between a pair of data types is unproblematic as it will result in two distinct edges of the type graph (the risk of unintended modeling errors is limited because of advanced modeling support provided by the IDE; cf. Subsection 22.3.4).

For the example partial models, this results in the complete data model depicted in Figure 22.7. However, two types of modeling errors need to be checked: First, a *type error* exists if any source data type has two properties of the same name pointing to different target data types. This cannot automatically be resolved and is marked as an error to the modeler. Second, a *cardinality conflict* exists if two properties with the same name differ with regard to their cardinality for any pair of data types. In this case, the modeler should be warned, but automatic resolution is possible.

Regarding directionality, if any of the affected properties is modeled as bidirectional relationship, the merged association is also regarded as bidirectional in order not to lose properties. In addition,

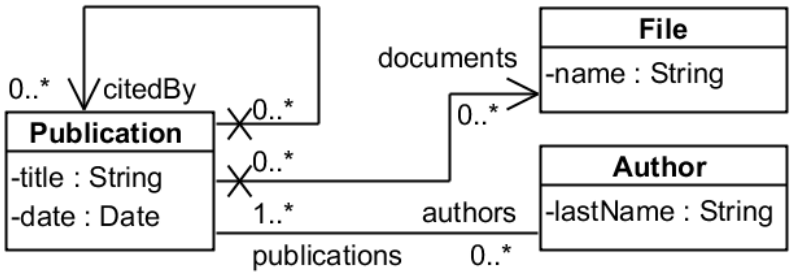


Figure 22.7: Inferred Global Data Model

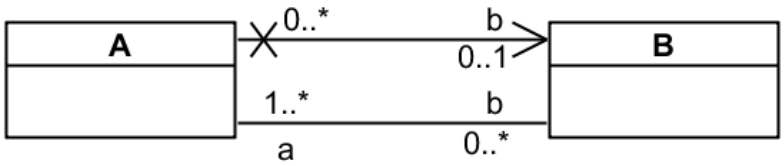


Figure 22.8: Exemplary Model With Cardinality Error

the resulting cardinality for each side of the association is calculated as the union between the conflicting cardinalities (ignoring not navigable ends) to avoid invalidating existing data. In the example class diagram depicted in Figure 22.8, the cardinality $0..1 \cup 0..* = 0..*$ is assigned to b and $1..*$ is assigned to a (as $0..*$ is not navigable), where $i..j = \{i, i+1, ..., j\}$ and $*$ represents infinity.

To sum up, this algorithm can be applied to partial models of different granularity in order to validate the models and derive a global data schema.

22.3.4 Modeling Support

The developed editor for MAML is based on Eclipse Sirius for visual editor creation, and the underlying Eclipse Modeling Framework [The16b]. Using the aforementioned model inference approach, tool support of the graphical DSL can be improved similar to general purpose programming languages and language workbenches such as Xtext which provide functionalities for scoping and validation of user input [Bet13]. Instant feedback can be provided to the user, for example by displaying the inferred data type within process elements as mental support. Also, expressions in the *Transform* element (to navigate through the data structure) are evaluated at runtime. In addition, the inference result are used to propose matching data types when adding attributes or to propose attributes for already known data types. Thus, model consistency is strengthened while at the same time reducing the overall modeling effort.

With regard to validation, the errors identified by the merge algorithm are traced back to the respective model elements, thus allowing for visual feedback directly within the model. By differentiating between errors, warnings and hints, more elaborate validation techniques such as name similarities or modeling conventions can seamlessly be integrated in the modeling editor. As a result, automatic data model inference enables feedback and support mechanisms known

from formal languages in a visual environment, progressing beyond the stage of just modeling "shapes filled with text".

22.3.5 App Generation

One of the main features of the MAML framework is the automatic generation of cross-platform apps in order to eliminate the need for manual programming. MAML relies on a model-driven software development approach, therefore the visual models are used as sole input for the app generation.

The current approach is based on previous work in the domain of business apps, notably the textual DSL MD²(see Section 22.2). Although both languages share common overall concepts, MD²is programming-oriented and follows a MVC-layered approach instead of the presented separation by tasks. For further details on MD²language features the reader is referred to [HM13; ME15].

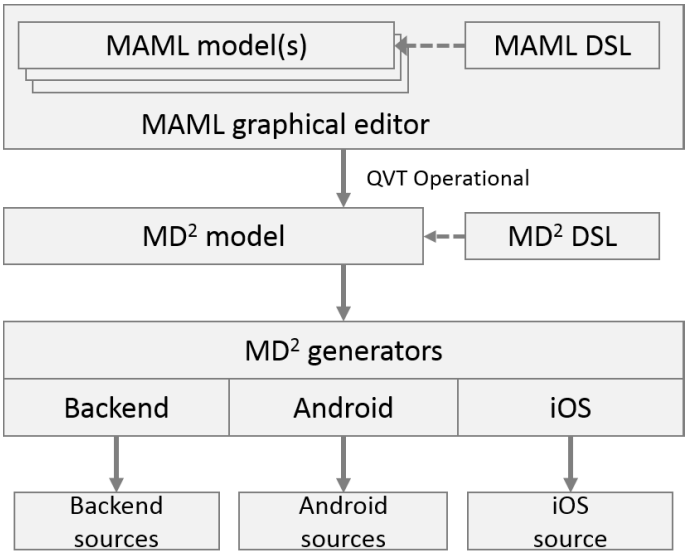


Figure 22.9: MAML Generation Process

A two-step generation process was established (cf. Figure 22.9): First, a model-to-model transformation translates all related MAML model instances to one MD²model instance. This transformation is based on mappings between the meta models of MAML and MD², specified using the QVT Operational framework [The16a]. Second, existing MD²generators perform the model-to-code transformation and output the app source code for the target platforms, in particular Android and iOS. Figure 22.10 presents screenshots of the resulting Android app for the first process steps shown in Figure 22.1.

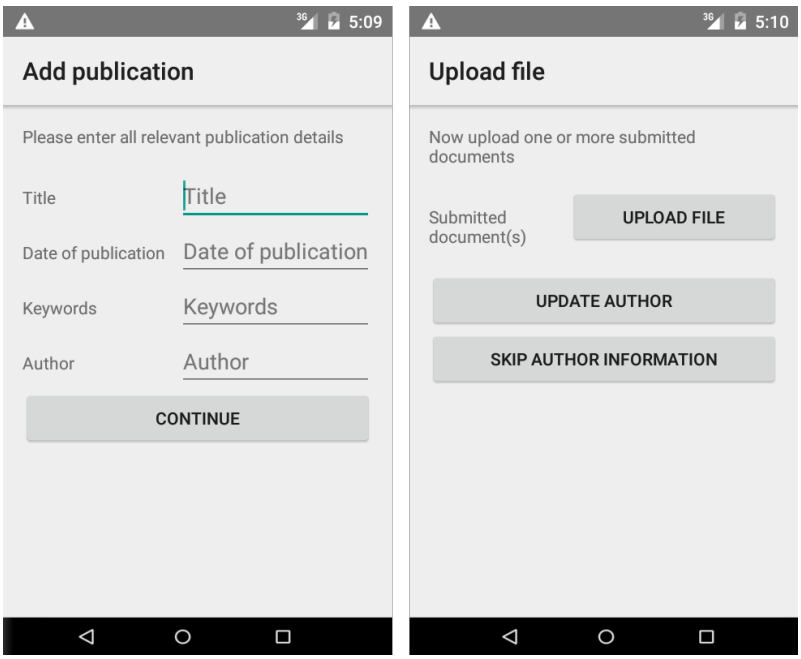


Figure 22.10: Generated Android App Screenshots

22.4 Evaluation and Discussion

The MAML framework should provide a user-centric approach to business app creation for non-technical users. However, it also aims for automatic generation of cross-platform apps from the graphical model, thus requiring a minimum of technical specificity to be interpretable by generators. When setting out for designing a language, a general trade-off needs to be found between detailedness and simplicity. There are other process modeling languages that may be easier to read but rely on substantial assumptions on the meaning of their elements. Influenced by different process modeling languages, the MAML DSL is positioned between BPMN 2.0 and IFML from a complexity point of view, enriched with elements specific to mobile apps.

It is not intended to create yet another workflow management system, instead processes are seen as reasonable level of abstraction for a domain expert in order to create an app without programming. In addition, MAML models have the advantage over e.g. IFML in that they are self-contained and display all pieces of information visually. MAML models can therefore also be seen as a means of communication, facilitating the discussion between involved stakeholders concerning interactions and data. In contrast, IFML models are connected to other UML standards and require multiple models for data, business logic, and user interaction to be interpreted together.

To compare both notations, a qualitative, observational study was performed with 26 student participants in individual sessions. One part of the study evaluated the readability and understandability of the respective notation. Therefore, one MAML and one IFML model, both depicting similar content (cf. online material [Rie16b]), were shown to the participants (in random

order to avoid bias) without prior introduction of the model elements. None of the participants had previous knowledge of either modeling language. Additionally, a System Usability Score (SUS) questionnaire was answered for each notation, comprising ten questions on a 5-point scale between strong disagreement and strong agreement. The result is obtained by converting and scaling the responses according to [Bro96].

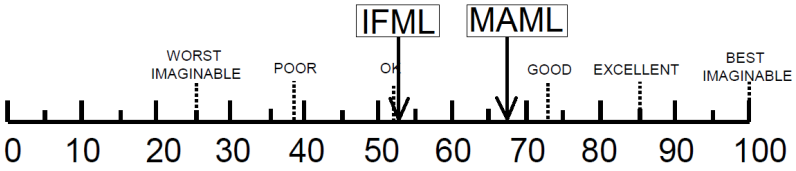


Figure 22.11: SUS Ratings for IFML and MAML

Overall, MAML surpasses IFML with 66.8 to 52.8 points (of maximum 100) regarding its initial readability without prior knowledge. It should be noted, however, that this does not represent a percentage but can instead be interpreted according to the adjective rating scale depicted in Figure 22.11 as proposed by Bangor et al. [BKM09].

For better comparability, the participants were clustered in the three groups of technical users (11), process modelers (9), and domain experts (6), according to their personal background in programming and process modeling. All groups rated MAML better (cf. Table 22.2), though to different degrees. The low spread in the group of process modelers may originate from a general familiarity with modeling notations.

Table 22.2: SUS Scores by Participant Group

Group	IFML score (σ)	MAML score (σ)
Technical users	45.91 (23.6)	64.09 (17.3)
Process modelers	64.17 (19.0)	69.44 (12.0)
Domain experts	48.33 (24.5)	67.92 (18.7)
Overall	52.79 (23.0)	66.83 (15.6)

Aspects regarding the ease of understanding and complexity of the notation scored better for MAML, particularly for the group of domain experts without knowledge of programming or process models. This reflects the preliminary results of the observations in which seven participants emphasized MAML's simplicity and reduction of "technical clutter". Interestingly, even software developers generally preferred MAML's process-oriented approach and only two participants critically noted the absence of explicit data models.

Using data model inference in MAML alleviates three problems: First, the modeler is disburdened of specifying required data upfront or maintaining a consistent data model separately, thus reducing effort and leading to a clean model of only the specific data properties required for each step. The suitability of such partial data models might be questioned in large-scale

enterprise models, but with regard to app development fully specified data models are often not pre-existing and need to be newly created. Under these circumstances, on-the-fly inference of the underlying data model is particularly helpful for process modelers and domain experts without programming experience. The study confirms this, as the inference mechanism and enhanced modeling support was welcomed as helpful guidance during the sessions, and scores favor MAML regarding consistency and explanatory power.

Second, improved modeling support is possible as described in Subsection 22.3.4 similar to code completion features known from statically typed programming languages. Further sophisticated inference techniques relying on ontology-based matching [Noy04] may be extensions to further improve model consistency. Third, the partial data models on a low granularity (i.e. per process flow element) allow for improved security. As one app is generated per annotated role, generators may tailor app-specific models by splitting the global data model accordingly. This not only reduces the amount of transferred data but allows for features such as automatic permission control.

Regarding app generation, all steps can be executed without additional configuration. The intermediate step is however no inherent limitation of the framework. Future generators targeting new platforms may just as well directly generate code from the MAML model instances. Apart from reusing existing MD²generators developed in the last years, the intermediate transformation adds a technical representation with detailed possibilities of configuration such as UI element styling [MEK15]. Technical users accustomed to MD²may therefore adapt the created model to their needs.

The usability study also showed potentials for further refinement: Although modeling activities were mostly performed correctly, the lack of a screen-oriented representation similar to IFML's nested elements was mentioned by four participants. Also, wording of elements proved to be not clear enough. For instance, data types are hard to understand for domain experts without further explanation and may be replaced by symbols for clarification.

22.5 Conclusion

In this paper, the MAML framework was proposed to model mobile apps using a declarative graphical DSL. In contrast to editors in which the low-level user interface is specified by positioning elements on a screen canvas, MAML focuses on a process-centric definition of apps using platform-agnostic process elements, and hence aligns with the business perspective of managing processes and data flows. The approach is based on existing work on cross-platform business app generation and uses model-driven techniques to transform the models first into an intermediate textual representation before generating platform-specific source code. In particular, a data model inference mechanism was presented that enables real-time validation and consistency checks on partial data models, and bypasses the need for explicitly modeling a global data schema. An empirical evaluation supports the advantage of MAML over the related technical IFML notation,

specifically with regard to its understandability by domain experts. MAML therefore achieves the desired balance of abstracting programming-heavy tasks to understandable process flows while keeping the technical expressiveness required for automatic app code generation.

The study results support the benefit of MAML and the participants can be seen as realistic sample for app-experienced adults in the general workforce. Still, the amount of student participants may pose a threat to validity and more extensive studies are necessary to confirm these results. Concerning a more general aspect that constitutes future work, applying the prototype to real-world problems might reveal further need for improvements. For example, data flow variations, currently limited to XOR elements, are deemed sufficient due to the sequential display on smartphone screens but advanced mechanisms such as loops and parallelism may be requested by practitioners. Furthermore, the platform-agnostic principle of MAML allows for its application to mobile devices beyond smartphones and tablets. Regarding the emergence of novel devices such as smart watches, interesting questions arise regarding best practices for implementing apps on such devices with different capabilities and user interaction patterns. Finally, programmers may want to deviate from the default configuration used for automatic app generation, causing manual changes to the intermediate representation. Research needs to be done on how to avoid interference of generated and custom DSL content given frequent re-generation.

References

[Ace+15] Roberto Acerbis et al. “Model-driven Development of Cross-platform Mobile Applications with WebRatio and IFML”. In: *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. MOBILESoft ’15. IEEE Press, 2015, pp. 170–171. URL: http://dl.acm.org/citation.cfm?id=2825041.2825090.

[Bar+15] Scott Barnett et al. “A Multi-view Framework for Generating Mobile Apps”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 305–306. DOI: 10.1109/VLHCC.2015.7357239.

[BDF09] M. Brambilla, M. Dosmi, and P. Fraternali. “Model-driven engineering of service orchestrations”. In: *SERVICES 2009 - 5th 2009 World Congress on Services* (2009). DOI: 10.1109/SERVICES-I.2009.94. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-72849106274&partnerID=40&md5=c059202c0abba845715d58a4ce570eb2.

[Bet13] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Community experience distilled. Birmingham, UK: Packt Pub, 2013.

[Biz16] Bizness Apps. *Mobile App Maker | Bizness Apps*. http://biznessapps.com/. 2016. URL: http://biznessapps.com/ (visited on 09/23/2016).

[BKF14] R. Breu, A. Kuntzmann-Combelles, and M. Felderer. “New Perspectives on Software Quality”. In: *IEEE Software* 31.1 (2014), pp. 32–38. DOI: 10.1109/MS.2014.9.

[BKM09] Aaron Bangor, Philip Kortum, and James Miller. "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale". In: *J. Usability Studies* 4.3 (2009), pp. 114–123. URL: http://dl.acm.org/citation.cfm?id=2835587.2835589.

[BMU14] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. "Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End". In: *Lecture Notes in Computer Science* 8640 (2014), pp. 176–191. DOI: 10.1007/978-3-319-10359-4_15.

[Bro96] John Brooke. "SUS-A quick and dirty usability scale". In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7.

[Bub16] Bubble Group. *Bubble - Visual Programming*. https://bubble.is/. 2016. URL: https://bubble.is/ (visited on 09/23/2016).

[Cap+12] Cinzia Cappiello et al. "MobiMash: End User Development for Mobile Mashups". In: *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web Companion* (2012), pp. 473–474. DOI: 10.1145/2187980.2188083.

[DP12] Jose Danado and Fabio Paternò. "Puzzle: A Visual-Based Environment for End User Development in Touch-Based Mobile Phones". In: *Human-Centered Software Engineering: 4th International Conference, HCSE 2012*. Ed. by Marco Winckler, Peter Forbrig, and Regina Bernhaupt. Springer Berlin Heidelberg, 2012, pp. 199–216. DOI: 10.1007/978-3-642-34347-6_12. URL: http://dx.doi.org/10.1007/978-3-642-34347-6_12.

[El-+15] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* (2015). DOI: 10.1016/j.asej.2015.08.004. URL: http://www.sciencedirect.com/science/article/pii/S2090447915001276.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *Proceedings of the 12th International Conference on Web Information Systems and Technologies (WEBIST 2016)* (2016), pp. 307–316. DOI: 10.5220/0005862103070316.

[Esp16] David Esperalta. *DecSoft - App Builder*. https://www.davidesperalta.com/appbuilder. 2016. URL: https://www.davidesperalta.com/appbuilder (visited on 09/08/2016).

[FMM14] Mirco Franzago, Henry Muccini, and Ivano Malavolta. "Towards a Collaborative Framework for the Design and Development of Data-intensive Mobile Applications". In: *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. MOBILESoft 2014. ACM, 2014, pp. 58–61. DOI: 10.1145/2593902.2593917. URL: http://doi.acm.org/10.1145/2593902.2593917.

[Fra+06] R. B. France et al. "Model-Driven Development Using UML 2.0: Promises and Pitfalls". In: *Computer* 39.2 (2006), pp. 59–66. DOI: 10.1109/MC.2006.65.

[Fra+15] Rita Francese et al. "Model-Driven Development for Multi-platform Mobile Applications". In: *Product-Focused Software Process Improvement: 16th International Conference, PROFES 2015*. Ed. by Pekka Abrahamsson et al. Springer Intl. Publishing, 2015, pp. 61–67. DOI: 10.1007/978-3-319-26844-6_5. URL: http://dx.doi.org/10.1007/978-3-319-26844-6_5.

[Gam+95] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995.

[Gra+15] David Granada et al. "Analysing the cognitive effectiveness of the WebML visual notation". In: *Software & Systems Modeling* (2015). DOI: 10.1007/s10270-014-0447-8.

[HM13] H. Heitkötter and T. A. Majchrzak. "Cross-Platform Development of Business Apps with MD^2". In: *Proc. of the 8th Int. Conf. on Design Science at the Intersection of Physical and Virtual Design (DESRIST)*. Vol. 7939. LNBIP. Springer, 2013, pp. 405–411.

[HV11] Zef Hemel and Eelco Visser. "Declaratively Programming the Mobile Web with Mobl". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. ACM, 2011, pp. 695–712. DOI: 10.1145/2048066.2048121. URL: http://doi.acm.org/10.1145/2048066.2048121.

[Int85] International Organization for Standardization. *ISO 5807:1985*. 1985.

[JJ15] Christopher Jones and Xiaoping Jia. "Using a Domain Specific Language for Lightweight Model-Driven Development". In: *ENASE 2014, CCIS 551*. 2015, pp. 46–62. DOI: 10.1007/978-3-642-23391-3. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-84879468495&partnerID=tZOtx3y1.

[Knu+13] David Knuplesch et al. "Visual Modeling of Business Process Compliance Rules with the Support of Multiple Perspectives". In: *Conceptual Modeling: 32th International Conference, ER 2013*. Ed. by Wilfred Ng, Veda C. Storey, and Juan C. Trujillo. Springer Berlin Heidelberg, 2013, pp. 106–120. DOI: 10.1007/978-3-642-41924-9_10. URL: http://dx.doi.org/10.1007/978-3-642-41924-9_10.

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. "DSMDiff: a differentiation tool for domain-specific models". In: *European Journal of Information Systems* 16.4 (2007), pp. 349–361. DOI: 10.1057/palgrave.ejis.3000685. URL: http://dx.doi.org/10.1057/palgrave.ejis.3000685.

[Liu+12] Qichao Liu et al. "Application of Metamodel Inference with Large-Scale Metamodels". In: *International Journal of Software and Informatics* 6.2 (2012), pp. 201–231.

[Lóp+15] Jesús J. López-Fernández et al. "Example-driven meta-model development". In: *Software & Systems Modeling* 14.4 (2015), pp. 1323–1347. DOI: 10.1007/s10270-013-0392-y. URL: http://dx.doi.org/10.1007/s10270-013-0392-y.

[ME15] T. A. Majchrzak and J. Ernsting. "Reengineering an Approach to Model-Driven Development of Business Apps". In: *8th SIGSAND/PLAIS EuroSymposium 2015*. 2015, pp. 15–31. DOI: 10.1007/978-3-319-24366-5_2. URL: http://dx.doi.org/10.1007/978-3-319-24366-5_2.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892. URL: http://doi.acm.org/10.1145/1118890.1118892.

[Noy04] N. F. Noy. "Semantic integration: A survey of ontology-based approaches". In: *SIGMOD Record* 33.4 (2004), pp. 65–70. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-11244252825&partnerID=40&md5=316e1e4275d67daeb2ba5d25de1a632b.

[Obj11] Object Management Group. *Business Process Model and Notation, Version 2.0*. 2011.

[Obj15a] Object Management Group. *Interaction Flow Modeling Language, Version 1.0*. 2015.

[Obj15b] Object Management Group. *Unified Modeling Language, Version 2.5*. 2015. URL: http://www.omg.org/spec/UML/2.5.

[Pen16] Pentaho Corp. *Data Integration - Kettle*. http://pentaho.com/product/data-integration. 2016. (Visited on 09/12/2016).

[RB01] E. Rahm and P. A. Bernstein. "A survey of approaches to automatic schema matching". In: *VLDB Journal* 10.4 (2001), pp. 334–350. DOI: 10.1007/s007780100057. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0035657983&partnerID=40&md5=9f1e1cfaf8b16ab43d34ecca0ff916fe.

[Rie16a] Christoph Rieger. "A Data Model Inference Algorithm for Schemaless Process Modelling". In: *Working Papers, European Research Center forInformation Systems No. 28*. Ed. by Jörg Becker et al. Münster, 2016.

[Rie16b] Christoph Rieger. *MAML Code Respository*. https://github.com/wwu-pi/maml. 2016.

[Rie17] Christoph Rieger. "Business Apps with MAML: A Model-Driven Approach to Process-Oriented Mobile App Development". In: *32nd Annual ACM Symposium on Applied Computing (SAC)*. Marrakech, Morocco: ACM, 2017, pp. 1599–1606. DOI: 10.1145/3019612.3019746.

[Rv14] Janessa Rivera and Rob van der Meulen. *Gartner Says By 2018, More Than 50 Percent of Users Will Use a Tablet or Smartphone First for All Online Activities*. http://www.gartner.com/newsroom/id/2939217. 2014. URL: http://www.gartner.com/newsroom/id/2939217 (visited on 09/23/2016).

[Sut+16] E. Sutanta et al. "Survey: Models and prototypes of schema matching". In: *International Journal of Electrical and Computer Engineering* 6.3 (2016), pp. 1011–1022. DOI: 10.11591/ijece.v6i3.9789. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84979258310&partnerID=40&md5=4cc65516ed76e5768e8c9e6b8021c528.

[SW07] C. Simons and G. Wirtz. "Modeling context in mobile distributed systems with the UML". In: *Journal of Visual Languages and Computing* 18.4 (2007), pp. 420–439. DOI: 10.1016/j.jvlc.2007.07.001. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-34548601270&partnerID=40&md5=33c025cdb39ab41de9b834c2620def71.

[The16a] The Eclipse Foundation. *Model-to-Model Transformation*. https://projects.eclipse.org/projects/ modeling.mmt. 2016. URL: https://projects.eclipse.org/%20projects/modeling.mmt (visited on 09/08/2016).

[The16b] The Eclipse Foundation. *Sirius*. https://eclipse.org/sirius/. 2016. URL: https://eclipse.org/sirius/ (visited on 09/08/2016).

[UB16] Eric Umuhoza and Marco Brambilla. "Model Driven Development Approaches for Mobile Applications: A Survey". In: *Mobile Web and Intelligent Information Systems: 13th International Conference, MobiWIS 2016*. Ed. by Muhammad Younas et al. Springer Intl. Publishing, 2016, pp. 93–107. DOI: 10.1007/978-3-319-44215-0_8. URL: http://dx.doi.org/10.1007/978-3-319-44215-0_8.

[van99] W.M.P. van der Aalst. "Formalization and verification of event-driven process chains". In: *Information and Software Technology* 41.10 (1999), pp. 639–650. DOI: 10.1016/S0950-5849(99)00016-6. URL: http://www.sciencedirect.com/science/article/pii/S0950584999000166.

[Wol11] D. Wolber. "App inventor and real-world motivation". In: *SIGCSE'11 - Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (2011). DOI: 10.1145/1953163.1953329. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-79954439194&partnerID=40&md5=b213de4b49567dedec7b552911af1a3a.

[ZS09] Uwe Zdun and M. Strembeck. "Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development". In: *Proceedings of 14th European Conference on Pattern Languages of Programs (EuroPLoP 2009)*. 2009, pp. 1–37. URL: http://eprints.cs.univie.ac.at/2327/.

[Żył15] Kamil Żyła. “Perspectives of Simplified Graphical Domain-Specific Languages as Communication Tools in Developing Mobile Systems for Reporting Life-Threatening Situations”. In: *Studies in Logic, Grammar and Rhetoric* 43.1 (2015). DOI: 10.1515/slgr-2015-0048.

23

Weighted Evaluation Framework for Cross-Platform App Development Approaches

Table 23.1: Fact sheet for publication P17

Title	Weighted Evaluation Framework for Cross-Platform App Development Approaches
Authors	Christoph Rieger[1] Tim A. Majchrzak[2] [1] *ERCIS, University of Münster, Münster, Germany* [2] *ERCIS, University of Agder, Kristiansand, Norway*
Publication Date	2016
Conference	Information Systems: Development, Research, Applications, Education: 9th SIGSAND/PLAIS EuroSymposium
Copyright	Springer International Publishing
Full Citation	Christoph Rieger and Tim A. Majchrzak. "Weighted Evaluation Framework for Cross-Platform App Development Approaches". In: *Information Systems: Development, Research, Applications, Education: 9th SIGSAND/PLAIS EuroSymposium 2016, Gdansk, Poland, September 29, 2016, Proceedings.* Ed. by Stanislaw Wrycza. Cham: Springer International Publishing, 2016, pp. 18–39. DOI: 10.1007/978-3-319-46642-2_2 This version is not for redistribution. The final authenticated version is available online at https://doi.org/10.1007/978-3-319-46642-2_2.

Weighted Evaluation Framework for Cross-Platform App Development Approaches

Christoph Rieger Tim A. Majchrzak

Keywords: App, Mobile Computing, Mobile Application, Cross-Platform, Multi-Platform, Evaluation

Abstract: Cross-platform app development is very challenging, although only two platforms with significant market share (iOS and Android) remain. While device fragmentation – multiple, only partly compatible versions of a platform – has been complicating matters already, the need to target different device classes is a new emergence. Smartphones and tablets are relatively similar but app-enabled devices such as TVs and even cars typically have differing capabilities. To facilitate usage of cross-platform app development approaches, we present work on an evaluation framework. Our framework provides a set of up-to-date evaluation criteria. Unlike prior work on this topic, it offers weighted assessment to cater for varieties in targeted device classes. Besides motivating and explaining the evaluation criteria, we present an exemplary application for one development approach and, as benchmarks, for native apps and Webapps. Our findings suggest that the proliferation of app-enabled devices amplifies the need for improved development support.

23.1 Introduction

Only two platforms for smartphone and tablet devices with significant market share remain [Wv16]. Even developing applications *only* for Apple's iOS and Google's Android is challenging (cf. e.g. [ME15]). Essentially, apps need to be realized separately for both, doubling the effort and prolonging the time-to-market [HMK13]. Moreover, device fragmentation – the parallel usage of several versions and possibly vendor-specific additions – complicates development, particularly for Android [Dob12]. Cross-platform development frameworks promise to relieve developers from the hardships of considering idiosyncrasies of several platforms and versions by providing uniform development interfaces [HHM13].

As an additional challenge for developers, an increasing number of devices is *app-enabled*. Arguably, most apps in today's sense target the smartphone but soon they will be routinely used on a much wider variety of platforms. Modern entertainment technology such as TVs, BluRay players and game consoles are capable of *running* apps. Cars are seen as a major target of tomorrow's apps [Wol13]. Wearable devices such as smartwatches and augmented reality glasses introduce novel usage scenarios [Liu+16]. Although it can be rightfully doubted that a fridge will be the main unit to install new apps on, it is likely that with the advent of the Internet of Things (IoT) many more devices will run apps. In consequence, catering for *all* device-specific particularities will become much harder.

Extending the already well-understood general requirements for cross-platform app development, we suggest taking into account the multitude of potential devices. Heitkötter, Hanschke

and Majchrzak have proposed an evaluation framework for cross-platform approaches in 2012 [HHM12]. The extension [HHM13] is still very useful and routinely cited in current papers on cross-platform development. However, even though the set of criteria they proposed is based on a thorough foundation, the rapid proliferation of the field mandates an overhaul. In addition, the former focus on smartphone and tablet devices ignores the plethora of novel devices reaching the consumer market.

We build on the existing work on cross-platform technology. In particularly, we use the existing set of criteria [HHM13] as the foundation to provide an extended, revised catalogue of them. This allows matching the criteria with the status quo of mobile computing. An approach that is best suited for smartphone apps might fail if apps also target entertainment systems. However, selecting a "catch-all" solution might be inferior to a specific one if only handheld devices should be supported. Therefore, we not only extend criteria but also embed them in a framework that includes a *weighting scheme*. The assessment of platforms can thereby be employed to make a per-scenario choice.

This paper makes several contributions. Firstly, it provides an evaluation framework for cross-platform development frameworks in the domain of mobile consumer devices. Evaluation criteria are explained in detail and the rationale for employing them is highlighted. Secondly, we provide the means to use our framework in an individual – particularly in a device-class specific – way by proposing the integration of balanced weights. Thirdly, we demonstrate the feasibility of our work with an exemplary evaluation. Accordingly, in Section 23.2 we discuss related work and in Section 23.3 we give the necessary background. Based on this foundation, Section 23.4 provides our catalogue of evaluation criteria. The exemplary use of the criteria for evaluation follows in Section 23.5. The findings are then discussed in Section 23.6, which leads to a conclusion in Section 23.7.

23.2 Related Work

Since cross-platform development of apps has been a topic for a few years now, there is plenty of scientific work on the topic in general. However, most papers tackle single frameworks or have an experimental nature. Consequently, there are relatively few papers that provide an overview, and even less than offer an evaluation. A comprehensive summary of related literature regarding covered tools, criteria and focal areas of comparison is given in Table 23.2. In the following, we only comment on particularly notable details.

The papers by Heitkötter et al. [HHM12; HHM13] have been used as basis for further research on apps. Examples include the definition of quality criteria for HTML5 frameworks [Soh+15], quantitative performance evaluations [WVN15], and the creation of the cross-platform development frameworks ICPMD [El-+14] and MD²/ [HKM15].

Early papers have typically taken into account few criteria only (if at all [CS12; SKS14]) – e.g. only seven [PSC12], and only from a developer's perspective [CGG14]. Few works take a

Table 23.2: Literature on Cross-Platform App Development Tool Evaluations

Paper	Year	Evaluated tools	Main categories (number of criteria)	Focal areas
[PSC12]	2012	RhoMobile, PhoneGap, DragonRad, MoSync	platform compatibility (2), development features (4), general features (4), device APIs (17)	qualitative tool comparison
[Dal+13]	2013	PhoneGap, jQuery Mobile, Sencha Touch, Titanium	Platform support, rich user interface, back-end communication, security, app extensions, power consumption, device features, open-source	performance evaluation (memory, CPU, power consumption)
[SK13]	2013	Titanium, Rhodes, PhoneGap, Sencha Touch	functionality (8), usability (6), developer support (4), reliability/performance (4), deployment (8)	criteria definition and qualitative tool comparison
[XX13]	2013	*none* (cross-platform approaches in general)	distribution, programming languages, hardware & data access, user interface, perceived performance	criteria definition
[CGG14]	2014	MoSync, Titanium, jQuery Mobile, PhoneGap	license, community, API, tutorials, complexity, IDE, devices, GUI, knowledge	qualitative tool comparison and apps with animations
[CG15]	2015	PhoneGap, Titanium	battery consumption, device resource usage	evaluation of battery consumption
[DM15]	2015	PhoneGap, Titanium, Adobe Air, MoSync	tool capabilities (9), performance (5), developer support (2)	performance benchmarks and development experience discussion
[HHH15]	2015	AngularJS, jQuery Mobile, HTML5/JS, RhoMobile, PhoneGap, Sencha Touch	Platform support (4), development support (7), deployment factors (6)	criteria definition and qualitative tool comparison

rather comprehensive approach. For example, Ohrt and Turau [OT12] have analysed nine tools with taking a developer focus and assessing user expectations. They e.g. have had a look at programming language, compilation without SDK, code completion, GUI designer, debugger, emulator, and extensibility with native code, as well as launch time, app package size, and memory usage.

Many papers focus on particular aspects, e.g. animations [CGG14], performance [Dal+13], and energy consumption [CG15]. Nonetheless, most authors at least provide criteria grouped into common categories [OT12; XX13; HHH15; SK13]. One problem typically found is a shortage of explanations (c.f. e.g. [HHH15]).

It can be summed up that many authors set out to conquer the field of cross-platform app development systematically. Without doubt, the papers shown in Table 23.2 provide substantial contributions. However, the rapid proliferation of the field and the only slowly emerging theory-

building mandate further work. This is also illustrated by many papers being published recently that – more or less isolated – address issues also discussed in this article. To conclude the study of related work, we highlight such works that address novel mobile devices.

Several papers address *smart TVs*. Typically, a combination of HTML5 and JavaScript is proposed to enable cross-platform development. Sub-topics are interactive ads [PG15b], serious games [Ryu+14], and 3D content [PG15a]. Work on *wearables* is more scarce [Kim+16]. Some authors have proposed middleware approaches to achieve a broader device-span [Chm13], in one case even on the hardware layer [Zha+11]. Despite much blurriness, *smart homes* could be a future area of cross-platform research [Jie+15].

Contrasting the hype around multimedia novelties for cars, few scientific papers tackle *in-vehicle apps*. Current discussions revolve around general challenges and potential applications [Wol13], the integration of non-automotive applications into the automotive environment [RP12], and usability [QG14]. A few papers provide experimental implementations of novel concepts such as a route planning app for head-up (HUD) displays [NBN14], an *Open Service Cloud* for cars [DRB15], and "remote" human machine interfaces (HMI) [DHH13]. While these papers help to understand the possibilities of cars as a potential target of apps, they are far away from actually discussing cross-platform challenges and chances.

23.3 Background

As a prerequisite for a differentiated evaluation of mobile platforms, we need to categorize the variety of devices. From our understanding, *mobile consumer devices* are designed to be used in absence of *stationary* workstation hardware by non-business users. While formerly it was possible to categorise by operation system, this is not feasible anymore: e.g. Windows 10 spans device classes. As no such classification exists in scientific literature, we propose a simple subdivision:

Traditional general-purpose devices

- Smartphone
- Tablets, including hybrids such as netbooks and (so called) ultrabooks

Novel mobile devices

- Smart TVs and entertainment devices[1]
- Wearables
  - Smart watches, e.g. iWatch, Pebble, Samsung Gear
  - Sensing devices, e.g. fitness trackers, GPS watches
  - Smart glasses for augmented reality, e.g. Google glass, MS Hololens
- Vehicles, e.g. from BMW, Tesla, Ford
- Smart home applications[2]

[1] While such devises impose themselves as being included in a categorisation as such, they arguably are not *mobile* in the sense of all other devices named here.

[2] This field is still very fuzzy but rapid proliferation mandates naming it here already.

This list is not meant as a *proven* categorisation but as a working scheme for this paper. Thus, we refrain from an elaborated explanation. Within each of these device classes, a multitude of devices based on different platforms has emerged. Whereas Android and iOS have divided most of the market share of smartphone platforms amongst themselves [Wv16], competition among the novel mobile device platforms is high and no clear winners are foreseeable. A short overview of this field is provided next.

App-enabled smart TVs are already present in 35% of U.S. households [Sta16] and two approaches of development exist: middleware and frameworks. Over 90% of connected TVs sold in Germany support the HTML5-based HbbTV standard that has evolved from previous approaches such as CE-HTML and Open IPTV [Sta16; Hbb16]. In addition, many individual frameworks emerged, for example the open-source media centre Kodi/XBMC with various forks, Android TV, Tizen OS for TV, and webOS [XBM; Goo16a; Lin16; LG 16].

With smartwatches, Google and Apple again compete for dominance with their respective Android Wear and watchOS platforms. Pebble OS, Tizen OS, and webOS are further players in this domain [Bou15]. Whereas several vendors open-sourced their operating system, few vendor-agnostic platforms exist such as AsteroidOS [Rev16]. Other wearable devices such as fitness trackers usually ship with proprietary platforms, e.g. Microsoft Band and Firefox OS for Wearables. Those devices often support pairing with smartphones of multiple platforms; however app development is still limited. Vendors such as Fitbit and Garmin do not even produce devices with modifiable operating system [Bou15].

Concerning the upcoming connected cars, there are four approaches for developing in-vehicle apps [SV14]. First, Android Auto, Blackberry QNX, and Windows Embedded are technologies that are rebranded by car manufacturers and run native apps on the car's head unit. Second, some cars allow access and control of features such as door locks through a remote API. Examples include General Motors, Airbiquity, and an unofficial API for Tesla cars [Dor16]. Third, platforms including Apple CarPlay and the MirrorLink alliance use screen mirroring of apps running on the smartphone and displayed on the car's screen [DHH13]. This approach honours security concerns by car manufacturers. Fourth, Dash Labs, Mojio, Carvoyant, or Automatic connect to the on-board diagnostics port to interact with the car. Although this requires a Bluetooth dongle as additional hardware, many cars can be supported that are not designed to be app-enabled in the first place. In addition to this variety of approaches, distribution of apps is a challenge because of the underlying fight for dominance between car manufacturers "owning" the car platform [SV14].

This overview of technologies shows similar characteristics of fragmentation as the smartphone market several years ago [HHM13]. However, few cross-platform approaches currently exist in the domain of novel mobile consumer devices. Interestingly from a cross-platform perspective, many smart TV platforms natively support app development using Web technologies such as HTML5 and JavaScript, thus being well-suited for cross-platform approaches. Some platforms such as Android and Tizen have branches that run on multiple devices from TVs to smartwatches, potentially allowing for a future development across device class borders. Samsung TOAST is an

early initiative to simultaneously develop for Samsung Smart TV, the new Tizen platform and browsers, based on the established Apache Cordova framework [Sam16].

The other way around, smartwatches can be paired with more than one platform [Dou15]. Such apps that act as a (smartphone) device extension are current practice and thus cross-platform development approaches must consider and support each combination of host and watch platform. However, some smartwatch platforms are announcing stand-alone capabilities [Goo16c].

Several platforms claim to be the adequate open platforms for smart home and IoT applications. Qualcomm's AllJoyn, Intel's IoTivity, Apple HomeKit, and Google Brillo are the most important players that try to establish their middleware as comprehensive solution [Car15].

Finally, for in-vehicle apps, no widespread cross-platform frameworks exist due to the novelty of devices and a lack of platform accessibility. Potentially, a middleware approach [DRB15] might be an option to provide an open ground for developers and at the same time guarantee security.

23.4 Criteria

In the following, we propose our categorisation framework. We start by discussing methodological considerations before explaining the criteria.

23.4.1 General Considerations

As argued in Section 23.2, we have been inspired by existing evaluation frameworks. Facilitating the requirements arising from the broad scope intended for our framework and catering for the progress in the field in the meantime, we propose numerous extensions and revisions. Most notably, we do not distinguish criteria by two perspectives (*infrastructure* and *development* originally [HHM13]) but by four.

The *infrastructure* perspective describes the general background and prospect of a cross-platform development approach. The *development* perspective takes into account aspects of using an approach for carrying out the actual programming activities. In addition to these, we introduce the *app* perspective and the *usage* perspective. The former offers an assessment of the capabilities of apps that can be realized with a given approach. This not only leads to more clarity with regard to the distinction of actual development activities and the outcome of development but also has multi-device class support in mind. While development might not differ much for different classes of devices, the capabilities of an app might vary significantly. The usage perspective considers usability, ergonomic, and performance aspects that are essential factors for user acceptance.

The categorisation into four perspective allows focussing on relevant aspects for particular needs. These needs might arise from the targeted device class(es) but may as well come from other sources. An example could be a specific focus on business apps [MWA15]. In the following, we

provide the rationale for each of our criteria following the above proposed categorisation. Besides referencing sources already discussed, we provide additional evidence where appropriate.[3]

23.4.2 Infrastructure Perspective

(I1) License: The license under which a framework is published is essential for the type of product to develop. It needs to be assessed whether a developer is free to create commercial apps, for example when using open source software [Dal+13; CGG14; PSC12]. In addition, the pricing model needs to be considered. A framework could be freely distributed, or require one-time or regular license payments with regard to the number of developers, projects, or as a flat fee [HHH15; SK13].

(I2) Supported Target Platforms: The number and importance of supported mobile platforms within a device class is a major concern for choosing a cross-platform approach [CGG14; PSC12]. Furthermore, support varies regarding different versions of each mobile operating system. The most recent version provides the newest features and its support is important to reach early adopters of a new technology [BB57]. However, the majority of users will use an old version of the system due to hardware limitations or slow update behaviour by users or vendors [Dob12]. Finally, it needs to be considered whether multiple device classes have to be bridged, for example a combination of smart TV and tablet application.

(I3) Supported Development Platforms: Flexibility regarding supported development platforms is beneficial for heterogeneous teams in which developers are accustomed to specific hardware and software such as an development environment [PSC12] (see also criterion D1). Moreover, the role within a team such as UI or UX design may require the approach to support multiple platforms.

(I4) Distribution Channels: With proprietary platform- or vendor-specific app stores typical for publication, the number of users who can be reached is critical. It needs to be weighted against fixed and variable costs of app store accounts and app publishing. The ease of the publication process itself also needs to be considered, regarding e.g. the average duration for initial app placement and update distribution as well as the strictness and detailedness of the review process [XX13]. Cross-platform frameworks vary by the degree of compatibility with app store restrictions and submission guidelines [SK13; DM15]. Further app store integrations include app rating to reach a better app store ranking as well as automatic update notifications within the app for rolling out updates fast [HHH15].

(I5) Monetisation: From a business perspective, the possibility and the complexity of selling the app itself and subsequent in-app purchases need to be considered as well as the availability of advertisement [DM15]. These features need to be traded off against direct costs and commissions to the app store operator. Again, cross-platform development frameworks can support this aspect, for example by providing interfaces to payment providers or advertising networks.

[3] However, we do *not* cite [HHM13] for each single criterion originating from this work.

(I6) Global App Distribution: Typically, a global distribution of apps is desired – unless specific reasons for restrictions exist. Approaches can offer built-in support for internalisation and localisation to create and distribute app versions targeted (and potentially restricted) to specific geographic regions. In addition, translation capabilities allow for easy delivery of multi-language content and provide format conversions for dates, currencies and location particularities [SK13].

(I7) Long-term Feasibility: Choosing an approach might be an important strategic decision considering the significant initial investment for training or hiring developers as well as the risk of technology lock-in, particularly for smaller companies. The maturity and stability of a framework can be evaluated concerning the historical and expected backwards incompatible changes of major releases. Other indicators are short update cycles, regular bug-fixes, and security updates. In an active community, developers exchange knowledge to solve issues. Ideally, several commercial supporters back the project with financial resources and steady contribution. Costs for professional support inquiries need also be considered, potentially increasing the attractiveness of a promising open-source project while safeguarding efficient solutions to development issues [HHH15; SK13].

23.4.3 Development Perspective

(D1) Development Environment: The maturity and features of an integrated development environment (IDE) heavily influence development productivity and speed. Tool support includes functionalities of the IDE such as auto-completion, debugging tools, and an emulator to enable rapid app development cycles [HHH15; SK13; CGG14; PSC12; DM15]. In addition to the IDE typically associated with the cross-platform approach, the freedom to use accustomed workflows, e.g. choose a preferred IDE, lowers the initial set-up effort for additional dependencies such as runtime environments or software development kits (SDK) [SK13].

(D2) Preparation Time: The learning curve, i.e. the subjective progress of a developer while exploring the capabilities and best practices of the approach, should foster rapid initial progress. To lower the entry barrier, the number and type of required technology stack and programming languages need to be considered [XX13; CGG14; SK13; PSC12], e.g. using known paradigms to further reduce the initial learning efforts [CGG14]. With unique benefits and characteristics, the quality of API documentation is also of major importance. "Getting started" guides, tutorials, and code examples initially clarify the framework's features and structure, whereas a corpus of best practices, user-comments, and technical specifications support in solving issues over the course of development [SK13; DM15].

(D3) Scalability: Scalability refers to the modularisation capabilities of the framework and generated apps in larg-scale development projects. Partitioning code in subcomponents and architectural design decisions such as the well-known Model-View-Controller pattern has implications on the app structure. Thus, the number of developers can be increased while extending the app's functionality [HHH15; PSC12]. However, a modular framework itself can guide and support this

division of labour. With specified interfaces and interactions between the components, developers can specialize themselves on few relevant components.

(D4) Development Process Fit: Departing from the traditional approach of implementing software from a fixed and comprehensive specification, a variety of methods with agile characteristic exist today. For such projects, the cross-platform approach can be evaluated regarding the effort to create the *minimum viable product*, e.g. the amount of boilerplate code and initial configuration, as well as the effort to subsequently modify its scope. This criterion also relates to the organisational aspect of scalability in terms of developer specialisation. In contrast to full-stack developers in small projects, modularizing development using roles can be supported through tailored views or specialized tools [Was10].

(D5) UI Design Approach: UI development is a major concern for cross-platform approaches. Graphical user interfaces are highly platform-specific and often just covered by a default appearance defined by the framework [HMK13]. In addition, a separate WYSIWYG editor to develop appealing interfaces for multiple devices can be beneficial and increases the speed of development compared to constantly deploying the full app to a device or an emulator.

(D6) Testing Support: App logic and user interfaces need to be tested with established concepts such as system and unit tests [HHH15; SK13]. To test context-sensitive mobile scenarios more authentically, external influences (such as *bad* connectivity) may be simulated [MS15]. Furthermore, possibilities of monitoring the app at runtime improve the testability, e.g. providing a developer console, meaningful error reporting, and logging functionalities for app-specific and system events. Tool support may also include remote debugging on a connected device rather than emulator environments, test coverage visualisation and metrics [HHH15].

(D7) Deployment Support: Build toolchain support immensely simplifies the deployment process, i.e. generating individual packages for all targeted platforms. Approaches vary from requiring all native SDKs to external build services and cloud-based techniques [HHH15; SK13]. Sophisticated projects additionally use *continuous integration* platforms to automate testing. Frameworks can be explicitly designed to integrate with such toolchains. Regarding production, the framework might also offer optimised build options (e.g. *minified* code) and app store integrations to automatically publish updates [HHH15].

(D8) Maintainability: In contrast to (I7), maintainability deals with the evolution of a code base over time [SK13] and difficult to quantify. Lines of code (LOC) for a specified reference app may be used for comparison with the assumption that less LOC are easier to support regarding readability of source code, amount of training and familiarisation, etc. This concept is similar to programming languages themselves, where so-called *gearing factors* try to compare the amount of code per unit of functionality [QSM09]. Advanced maintainability metrics are hard to apply due to the heterogeneity and varying complexity of frameworks, especially in case of apps composed of different programming languages. Furthermore, the reusability of source code across development projects can be evaluated, for instance concerning the portability to other software projects [SK13].

(D9) Extensibility: Special requirements may introduce the need for features that go beyond the core of the framework. These might not be put into practice with high priority. Therefore, the possibility to extend the framework with custom components and third-party libraries should be evaluated. Examples for such extensions include additional UI elements, functionalities to access device features, and solutions to common challenges such as data transfer [HHH15; PSC12].

(D10) Integrating Native Code: For some applications, running native code within the application is a requirement. This seemingly invalidates the idea of cross-platform development; nonetheless, it *can* be beneficial: Previously existing code can be reused, e.g. when migrating apps to a cross-platform development approach and replacing platform-specific code over time. Also, native platform APIs might enable access to platform functionalities and device features currently not available on the framework's level of abstraction [SK13; PSC12].

(D11) Speed of Development: Rapid development is influenced by the amount of boilerplate code necessary for functional app skeletons (cf. [Hei+14]) and the availability of typical app functions such as user authentication. Assuming salaries to be independent of programming language proficiency, development speed directly influences the variable costs and ultimately the return-on-investment.

23.4.4 App Perspective

(A1) Access to Device-specific Hardware: For cross-platform approaches, the coverage of platform- and device-specific hardware is of supreme importance [HHH15; Dal+13; CGG14; SK13; DM15]. Especially regarding the capabilities of novel mobile devices, a plethora of device hardware is present today. This includes sensors such as camera, microphone, GPS, accelerometer, gyroscope, magnetometer, temperature sensor, and heart rate monitor. In addition, cyberphysical systems enable bidirectional interaction that can modify the environment through actuators. The set of individual features is evaluated according to the framework's documentation.

(A2) Access to Platform-specific Functionality: Regarding the software side of the various mobile platforms, functionalities include a persistence layer such as the file system and access to a database on the device, contact lists, information on the network connection, and battery status [HHH15; DM15]. In addition, in-app browser support may be desirable for fetching additional content from the Internet without leaving the app [HHH15]. Advanced features like monitoring or push notifications can be realised using background services [SK13].

(A3) Support for Connected Devices: Current wearable devices and also sensor/actuator networks of cyberphysical systems often require to be coupled to a respective master device (e.g. a smartphone). This trend of "device extensions" needs to be evaluated regarding viable device combinations. More specific this includes the level of access to coupled device data and sensors as well as additional user interfaces. This may be trivial if the platform provides a layer of abstraction that exposes the coupled device similar to other device components. Yet, in many cases the cross-platform approach needs to take care of this additional complexity.

(A4) Input Device Heterogeneity: The input device criterion evaluates the support of the approach with regard to the variety of input devices that can be used to interact with the app. This includes traditional devices such as keyboard and mouse as well as (multi-) touch screens, voice recognition, remote controls, hardware buttons and more. Each of these devices can process many interaction mechanisms, for example multi-touch screens reacting to gestures such as different types of taps, swipes, pinches, pressure, orientation changes etc., all of which the cross-platform tool needs to make available to the app developer.

(A5) Output Device Heterogeneity: Mobile devices provide a huge variety of different output devices differing in device size, resolution, format (e.g. *round* smartwatches), colour palette, frame rate (e.g. E-Ink screens), and opacity (e.g. augmented reality projections). This poses challenges as adaptability is already a major challenge for traditional devices [AK14]. In addition, the app has to adapt to device class-specific context changes, thereby realizing well-understood design ideals [SAW94] such as day/night-mode appearances for in-vehicle apps.

(A6) Application Life Cycle: This criterion refers to how far a framework supports the life-cycle inherent to an app. Platforms may differ in starting, pausing, continuing, and exiting an app [SK13]. Additional differences arise from the life cycle of individual views and view elements.

(A7) Business Integration: To integrate with the overall business, support for data exchange protocols, serialisation, and multiple data formats are often required [Dal+13]. Apps may communicate with existing Web service back-ends for data storage and processing, or initiate inter-app communication. E.g., business processes often require collaboration of different user roles. Business integration also refers to customizability, e.g. being adaptable to a *corporate identity* [SK13].

(A8) Security: Frameworks can support the development of secure apps on several levels. First, mobile platforms are usually restrictive regarding access permissions. Requesting permissions on demand increases not only the perceived security of an app. Second, data loss can be avoided by using data encryption mechanisms on the device as well as secure data transfer protocols against eavesdropping [Dal+13; HHH15]. Third, the framework may provide user input validation and prevent cross-site forgery and code injection [HHH15].

(A9) Fallback Handling: Considering the device and platform heterogeneity of (A1)-(A5), intelligent fallback mechanisms aid in case individual features are unsupported or restricted. As a naïve approach, the user may be redirected to a Web page. Sophisticated actions include *graceful degradation* techniques with simpler representations [Ern+16], or alternative functions to fulfil the user's task.

23.4.5 Usage Perspective

(U1) Look and Feel: This criterion considers whether available UI elements have a native look & feel or rather behave like a Web site [SK13]. The set of elements can be evaluated according to the human interface guidelines of the respective platform. Particularly, rich user interfaces with 2D/3D animation and multimedia features are challenging for cross-platform tools [Dal+13].

In addition, it should be considered to which degree a framework supports the platform-specific usage philosophy, e.g. the position of navigation bars, scrolling, and gestures [SK13].

(U2) Performance: Application speed, stability, and responsiveness of the app on user interaction are essential performance aspects. Apart from the subjective user experience, the speed at start-up, after interruptions and for shut-down can be measured [DM15]. Moreover, resource usage can be assessed, e.g. CPU, RAM and battery utilisation at runtime, or download size [SK13; CGG14; Dal+13; CG15].

(U3) Usage Patterns: Apps are frequently used for a short amount of time and are likely to be interrupted. Users want an "instant on" experience and continue where they left the app. To match usage patterns, apps have to integrate into personal workflows for information processing such as sharing with other apps or saving to persistent storage, and community interactions such as messaging, e-mail, and social media. For some use cases, support for synchronisation of app data across multiple devices of the user for seamless context switching is beneficial. In addition, notification centres of the platform are gaining importance for app interaction.

(U4) User Management: Cross-platform frameworks may support different types of user handling, reaching from purely local apps to user accounts across multiple devices and role-based authentication. Authentication may therefore be performed in-app or server-based, and potentially connected to session management. In addition, mobile devices may provide various login mechanisms, including traditional passwords, gestures, and biometric information such as fingerprints, voice recognition, or other characteristics [LL15].

23.5 Evaluation

Due to their recent emergence, cross-platform approaches barely exist for novel mobile device classes (cf. Section 23.2). Therefore, the following evaluation compares PhoneGap to Web apps and native applications with regard to traditional smartphone mobile devices. PhoneGap was chosen due to its perennial popularity as leading cross-platform development tool [Vis15]. The evaluation is by no means a comprehensive survey of the cross-platform framework itself (as provided by [DM15; HHH15]) but should serve as exemplary comparison in order to discuss our approach of weighted criteria evaluation. Thereby, this evaluation particularly serves as a benchmark for our evaluation framework.

23.5.1 Weight Profiles

To cater for differences across heterogeneous and evolving mobile device classes, our approach to cross-platform tool evaluation applies a weighting mechanism. Each of the 31 criteria receives between 1 and 7 points with a total of 100 points assigned (not necessarily distributed equally across the categories), constituting the so-called *weight profile*. Each criterion is evaluated on a scale from 0 (criterion unsatisfied) to 5 (optimally fulfilled). The weighting points directly

translate to percental values used in calculating the *weighted score*. A weight profile reflects the requirements of a specific device class regarding cross-platform development. It can be individually adapted to the future evolution of the mobile ecosystems, as well as changed to reflect particular needs, e.g. regarding the background of developers. The proposed weight profiles for the device classes presented in Section 23.3 are depicted in Table 23.3 along with an exemplary evaluation.

In the following, we focus on the *smartphone* device class, which can be backed with empirical and theoretical work. Studies have shown that cross-platform approaches are often developer-oriented [Vis15; res14]. From an infrastructure perspective, this means that free and open approaches are considered particularly important. Long-term feasibility benefits from a stabilized smartphone ecosystem with Android and iOS as main players [Wv16]. Distribution channels are mostly limited to platform-specific app stores with a broad set of features.

App developers want to use existing standards and previous knowledge for fast-paced development [Vis15; res14]. In contrast, the current practice of smartphone apps apparently does not cover large development teams. As a result, organisational aspects such as scalability, maintainability, and development process integration are not requested by practitioners [res14]. UI design seems to be an ongoing challenge for cross-platform frameworks and may even become more important for "standing out from the mass of apps" [AK14].

On the application side, access to a broad rage of device functionalities is requested, while support for smartphone screens as both input and output device has matured [Vis15]. Apps are still rather developed for social and communication purposes [LLM15], thus business integration and security issues are not prioritized.

With mobile usage soon surpassing desktop usage [LLM15], performance and native look and feel remain important topics for smartphone development. Finally, user management and usage patterns play an inferior role on smartphones as these are mainly designed for single-person usage.

23.5.2 Web Apps

Web apps are mobile-optimized Web sites built with HTML5 and JavaScript (JS), and executed within the smartphone's browser. They rely on open standards and are highly cross-platform compatible while using Web development tools (I1-I4). While profiting from an immense community of developers, "app-like" behaviour needs to be implemented manually and distribution cannot be controlled (I5-I6).

Only Web development skills are required; many tutorials and profound tool support is available (D1-D2). The universality of the Web at the same time limits the application to apps, e.g. requiring boilerplate code or providing no guidance on the structure of source code and development (D3-D5). Testability is problematic: desktop browsers emulate the respective mobile counterpart inconsistently and mobile in-browser debugging is hardly supported (D6). Various libraries

Table 23.3: Comparison of Approaches and Device Class Weight Profiles

	Smartphone Comparison				Category Weights				
Criterion	**Weight (%)**	**Web apps**	**PhoneGap**	**Native apps**	**Tablets**	**Entertainment**	**Wearables**	**Vehicle**	**Smart Home**
I1: License	6	5	5	5	5	6	5	3	5
I2: Target Platforms	7	5	5	1	5	6	7	4	7
I3: Development Platforms	2	5	5	2	2	2	1	1	1
I4: Distribution Channels	2	5	3	4	2	3	4	3	3
I5: Monetisation	2	0	3	5	2	1	1	2	2
I6: Global Distribution	2	1	3	5	2	2	2	0	1
I7: Long-term Feasibility	5	5	5	4	5	3	3	6	5
D1: Dev. Environment	7	4	5	5	7	7	5	5	6
D2: Ramp-up Time	7	5	4	3	7	7	5	1	5
D3: Scalability	2	3	3	3	2	3	2	3	2
D4: Development Process Fit	2	3	4	2	2	3	1	4	2
D5: UI Design	4	3	3	4	4	5	5	6	3
D6: Test Support	3	3	4	5	3	3	4	7	3
D7: Deployment Support	3	5	5	3	3	3	4	5	2
D8: Maintainability	2	2	4	2	2	2	1	5	2
D9: Framework Extensibility	2	5	5	0	2	2	2	1	2
D10: Native Extensibility	2	0	3	5	2	2	1	0	0
D11: Speed of Development	4	2	3	0	4	3	3	2	4
A1: Hardware Access	5	2	4	5	3	1	6	4	7
A2: Platform Functionality	5	2	4	5	5	3	2	3	3
A3: Connected Devices	3	0	0	5	2	1	7	4	7
A4: Input Heterogeneity	1	4	4	5	3	3	2	2	2
A5: Output Heterogeneity	1	4	4	5	1	1	6	3	4
A6: App Life Cycle	3	0	4	5	3	3	3	3	2
A7: Business Integration	2	3	3	5	3	3	1	2	1
A8: Security	3	0	0	1	4	1	3	7	5
A9: Fallback Handling	2	2	4	0	1	4	3	2	1
U1: Look and Feel	4	1	3	5	4	2	4	5	3
U2: Performance	4	3	2	5	4	6	3	3	2
U3: Usage Patterns	2	0	1	2	4	4	4	3	4
U4: User Management	1	0	0	0	2	5	0	1	4
Weighted Score		**2.99**	**3.66**	**3.56**					

simplify development, yet native code is unsupported (D9-D10). As a result, Web apps are rather easy to create and modify using established toolchains. However, all aspects regarding app life cycle, integration, security, and fallback have to be built manually without platform-specific abstraction (D7-D8, D11, A6-A9).

Accessing device components is possible only via HTML5 APIs such as Media Capture Stream, which are scarcely supported by mobile browsers (A1-A2) [Mob15]. As the execution happens in the browser, keyboard and gesture support are well established through JS events but limited to Web page behaviour and browser controls (A3, U1). Furthermore, CSS can be used customise the design and target different outputs devices, with the exception of connected devices (A4-A5, A7). Finally, usage patterns and user management are completely up to the developer, whereas the overall performance depends on the smartphone browser and is likely be optimized by the platform provider (U2-U4).

23.5.3 PhoneGap

PhoneGap was initially developed around 2009 and is still the top-used cross-platform development tool [Dav09; Vis15]. Freely available under the permissive Apache License, PhoneGap targets all major smartphone platforms in various versions (I1). Technically, apps are developed using HTML5/JS/CSS and executed within a *Web view wrapper* component without browser controls. This allows installing the apps and providing API access to native functionality. Thus, the framework does not adhere to specific platform guidelines but provides a general mobile appearance that can be distributed through any app store but without advanced features such as in-app purchases (I4-I6). Its long existence and stable API has created a large community that in turn supports the long-term perspective (I7).

Similar to Web apps, developers can freely choose their preferred Web development environment and profit from previous knowledge. The framework's structure requires little knowledge and it is well documented (I3, D1-D2). PhoneGap generates a running app skeleton and file structure but does not impose further implications on the development process (D3-D5). All app functionality needs to be implemented manually (A8-A9, U3-U4). Outstanding features are the *cloud deployment* that requires no locally installed SDKs as well as the *remote debugging interface* that connects to real devices (D6-D7). Maintainability is enabled through the extensive and stable API abstracting from platform differences and increasing the speed of development (D8, D11, A1-A2). In addition, numerous plug-ins exist, extending the functionality and covering many of the aforementioned drawbacks, also allowing the execution of native code (D9-D10).

Regarding native behaviour and appearance, the *Event API* provides access to life cycle events and platform settings can be retrieved via the *Device API* to target specific platforms (A6-A7,A9,U1). Support for input and output is similar to Web apps and likewise restricted to the main device (A3-A5). The app performance depends again on the smartphone's browser capabilities with additional framework overhead (U2) [Apa16; Ado15].

23.5.4 Native Apps

Any cross-platform development approach can be benchmarked against native app development. While it naturally is closest to a platform's capabilities, developing natively not necessarily is the most efficient or elegant option.

Platform SDKs are freely available and fully integrated into the respective app stores. The latter provide a broad set of features for distribution and monetisation (I1, I4-I6). Whereas development might be possible with several technologies, the target platforms are limited to one (I2-I3).

iOS and Android as prevailing platforms and can be treated as reliable on long-term [Wv16]. Platforms typically require specific programming language knowledge, although extensive documentation and community support are available (D2). Moreover, a full ecosystem with tool support for all phases of development is usually provided, with varying degrees of alternatives (D1, D5-D7).

The flexibility of implementation comes at the cost of few guidelines on structuring and subdividing development work (D3-D4). The platforms usually do not provide support for recurring programming tasks (A8-A9, U3-U4). Obviously, the speed of developing multiple native apps is unmatched low (D11).

Native apps can access all possible features of a given platform (A1-A7). Ultimately, a fully native appearance and behaviour as well as performance without runtime overhead can only be reached with native apps (A1-A2).

23.6 Discussion

The framework presented in the prior sections should provide a step towards a sound theory of cross-platform app development. However, it is by no means static. In fact, we hope it can be the foundation for application and extension by others. Thereby, the framework can stay at eye level with further developments in the field. Specifically, depending on the emergence of novel device classes and the possible proliferation of further kinds of devices, revisions can be applied.

In the following we reflect on our work, starting with a *synthesis of findings*. Our criteria have proven to be useful and applicable in the exemplary use. Categorisation into four perspectives worked well, although it remains to be seen whether an even finer scheme might be advisable to cater for future developments. While the weighting profiles will need further tweaking (see also below), they lead to producible results. In particular, the smartphone profile has proven to be feasible. As could be expected and is widely affirmed by related literature (cf. Sections 23.2 and 23.3), PhoneGap as the leading approach is better suited for cross-platform development that targets smartphones than pure native or Web apps. It should be noted that native development not simply satisfies all criteria *but* cross-platform capabilities; working natively can have its own *overhead*.

The additional weight profiles for now have to be seen as proposals. They should nonetheless be reasonable starting points. In particular, they are well-suited to address idiosyncrasies of specific device classes, e.g. to put weight on security for apps in cars or smart homes.

The tablet profile is rather similar to smartphones, particularly from the infrastructure and the development perspective. Multi-user scenarios and business integration need more attention, and additional means of input play a role. Quite differently, the entertainment profile has less business implications and is less focused on security, sensors and platform-features. Performance requirements might add complexity and support for multiple users is a prerequisite.

For the wearable profile, yet other specialities need to be taken into account. Deploying to and testing on devices is quite hard. User interfaces differ much from platform to platform. Apps typically are very small and must perform with low resource utilisation. At the same time, usage scenarios are simpler and due to high-fluctuation of devices a long-term focus needs to be less emphasized.

Security is of foremost concern in the vehicle profile. Due to the field's fuzziness, it shares similarities with the wearable profile but has more focus on professional software development. Most blurry is the smart home profile, which needs to address the heterogeneity of possible devices along with security concerns.

A number of open questions can be raised. While we deem the evaluation framework to be readily usable, particularly due to its solid literature foundation, the weighting remains open for revisions. Future research will scrutinize whether the device classes have been chosen wisely. There is no easy answer to this since new kinds of devices might be designed with a focus on app-enablement – or not. For example, Tesla announced an own SDK but current work obviously has taken another direction due to security concerns [Lam16]. Moreover, it is hard to predict market development. For example, Android Wear [Goo16b] *might* unify development for Wearables or at least consolidate different streams.

It remains to be seen whether the success of Web technology (including frameworks such as PhoneGap) will be repeated for new device classes. On the one hand, devices with hardware that is not powerful enough to run a WebKit-Engine such as some watches might require different approaches. Other devices, such as arguably fitness trackers, do not even pose a platform that would be comparable to Android or iOS. On the other hand, Web technology might be the bridging element for heterogeneity. It is still very hard to image the proper abstraction for devices that fall under the umbrella of smart home technology.

Furthermore, it needs to be questioned whether for all device classes full ecosystems as for smartphones will be established. A Cloud-based middleware, mirroring, or other "remote" approaches could solve issues such as low performance, hardware heterogeneity and security without even relying on devices directly. Moreover, device classes might converge. Modern fitness trackers have smartwatch functionality; a smartwatch was recently hacked to run applications only imaginable on smartphones before [Kra14]. So called *instant apps* can be run without installation [Gan16] and might also contribute to future changes.

Due to the breadth of our work and also due to the novelty of some of the tackled topics, this paper is bound to limitations. While we built upon the literature both for the derivation of criteria and for their exemplary usage, we have not evaluated our work empirically. This is particularly an issue for the weight profiles, which need to be assessed based on the input from practitioners. Moreover, we have made assumptions about the future, most notably considering device classes. It seems unlikely but it might turn out that e.g. app-enabled cars will not gain importance. Even if they do, it is not given that cars (or other device classes) will allow for reasonable cross-platform app development support. Looking towards the future is part of our work but a boundary at the same time.

The limitations do not impede the value of our work, though. In fact, in combination with the above discussed open questions they provide the foundation for our future work. Writing this paper has been *more* than setting out to refresh the view on the topic of mobile computing. It has brought up a host of new ideas for us, particularly revolving around the differences in device classes. We will strive to provide a unified understanding while honouring the particular strengths and possibilities offered by devices. A major source of our future work will be the above mentioned limitations. As a next step, we will work on a broader evaluation of current approaches based on our criteria. Moreover, we will assess possibilities how to get practitioners' feedback on the framework, ideally leading to an empiric validation of our work. This will include a revision of the weights and more concrete advice on approach choice. In particular, we would like to provide recommendations in form of case study-like scenarios for future applications. Finally, we will also seek to make further theory contributions, especially concerning an abstraction from device classes.

23.7 Conclusion

In this paper, we have presented work on an extended cross-platform app development evaluation framework. It extends existing papers and revised the criteria formerly proposed. In particularly, it takes into account differences in the increasing number of device classes and provides a weighted evaluation. We have not only comprehensively introduced our framework but given an exemplary evaluation. The findings suggest that the framework is well-suited. Nonetheless, much work remains due to the novelty and breadth of the field.

References

[Ado15] Adobe Systems Inc. *PhoneGap Documentation*. http://docs.phonegap.com. 2015. URL: `http://docs.phonegap.com` (visited on 05/30/2016).

[AK14] Suyesh Amatya and Arianit Kurti. "Cross-Platform Mobile Development: Challenges and Opportunities". In: *ICT Innovations 2013*. Vol. 231. Springer, 2014, pp. 219–229.

[Apa16] Apache Software Foundation. *Apache Cordova Documentation*. https://cordova.apache.org/docs/en/. 2016. URL: https://cordova.apache.org/docs/en/ (visited on 05/30/2016).

[BB57] George M. Beal and Joe M. Bohlen. *The diffusion process*. Agricultural Experiment Station, Iowa State College, 1957.

[Bou15] Gil Bouhnick. *A List of All Operating Systems Running on Smartwatches [Wearables]*. 2015. URL: http://www.mobilespoon.net/2015/03/a-list-of-all-operating-systems-running.html (visited on 05/31/2016).

[Car15] Jamie Carter. *Which is the best Internet of Things platform?* 2015. URL: http://www.techradar.com/news/-1302416 (visited on 05/31/2016).

[CG15] M. Ciman and O. Gagg. "Measuring energy consumption of cross-platform frameworks for mobile applications". In: *LNBIP* 226 (2015), pp. 331–346. DOI: 10.1007/978-3-319-27030-2{\textunderscore}21.

[CGG14] M. Ciman, O. Gaggi, and N. Gonzo. "Cross-platform mobile development: A study on apps with animations". In: *Proc. ACM Symposium on Applied Computing*. 2014. DOI: 10.1145/2554850.2555104.

[Chm13] J. Chmielewski. "Towards an architecture for future internet applications". In: *Lecture Notes in Computer Science* 7858 (2013), pp. 214–219. DOI: 10.1007/978-3-642-38082-2{\textunderscore}18.

[CS12] C. P. Rahul Raj and Seshu Babu Tolety. "A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach". In: *2012 Annual IEEE India Conference (INDICON)*. 2012, pp. 625–629. DOI: 10.1109/INDCON.2012.6420693.

[Dal+13] I. Dalmasso et al. "Survey, comparison and evaluation of cross platform mobile application development tools". In: *Proc. 9th IWCMC*. 2013. DOI: 10.1109/IWCMC.2013.6583580.

[Dav09] Lidija Davis. *PhoneGap: People's Choice Winner at Web 2.0 Expo Launch Pad*. 2009. URL: http://readwrite.com/2009/04/02/phone_gap (visited on 05/31/2016).

[DHH13] S. Durach, U. Higgen, and M. Huebler. "Smart automotive apps: An approach to context-driven applications". In: *LNEE* 200.VOL. 12 (2013), pp. 187–195. DOI: 10.1007/978-3-642-33838-0-17.

[DM15] S. Dhillon and Q. H. Mahmoud. "An evaluation framework for cross-platform mobile application development tools". In: *Software – Prac. and Exp.* 45.10 (2015), pp. 1331–1357. DOI: 10.1002/spe.2286.

[Dob12] Alex Dobie. *Why you'll never have the latest version of Android*. 2012. URL: http://www.androidcentral.com/why-you-ll-never-have-latest-version-android (visited on 05/25/2016).

[Dor16] Tim Dorr. *Tesla Model S JSON API*. 2016. URL: http://docs.timdorr.apiary.io (visited on 05/31/2016).

[Dou15] Adam Doud. *How important is cross-platform wearable support?* 2015. URL: http://pocketnow.com/2015/05/10/cross-platform-wearable-support (visited on 05/31/2016).

[DRB15] M. Deindl, M. Roscher, and M. Birkmeier. "An architecture vision for an open service cloud for the smart car". In: *Green Energy and Technology* 203 (2015), pp. 281–295. DOI: 10.1007/978-3-319-13194-8{\textunderscore}15.

[El-+14] W. S. El-Kassas et al. "ICPMD: Integrated cross-platform mobile development solution". In: *Proc. 9th ICCES*. 2014. DOI: 10.1109/ICCES.2014.7030977.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *Proc. of the 12th WEBIST*. SciTePress, 2016, pp. 307–316.

[Gan16] Suresh Ganapathy. *Introducing Android Instant Apps*. 2016. URL: http://android-developers.blogspot.no/2016/05/android-instant-apps-evolving-apps.html.

[Goo16a] Google Inc. *Android TV*. 2016. URL: https://www.android.com/tv/ (visited on 05/31/2016).

[Goo16b] Google Inc. *Android Wear 2.0 Developer Preview*. 2016. URL: https://developer.android.com/wear/preview/index.html.

[Goo16c] Google Inc. *Android Wear 2.0 Developer Preview*. 2016. URL: https://developer.android.com/wear/preview/index.html (visited on 05/31/2016).

[Hbb16] HbbTV. *HbbTV Overview*. 2016. URL: https://www.hbbtv.org/overview/ (visited on 05/31/2016).

[Hei+14] Henning Heitkötter et al. "Comparison of Mobile Web Frameworks". In: *LNBIP*. Vol. 189. Springer, 2014, pp. 119–137.

[HHH15] A. Hudli, S. Hudli, and R. Hudli. "An evaluation framework for selection of mobile app development platform". In: *Proc. 3rd MobileDeLi*. 2015. DOI: 10.1145/2846661.2846678.

[HHM12] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. "Comparing Cross-platform Development Approaches for Mobile Applications". In: *Proceedings 8th WEBIST*. SciTePress, 2012, pp. 299–311.

[HHM13] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. "Evaluating Cross-Platform Development Approaches for Mobile Applications". In: *LNBIP*. Vol. 140. Springer, 2013, pp. 120–138.

[HKM15] Henning Heitkötter, Herbert Kuchen, and Tim A. Majchrzak. "Extending a model-driven cross-platform development approach for business apps". In: *Science of Computer Programming* 97, Part 1.0 (2015), pp. 31–36.

[HMK13] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. "Cross-platform model-driven development of mobile applications with MD^2". In: *Proc. SAC '13*. ACM, 2013, pp. 526–533.

[Jie+15] G. Jie et al. "Cross-Platform Android/iOS-Based Smart Switch Control Middleware in a Digital Home". In: *Mobile Inf. Sys.* 2015 (2015). DOI: 10.1155/2015/627859.

[Kim+16] H. Kim et al. "Wearable device control platform technology for network application development". In: *Mobile Information Systems* 2016 (2016). DOI: 10.1155/2016/3038515.

[Kra14] Konrad Krawczyk. *Hacker installs Windows 95 and Doom on a Samsung Gear Live smartwatch*. 2014. URL: http://www.digitaltrends.com/computing/hacker-installs-windows-95-and-doom-on-a-samsung-gear-live-smartwatch/.

[Lam16] Fred Lambert. *Tesla is moving away from an SDK*. 2016. URL: http://9to5mac.com/2016/01/28/tesla-sdk-iphone-apps-mirror/.

[LG 16] LG Electronics. *WebOS for LG Smart TVs*. 2016. URL: http://www.lg.com/uk/smarttv/webos (visited on 05/31/2016).

[Lin16] Linux Foundation. *Tizen*. 2016. URL: https://www.tizen.org/ (visited on 05/31/2016).

[Liu+16] Xin Liu et al. "Wearability Factors for Skin Interfaces". In: *Proc. 7th Augmented Human Int. Conf.* ACM, 2016, 21:1–21:8. DOI: 10.1145/2875194.2875248.

[LL15] A. De Luca and J. Lindqvist. "Is secure and usable smartphone authentication asking too much?" In: *Computer* 48.5 (2015), pp. 64–68. DOI: 10.1109/MC.2015.134.

[LLM15] Adam Lella, Andrew Lipsman, and Ben Martin. *The 2015 U.S. Mobile App Report*. 2015. URL: https://www.comscore.com/ger/Insights/Presentations-and-Whitepapers/2015/The-2015-US-Mobile-App-Report.

[ME15] Tim A. Majchrzak and Jan Ernsting. "Reengineering an Approach to Model-Driven Development of Business Apps". In: *LNBIP*. Vol. 232. Springer, 2015.

[Mob15] MobileHTML5. *Mobile HTML5 compatibility*. 2015. URL: http://mobilehtml5.org/ (visited on 05/30/2016).

[MS15] Tim A. Majchrzak and Matthias Schulte. "Context-Dependent Testing of Applications for Mobile Devices". In: *Open Journal of Web Technologies (OJWT)* 2.1 (2015), pp. 27–39.

[MWA15] Tim A. Majchrzak, Stephanie Wolf, and Puja Abbassi. "Comparing the Capabilities of Mobile Platforms for Business App Development". In: *LNBIP*. Vol. 232. Springer, 2015, pp. 70–88. DOI: 10.1007/978-3-319-24366-5_6.

[NBN14] M. Noreikis, P. Butkus, and J. K. Nurminen. "In-vehicle application for multimodal route planning and analysis". In: *Proc. IEEE 3rd CloudNet*. 2014. DOI: 10.1109/CloudNet.2014.6969020.

[OT12] Julian Ohrt and Volker Turau. "Cross-Platform Development Tools for Smartphone Applications". In: *Computer* 45.9 (2012), pp. 72–79. DOI: 10.1109/MC.2012.121.

[PG15a] E. Perakakis and G. Ghinea. "A proposed model for cross-platform web 3D applications on Smart TV systems". In: *Proc. 20th Web3D*. 2015. DOI: 10.1145/2775292.2778303.

[PG15b] E. Perakakis and G. Ghinea. "HTML5 Technologies for Effective Cross-Platform Interactive/Smart TV Advertising". In: *IEEE Trans. HMS* 45.4 (2015), pp. 534–539. DOI: 10.1109/THMS.2015.2401975.

[PSC12] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. "Comparison of cross-platform mobile development tools". In: *Proc. 16th ICIN*. IEEE, 2012, pp. 179–186. DOI: 10.1109/ICIN.2012.6376023.

[QG14] M. Quaresma and R. Gonçalves. "Usability analysis of smartphone applications for drivers". In: *Lecture Notes in Computer Science* 8517 (2014), pp. 352–362. DOI: 10.1007/978-3-319-07668-3{\textunderscore}34.

[QSM09] QSM. *Function Point Languages Table: Version 5.0*. 2009. URL: http://www.qsm.com/resources/function-point-languages-table (visited on 05/27/2016).

[res14] research2guidance. *Cross-Platform Tool Benchmarking 2014*. 2014. URL: http://research2guidance.com/product/cross-platform-tool-benchmarking-2014/.

[Rev16] Florent Revest. *AsteroidOS*. 2016. URL: http://asteroidos.org/ (visited on 05/31/2016).

[RM16] Christoph Rieger and Tim A. Majchrzak. "Weighted Evaluation Framework for Cross-Platform App Development Approaches". In: *Information Systems: Development, Research, Applications, Education: 9th SIGSAND/PLAIS EuroSymposium 2016, Gdansk, Poland, September 29, 2016, Proceedings*. Ed. by Stanislaw Wrycza. Cham: Springer International Publishing, 2016, pp. 18–39. DOI: 10.1007/978-3-319-46642-2_2.

[RP12] S. Rodriguez Garzon and M. Poguntke. "The personal adaptive in-car HMI: Integration of external applications for personalized use". In: *LNCS* 7138 (2012), pp. 35–46. DOI: 10.1007/978-3-642-28509-7{\textunderscore}5.

[Ryu+14] D. Ryu et al. "A serious game design for english education on Smart TV platform". In: *Pro. ISCE*. 2014. DOI: 10.1109/ISCE.2014.6884479.

[Sam16] Samsung Electronics Co. Ltd. *Let's TOAST - Samsung Smart TV Apps Developer Forum*. 2016. URL: https://www.samsungdforum.com/Features/TOAST (visited on 05/31/2016).

[SAW94] B. Schilit, N. Adams, and R. Want. "Context-Aware Computing Applications". In: *Proc. of the 1994 1st WMCSA*. IEEE CS, 1994, pp. 85–90.

[SK13] A. Sommer and S. Krusche. "Evaluation of cross-platform frameworks for mobile applications". In: *LNI* P-215 (2013).

[SKS14] R. N. Sansour, N. Kafri, and M. N. Sabha. "A survey on mobile multimedia application development frameworks". In: *Proc. ICMCS*. 2014. DOI: 10.1109/ICMCS.2014.6911207.

[Soh+15] H.-J. Sohn et al. "Quality evaluation criteria based on open source mobile HTML5 UI Framework for development of cross-platform". In: *IJSEIA* 9.6 (2015), pp. 1–12. DOI: 10.14257/ijseia.2015.9.6.01.

[Sta16] Statista Inc. *Statista*. 2016. URL: http://www.statista.com/ (visited on 05/31/2016).

[SV14] Stijn Schuermans and Michael Vakulenko. *Apps for connected cars? Your mileage may vary*. 2014. URL: http://www.visionmobile.com/product/apps-for-cars-mileage-may-vary/ (visited on 05/30/2016).

[Vis15] Visionmobile Ltd. *Cross-Platform Tools 2015*. 2015. URL: http://www.visionmobile.com/product/cross-platform-tools-2015/ (visited on 05/27/2016).

[Was10] Anthony I. Wasserman. "Software engineering issues for mobile application development". In: *Proc. FoSER '10*. Ed. by Gruia-Catalin Roman and Kevin Sullivan. 2010, p. 397. DOI: 10.1145/1882362.1882443.

[Wol13] F. Wolf. "Will vehicles go the mobile way?: Merits and challenges arising by car-apps". In: *Proc. 10th ICINCO*. Vol. 2. 2013.

[Wv16] Viveca Woods and Rob van der Meulen. *Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015*. 2016. URL: http://www.gartner.com/newsroom/id/3215217 (visited on 05/25/2016).

[WVN15] M. Willocx, J. Vossaert, and V. Naessens. "A Quantitative Assessment of Performance in Mobile App Development Tools". In: *Proc. 3rd Int. Conf. on Mobile Services*. 2015. DOI: 10.1109/MobServ.2015.68.

[XBM] XBMC Foundation. *Third-party forks and derivatives*. URL: http://kodi.wiki/view/Third-party_forks_and_derivatives.

[XX13] Spyros Xanthopoulos and Stelios Xinogalos. "A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications". In: *Proc. 6th BCI*. ACM, 2013, pp. 213–220. DOI: 10.1145/2490257.2490292.

[Zha+11] J. Zhang et al. "USink: Smartphone-based moible sink for wireless sensor networks". In: *Proc. CCNC'2011*. 2011. DOI: 10.1109/CCNC.2011.5766639.

24

Refining a Reference Architecture for Model-Driven Business Apps

Table 24.1: Fact sheet for publication P18

Title	Refining a Reference Architecture for Model-Driven Business Apps
Authors	Jan Ernsting[1] Christoph Rieger[1] Fabian Wrede[1] Tim A. Majchrzak[2] [1] *ERCIS, University of Münster, Münster, Germany* [2] *ERCIS, University of Agder, Kristiansand, Norway*
Publication Date	2016
Conference	12th International Conference on Web Information Systems and Technologies (WEBIST)
Copyright	SciTePress CC BY-NC-ND 4.0
Full Citation	Jan Ernsting, Christoph Rieger, Fabian Wrede, and Tim A. Majchrzak. "Refining a Reference Architecture for Model-Driven Business Apps". In: *12th International Conference on Web Information Systems and Technologies (WEBIST)*. Rome, Italy: SCITEPRESS, 2016, pp. 307–316. DOI: 10.5220/0005862103070316

Refining a Reference Architecture for Model-Driven Business Apps

Jan Ernsting Christoph Rieger Fabian Wrede Tim A. Majchrzak

Keywords: reference architecture, MDSD, app, mobile, mobile app, business app, architecture

Abstract: Despite much progress, cross-platform app development frameworks remain a topic of active research. While frameworks that yield native apps are particularly attractive, their spread is very limited. It is apparent that (theoretical) technological superiority needs to be accompanied with profound support for developers and adequate capabilities for maintaining the framework itself. We deem so called reference architectures to be a major step for building better cross-platform app development frameworks, particularly if they are based on techniques of model-driven software development (MDSD). In this paper, we describe a refinement of a reference architecture for business apps. We employ the model-driven cross-platform development framework MD² for this purpose. Its general design has been described extensively in the literature. The framework has a sound foundation in MDSD, yet lacks a generator support that fulfils the above sketched goals. After describing the required background, we argue in detail for a suitable reference architecture. While it will be a valuable addition to the MD² framework, the discussion of our findings also makes a contribution for generative app development in general.

24.1 Motivation

Cross-platform development frameworks for apps have gained much popularity [OT12; HTS12]. However, most of them employ Web technology and yield inferior results when striving for a *native look and feel* [JMK13; HHM13]. Creating cross-platform frameworks is a profound technological challenge (cf. [HMK13a]). At the same time, a framework needs to be comprehensible for developers. Moreover, it should provide adequate development support. The latter is particularly true when a framework seeks to improve the app development process in general (cf. [HKM15]).

In this paper, we build on previous work on MD², a cross-platform development framework that provides fully native apps for the supported target platforms. It employs techniques from model-driven software development (MDSD). Apps are rather modelled than programmed; for this purpose, a domain-specific language (DSL) is used, which has been tailored to business app development [HMK13b].

Despite its undoubted technological soundness, MD² has not yet found widespread adoption. This without question can be attributed to a missing community and user base (i.e. economies of scale). We believe that part of the reason lies in the complexity of generation [EEM16], though. Code generation might seem negligible from an app developer's point of view since capabilities are taken for granted when developing a specific app. However, improvements in the generation step are also reflected in development in form of extended capabilities and the possibility to benefit

from improvements to the DSL. We have, therefore, argued to overcome the existing challenges with profound work on the app generation step [EEM16].

With a more detailed look at the problem, additional questions need to be asked:

- Can a design pattern such as Model-View-Controller (MVC) or Model-View-ViewModel (MVVM) [Smi09] be implemented throughout the development process?
- How can the fragmentation of devices and the heterogeneity of software be overcome?
- How can frequent releases of platforms (such as Android and iOS), changes to the underlying programming languages (such as Apple's transitioning from Objective-C to Swift [App15]), and modifications to the ecosystems be effectively reflected in the generators?
- How can redundancy in a typical MVC description of apps be avoided?

Even though generators are seldom implemented from scratch and only updated occasionally, their development demands great skill. Code generators facilitate the embedding and have to account for two driving forces: a model instance on the one side and the target platform on the other. With this paper, we refine our previous work [EEM16]. This refinement builds on two shortcomings that we perceived:

- Currently, MD² focuses on object structure and behaviour, and not on interaction.
- The employed top-down approach (from model to reference architecture to platform-specificity) does not take into consideration platform-specific features in an effective fashion.

Therefore, this paper presents a more effective reference architecture that advanced from the feedback to realising a previously proposed reference architecture with two distinct platforms. Thereby, it greatly helped to fix the ecosystem under test except for the code generation stage. We were thus enabled in assessing and revising the reference architecture accordingly.

The main contributions of this paper are twofold. Firstly, we propose a detailed, sophisticated reference architecture for the model-driven creation of business apps. While it has been tailored to use in MD², it is applicable to MDSD for apps (as well as in Web technology-based frameworks) in general. Secondly, we discuss our findings. This further increases the generalizability of our work.

This paper is structured as follows. Section 24.2 draws the background of related work on MDSD for business app development and the corresponding use of reference architectures. Section 24.3 then presents an evaluation of the existing reference architecture to provide the foundation for the new proposal. This is presented as a detailed refinement in Section 24.4. Our findings are then discussion in Section 24.5. Finally, we draw a conclusion in Section 24.6.

24.2 Related Work

Apart from business apps and related work in the area of cross-platform development approaches, this section specifically highlights a concrete approach for business apps, namely MD². In addition, the general area of reference architectures is considered to base the refinements on.

24.2.1 Business Apps and App Development

Besides common apps for purposes such as social networking or entertainment, business apps encompass those that are directed at interacting with businesses' backend information systems. In addition, they can be categorised by their form-based and data-driven nature [MEK15].

For apps, in general, to overcome the heterogeneity of mobile platforms, various approaches exist. Their spectrum ranges from Web apps to native apps [MEK15]. While the former rely on Web technologies such as HTML5, Cascading Style Sheets (CSS), and JavaScript, the latter requires sufficient proficiency of the respective platforms, the applicable programming languages, and the respective software development kits (SDKs). Generative approaches aim at providing a native look and feel as well as native performance while striving to be as convenient to use as Web-based frameworks [HHM13]. There are two kinds of generative cross-platform frameworks: *transpilers* and *MDSD*-based approaches.

Transpilers allow compiling source, intermediate, or binary code for one platform to another. However, current transpilers such as J2ObjC [Goo15] deliberately focus on non-UI code. Therefore, they do not support a complete transformation and require at least some finishing touches to let apps run on targeted platforms. Consequently, they are neither widely used nor can be considered to be particularly mature.

Model-driven generative approaches employ the techniques of MDSD [SV06]. Typical examples for approaches encompass commercial products such as WebRatio [15b]. It utilises graphical models that are expressed in the Interaction Flow Modeling Language (IFML), which the backing company pushed to standardisation by the Object Management Group (OMG). Though restricted to displaying data, applause [15a] features a textual domain-specific language (DSL) to express models in. While applause's development was discontinued for more than a year and Xmob [LW13] has yet to present concrete modelling facilities, development on MD² progressed.

24.2.2 Creating Business Apps with MD²

MD² is based on a textual DSL that is structured along the Model-View-Controller (MVC) design pattern [Gam+95; Bus+96]. To overcome platform heterogeneity, its syntax focuses on describing business apps. Thus, consolidated abstractions are used to express archetypes in MD²-DSL [HMK13a]. This has obvious limitations, as featured elements correspond to the lowest common denominator of platform features. This is particularly true for view related aspects (cf. Section 24.4). On the other hand, users benefit from business apps automatically derived from archetypes without needing in-depth knowledge of target platforms. In consequence, users have to weigh the pros and cons of using a model-driven approach.

To mince matters, approaches can boost their usefulness by increasing the number of supported platforms. This is achieved by adding code generation facilities for desired platforms. An archetype passes through a preprocessor as shown in Figure 24.1. That stage augments the input archetype with concrete elements to reduce generation effort [ME15]. Next, the augmented archetype is

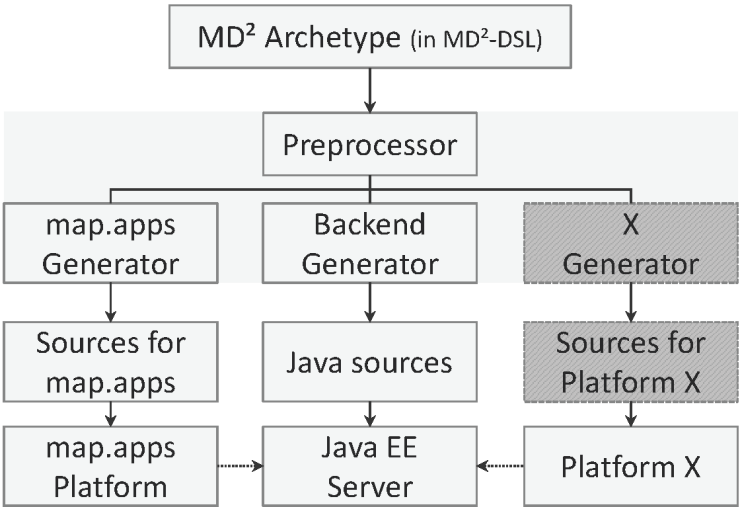


Figure 24.1: MD² Modelling Process

handed over to code generators. Currently, these output sources for the commercial map.apps platform [con15] and a Java EE based backend. Nevertheless, the modelling process incorporates provisions for adding generators such that these dispense applications for arbitrary platforms such as Windows Phone or Symbian.

Developing a code generator requires its engineers to comprehend the archetype's structure (i.e. the metamodel and its instances) as well as the targeted platform ecosystem (i.e. the programming environment, common libraries, etc.). Thus, engineers have to account for these two forces that drive generator development. In the following, assistive means that reduce the overall development effort are illustrated.

24.2.3 Reference Architectures

To understand the relevance of developing and further improving a reference architecture for MD² apps, knowledge of the potential benefits of reference architectures is advisable. This question was already subject of academic research; however, it was discussed in a broader context and not specifically tailored to model-driven software development and mobile apps in particular (cf. [Clo+10; AGG09]). Nonetheless, the findings can be transferred to this area.

First of all, as any other form of architectures, a reference architecture helps to control complexity [Clo+10]. The design of software is expressed in a standardized form such that it is easily understandable for developers. However, in the context of MD², a reference architecture main benefit consists in the preserved knowledge and guarantees a common understanding. The reference architecture is supposed to build a common ground on which the development of generators for new platforms takes place. Consequently, it has to embrace the knowledge and insights which were gained during the implementation of other generators and applications in order to provide these to its consuming developers.

A common understanding of MD² application components is necessary to keep the effort for development and maintenance of existing generators manageable. The architecture of the generators and applications should be similar to a certain degree. This facilitates a quick understanding of the concrete architecture on different platforms. Thus, developers who implemented a generator for a certain platform could perform maintenance tasks on other platform generators, as the underlying concepts are the same.

24.3 Evaluation of a Reference Architecture for Model-Driven Business Apps

Various best practices and established patterns for mobile platforms exist. Unsurprisingly, these typically neglect code generation as they do not distinguish between model-specific elements and components concerned with the general application execution.

However, in the context of model-driven approaches, the aforementioned code generation stage creates a tie between the model and its targeted platforms. Thus, the generation stage forms a crucial component of these approaches.

When considering MD²'s use of the MVC pattern in its DSL model, sufficient guidance with regard to engineering apt code generators could be expected. Yet, previous efforts showed that redundant or recurring ideas could not be removed by the MVC pattern itself nor did the tool's underlying metamodel offer substantial guidance in developing code generators. MD²'s constitutional development had focused on Android and iOS as target platforms. At that stage, its engineers noticed to some degree that numerous architectural choices arose for both platforms [EEM16].

Prior to targeting a third, previously unconsidered commercial platform, [EEM16] compiled a reference architecture for model-driven business apps. Establishing support for map.apps [con15] not only illustrated the reference architecture's applicability, but also showed that MD²'s scope could surpass the limits of mobile platforms.

The reference architecture explicates dependencies between and usages of MD²-DSL's metamodel elements. On a coarse level, the MVC pattern alone did not facilitate development of code generators in an intuitive fashion. Nevertheless, architectural elements can easily be partitioned to accentuate MD²'s MVC provenance as shown in Figure 24.2. To provide guidance for generator development, [EEM16] described architectural key elements with regard to their instantiation and runtime behaviour.

For example, concrete `EventHandler` account for different event types that exist in MD². These architectural elements assist in orchestrating an app's runtime behaviour. They determine which associated `Action` has to be executed in response to triggered events. Here, the architecture's explication guides code generators to provide facilities to handle global (e.g. context related)

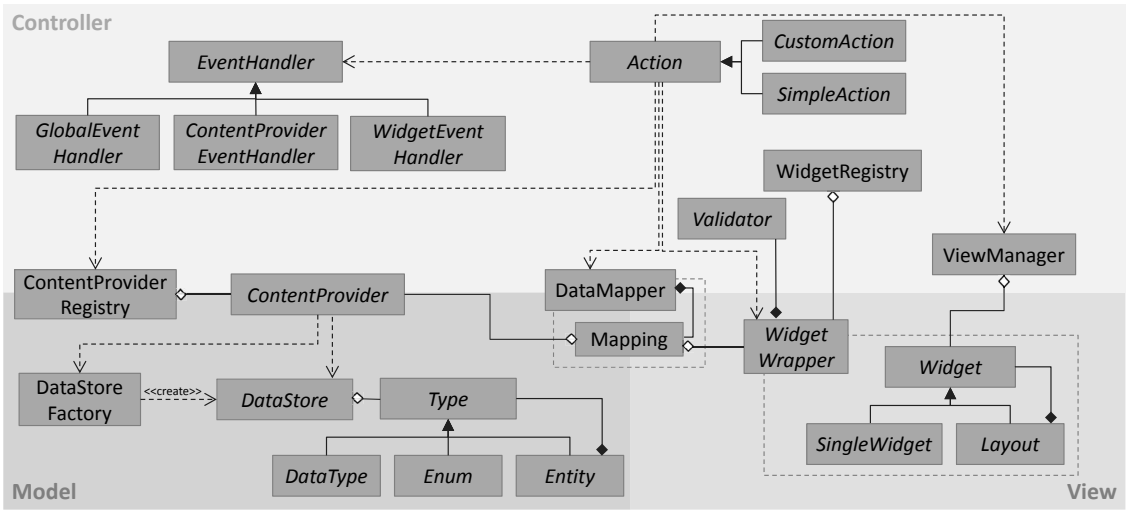


Figure 24.2: Original Reference Architecture [EEM16]

events such as connection lost, view related (i.e. widget) events, and data persistence (i.e. content provider) events in their respective generates (i.e. apps).

Of course, [EEM16] elaborated on the other elements in Figure 24.2. For now, we highlight their discussion in which they point out that little backing empiricism exists and that their architecture may require minor changes to be universally applicable to other platforms. Put straight, we strove to evaluate the architecture's suitability by targeting the two platforms currently dominating the mobile market: iOS and Android [Gar15].

The implementation was conducted as a greenfield approach, respecting the development process proposed by [SV06, p. 27]. Two independent developers, who were not involved in the development of the original reference architecture, implemented the reference apps and generators for the corresponding platforms, leveraging the original reference architecture as a guideline for the respective concrete architectures.

The evaluation of the reference architecture revealed that it was already quite well suited to provide guidance regarding the structural composition of required components. However, overall it was difficult to instantiate the reference architecture because of two main shortcomings: first, the reference architecture lacks to provide an overview of the runtime behaviour and the interaction between components and second, it chooses a misleading level of abstraction at some points, thus slackening the platform-specific development progression. Therefore, the reference architecture was revised and applied as presented in the following.

24.4 Revising the Reference Architecture

Because of the aforementioned shortcomings, structural aspects of the original reference architecture were improved and developers relieved with increased flexibility regarding the implementation in programming code as well as increased clarity towards interaction patterns.

24.4.1 Reference Architecture Structure

The revised reference architecture improves the structure three-fold: first, the progression of the MD² language introduced a process layer extension that is reflected in the current architecture version. Furthermore, the previously unspecified view layer of the application is substantiated and a mobile-centric task queue component is added.

Workflow Layer

Since the initial development of the reference architecture, the MD² language further evolved. Most prominently, an additional layer of so-called workflow elements was added to its specification, enabling development of cross-app workflows and app product lines [Dag+16]. This process-oriented layer is now also considered in the revised reference architecture. For a seamless integration, the current building blocks of events and actions were reused. To trigger a workflow event, an action called `SetWorkflowElementAction` extends the existing architecture. Newly added `WorkflowElement` objects represent self-contained process steps. Every workflow element is supplied with a workflow *event* to identify transition points within the overall process. Also, a workflow action indicates whether to start or end the specified workflow. Together, conditional workflow *paths* can be modelled.

The actual processing of the workflow elements is performed by the `WorkflowManager` component. For local workflow elements, this includes finalizing the currently active workflow element, deciding upon the further workflow path, and starting the next element. In case the workflow element has to be continued in another app, the intermediate state of the workflow and its associated entities is additionally sent to the backend server. The `WorkflowManager` component of the subsequent app is then notified about a newly available workflow instance and app users can eventually continue its execution.

View specification

The original version of the reference architecture did not consider the implementation of the view as a part of the architecture itself. It was claimed that the implementation of the view was too platform-specific and, therefore, could not be included. However, our evaluation of the reference architecture on iOS and Android showed that there are similarities regarding the structure of views. That is on both platforms, the view is defined as a hierarchy of layouts and widgets that can be arbitrarily nested. This pattern is also common on other platforms. Therefore, objects

representing the structure of the view were included in the reference architecture. In case that a platform handles the view differently, this part could easily be adapted when transforming the reference architecture into a platform-specific one.

The `WidgetWrapper` object was removed in the revised reference architecture. At this point, the original reference architecture stated that a widget should provide a certain set of methods. However, this was already done by applying a specific design pattern, which is not necessarily the best choice for the concrete implementation. On Android, for example, it turned out to be more convenient to use Custom Views instead of wrapper objects. Therefore, we claim that the application of concrete design patterns should be avoided in order not to push developers in a direction that might not be the best choice. Rather, the reference architecture should be more general and show options on how to develop the concrete implementation.

Tasks

In MD², custom actions consist of atomic tasks. These tasks perform operations such as binding the value of a widget to an entity. Tasks were not a part of the original reference architecture as they were supposed to be directly transformed into plain code by app generators. However, the evaluation showed that tasks can still be further generalized. Consequently, it is beneficial to add classes for the different tasks in the platform-specific libraries and to instantiate them with required parameters in the generated code. Tasks differ from actions in a way such that actions perform high-level, control flow-oriented operations, such as switching to another view, whereas tasks perform low-level, view-oriented operations, such as enabling data binding to a widget.

TaskQueue

In general, mobile applications should be designed in a resource efficient way as mobile devices typically provide limited resources compared to personal computers. The Android platform, for example, provides a built-in memory management that is enabled to free up memory allocated to paused activities so that it can be used by activities with a higher priority, e.g. active ones. Consequently, it is not guaranteed that widget objects exists all the time an app is running. However, for certain tasks, such as data binding, it is necessary to have access to a widget, e.g. to register event listeners. The first version of the reference architecture suggested overcoming that issue by creating `WidgetWrappers` on start-up of the application. As these `WidgetWrappers` keep a reference to the actual widget, the Android memory management does not destroy the objects so that they can be accessed if required during the execution of the app. However, this approach leads to memory leaks, i.e. objects that blocked memory and could not be deleted, because they were referenced by a running app. To avoid these leaks and to facilitate a more dynamic handling of application resources, the concept of the `TaskQueue` is introduced in the revised version of the reference architecture.

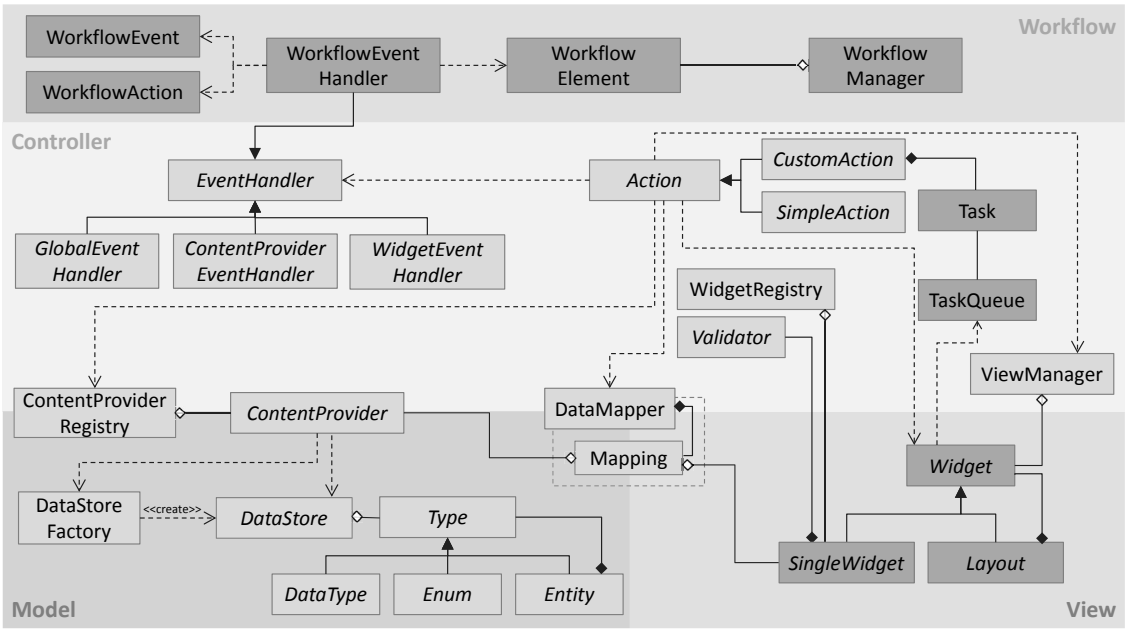


Figure 24.3: Revised Reference Architecture

The `TaskQueue` provides functionality to store tasks that cannot be executed because of required objects missing. As soon as the required objects are created, the execution is triggered again. An example from the Android platform is the transition between activities. When a new activity is started, all widget objects that belong to the activity's view are created. Therefore, it might be possible to execute tasks that could not be executed before. Thus, the transition between activities is one point in the flow of an Android app where the execution of formerly stored tasks should be triggered again.

24.4.2 Platform-Specific Implementation Variability

Possible alternatives for the implementation of the architecture in object-oriented programming languages include the approaches that are discussed in the following.

Status Quo

The original reference architecture was developed with a top-down approach starting from the MD² language definition. As a consequence, platform-specific characteristics were mainly treated as limitations that needed to be bypassed by additional generalised components. For example, the event handling mechanism is designed as explicit component because platforms not necessarily provide the possibility to extend their native event system.

As further contribution, the revised architecture incorporates the implementation learning in a bottom-up manner. Particularly, the main pain point of over-generalisation, resulting in tedious re-implementations of existing platform features, should be avoided. Mapping the reference

architecture to appropriate platform implementations is a problem to be solved by generator developers with knowledge of the target platform's characteristics and advanced language constructs. This choice of implementation allows for the necessary variability to avoid implementation overhead and benefit from available language features. [FMO11] discuss several techniques for object-oriented programming languages that are applicable for the MD² reference architecture.

Delegation

Glue interfaces bridge potentially incompatible code parts by specifying an interface towards the rest of the application [FR07]. As a basis, all components of the revised reference architecture can be regarded as such interfaces instead of explicit class requirements. Components that cannot easily be specified as one object on the target platform can use available aggregation or delegation mechanisms to perform the desired action as long as the interface towards the remaining objects is fulfilled. Developers can therefore benefit from unique platform features to reduce development time, increase runtime performance, or improve maintainability and readability of the resulting code.

For example, the implementation of data mappings depends on available platform features. Where possible, an efficient bidirectional map of model entity and view element is a good solution. Other implementations are likely suitable as long as the data mapper can be queried to look up the respective widget to an entity attribute, and vice versa.

Inheritance

Extending classes of the platform is a viable solution to reuse existing functionality contained in the platform libraries. Using class inheritance mechanisms, provided classes only need to be enriched with missing operations in order to comply with the reference architecture specification, again reducing implementation efforts.

Native widget extensions are a possible application of this approach: Platforms such as Android support the creation of custom widgets by sub-classing a generic widget class. Consequently, only additional functionalities for validation and value access need to be implemented manually. Alternatively, wrapper objects can be used that refer to native widget elements as fallbacks. As a second option, the native platform event system can be extended with custom events on some platforms; thus limiting implementation overhead. Where this is not possible, for instance on iOS, all event-related components of the reference architecture need to be implemented.

Overloading

Overloading methods or using generic classes is a further technique to flexibly implement the required methods of the reference architecture's component specifications. By choosing appropriate parameters, the desired functionality can be provided while leveraging the potential of the respective programming language concepts.

For instance, dynamically typed languages such as JavaScript might look up view element references using String objects. However, statically typed languages may benefit from enumeration types by providing additional compile-time security with regard to the existence of such references.

Decentralized processing

Several components of the reference architecture manage other components. However, it is deliberately unspecified whether a single object should perform all management activities or whether responsibilities can be shared by a distributed set of instances. For example, event handlers can be implemented as singleton objects that handle specific events on a global application level. On the other hand, the Android implementation initializes individual event handlers that use the observer pattern to directly capture changes of the widgets and content providers [Gam+95, p. 293].

As a result, the revised reference architecture combines the previous model-driven top-down approach with a platform-driven bottom-up perspective while at the same time encouraging implementation choices by generator developers.

24.4.3 Reference Architecture Interactions

To assist developers with respect to their implementation choices and expatiate on desired component interactions in the respective implementations, further behavioural aspects require clarification complementing the revised structure of the reference architecture. Particularly, three main interactions are essential for the application execution: basic widget control flows, process-oriented workflow control flows, and data flows.

Widget control flow

In MD², apps' business logic is modelled using MD² actions which, amongst others, can trigger view updates. Events and actions are created and mutually registered by the `Initializer` component which sets up all interacting objects on application launch and triggers the first event. From this point onwards, the event-action loop is the backbone of the application runtime. As depicted in Figure 24.4, widget changes are observed through respective gesture or value change events. Event handlers then execute the respective actions registered on start-up of the application. Depending on the state of the targeted view element, updates such as widget state changes or view transitions can directly be applied on visible elements. Update tasks on currently inactive elements, for instance data binding changes, are temporarily queued in the `TaskQueue` as described in Subsection 24.4.1.

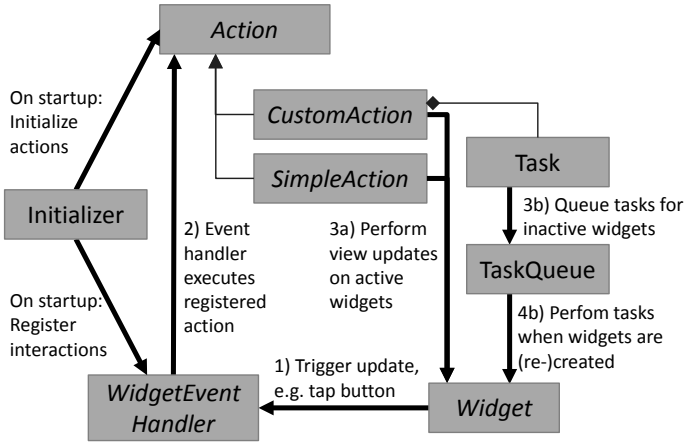


Figure 24.4: Interaction Diagram for Widget Control Flows

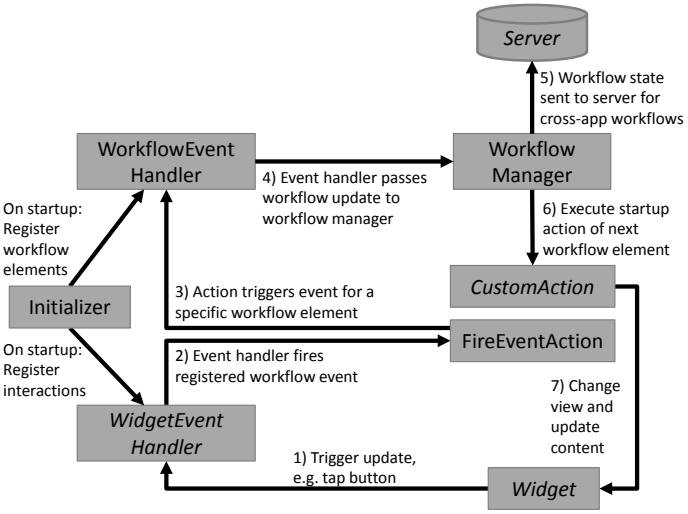


Figure 24.5: Interaction Diagram for Workflow Control Flows

Workflow control flow

In addition to this app-internal view update mechanism, the overall business process modelled as multiple workflow elements [Dag+16] also leverages the event-action loop concept. Instead of updating a view element, the widget event handler executes a `FireEventAction` that further triggers a workflow event (cf. Figure 24.5). The respective event is handled by the `WorkflowEventHandler` that notifies the `WorkflowManager` component to process the event as described in Subsection 24.4.1. When starting the next workflow element, its start-up action is executed which may contain model-specific initialization tasks. It also switches to the respective view and updates widget contents according to the regular widget control flow description.

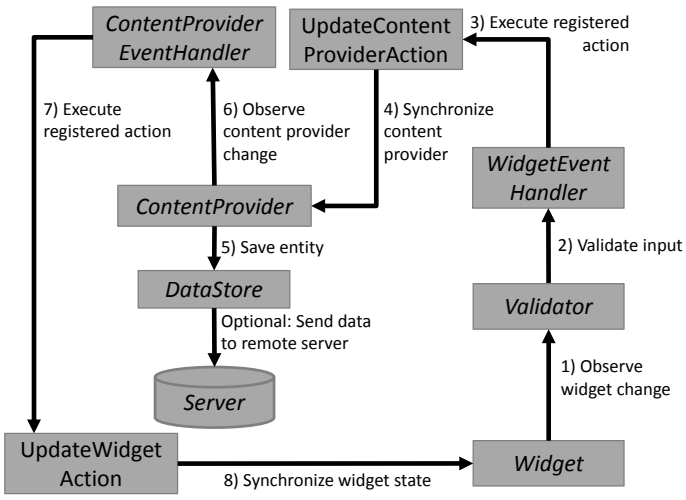


Figure 24.6: Interaction Diagram for Data Flows

Data flow

As MD² apps are data-driven, data flows within the application are a major element of interest. Starting from observed widget changes again, first input validation is applied to avoid unnecessary data flows. Next, event handlers execute the registered action which notifies the content provider of an update. This content provider manages the underlying entity and saves the field to either local storage or a remote location accessed by a data store. Finally, the content provider also emits an update event such that the respective event handler executes update actions for (potentially multiple) associated widgets. This loop, depicted in Figure 24.6, can also be started when an external data source changes and the data store notifies the content provider of updated content.

These interaction loops additionally guide the behavioural implementation of the reference architecture on new platforms without enforcing any technical implementation approach with regard to the event and synchronization techniques.

24.5 Discussion and Outlook

The revised reference architecture incorporates insights gained from the application of the original architecture in three distinct generator implementations. However, the focus on the MD² language still limits the empirical validation of the approach. On the one hand, the balance of component generalisation and flexibility of implementation derived from the evaluated generator implementations needs to be validated. On the other hand, the described interaction illustrations are also based on the currently available generator implementations. In retrospect, this information would have provided important guidance for the understanding of the reference architecture. Still, for both aspects empirical validation can only be achieved by applying the refined reference architecture to another platform *without* prior knowledge of the subject matter.

While the interaction patterns do not impose any restrictions on the actual implementation, there might also be platforms for which the basic event-action loop is not well applicable. With additional communication emerging from cross-app workflow coordination, it should be considered to further specify the behaviour of the existing external interfaces as guidance for generator developers. For instance, data exchange formats on mobile platforms should be assessed to provide further standardisation. Also, different communication mechanisms such as asynchronous backend requests or servers pushing messages to apps may be considered with regard to user experience improvements. Incorporating such changes into the reference architecture may result in changes to the interaction mechanisms that were presented in this paper.

Finally, the derived best practices need to be reapplied to the generator implementations fostering a maintainable code base following common architectural design decisions. The revised reference architecture already incorporates changes resulting from the MD² language evolution as for instance the workflow layer extension integrates nicely into the existing architecture. Yet, it has to be shown whether the current reference architecture is flexible enough to adapt to future DSL language changes.

Despite some minor drawbacks, the evaluation showed that reference architectures can serve as supportive means for extending model-driven approaches such as MD². In addition, these limitations were leveraged to assist in revisiting and applying these newly gained insights as presented in Section 24.4.

24.6 Conclusion

In this paper, we have presented work on the refinement of a reference architecture for MD². It extends the model-driven cross-platform framework with means to provide unified, maintainable, and scalable code generation. Thereby, it ultimately also contributes to the framework's ease-of-development.

Based on the study of related work and the first suggestion by [EEM16], we proposed steps for the refinement. Despite some minor drawbacks, the evaluation showed that reference architectures can serve as supportive means for extending model-driven approaches such as MD². These limitations were leveraged to assist in revisiting and applying the newly gained insights. The actual work consists of detailed suggestions for the structure of the reference architecture, ways to address platform-specificity while keeping an abstract interface, and suggested guidelines for component interactions.

There is a fine line between being too specific and too general (or, rather, abstract) with regard to reference architectures. We are confident that we have found a balanced approach with the proposals made in this paper. However, the feasibility of our ideas will need to be proven empirically – first qualitatively and ultimately quantitatively. In the meantime, we will keep up our work and also seek to contribute to the core of the MD² framework including its domain-specific language. Currently, an alternative, graphical modelling front-end that utilises

the revised reference architecture and generators is being designed to assess its accessibility to modellers; thus, making a case in favour of reusable and maintainable generation facilities. We hope that MD² will get more attention by industrial users in the future, probably even stimulating work on complementary or competing approaches.

References

[15a] *applause*. https://github.com/applause/. 2015.

[15b] *WebRatio*. http://www.webratio.com/. 2015.

[AGG09] S. Angelov, P. Grefen, and D. Greefhorst. "A classification of software reference architectures: Analyzing their success and effectiveness". In: *European Conf. on Software Architecture (ECSA)*. 2009, pp. 141–150. DOI: 10.1109/WICSA.2009.5290800.

[App15] Apple Inc. *Swift Blog - Apple Developer*. https://developer.apple.com/swift/blog/. 2015.

[Bus+96] Frank Buschmann et al. *Pattern-Oriented Software Architecture*. Wiley, 1996.

[Clo+10] Robert Cloutier et al. "The Concept of Reference Architectures". In: *Systems Engineering* 13.1 (2010), pp. 14–27.

[con15] con terra. *map.apps product description*. 2015. URL: http://conterra.de/en/produkte/con-terra-solutionplatform/mapapps/beschreibung.aspx.

[Dag+16] Jan C. Dageförde et al. "Generating App Product Lines in a Model-Driven Cross-Platform Development Approach". In: *49th Hawaii International Conference on System Sciences (HICSS)*. 2016.

[EEM16] Sören Evers, Jan Ernsting, and Tim A. Majchrzak. "Towards a Reference Architecture for Model-Driven Business Apps". In: *49th Hawaii International Conference on System Sciences (HICSS)*. 2016.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *12th International Conference on Web Information Systems and Technologies (WEBIST)*. Rome, Italy: SCITEPRESS, 2016, pp. 307–316. DOI: 10.5220/0005862103070316.

[FMO11] Fazal-e-Amin, Ahmad Kamil Mahmood, and Alan Oxley. "An analysis of object oriented variability implementation mechanisms". In: *ACM SIGSOFT Software Engineering Notes* 36.1 (2011), p. 1.

[FR07] Robert France and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap". In: *Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54. (Visited on 06/09/2014).

[Gam+95] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1995.

[Gar15] Gartner. *Gartner Press Release*. http://www.gartner.com/newsroom/id/3169417. 2015.

[Goo15] Google Inc. *J2ObjC*. 2015. URL: %7Bhttp://j2objc.org/%7D.

[HHM13] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. "Evaluating Cross-Platform Development Approaches for Mobile Applications". In: *Revised Selected Papers WEBIST 2012*. Vol. 140. LNBIP. Springer, 2013, pp. 120–138.

[HKM15] Henning Heitkötter, Herbert Kuchen, and Tim A. Majchrzak. "Extending a Model-Driven Cross-Platform Development Approach for Business Apps". In: *Science of Computer Programming (SCP)* 97.1 (2015), pp. 31–36.

[HMK13a] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. "Cross-Platform Model-Driven Development of Mobile Applications with MD^2". In: *Proc. of the 28th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2013, pp. 526–533.

[HMK13b] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. "MD^2-DSL - eine domänenspezifische Sprache zur Beschreibung und Generierung mobiler Anwendungen". In: *6. Arbeitstagung Programmiersprachen (ATPS)*. Vol. 215. LNI. Gesellschaft für Informatik e.V. (GI), 2013, pp. 91–106.

[HTS12] Andreas Holzinger, Peter Treitler, and Wolfgang Slany. "Making Apps Useable on Multiple Different Mobile Platforms: On Interoperability for Business Application Development on Smartphones". In: *Multidisciplinary Research and Practice for Information Systems*. Vol. 7465. LNCS. Springer, 2012, pp. 176–189.

[JMK13] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. "Real Challenges in Mobile App Development". In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 2013, pp. 15–24.

[LW13] Olivier Le Goaer and Sacha Waltham. "Yet another DSL for cross-platforms mobile development". In: *Proc. of the First Workshop on the Globalization of Domain Specific Languages*. ACM. 2013, pp. 28–33.

[ME15] Tim A. Majchrzak and Jan Ernsting. "Reengineering an Approach to Model-Driven Development of Business Apps". In: *8th SIGSAND/PLAIS EuroSymposium*. 2015, pp. 15–31.

[MEK15] Tim A. Majchrzak, Jan Ernsting, and Herbert Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *Open Journal of Information Systems (OJIS)* 2.2 (2015), pp. 3–14.

[OT12] Julian Ohrt and Volker Turau. "Cross-Platform Development Tools for Smartphone Applications". In: *IEEE Computer* 45.9 (2012), pp. 72–79.

[Smi09] Josh Smith. *Patterns – WPF Apps With The Model-View-ViewModel Design Pattern*. https://msdn.microsoft.com/en-us/magazine/dd419663.aspx. 2009.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

25

How Cross-Platform Technology Can Facilitate Easier Creation of Business Apps

Table 25.1: Fact sheet for publication P19

Title	How Cross-Platform Technology Can Facilitate Easier Creation of Business Apps
Authors	Tim A. Majchrzak[1] Jan C. Dageförde[2] Jan Ernsting[2] Christoph Rieger[2] Tobias Reischmann[2] [1] *ERCIS, University of Agder, Kristiansand, Norway* [2] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2017
Publication Outlet	Apps Management and E-Commerce Transactions in Real-Time
Copyright	IGI Global
Full Citation	Tim A. Majchrzak, Jan C. Dageförde, Jan Ernsting, Christoph Rieger, and Tobias Reischmann. "How Cross-Platform Technology Can Facilitate Easier Creation of Business Apps". In: *Apps Management and E-Commerce Transactions in Real-Time*. Ed. by In Lee and Sajad Rezaei. Advances in E-Business Research. IGI Global, 2017, pp. 104–140. DOI: 10.4018/978-1-5225-2449-6.ch005

How Cross-Platform Technology Can Facilitate Easier Creation of Business Apps

Tim A. Majchrzak Jan C. Dageförde Jan Ernsting Christoph Rieger Tobias Reischmann

Keywords: Mobile, App, Mobile App, Business App, Cross-Platform, Multi Platform, App Development

Abstract: Applications for mobile devices – apps – have seen unprecedented growth in importance. Ever better apps keep propelling the proliferation of mobile computing. App development is rather easy, particularly if it is based on Web technology. However, implementing apps that are user friendly and useful in the long-run is cumbersome. Thereby, it typically is expensive for corporate developers. Nonetheless, business apps are embraced by enterprises. To overcome the overhead of developing separately for multiple platforms and to mitigate the problems of device fragmentation, cross-platform development approaches are employed. While many such approaches exist, few have found widespread usage. In this chapter, we argue what the path towards future solution could look like. We thereby take a rather technological look, but always keep a business-focus in mind. Our findings suggest that much effort is needed to enable the next generations of business apps. However, such apps will provide many merits and possibilities. Moreover, they provide the chance to master several of today's challenges.

25.1 Introduction

Applications for mobile devices – apps – have seen an unprecedented growth in importance over the course of the last years [RM14], as an increasing number of companies embrace mobile computing [MH14]. They employ the capabilities and versatility of apps to support business processes as well as to provide tools to connect with (existing and future) customers [McL14]. Even if technological progress was stalled in terms of device hardware, ever better apps would likely keep propelling the proliferation of mobile computing. App *development*, however, is cumbersome considering the multitude of mobile platforms that need to be supported [HMK13].

Developing apps is, in general, facilitated by rather easy to use software development kits (SDKs) released by the mobile platform vendors (such as Apple for iOS [Appc] and Google for Android [Goo16a]). *Platform* in this context refers to the operation system for a mobile device along with the accompanying ecosystem of SDK, frameworks, tools, means of distribution etc. Moreover, Web-based development is widely used: Either Web apps are chosen as a kind of least common denominator or, alternatively, cross-platform tools such Apache Cordova [Apa] are employed. In addition, numerous frameworks exist that intend supporting development activities. Nonetheless, creating good apps is both complex and expensive.

It can be witnessed that app development even in companies with experience in software development differs from traditional development activities [MH14]. Team sizes are often smaller – frequently, app development is a single-person project [Hei+12]. While developing an app in general requires just a few steps, implementing sound concepts requires knowledge, experience,

and arduous work. Unsurprisingly, there is an endless stream of complaints about low app quality in app stores (cf. e.g. [Kha+15]) as well as a daily emergence of security-related news, typically reporting breaches (cf. e.g. [CYA12; Man12; Wan+13]). Even without serious flaws, it is hard for companies to leverage the potential of mobile computing if development is more expensive than it should be.

Despite the problems, companies virtually cannot avoid apps. Investigating the possibilities provided by modern mobile computing ought to be a task every company should pursue, almost independent of size, sector, and corporate culture [MH14]. For many enterprises, so called business apps will provide benefits, or at least promise to facilitate future improvement to process management, workflows, and customer relations. Business apps are typically based on the standardized GUI elements (i.e. essentially provide forms) and are data-driven (i.e. receive inputs, process them, and provide some information or request more input) [MEK15]. Moreover, they typically make use of a backend system [MEK15; MH14], which contrasts many consumer apps, which entirely operate on the mobile device.

Particularly business apps should provide support for multiple platforms to include all potential users. Even though Android and iOS have divided most of the market share of mobile devices platforms amongst themselves [WM16], typically more platforms should be supported. This can be explained by the fact that additional platforms still have millions of users who would otherwise be excluded from a company's services [WM16]. In addition, future areas such as operating systems used in cars (see e.g. [Goo16b]) might shift market shares of established platforms, introduce new ones, or – possibly worse – lead to partly incompatible derivatives (or *forks*) of existing platforms. Even with the two initially mentioned platforms alone, development effort is almost doubled compared to supporting just one of them.

Moreover, it would be beneficial if existing business backend systems could be used. Ideally, established workflows should be included; the re-usage of business process models would be excellent [EEM16]. Besides that, it needs to be assessed whether business apps can partially or even fully be created by domain experts rather than by developers [HM13]. With their relative simplicity on the one hand and business purpose on the other hand, it should be possible for people without development experience to create business apps. Here, *should* can be read in the ways of both *ought to* and *considered to be*.

The above described targets have led to numerous developments (see also Section 25.2). By now, many cross-platform app development frameworks exist, even though few of them have found widespread usage. Frameworks and techniques for business integration have been developed, even though again their applications are limited. Finally, methods and tools have been described that enable app development for non-developers, e.g. as learning tools. However, a cross-platform app development solution with tight business integration and accessibility for people without considerable development experience does not exist.

In this chapter, we argue what the path towards such a solution can look like. For this purpose, we assess the potential merits, what current and future obstacles look like, and which limitations

are likely to remain. We will start with a look at the past, scrutinize the status quo, and provide an elaborated outlook.

The chapter will be based on our own work on a cross-platform business app framework (cf. [HMK13]) but goes much beyond the technological and business process scope that existing articles have taken. Our work is structured as follows, honouring the chronological approach sketched above:

In Section 25.2 we first motivate the topic economically and give a brief look at the past. We then introduce existing approaches, both technology-oriented and domain-oriented. The aim of this section is particularly to meet the readers and to introduce them to the technological idiosyncrasies and ramifications that the topics of business apps and app-based commerce pose.

We then study related work in Section 25.3. This is followed by an extensive study of what cross-platform technology is capable of already in Section 25.4. Our synopsis is not only based on theory but practical findings as well as experiences from designing a cross-platform development framework. Moreover, we will highlight existing research activities that are attractive from a business point of view, yet have failed to find wider adoption in usage by companies. This includes a stronger business process focus of development tools and frameworks as well as the integration of graphical modelling with the creation of apps.

The assessment of the status quo leads to our vision of the future as laid out in Section 25.5. We will describe required technology and the app commerce possibilities that could be provided by advanced solutions. In Section 25.6 we then present a discussion of our findings. This is started with the presentation of usage scenarios. An outlook follows along with a discussion of open questions, both leading to an agenda for research. Eventually, we name limitations of our work and sketch our own goals for the near future. Finally, we conclude in Section 25.7.

25.2 Status Quo of Cross-Platform App Development

To lay out the foundation for this chapter, we introduce the basics of cross-platform app development. For this purpose, we first sketch the economic importance of business apps in particular and draw a brief history of app development in general. We then introduce a categorization of the possibilities for developing apps that will run on more than one platform. Eventually, we give a short overview of currently employed frameworks and tools in the context of cross-platform app development.

25.2.1 On the Economic Value of App Development

After years of immense growth (cf. e.g. [GP12; RM14]), the market for smartphones seems to be saturated [WM16]. However, there is no reason to believe that this saturation will also affect the development of apps. In fact, it is even too early to determine whether growth of the smartphone market will not resume. In the meantime, existing markets for devices that run apps will likely

keep growing. This not only considers established mobile devices such as tablets but also novel fields. Examples are wearables [CMK14], which increasingly are customizable by the means of apps, and cars (cf. e.g. [Goo16b]).

Reasons for expecting the further proliferation of apps are manifold. Many businesses have only slowly embraced the possibilities and continue to explore [MH14]. New possibilities due to hardware improvements have not been fully leveraged; particularly sensor usage is poised to gain much more momentum (see e.g. [FNP15; GR14]). And there are many more ideas for sector- and domain-specific usage, e.g. in the field of healthcare [Mar+14]. With Facebook already reporting 76% of its advertising revenue being generated on mobile devices [Fac15] and mobile spam becoming *reasonable* (from a spammers point of view) [Sen+15], there are no signs of a saturation in terms of app development activities.

Unsurprisingly, *m-commerce* has been coined as a term that describes revenue creation in the mobile world. m-commerce thereby is an expansion, or rather a subcategory of the even longer discussed e-commerce. Despite the blurriness of such terms – particularly considering multi-channel approaches, which might make it hard to determine if commerce is mobile, online, or offline – they have impact. A study that set out to provide a taxonomy for m-commerce [KZG12] included an analysis of 2,300 patents related to m-commerce. It might well be the case that some countries allow filing software patents with questionably low innovation height; nonetheless, this is a profound indicator for the topic's perceived importance by the industry.

The research dimension is rather limited until now. Myriads of papers exist that shed light on distinctive aspects. Very popular for analysis are for example the app stores (cf. e.g. [Al-+15; Fu+13; Kim10]). However, a comprehensive, holistic view of the importance of enterprise mobile computing as well as m-commerce is missing.

Some app store related papers provide findings concerning the economic importance of apps. For example, a study of Kang, Mun, and Johnson [KMJ15] has affirmed positive effect of mobile apps on shopping behaviour. Similarly, a study by Google [Goo13] found that 90% of smartphone shoppers use their phone for pre-shopping activities. 84% of smartphone shoppers use their devices to help shop while in a store. It might be suspected that this is economically objectionable, e.g. because shopper use stores to try out product but buy them from the cheapest online vendor. The study suggests, however, that shoppers who use mobile more, buy more. Even though such findings have to be taken with care (in this case, an increase of the median of a shopping basket size was observed), they provide hints. Obviously, businesses cannot neglect a mobile strategy.

Existing overview articles are limited in scope but can give important hints. A Delphi study with "14 leading mobile commerce scholars" [Pou+15] yielded several interesting observations:

- *Mobile enterprises* are expected to proliferate.
- Mobile services and applications are no new emergence but have changed in the course of time.
- m-commerce is seen as an important part of the ongoing *digitalization* of societies.

- Integration of mobile into the overall business is seen to be a driven topic of the near future.

At the same time, it was acknowledged that a theoretical foundation is missing. This finding is not only an indicator for the rapid involvement of the field but also a motivation for work like the one provided in this chapter.

Summing up, there is still much theoretical work required to provide a better understanding of the economic importance of mobile computing. In particular, better advice for companies is required. Nevertheless, the already high and still increasing impact of m-commerce is unquestionable.

25.2.2 A Brief History of App Development

Mobile apps in general are not an entirely new emergence. Provisioning apps for so called *feature phones* or personal digital assistants (PDAs) turned out to be a major issue in the pre-app store era: Inconveniently, mobile apps had to be installed by their end users through desktop PCs that utilized software and hardware dedicated to installing applications on mobile devices.[1] In addition, the absence of economic data plans restricted mobile app use cases to a few scenarios that required little to no connectivity [WM10].

These devices offered a modicum of features; in fact, they often were mobile telephones with very limited computing ability. Seldom did they include localization sensors, i.e. GPS, or varied connectivity means, e.g. WiFi, Bluetooth, NFC; and if they did, these features where typically reserved to *handheld* PCs and PDAs, targeting resourceful – and typically professional – end user segments.

From an app developer's point of view, the dissemination of platforms required them to deal with vendors' proprietary development environments. As a consequence, enterprises utilizing mobile devices resorted to purchasing from specific vendors in order to reduce platform diversity. Sun Microsystems' mobile edition of Java (J2ME) [Ora11] promised to overcome this heterogeneity, but failed to honour this. Despite its intention, concrete runtime implementations differed across device vendors, thus limiting a wider adoption through incompatibilities [GHG10].

In 2007, the mobile game changed when Apple introduced the iPhone [Mac07]. At first, the iPhone provided few substantial benefits over previous mobiles [WM10, p. 275]. In fact, its introduction lacked the concept of apps as we know it; third parties were meant to run their applications as Web sites.

The same year, Google along with the Open Handset Alliance announced the Android platform [Ope07]. It provided access to sensors such as GPS and accelerometers by default and facilitated their utilization as part of its software development kit (SDK). Consequently, hardware vendors embrace the new possibilities.

Apple followed and published its SDK for iOS along with the App Store concept in 2008 [Fle07]. Apps are exclusively exposed and centrally provisioned through the App Store, which

[1] Actually, some distribution alternatives existed quite early, but their usage was inconvenient [Kim+13]

also supports billing of apps and purchases within apps [WM10, p. 280]. For Android, the Android Market [Goo16c] (nowadays known as the Google Play Store) provides the same functionality.

Other platform vendors such as Microsoft (Windows Phone) and Research in Motion (Blackberry OS) also pursue mobile ecosystems. By mid-2015, however, their cumulative sales diminished to less than five percent [IDC16]. To demonstrate the challenges of targeting heterogeneous platform, it suffices to focus on iOS and Android – we do so in the following. It must nonetheless not be assumed that two platforms are the fixed goal to reach customers. In fact, the above named competitors still sell millions of devices [WM16]. Moreover, the already mentioned future trends with more and more devices being *app-enabled* might lead to shifts in shares.

In parallel, mobile network operators upgraded their networks (from 2G/EDGE over 3G/UMTS to 4G/LTE) enabling bandwidth-intense contents such as video streaming as well as improving their availability [BK11]. They also pitched data plans making mobile data usage affordable, thus benefiting competition among operators. However, mobile bandwidth has not yet become a ubiquitous resource: Availability as well as bandwidth remain limited in rural areas and developing countries. Along with the *device fragmentation* (cf. [IDC16; HKM15]) that particularly impedes development for Android, particularly feature-rich apps need to be developed in a way that keeps them flexible.

25.2.3 General Approaches Towards Cross-Platform Development

Despite Apple's and Google's efforts in making mobile app development more accessible, their respective platforms differ substantially, e.g. with regard to app development but also with regard to user experience. Consequently, developing for mobile platforms requires versatile knowledge of their particular ecosystems, such as development environment, software development kits, and human computer interaction guidelines [JMK13].

In the following, we refer to native development as utilizing a platform's abstractions and SDK along with its provided programming language(s). Native development, in this context, does not necessarily mean development on a low-level, e.g. writing C code, using Application Binary Interfaces (ABIs), nor directly interacting with physical interfaces.

Research indicates an increasing fragmentation of mobile devices and platforms; however, app development endeavours fail to transfer their knowledge [JMK13]. In addition, mobile devices' market shares entail the support of at least two platforms [IDC16]. Typically, projects take place per platform, i.e. employ *native, platform-specific* SDK. Each of these projects require capable personnel, well-versed on the respective platform, and also necessitate additional testing to ensure consistent functionality across these platforms. Due to the nature of development, only analysis requirements engineering partly overlap, leading to an almost linear increase in development effort with the number of supported platforms [HMK13].

Various approaches that seek to overcome this platform heterogeneity exist. So-called cross-platform development exposes varying degrees with regard to platform peculiarities it supports.

Figure 25.1 shows a classification of approaches, based on their nativeness, i.e. how apps are developed, how they look and feel, and how they are executed.

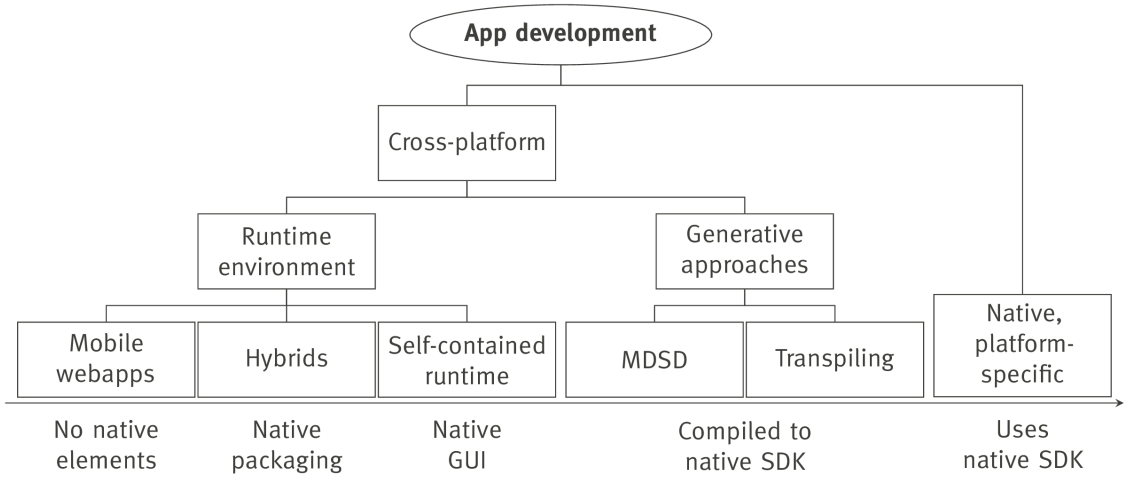


Figure 25.1: Classification of Cross-Platform Development Approaches [MEK15]

Approaches from the broader category of runtime environments require a dedicated runtime that has to be available for the targeted platforms and interprets an app's description. *Mobile webapps* feature no native elements and are displayed in a Web browser, i.e. utilize a browser engine. They are provisioned, specifically hosted, as any Web site; thus, require no app store interactions.

Hybrids are similar to webapps in that they are typically developed using Web technologies but grant access to features not available through these technologies, i.e. hybrid approaches often provide interfaces to use platform-specific features and run apps in a modified browser engine. These approaches offer mechanisms to mimic platform-specific UI elements but do not offer the same user experience native widgets do. Bundling as a native app enables provisioning of hybrids through app stores.

Approaches that are based on a *self-contained runtime* provide their own SDKs, instead of relying on Web technologies. These SDKs often bridge gaps between platforms and grant developers access to platform features and use native UI elements. As with hybrids, self-contained runtimes have to be available per platform. These apps are natively bundled along with the runtime engine that the app runs in, which might pose an overhead [HHM12].

Generative approaches forgo runtime environments and utilize native SDKs and the corresponding programming language. Development is either done in the programming language of *one* of the supported platform, or in a separate language. The latter typically is a so called domain-specific language (DSL). DSLs are tailored to a specific purpose and – usually no general-purpose programming languages.

The category of generative approaches distinguishes model-driven software development (*MDSD*) and *transpilers*. The former transforms an app model into source code that executes on

the targeted platforms. App modelling can be done in a textual modelling language, which might be not too different from a general programming language, or graphically. The latter translate source code for one SDK into source code for another SDK. This possibly includes translation into a different programming language. Transpilers not necessarily need to cover all aspects of an app; it is possible to only translate certain *layers*, such as the business logic.

Thus, developers face a major decision. Either, they write native, platform-specific code using the respective SDK and accompanying tools or they leverage a cross-platform development approach. A combination of both will be a waste of resources save for particular circumstances, e.g. optimization for one platform while providing general support for many others. Even with progress in Web technology such as HTML5 [W3C14] and widely popular runtime-based cross-platform approaches (see next subsection), native development is the richest and most flexible approach. However, it requires more resources (knowledgeable personnel, time (for testing), etc.) [JMK13]. Moreover, it might be less suited for future developments since changes to each supported platform will need to be addressed when developing new versions of an app.

25.2.4 Overview of Current Frameworks and Tools

In the following, a subset of current frameworks – one for each category – is presented. A thorough overview would be outdated swiftly and is difficult to accommodate in a single place. Nevertheless, an evaluation scheme supports decision makers in assessing frameworks' suitability for their purposes and choosing the fittest [Hei+13].

Native, platform-specific development relies on the respective SDKs. For example, when developing for iPhones and iPads the iOS SDK is used in conjunction with the Objective-C and/or Swift programming language. Android development typically employs Android SDK using the programming language Java.

The troika of HTML5, Cascading Style Sheets (CSS), and JavaScript resemble the foundation for *mobile webapps*. While HTML5 and CSS deal with the presentation of content, JavaScript enables webapps to expose interactive elements, e.g. update contents from a remote service on the fly. Apache Cordova [Apa] (a.k.a. Adobe PhoneGap) leverages HTML5, CSS, and JavaScript to create *hybrids* that are provisioned as native apps. In addition, Cordova fills gaps where Web standards are lacking to access platform specific features, e.g. access user's contacts.

Whereas the AppCelerator Platform (formerly Titanium Appcelerator) [Appa] *self-contained runtime* interprets JavaScript, it utilizes native UI elements, i.e. native widgets from the respective platform SDKs, through its custom abstraction layer. If need be, developers may directly employ platform-specific UI elements or concepts from iOS, Android, or Windows. Due to its interpretative approach, loading of apps takes longer.

Applause [Appb], in its second version, is a prototypical, *generative approach* that utilises MDSD techniques to generate native apps from code expressed in Applause's domain specific language (DSL). In addition, it provides facilities to express presentation-unrelated business logic

in JavaScript. We will learn about another MDSD approach in Section 25.4. *Transpiling* as provided by J2ObjC enables sharing of non-UI code, i.e. data models and application logic. It translates Java source to iOS compatible Objective-C code. However, even J2ObjC as the only tool with a relevant user base, is not capable of full translation. In particular, the code for the graphical user interface (GUI) *cannot* be converted. It would, however, be this code that would be most appealing for automated conversion.

25.3 Related Work

The work presented in this chapter is literature-driven by its nature. Thus, no generic study of related work is mandated. This particularly takes in mind that on the one hand, similar articles are mentioned in Subsection 25.2.1. And, on the other hand, no articles that are very close in presentation to this chapter exist. However, in the following we will highlight articles that relate in so far as they assess the status quo of cross-platform app development or take a look at the future of app development.

Several papers have discussed general ideas of native development in comparison to other approaches [HHM12; HKM15; ME15; OT12]. Due to the rapid progress of both hardware and platform development, these articles naturally have taken a look into the future. The status quo described is a mere snapshot and the ideas proposed a peak of what might be. It is needless to say that bold propositions have hardly been made in these scientific papers, since it would have been very hard to assess them. In general, the earlier papers faced an even more dynamic nature than we now have and thereby the author had an even harder time to make educated predictions.

Marinho and Resende [MR15] have taken a look at "native and multiple targeted mobile applications". They have compiled "positive and negative aspects" and explicitly target "decision-makers". Thereby, their work is complementary to ours.

There are a number of articles on cross-platform development approaches that also make proposals about the future of cross-platform app development. While typically a focus on business apps (or, actually, a domain at all) is missing, the ideas presented in such work are notable.

Mobl is a DSL for mobile Web applications [HV11]. Even though it targets Web browsers that are inherently cross-platform, their DSL is worth further consideration. In fact, by relying on browsers while still proposing the usage of a DSL, the approach brings together several existing ideas in a novel fashion. Mobl succeeds WebDSL [Weba] in that it exposes a versatile approach to user interactions and extending on a library model [HV11, p. 710f.]. It dispenses the typical controller component in favour of automatic inference. Mobl's data model and view descriptions can be annotated to augment the inferred controller [HV11, p. 696.]. Thereby, it positions itself as one integrated language (rather than using HTML5, CSS, and JavaScript as in classical webapp development). *MobiDSL* [Kej09] is similar to Mobl, but restricted with regard to custom controls or IDE support.

RAPPT [Bar+15; BVG15] is a *bootstrapping* tool for mobile app developers. It implements ideas that are similar to frameworks for rapid JavaScript development. Three different views are taken in RAPPT:

- A *domain-specific visual language* (DSVL), i.e. a graphical DSL, is used.
- In addition, developers can employ an *app modelling language* (AML) as well as a *domain-specific textual language* (DSTL).
- Additionally, raw source code can be written. The three-fold approach is novel. The author "made a conscious decision not to support round trip engineering. The primary reason for this is that the generated code is the scaffolding for a new project and is intended to be heavily modified" [Bar+15, p. 1]. Their approach keeps DSVL and DSTL in sync. However, RAPPT targets Android only. So far, no plans to extend support to other platforms exist. Thus, the approach can rather be seen as an attempt to easy development in general and to bridge different development philosophies. The authors' claim to produce modifiable code will need to be assessed.

The cross-platform development framework AXIOM [JJ15] provides a multi-stage transformation (structural and with regard to style). Generators and transformations have to be specified by developers. AXIOM's authors are in the process of developing an approach to extract metamodels from AXIOM mappings. Along with other approaches that mean to provide an approached of more structure an abstraction (cf. [EEM16]), this could lead to a standardized way of (possibly domain-specific) cross-platform development.

Finally, the Interaction Flow Modeling Language (IFML) [IFM] has to be mentioned. Its development has been driven by WebRatio [Webb], even though IFML is not directly focused on mobile development. However, since it addressed GUI applications it is highly relevant for apps, which require a well thought out user interface but due to the reliance on backend systems often a relatively simple business logic. Extensions such as Brambilla [BMU14] explicitly support mobile specific features. This includes event processing, for example for sensor data and user interface events (such as the "long press" on the latest Apple devices).

Summing up, the rapid progress in the very field of cross-platform development and in related areas in combination with the rapid proliferation of everyday mobile computing has led to many ideas about the future course of actions. Some of them are mere proposals how *better* apps could be enabled; others are able to sketch the path more clearly, particularly if existing developments are extrapolated.

25.4 Current Contributions to Business App Development

This chapter introduces the current status of business apps from the perspective of cross-platform development. The general outset is to illustrate what is already possible, which chances and limitations exist, and where we are currently heading. For this purpose, we start with a introduction

to the topic based both on the literature and contemporary practical approaches. We continue by introducing MD², which is our framework aimed at cross-platform development of business apps.

25.4.1 Understanding the Current State of Cross-Platform Utilization

Even though the idea of cross-platform development is appealing, many companies opt for developing individual apps for every platform [MH14]. As a result, they are faced with overhead expenditures for developing and continuously maintaining each app on its own. Also, features developed for one platform can frequently not be ported directly from one platform's app to another, but instead have to be re-implemented from scratch due to differences between the platform [JMK13]. Naturally, this leads to further unnecessary spending. Furthermore, customers changing from one platform to the other, or customers using a company's app on different devices, might be confused. This confusion might arise from two seemingly contradicting design decisions. A different look & feel of an app's version for different platforms leaves users wondering if they got the *right* app. An app looking similarly on all platforms but not being responsive to platform-specific characteristics (e.g. particular gestures) annoys users. In addition, this approach requires companies to choose their target platforms.

Needless to say, deciding against supporting certain platforms always implies that those customers who are using that platform are excluded from a company's services. Thus, intentionally leaving out platforms potentially neglects business opportunities or even disgruntle customers. Given that backdrop, it is apparent that current cross-platform approaches are unsatisfactory, so that companies perceive greater benefit from pursuing fragmented platform-specific approaches.

Mobile webapps and hybrid apps share the shortcoming that they cannot provide a truly native look-and-feel: They are always limited to the Web browser runtime in which they are executed (cf. Subsection 25.2.3), over which app developers hardly have any control. Web technology can be used to recreate visual elements of native user interfaces, thus theming the Web-based app to look and act like their native counterparts. However, this requires extensive use of CSS and JavaScript, thus harming execution performance of the app [HHM12]. Platforms offer no standardised means to imitate their look and feel on webapps. Consequently, every company has to develop this on their own repeatedly for each supported platform, and the respective runtime environments are unable to optimise execution of the required scripts. Web development frameworks only partly mitigate this circumstance [Hei+13]. Execution of Web-based apps therefore tends to be slow and battery consuming. The approach of imitating the look and feel is further harmed, as some devices do not provide a Web browser that complies with current Web technology standards [Mob]. In addition, older Android devices that are discontinued by their vendors do not receive updated Web browsers [Dob12].

Development based on self-contained runtimes results in apps that are packaged together with platform-specific runtime libraries. Platform-independent app code accesses features of all kinds of devices in a consistent way which is implemented by the runtime libraries. Consequently,

these platform-specific runtimes can be *optimised per platform*, resulting in better performance than that of Web-based apps. A general downside of that approach is that the least common denominator of the targeted platforms and the chosen approach determines the expressiveness for developers, and, therefore, the capabilities (or *richness*) of developed apps. This extends to multiple aspects of apps: For UI elements and hardware-provided platform features, such as device orientation, camera, GPS, and telephony, cross-platform development approaches can only offer features that are offered similarly by all platforms. Otherwise, platform-specific development tasks would remain. Furthermore, runtime libraries – although optimised – always constitute an execution overhead, resulting in worse performance in comparison with truly native apps. Consequently, native apps can always leverage more features of a platform than cross-platform apps; they remain more efficient at the same time.

Generative (in particular model-driven) approaches are suited to overcome the above discussed deficiencies: They allow cross-platform development using a single code base that serves all platforms. In the context of model-driven software development this code base is referred to as a *model* of an app. From the specified model, a generator can derive one or more programs. Therefore, in the context of cross-platform development, the generator can use a single model as an input to derive one app per target platform. During this process, the generator is not required to take the model literally: Instead, if required, it can deviate from ideal implementations in order to implement the developers' *intentions*. This is necessary if the model requires certain capabilities that the target platform does not provide. Take, for example, an app that requires a user's location data. During app generation for a platform that does not provide hardware support for this, the generator can substitute the corresponding part of an app with a software emulation of that feature. Alternatively, it could add a message to the add that informs users that their device does not provide the necessary support. For those platforms that support the feature, appropriate hardware calls would be generated. In this way not only multiple platforms are addressed but also the complexity that arises from device fragmentation is tackled.

As a result, generative approaches are not restricted to the least common denominator of all target platforms, but can mitigate this problem by providing adequate substitutions for required capabilities. Consequently, the generator's capabilities define the feature set usable for cross-platform app developers. Of course, conceiving and providing such substitutions has to be done by generator developers, which results in some development overhead. This overhead does not incur for each app, though, but only for one generator that will then be employed for multiple apps. In other words: the more apps are realized the less significant is the effort for generator implementation. If the generator creates purely native apps, this minimises runtime overhead. It results in *ideal* performance for end-users. With some experience and effort, the runtime performance of generated app rivals that of carefully manually implemented native apps.

Another benefit of model-driven generative approaches is that the programming language used for specifying the model is defined by generator developers, thus enabling programming languages that abstract from details. Rather than using a general purpose level, a domain-specific

language focuses on what is important for the intended purpose. This provides two major benefits: Firstly, repetitive boilerplate code of apps can be generated automatically, thus reducing clutter from the simpler, app-specific models. This can significantly reduce the number of lines of code (LOC) that needs to be written, easily outmatching even rapid development languages that honour the convention-over-configuration paradigm. Secondly, simpler programming languages can be created that are appealing to non-programmers. In fact, the language does not even need to resemble one of the widely used imperative programming languages. Declarative programming, graphical modelling and similar approaches are feasible. Consequently, a cross-platform app development solution with tight business integration and accessibility for people with little development experience enables them to develop parts of an app or even entire apps.

25.4.2 MD² as a Contemporary Generative Approach

Aiming to provide a cross-platform development solution for business apps, the framework MD² has been developed [MEK15]. The chapter's authors combine a variety of viewpoints on MD² and, thereby, on cross-platform app development in general:

- One of the authors was a member of the original research group of 2013 [HMK13; HM13].
- One of the authors supervised the ongoing development [MEK15; ME15].
- Another three have recently joined the work on the approach and contribute to the revision and rejuvenation of the approach [Dag+16; Ern+16].

MD² follows a model-driven generative approach. It provides a consolidated programming language, MD²-DSL, which is employed to develop a business app. Following the Model-View-Controller (MVC) pattern [Bus+96], a developer specifies a business app as a so-called *artifact*, which comprises multiple files in MD²-DSL.

The MD²-DSL is accompanied by software that derives actual apps from a developed artifact (cf. Figure 25.2). MD² contains a *preprocessor* which reads an artifact and performs platform-independent transformations on it, converting shorthand notations to more complex statements. More importantly, MD² comprises a set of generators, one per target platform. Each generator operates on the preprocessed artifact and creates platform-specific, native code for its respective target. In result, the generators create one native app per target platform, thus optimizing the generated code for execution on its target. MD² currently provides generators for iOS, Android, and the proprietary map.apps [EEM16]. Furthermore, a generator for a common Java EE-based backend is included, which provides RESTful Web services for common data storage of all generated apps. The latter can be seen as a blueprint for access to arbitrary backend systems. This is particularly important since business apps typically rely heavily on (existing) corporate systems [MEK15].

While further extension of the supported platforms is desirable, this target is currently not of foremost importance. Implementing an additional generator would pose only minor challenges

but be very time consuming. Moreover, if would hardly lead to additional research insights. Thus, this task ought to be reserved for the effort of a to-be-established community rather than to be a research activity.

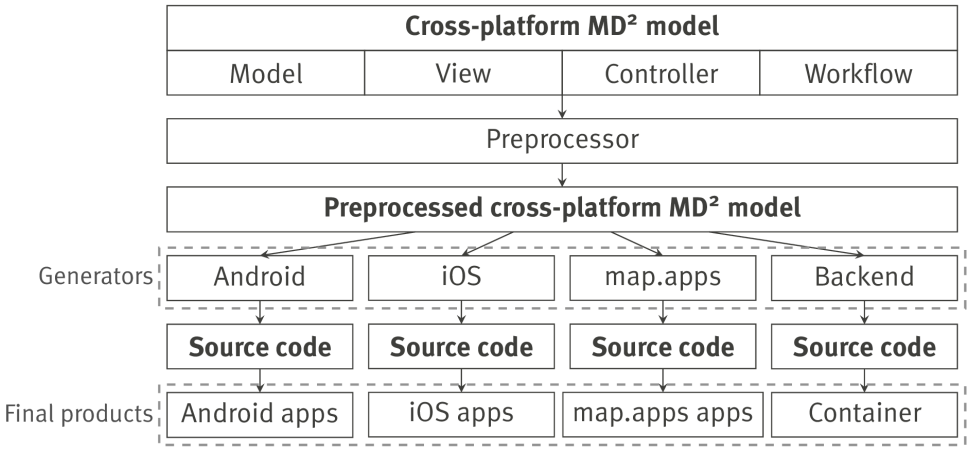


Figure 25.2: Flow From the Cross-Platform MD² Model Through the Set of Generators, Creating Multiple Platform-Specific Native Apps

The focus of MD² is on data-driven business apps, i.e. apps that heavily rely on display and manipulation of business data using forms [MEK15]. The MD²-DSL is designed such that creating, managing, and using UI elements for forms is as simple as possible for a developer. In fact, MD² should even be suited to people with some prior knowledge in textual modelling but no specific background as a programmer. Moreover, the developer does not have to consider peculiarities of the targeted platforms; instead, this task is delegated to the respective generators. Therefore, the amount of required boilerplate code is reduced. At the same time, the complexity is lowered since less platform-specific experience is required to realize *good* apps. The same holds true for all operations on the underlying data, as well as for connecting UI elements to data that are stored either locally or on a generated or manually developed backend: The developer only specifies the relationships between data and its display, whereas the generators will create corresponding code for the apps. An additional perk of this approach is rapid development once basic experience is gained with MD².

Exemplary, an app developer could use MD² for the development of a public sector app that lets citizens capture and describe a deficit observed in public spaces and send the report to the local authorities [Dag+16]. Creating such a report could start with determining the citizen's location: Depending on a platform's or hardware's capabilities, this can be realised using sensor data or software emulation, but the developer does not need to consider this during development. Considering the entire reporting workflow as a process model, several steps like this are involved (cf. Figure 25.3). Using MD²-DSL, each step can be developed individually as a self-contained component. Afterwards, these components can be arranged and re-arranged into a particular order, based on the final requirements of the apps. Furthermore, MD²-DSL enables to specify a set

of apps that, in combination, fulfil a workflow. In this example, there would be one app targeted at citizens, which reports to another app for the public administration. The latter can also send feedback to the reporting citizen. The generators use the developed MD² artifact and generate an according set of apps, as well as a backend that coordinates the interaction of individual apps.

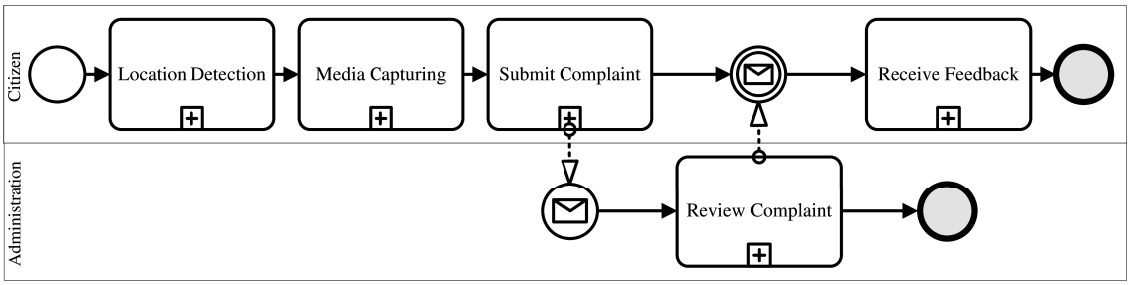


Figure 25.3: Process Model of an Exemplary Business App for the Public Sector [Dag+16]

Similarly, MD²-DSL could be used to create apps for insurance companies, integrating data-centric workflows of field staff and office staff together with corporate backend. A further use case that could be realised is a pharmacy's inventory control system which is combined with point-of-sale data.

25.5 Paving the Road to Future Cross-Platform Business App Development

In this section we present our vision of cross-platform business app development. We start by sketching technological progress that we deem as a requirement. Next, we propose how a stronger domain-orientation will facilitate a wider adoption of cross-platform approaches. Based on this background, we sketch our view of the future. This is detailed in the following subsections, which successively discuss merits, challenges, and limitations.

25.5.1 Required Technological Work

From a technical perspective, some challenges arise that need to be overcome before leveraging the benefits of cross-platform approaches for mobile e-commerce apps. Firstly, reliable integration with current e-commerce back-end systems needs to be available to reuse existing business logic. Whereas in the B2B sector basic electronic data interchange standards exist for many years [MKK95], integrations of mobile software with shop back-ends, ERP systems, mobile payment brokers, recommender software, pricing systems, and other specialized software will be required. This is necessary for the multitude of players on the market, many of them having differing proprietary interfaces. In general, managing the complexity of e-commerce systems will be essential to develop intuitive and well-performing applications for mobile devices. Model-driven

cross-platform approaches therefore also need to tackle the complexity of DSLs, for example regarding developing methods for language composition, interplay, and embedding [Völ13].

As a second domain of technological requirements, context-sensitivity needs to be drastically improved to provide a significant benefit compared to regular Web sites (or webapps). Recent studies indicate that the mood of a potential customer influences the susceptibility for recommendations [SM16]. Apps need to consider such psychological insights together with context information in order to provide an improved shopping experience for customers and higher conversion rates for merchants. As a consequence, smart notifications, geotargeted advertising, and meaningful recommendations need to rely on intelligent functionalities [GWH13; Sha+16]. In the long term, further technical innovations with regard to sensor usage and novel wireless communication mechanisms (e.g. beacons) will foster mobile e-commerce. As such technologies are usually platform-dependent. Cross-platform approaches will play a major role in bridging different implementations and add to an enjoyable shopping experience. Moreover, they will be required for keeping the implementation cost-effective.

25.5.2 Improving Domain-Orientation

Mobile apps in the domain of e-commerce have not yet been researched extensively (cf. e.g. [BRS07; TL14]). Apart from studies and guidelines on apps in general, which focus on user interfaces, feature fit and functionality on mobile devices [Appd; Kha+15], best practices and guidelines specifically for e-commerce apps are scarce. For example, the representation of shopping carts and product catalogues on mobile devices remains basically equal to traditional e-commerce Web pages. While this is not a problem per se, it hinders the adoption of more user-friendly possibilities. Thus, research needs to be conducted in order to uncover suitable patterns for domain-specific interactions, ultimately leading to more engaging user experiences for the app user.

Concerning MDSD as specific approach for cross-platform development, and DSLs in particular, their appropriateness and eventual success highly depends on the apt tailoring to the targeted domain. A key element to leverage the potential gain in productivity compared to traditional programming model is the manageability of complexity though increased abstraction [SV06]. Therefore, representing domain-specific concepts of e-commerce as abstract concepts in the cross-platform approach is essential for integrating the business needs with app development. To support this development, domain experts in the field of e-commerce should be involved in the development process. This can be achieved by tightly working together with the app developers. In a model-driven context, the domain expert can even actively participate in the app creation if the language is designed to support abstract domain concepts in a declarative manner such that the result can be modelled rather than traditionally programmed. In the long run, a *community* must be created. Developer communities greatly contribute to the ease-of-use of a platform [MWA15], and will thus be required for frameworks to become successful.

25.5.3 Sketching the Future

As described in Section 25.4, our cross-platform development framework MD² already supports the development of business apps for multiple platforms and may serve as example for a glimpse into the future of cross-platform development. There is a wide range of possibilities of how to advance the current approach, considering both technical and domain-specific aspects, while paving the road for future business app development.

From a business perspective, apps should support the existing digital ecosystem in e-commerce. Therefore, a strong focus on business processes is essential to match domain experts' knowledge and expectations. This is reflected in research on MD², where workflows have been introduced in order to group sequences of views in an app into larger, business process-oriented steps and allow for interaction between apps based on user roles [Dag+16]. In addition, there is ongoing research towards modelling use cases as self-contained archetypes and automatically inferring app structure, navigation, and data models according to user roles. In the future, manual interventions on process interfaces can be avoided, allowing fully integrated process flows towards the customer and within the organization.

Concerning technical aspects, overcoming the increase of complexity can achieve significant progress on the adoption of model-driven cross-platform approaches. While the domain-specific language has grown to a powerful tool over time, its complexity has risen, too, and may alienate new developers and modellers [HMK13]. To support developers for the target platform, a reference architecture for mobile business apps was developed [Ern+16; EEM16]. Ongoing research also deals with the development of a graphical DSL in order to ease the actual modelling of apps. Coinciding, the level of abstraction will be further increased towards a rather declarative style of creating apps in contrast to an explicit and detailed specification of user interfaces and navigation. Ideally, business users without much technical knowledge will be able to express their domain knowledge as an archetype and use them as tool for communication [Voe+13].

Finally, the current approach is characterised by a form-based approach using standardised view elements. In the future, improved visualizations and new widget types can improve the apps' aesthetics. For example, new layout types such as cards and infinite scrolling (as in Twitter) are visually appealing representations of content still missing in the current version of MD². Although not exclusive to this domain, well-designed user interfaces with rich imagery and typography have a direct influence on user behaviour and engagement, which makes their design particularly relevant for e-commerce. Ultimately, a theming approach may be a suitable option to model the apps' appearance while taking up the challenge of balancing cross-platform abstraction and platform-specific characteristics.

25.5.4 Possible Merits and Positive Accessory Phenomena

The relevance for app development in the field of m-commerce has two different facets. On the one hand, apps change the way a company can interact with its customers. On the other hand,

mobile devices can be leveraged for employee usage to improve internal processes, such as order handling. However, both areas profit from advantages of model-driven software-development being applied to the development of mobile apps. Apart from the general benefits of model-driven development such as higher productivity, fast development cycles, or integration of domain experts [Voe+13], the application of model-driven aspects on cross-platform development offer additional value. Through the native implementation of apps generated with the MD² framework, these apps can deliver a native look and feel and thus increase the emphasized user experience [HHM12]. Additionally, they are able to integrate certain hardware features of the mobile device, such as geolocation, camera, and microphone in a way that enables new use cases.

With the aforementioned growth of the mobile computing market and the ever widening capabilities of mobile technology, it is hardly possible to predict what may be possible in ten years. However, a mid-term look is well possible. While the number of different platform is a core issue in mobile app development, a similar issue can be observed, for instance, for the interfaces of *smart devices*. Current sharing protocols, such as Apple Airplay or UPnP, are limited in terms of collaboration among each other [Bus+15]. Cross-platform development could help to overcome these limitations by generating interfaces for every possible combination of device or protocol. Enabling the interconnection of different smart devices in this way facilitates a much wider range of usage scenarios. Thus, when talking about the target platforms of cross-platform app-development, in the future smartphones and tablets may not be the only conceivable target, but new technologies for augmented reality (for instance Google Glass, Microsoft HoloLens), smart watches, or other smart devices may be considered as well. There is already research for augmented reality (AR) in the context of e-commerce [LS07]. With a fully integrated software system landscape on devices through the whole supply chain, new scenarios are imaginable, which are profitable both for customers and enterprises. The customer can be given a reinvented shopping experience through in-shop information, a close contact to the company, and the possibility of trying while shopping enabled by AR. Additionally, the customers have the freedom to interact with the company when and wherever they like to. Companies can profit from timely information of the customers' behaviour and needs. Further, analytical details offered by mobile technology and the innovations in context with the Internet of Things (IoT) can improve the information needed for operational, tactical, or strategical decisions.

To embed the usage of different sorts of mobile devices into the internal workflows, a close orientation on existing business process models can be useful. A model to model transformation could be leveraged to make excerpts of the company's business process models usable by model-driven approaches. Using a unified graphical modelling language, the process model excerpts could then be enriched by data necessary for app development in a way comprehensible by domain experts, which may be not IT-affine. Web services or other interfaces can help to interconnect the generated apps with the existing application landscape of the company. In this way, the apps can show and generate valuable information from and to the different actors of the whole supply chain in-time and thus increase the overall value generation.

25.5.5 Challenges, Obstacles, and Possibly Negative Ramifications

Several prerequisites from a technical and domain-oriented viewpoint have already been discussed before. However, additional challenges need to be overcome when implementing and maintaining business apps using cross-platform approaches. A robust framework suffers from a chicken-and-egg problem, being adopted only when it proves to be useful and simultaneously profiting from the application to the real world. Ultimately, the major hurdle for the initial adoption of a cross-platform approach is its usability. Therefore, improving its fit to the desired target users and usage scenarios needs to be considered permanently. Each approach needs to develop mechanisms in order to manage the complexity and abstract from individual platform issues. For instance, model-driven approaches have to balance and reduce the complexity both on the modelling side (targeted for non-technical business users) and the generation side (targeted for individual platform generator developers). Apart from a sound technical design, proper tools and documentation are key for the initial adoption of an approach as well as the integration into the overall software development cycle including testing, build and deployment.

25.5.6 Remaining Issues

In the prior section we have identified challenges that need to be overcome in order to benefit from cross-platform approaches in mobile e-commerce. In addition to these, more fundamental problems exist with regard to user acceptance and economic reasons. Firstly, the willingness of users to install separate apps for different shops needs to be questioned. The growing number of entries in app markets leads to a competition for storage space on customers' mobile devices: If an app does not perform as expected, it is ignored in future or removed. This high competition leads to increasing challenges in the app market in view of value proposition and user experience [ZM13]. This observation indicates that only customers with frequent orders are likely to install a vendor-specific mobile app unless unique benefits are proposed through its usage. Web apps have lower entry barriers as no installation is required, yet customer engagement and loyalty can be expected to be comparable with traditional Web sites.

From a development perspective, cross-platform approaches offer significant benefits in terms of reusability. Yet, initial development effort for multiple platforms is high. Particularly, model-driven techniques that drastically simplify and accelerate the app creation process require substantial initial efforts for generator development. In many scenarios, no more than one application with infrequent new versions will be needed by a company, consequently incentivizing Web-based approaches. As an alternative business model, customized apps may be provided through a service provider that develops an industry-wide DSL and benefits from economies of scale. However, legal issues remain regarding the responsibility for sensitive customer information and transaction data.

Finally, mobile devices offer the possibility to track users much more comprehensively, for instance regarding customer behaviour, location profiles, or personal preferences. Potential misuse

of this information leads to privacy concerns and may harm the acceptance of such applications by customers.

25.6 Discussion

Building on the previous sections, we now discuss our findings. Firstly, we set out with several scenarios that fill our before sketched vision with life. We then provide an outlook and name open questions for research and practice. Eventually, we name limitations of the work presented in this chapter and sketch our own research goals, which are derived from the work presented here and which honour the limitations.

25.6.1 Scenarios

In the following, we have compiled three future usage scenarios. In fact, parts of them have already been demonstrated individually and most of them should be possible with today's technology. We believe that we will see such applications much more widely in the near future. Please note that these scenarios are for illustration, and might pose some inherent problems. In particular, privacy and security will need to be carefully considered on top of the extended functionality and comfort. The future progress regarding cross-platform business apps can also be transferred to other domains. Therefore, a fourth scenario depicts a possible application of mentioned features to crisis management.

Scenario 1: Multi-channel shopping experience

Imagine Peter is looking for a new technology gadget on his tablet device while relaxing on the couch in the evening. To be well prepared, he uses an app to inform himself about different devices and their specific characteristics and capabilities. Reviews by other customers are also available within the application so that he can get a first impression based on the opinion of other customers and bookmark interesting items. To get hands-on experience, Peter decides to try out some devices in a (physical) store and therefore looks up the directions to the nearest shop location. From within the app, he can see tomorrow's opening hours and schedule an appointment with a salesperson, which is automatically added to his calendar. The next day, Peter uses the smartphone app, which synchronized the data, to show and discuss his favourite devices with the salesperson in store. While strolling through the store for a few more minutes in order to decide on a device, his phone vibrates and offers him a break with a free drink at the shopping centre lounge area. While enjoying his drink, he uses one of the tablets which are available for the customers, and is instantly redirected to a social media platform, which exchanges information about the gadgets he is about to buy. Peter raises a question to the community and after a few seconds has an answer from a former customer, who praises the quality of one device and recommends him an accessory which connects to his multimedia system at home. With intelligent app support, the

multi-channel shopping experience is dramatically improved by seamlessly integrating into the customers' shopping behaviour and combining the best of the digital and physical world.

Scenario 2: Technology-assisted fashion shopping

Claudia is visiting the close-by shopping centre, since she likes the shopping experience and is looking for a new fashionable outfit. For some inspiration she looks around through the shop's assortment. Through augmented reality (AR) glasses that she is wearing, her eye movements are tracked. After some minutes of gathering information and based on previous orders, Claudia's smartphone recommends her some additional trousers, which might suit what she is looking for. The glasses are pointing her the way to where the trousers are positioned and then to the changing rooms. Her former purchases are saved in the cloud and the 3D models of each piece is offered through the retailers web interface. Claudia scrolls through her purchases using her smartphone, which are then directly projected on her body by the AR glasses. Thus, she can directly try on the new trousers in combination with the tops she bought last week in her parents' home town. Unfortunately, she is not satisfied with the colour of the tops presented in the store. Luckily, she can order unavailable items which are sent to her from the central warehouse. She does not even need to specify her size, since the combination of AR and order history allows the app to submit the proper size for the chosen apparel brand. Meanwhile, the company's purchasing department was already informed about the missing tops, including detailed information about the whole outfit Claudia is pleased with. This information is propagated through the supply chain and reaches the designer, who is now able to rethink his product line for the next season. As a result, business apps for AR devices offer novel shopping experiences to customers' based on their preferences while at the same time creating the potential for cross-selling for vendors.

Scenario 3: B2B procurement

In a B2B context, the advantage of business process integration can be highlighted. John needs to buy new consumables for his department within a large organisation. To check the current stocks in the storeroom and reorder items running short, he can use a webapp on his mobile device. In order to decide on the reorder quantity, the system provides visualisations for the consumption of the previous months. If required, a trend analyses and utilization simulation can be run. As usual in large enterprises, he has to honour existing procurement contracts. Therefore, the application acts as a marketplace in which merchants can list products that were previously negotiated in master agreements by the procurement department. After completing the activity, the order is automatically passed to the department leader who is in charge of the budget decisions for formal approval.

Scenario 4: Tunnel Fire

Judy is using her municipality's self-service app. This app provides basic e-government functionality. For example, Judy can prepare administrative formalities or report potholes. The app, however, also has an emergency functionality that only activates in case of disasters. When driving home from a visit to the mountains with her family, Judy enters a long tunnel. After hearing a loud, shaking sound, she has to break strongly since the cars in front of her come to a halt. Smoke quickly fills the tunnel, making orientation impossible. The children scream. While the mobile network breaks down, the self-service app requests Judy to turn on WiFi. Within seconds, her mobile phone and the devices of other people in the tunnel form an ad hoc network. Since many devices have temperature sensors, and the devices' cameras can be used to determine the density of the smoke, in a joint effort the network can determine in which direction the fire must be located. Based on a plan of the tunnel, Judy's device displays hints on how to get to the nearest – and safest – emergency exit. When Judy and her family safely exit the tunnel, the fire fighters already arrive. They have been called automatically by the device of the first person that made it out. The fire is condemned quickly since the brigade leader can utilize the information recorded by the ad hoc network.

25.6.2 Outlook and Open Questions

Throughout the prior sections of this chapter, a need for much further research has become apparent. The observations of what is possible so far, and the study of how these possibilities have been enabled, allow for an outlook. As suggested earlier, this outlook cannot be presented in a form of looking ten (or even more years) into the future and proposing the then to-be-encountered reality.[2] Rather, it can be argued which questions will need to be answered to enable what could be a desirable future of mobile computing, specifically multi-platform business apps.

The possibly most profound question concerns technology choice: Which cross-platform app technology will prevail? While mobile platforms have become more similar than they have been a few years ago, there is no trend towards full unification in terms of a common platform, programming model or the like. Distinctive ecosystems will remain. As we have shown, no cross-platform solution has found universal acceptance, although those based on Web technology are quite widely used. At the same time, the technological superiority of generative approaches is apparent – as is the ease-of-use and development simplicity of webapps. While this makes a prediction impossible, it is not too bold to expect further unification and a strive for easier multi-platform development. We strongly believe that contributing to advanced cross-platform technology is important for pursuing this goal.

Many open questions remain when bringing together the technological and the economical dimension of business app development. Traditionally, technological progress is made by engineers, in this case particularly by computer scientist and electric engineers. Observations are

[2] Even forecasts that are less than 10 years old need to be considered outdated by now [WTS07].

made by behaviourist, i.e. typically social scientists. Unfortunately, bringing together both worlds to provide a kind of "round-trip" research often is missing. What would be needed to pave the future are circles of observation and engineering (also cf. [ME15]). How these can be designed to keep pace with the rapid proliferation of the field remains an open question, though. For rigorous research in particular, there is a very high risk that profound work is not only practically irrelevant but literally outdated once accomplished. This might be one of the reasons for the current lack of a theoretical foundation.

Considering the technological dimension of cross-platform business app development, a host of questions remains open. These typically have a high level of detail and, thus, are out of scope of discussion here. Bridging the technological dimension to the domain-oriented one remains a profound challenge, though. Our vision includes app development by people who are no seasoned programmers. How to achieve this has yet to be solved. Progress in this regard would also be beneficial for senior development staff. Nowadays, grasping the total complexity of software development is very hard. It, for example, cannot be expected that all developers have proficiency considering system security. Therefore, if next-generation tools can provide reasonable "default" security, developing would become much easier. Even more, this would be highly beneficial from an economic standpoint as investment in apps would be less risky. Again, how to achieve such reasonable default provision of functionality remains an open question. Undoubtedly, integration with cross-platform frameworks would be fascinating. Of course, overcoming heterogeneity in this regard will be a challenge of its own.

While some of these questions might seem to be hard to be answered, we are very optimistic. Economic pressure and competition in searching for solutions to urgent problems can lead to the rapid development of sustainable solutions. An example for this is Mobile Device Management (MDM). Desperately required only a few years ago (cf. [MH14]), it now can be regarded as pretty well conquered [MWA15].

25.6.3 Limitations

A research article should name any limitations that could diminish its accuracy or narrow its applicability. In case of this chapter, it of course has to be kept in mind that we take a rather bold look at the future. All ideas presented in this chapter are based on thorough work. However, neither of us is a prophet and, therefore, some of the developments we propose might turn out wrong. Moreover, predicting fundamental changes is notoriously hard and it is easy to get the future work. At the same time, it is very regarding if you get it right, particularly, if the vision has been profound (cf. e.g. with Weiser's *Computer for the 21st Century* [Wei99]). Therefore, it has to be kept in mind that the further one of our proposals looks into the future, and the greater the change we propose is, the greater is the probability that *something different* will happen. We have tried to mitigate this general problem of future-oriented work by sketching alternatives and explicating uncertainties.

The second limitation of the work presented here is its non-empirical onset. This is a deliberate choice; in fact, we believe that a future-oriented article that tried to combine work from a theoretical and observational standpoint (such as ours), and empirical data is likely to become confusing. However, we deem both quantitative and qualitative studies to be necessary to tackle some of the open questions sketched above. Probably, a future article can join the non-empirical and the empirical approach to a more comprehensive vision.

25.6.4 Future Work

While a general need for extensive future activities in the field of cross-platform app development and many neighbouring activities has been stressed when giving an outlook, this section eventually sheds light on our own plans.

Concerning model-driven software development, our research focuses on lowering the complexity of development. In particular, the unique advantage of MDSD regarding easy-to-understand models for domain-specific purposes is further investigated. Currently, a graphical DSL with a respective modelling environment is developed that allows non-technical domain experts to model apps. To exploit its full potential, the language is designed to be highly declarative and rely on inference mechanisms in order to derive fully functional business apps. Furthermore, several ideas exist for the continuation of the MD² framework itself (see also Subsection 25.5.3), for example regarding improved business processes integration as well as language modularity.

Considering app development in general, we will look at more domain-specific applications as well as on better domain-specificity. It has to be asked whether the business app as we frame it is the perfect constraint for domain-specific app development. Possibly, additional domains can be added, some being related, while others being widely different. An example for a related domain could by graphically intensive apps that are still used for business purposes. For this purpose, a graphical DSL might be more suited. Another example would be a DSL that is specifically tailored to apps that are used in crisis management. This field could greatly benefit from the possibility to tailor apps in the immediate wake of a disaster. Such an app should be flexible enough to run on a multitude of devices because in disaster situations you usually have to cope with whatever might be available.

25.7 Conclusion

In this chapter we presented a vision of improved business apps that have been developed using cross-platform development approaches. Apps already have a profound economic (and societal) impact. Their importance will continue growing. However, even with only two mobile platforms that have significant market shares remaining, cross-platform development remains an important topic. Current approaches often employ Web technology, which enables quick results. However, apps often do not perform well and have user experience shortcomings. Solutions based on

model-driven software development provide the benefits of cross-platform development while mitigating many current shortcomings faced when implementing apps. Based on a discussion of MD², being an MDSD cross-platform approach with a comparatively long development history, we have sketched the path to future business app development. To enable the next generation of apps, both technological progress and a strong domain-orientation are required. Then, however, the outlook is bright with many possibilities and merits. These also have been demonstrated in a number of scenarios. The required progress and remaining challenges and limitations hint to a number of open questions. For some of these, our own work will seek to provide answers.

Looking back at the development of mobile computing in the age of smartphones, a ludicrous pace of innovation has to be attested. While such a dynamic always poses risks and poses problems, the future does not look grim. In fact, the promise of future possibilities is impressive. We believe that apps will continue to enrich enterprise activities and society. As demonstrated not only in this chapter but in the whole book, the research's journey for this has by far not come to an end.

References

[Al-+15] Afnan Al-Subaihin et al. "App Store Mining and Analysis". In: *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*. DeMobile 2015. New York, NY, USA: ACM, 2015, pp. 1–2. DOI: 10.1145/2804345.2804346.

[Apa] Apache. *Apache Cordova*. URL: https://cordova.apache.org/ (visited on 04/28/2016).

[Appa] AppCelerator. *AppCelerator*. URL: http://www.appcelerator.com/ (visited on 04/30/2016).

[Appb] Applause. *Applause Repository*. URL: https://github.com/applause/applause (visited on 04/29/2016).

[Appc] Apple. *Apple iOS*. URL: http://www.apple.com/de/ios/ (visited on 04/30/2016).

[Appd] Apple Inc. *iOS Human Interface Guidelines*. URL: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/index.html (visited on 04/21/2016).

[Bar+15] Scott Barnett et al. "A Multi-view Framework for Generating Mobile Apps". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* 1 (2015), pp. 305–306. DOI: 10.1109/VLHCC.2015.7357239.

[BK11] Rahul C. Basole and J??rgen Karla. "On the evolution of mobile platform ecosystem structure and strategy". In: *Business and Information Systems Engineering* 3.5 (2011), pp. 313–322. DOI: 10.1007/s12599-011-0174-4.

[BMU14] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. "Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End". In: *MobiWIS* 2000 (2014), pp. 176–191. URL: http://dblp.org/db/conf/mobiwis/mobiwis2014.html%7B%5C#%7DBrambillaMU14.

[BRS07] Enrique Bigné, Carla Ruiz, and Silvia Sanz. "Key Drivers of Mobile Commerce Adoption. An Exploratory Study of Spanish Mobile Users". In: *Journal of Theoretical and Applied Electronic Commerce Research* 2.2 (2007), pp. 48–60.

[Bus+15] Christoph Busold et al. "Smart and secure cross-device Apps for the Internet of advanced things". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8975 (2015), pp. 272–290. DOI: 10.1007/978-3-662-47854-7_17.

[Bus+96] Frank Buschmann et al. *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley, 1996.

[BVG15] Scott Barnett, Rajesh Vasa, and John Grundy. "Bootstrapping Mobile App Development". In: *Proceedings - International Conference on Software Engineering* 2 (2015), pp. 657–660. DOI: 10.1109/ICSE.2015.216.

[CMK14] Jagmohan Chauhan, Anirban Mahanti, and Mohamed Ali Kaafar. "Towards the Era of Wearable Computing?" In: *Proceedings of the 2014 CoNEXT on Student Workshop*. CoNEXT Student Workshop '14. New York, NY, USA: ACM, 2014, pp. 24–25. DOI: 10.1145/2680821.2680833.

[CYA12] Pern Hui Chia, Yusuke Yamamoto, and N Asokan. "Is This App Safe?: A Large Scale Study on Application Permissions and Risk Signals". In: *Proceedings of the 21st International Conference on World Wide Web*. WWW '12. New York, NY, USA: ACM, 2012, pp. 311–320. DOI: 10.1145/2187836.2187879.

[Dag+16] Jan C. Dageförde et al. "Generating App Product Lines in a Model-Driven Cross-Platform Development Approach". In: *Proceedings of the 2016 49th Hawaii International Conference on System Sciences*. 2016, pp. 5803–5812. DOI: 10.1109/HICSS.2016.718.

[Dob12] Alex Dobie. *Why you'll never have the latest version of Android*. 2012. URL: http://www.androidcentral.com/why-you-ll-never-have-latest-version-android (visited on 04/30/2016).

[EEM16] Sören Evers, Jan Ernsting, and T A Majchrzak. "Towards a Reference Architecture for Model-Driven Business Apps". In: (2016), pp. 5731–5740. DOI: 10.1109/HICSS.2016.708.

[Ern+16] Jan Ernsting et al. “Refining a Reference Architecture for Model-Driven Business Apps”. In: *WEBIST 2016 - Proceedings of the 12th International Conference on Web Information Systems and Technologies, Rome, Italy, 23-25 April, 2016* (2016).

[Fac15] Facebook. “Facebook reports second quarter 2015 results”. In: *facebook Investor Relations* (2015). URL: http://investor.fb.com/releasedetail.cfm?ReleaseID=924562.

[Fle07] Nik Fletcher. *Apple: ”we plan to have an iPhone SDK in developers’ hands in February”*. 2007. URL: http://www.engadget.com/2007/10/17/apple-we-plan-to-have-an-iphone-sdk-in-developers-hands-in-fe/ (visited on 04/30/2016).

[FNP15] Gabriel Filios, Sotiris Nikoletseas, and Christina Pavlopoulou. “Efficient Parameterized Methods for Physical Activity Detection Using Only Smartphone Sensors”. In: *Proceedings of the 13th ACM International Symposium on Mobility Management and Wireless Access*. MobiWac ’15. New York, NY, USA: ACM, 2015, pp. 97–104. DOI: 10.1145/2810362.2810372.

[Fu+13] Bin Fu et al. “Why People Hate Your App: Making Sense of User Feedback in a Mobile App Store”. In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’13. New York, NY, USA: ACM, 2013, pp. 1276–1284. DOI: 10.1145/2487575.2488202.

[GHG10] Tor-Morten Grønli, Jarle Hansen, and Gheorghita Ghinea. “Android vs Windows Mobile vs Java ME: A Comparative Study of Mobile Development Environments”. In: *Proceedings of the 3rd International Conference on Pervasive Technologies Related to Assistive Environments*. PETRA ’10. New York, NY, USA: ACM, 2010, 45:1–45:8. DOI: 10.1145/1839294.1839348.

[Goo13] Google. “Mobile In-Store Research: how in-store shoppers are using mobile devices”. In: *Google shopper marketing council* April (2013).

[Goo16a] Google. *Android*. 2016. URL: https://www.android.com/ (visited on 04/30/2016).

[Goo16b] Google. *Android Auto*. 2016. URL: https://www.android.com/intl/de%7B%5C_%7Dde/auto/ (visited on 04/28/2016).

[Goo16c] Google. *Google Play Store*. 2016. URL: https://play.google.com/ (visited on 04/30/2016).

[GP12] Laurence Goasduff and Christy Pettey. *Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth*. 2012. URL: http://www.gartner.com/newsroom/id/1924314 (visited on 04/30/2016).

[GR14] Arindam Ghosh and Giuseppe Riccardi. “Recognizing Human Activities from Smartphone Sensor Signals”. In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM ’14. New York, NY, USA: ACM, 2014, pp. 865–868. DOI: 10.1145/2647868.2655034.

[GWH13] Daniel Gallego, Wolfgang Woerndl, and Gabriel Huecas. "Evaluating the impact of proactivity in the user experience of a context-aware restaurant recommender for Android smartphones". In: *Journal of Systems Architecture* 59.9 (2013), pp. 748–758. DOI: 10.1016/j.sysarc.2013.02.004.

[Hei+12] H Heitkötter et al. *Business Apps: Grundlagen und Status quo*. Working Papers 4. Münster: Förderkreis der Angewandten Informatik an der Westfälischen Wilhelms-Universität Münster e.V., 2012.

[Hei+13] H Heitkötter et al. "Evaluating Frameworks for Creating Mobile Web Apps". In: *WEBIST 2013 - Proceedings of the 9th International Conference on Web Information Systems and Technologies, Aachen, Germany, 8-10 May, 2013*. Ed. by Karl-Heinz Krempels and Alexander Stocker. SciTePress, 2013, pp. 209–221.

[HHM12] H Heitkötter, S Hanschke, and T A Majchrzak. "Comparing Cross-platform Development Approaches for Mobile Applications". In: *Proceedings of the 8th International Conference on Web Information Systems and Technologies (WEBIST)*. Porto, Portugal, 2012, pp. 299–311. DOI: 10.5220/0003904502990311.

[HKM15] H Heitkötter, H Kuchen, and T A Majchrzak. "Extending a Model-Driven Cross-Platform Development Approach for Business Apps". In: *Science of Computer Programming (SCP)* 97.Part 1 (2015), pp. 31–36.

[HM13] H Heitkötter and T A Majchrzak. "Cross-Platform Development of Business Apps with MD²". In: *Proc. of the 8th Int. Conf. on Design Science at the Intersection of Physical and Virtual Design (DESRIST)*. Vol. 7939. LNBIP. Springer, 2013, pp. 405–411.

[HMK13] H Heitkötter, T A Majchrzak, and H Kuchen. "Cross-Platform Model-Driven Development of Mobile Applications with MD²". In: *Proc. 28th SAC*. ACM, 2013, pp. 526–533.

[HV11] Zef Hemel and Eelco Visser. "Declaratively Programming the Mobile Web with Mobl". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 695–712. DOI: 10.1145/2048066.2048121.

[IDC16] IDC. *Smartphone OS Market Share*. 2016. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (visited on 04/30/2016).

[IFM] IFML. *IFML: The Interaction Flow Modeling Language*. URL: http://www.ifml.org/ (visited on 04/30/2016).

[JJ15] Christopher Jones and Xiaoping Jia. "Using a Domain Specific Language for Lightweight Model-Driven Development". In: *ENASE 2014, CCIS 551*. 2015, pp. 46–62. DOI: 10.1007/978-3-642-23391-3.

[JMK13] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. "Real challenges in mobile app development". In: *International Symposium on Empirical Software Engineering and Measurement*. 2013, pp. 15–24. DOI: 10.1109/ESEM.2013.9.

[Kej09] Ankita Arvind Kejriwal. "MobiDSL - a Domain Specific Language for Mobile Web Applications: developing applications for mobile platform without web programming". In: *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling* (2009).

[Kha+15] Hammad Khalid et al. "What do mobile app users complain about?" In: *IEEE Software* 32.3 (2015), pp. 70–77. DOI: 10.1109/MS.2014.50.

[Kim+13] Jieun Kim et al. "Mobile application service networks: Apple's App Store". In: *Service Business* 8.1 (2013), pp. 1–27. DOI: 10.1007/s11628-013-0184-z.

[Kim10] K Kimbler. "App store strategies for service providers". In: *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on*. 2010, pp. 1–5. DOI: 10.1109/ICIN.2010.5640947.

[KMJ15] Ju Young M Kang, Jung Mee Mun, and Kim K P Johnson. "In-store mobile usage: Downloading and usage intention toward mobile location-based retail apps". In: *Computers in Human Behavior* 46 (2015), pp. 210–217. DOI: 10.1016/j.chb.2015.01.012.

[KZG12] Lara Khansa, Christopher Zobel, and Guillermo Goicochea. "Creating a Taxonomy for Mobile Commerce Innovations Using Social Network and Cluster Analyses". In: *Int. J. Electron. Commerce* 16.4 (2012), pp. 19–52. DOI: 10.2753/JEC1086-4415160402.

[LS07] Yuzhu Lu and Shana Smith. "Augmented Reality E-Commerce Assistant System: Trying While Shopping". In: *Human-Computer Interaction. Interaction Platforms and Techniques: 12th International Conference*. Ed. by Julie A Jacko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 643–652. DOI: 10.1007/978-3-540-73107-8_72.

[Mac07] Michael Macedonia. "iPhones Target the Tech Elite". In: *Computer* 40.6 (2007), pp. 94–95. DOI: 10.1109/MC.2007.212.

[Maj+17] Tim A. Majchrzak et al. "How Cross-Platform Technology Can Facilitate Easier Creation of Business Apps". In: *Apps Management and E-Commerce Transactions in Real-Time*. Ed. by In Lee and Sajad Rezaei. Advances in E-Business Research. IGI Global, 2017, pp. 104–140. DOI: 10.4018/978-1-5225-2449-6.ch005.

[Man12] Steve Mansfield-Devine. "Paranoid Android: just how insecure is the most popular mobile platform?" In: *Network Security* 2012.9 (2012), pp. 5–10. DOI: 10.1016/S1353-4858(12)70081-8.

[Mar+14] Daniel Martin et al. "I Will Prescribe You an App". In: *Proceedings of the 2014 Summer Simulation Multiconference*. SummerSim '14. San Diego, CA, USA: Society for Computer Simulation International, 2014, 58:1–58:8. URL: http://dl.acm.org/citation.cfm?id=2685617.2685675.

[McL14] Charles McLellan. *Enterprise mobility in 2014: App-ocalypse Now?* 2014. URL: http://www.zdnet.com/article/enterprise-mobility-in-2014-app-ocalypse-now (visited on 04/30/2016).

[ME15] T A Majchrzak and J Ernsting. "Reengineering an Approach to Model-Driven Development of Business Apps". In: *8th SIGSAND/PLAIS EuroSymposium 2015, Gdansk, Poland*. Danzig, Polen, 2015, pp. 15–31. DOI: 10.1007/978-3-319-24366-5_2.

[MEK15] T A Majchrzak, J Ernsting, and H Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14.

[MH14] T A Majchrzak and H Heitkötter. "Status Quo and Best Practices of App Development in Regional Companies". In: *Revised Selected Papers WEBIST 2013*. Ed. by Karl-Heinz Krempels and Alexander Stocker. Vol. 189. LNBIP. Springer, 2014, pp. 189–206.

[MKK95] T Mukhopadhyay, S Kekre, and S Kalathur. "Business value of information technology: a study of electronic data interchange". In: *MIS quarterly* 19.2 (1995), pp. 137–156. DOI: 10.1145/243350.243363. URL: http://www.jstor.org/stable/249685.

[Mob] MobileHTML5. *Mobile HTML5 compatibility*. URL: http://mobilehtml5.org/ (visited on 04/29/2016).

[MR15] Euler Horta Marinho and Rodolfo Ferreira Resende. "Native and Multiple Targeted Mobile Applications". In: *Computational Science and Its Applications ICCSA 2015: 15th International Conference*. Ed. by Osvaldo Gervasi et al. Springer International Publishing, 2015, pp. 544–558. DOI: 10.1007/978-3-319-21410-8_42.

[MWA15] T A Majchrzak, S Wolf, and P Abbassi. "Comparing the Capabilities of Mobile Platforms for Business App Development". In: *Information Systems: Development, Applications, Education: 8th SIGSAND/PLAIS EuroSymposium 2015*. Ed. by Stanislaw Wrycza. Cham: Springer International Publishing, 2015, pp. 70–88. DOI: 10.1007/978-3-319-24366-5_6.

[Ope07] Open Handset Alliance. *Industry Leaders Announce Open Platform for Mobile Devices*. 2007. URL: http://www.openhandsetalliance.com/press%7B%5C_%7D110507.html.

[Ora11] Oracle Corp. *JSR 68: J2ME Platform Specification*. 2011. URL: https://www.jcp.org/en/jsr/detail?id=68 (visited on 04/30/2016).

[OT12] Julian Ohrt and Volker Turau. "Cross-Platform Development Tools for Smartphone Applications". In: *IEEE Computer* 45.9 (2012), pp. 72–79.

[Pou+15] Key Pousttchi et al. "Introduction to the Special Issue on Mobile Commerce: Mobile Commerce Research Yesterday, Today, Tomorrow - What Remains to Be Done?" In: *Int. J. Electronic Commerce* 19.4 (2015), pp. 1–20. DOI: 10.1080/10864415.2015.1029351.

[RM14] Janessa Rivera and Rob van der Meulen. *Gartner Says Sales of Smartphones Grew 20 Percent in Third Quarter of 2014*. 2014. URL: http://www.gartner.com/newsroom/id/2944819 (visited on 04/30/2016).

[Sen+15] Suranga Seneviratne et al. "Early Detection of Spam Mobile Apps". In: *Proceedings of the 24th International Conference on World Wide Web*. WWW '15. New York, NY, USA: ACM, 2015, pp. 949–959. DOI: 10.1145/2736277.2741084.

[Sha+16] Venkatesh Shankar et al. "Mobile Shopper Marketing: Key Issues, Current Insights, and Future Research Avenues". In: *Journal of Interactive Marketing* 34 (2016), pp. 37–48. DOI: 10.1016/j.intmar.2016.03.002.

[SM16] Fanjuan Shi and Jean-luc Marini. "Can e-Commerce Recommender Systems be More Popular with Online Shoppers if they are Mood-aware?" In: *WEBIST* 2 (2016), pp. 173–180.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Chichester: John Wiley & Sons, 2006.

[TL14] David G. Taylor and Michael Levin. "Predicting Mobile App Usage for Purchasing and Information-Sharing". In: *International Journal of Retail & Distribution Management* 42.8 (2014), pp. 759–774. DOI: 10.1108/IJRDM-11-2012-0108.

[Voe+13] Markus Voelter et al. "DSL Engineering Designing, Implementing and Using Domain-Specific Languages". In: (2013), p. 558. URL: http://dslbook.org.

[Völ13] Markus Völter. *DSL engineering: Designing, implementing and using domain-specific languages*. Lexington, KY: CreateSpace Independent Publishing Platform, 2013.

[W3C14] W3C. *HTML5: A vocabulary and associated APIs for HTML and XHTML, 28. October 2014*. Tech. rep. W3C, 2014. URL: https://www.w3.org/TR/html5/single-page.html.

[Wan+13] Tielei Wang et al. "Jekyll on iOS: When Benign Apps Become Evil". In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 559–572.

[Weba] WebDSL. *WebDSL Repository*. URL: https://github.com/webdsl/webdsl (visited on 01/01/2016).

[Webb] WebRatio. *WebRatio*. URL: http://www.webratio.com/ (visited on 04/30/2016).

[Wei99] Mark Weiser. "The Computer for the 21st Century". In: *SIGMOBILE Mob. Comput. Commun. Rev.* 3.3 (1999), pp. 3–11. DOI: 10.1145/329124.329126.

[WM10] Joel West and Michael Mace. "Browsing as the killer app: Explaining the rapid success of Apple's iPhone". In: *Telecommunications Policy* 34.5–6 (2010), pp. 270–286. DOI: 10.1016/j.telpol.2009.12.002.

[WM16] Viveca Woods and Rob van der Meulen. *Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015*. 2016. URL: http://www.gartner.com/newsroom/id/3215217 (visited on 04/30/2016).

[WTS07] M Walker, R Turnbull, and N Sim. "Future Mobile Devices: An Overview of Emerging Device Trends, and the Impact on Future Converged Services". In: *BT Technology Journal* 25.2 (Apr. 2007), pp. 120–125. DOI: 10.1007/s10550-007-0035-3.

[ZM13] Nan Zhong and Florian Michahelles. "Google play is not a long tail market: an empirical analysis of app adoption on the Google play app market". In: *Proceedings of the 28th Annual ACM* (2013), pp. 499–504. DOI: 10.1145/2480362.2480460. URL: http://dl.acm.org/citation.cfm?id=2480460.

26

A Data Model Inference Algorithm for Schemaless Process Modeling

Table 26.1: Fact sheet for publication P20

Title	A Data Model Inference Algorithm for Schemaless Process Modeling
Authors	Christoph Rieger[1] [1] *ERCIS, University of Münster, Münster, Germany*
Publication Date	2016
Publication Outlet	Working Papers, European Research Center for Information Systems No. 29
Copyright	ERCIS, University of Münster
Full Citation	Christoph Rieger. "A Data Model Inference Algorithm for Schemaless Process Modelling". In: *Working Papers, European Research Center for Information Systems No. 29*. Ed. by Jörg Becker, Klaus Backhaus, Martin Dugas, Bernd Hellingrath, Thomas Hoeren, Stefan Klein, Herbert Kuchen, Ulrich Müller-Funk, Heike Trautmann, and Gottfried Vossen. Münster: ERCIS, University of Münster, 2016, pp. 1–17

A Data Model Inference Algorithm for Schemaless Process Modeling

Christoph Rieger

Keywords: Graphical DSL, Mobile Application, Business App, Model-driven software development, Data model inference

Abstract: Mobile devices have become ubiquitous not only in the consumer domain but also support the digitalization of business operations though business apps. Many frameworks for *programming* cross-platform apps have been proposed, but only few *modeling* approaches exist that focus on platform-agnostic representations of mobile apps. In addition, app development activities are almost exclusively performed by software developers, while domain experts are rarely involved in the actual app creation beyond requirements engineering phases.

This work concentrates on a model-driven approach to app development that is also comprehensible to non-technical users. With the help of a graphical domain-specific language, data model, view representation, business logic, and user interactions are modeled in a common model from a process perspective. To enable such an approach from a technical point of view, an inference mechanism is presented that merges multiple partial data models into a global specification. Through model transformations, native business apps can then be generated for multiple platforms without manual programming.

26.1 Introduction

Mobile devices have become ubiquitous not only in the consumer domain but also support the digitization of business operations [Rv14]. In particular since the advent of Apple's first iPhone in 2007 [App07], a trend towards small-scale applications for specific use cases, so-called apps, has emerged. Whereas apps are now *used* in various consumer and business contexts, app *development* is still a task exclusively executed by programmers. Other stakeholders and future users are involved primarily in requirements engineering phases upfront implementation, following established software engineering methodologies.

However, Gartner predicts that in the next two years, more than half of all business apps for company-internal purposes will be created using codeless tools [Rv14]. One approach to codeless app creation is model-driven software development. Modeling such apps can be achieved using two kinds of notations: On the one hand, a wide variety of general purpose process modeling notations such as Business Process Model and Notation (BPMN) or Event-driven Process Chains (EPC) exists [Obj11; van99]. Usually, those models cannot be directly transformed into mobile apps because of lacking mobile-specific details and semantics. From a technical perspective, these notations often just represent connected, non-interpretable boxes filled with text. On the other hand, technical notations such as the Interaction Flow Modeling Language (IFML) are too complex

to understand for domain experts and require software engineering knowledge [Obj15a; Żył15]. In addition, to adequately model mobile apps, a suitable notation needs to be platform-independent to cover the variety of platforms and device types.

Many frameworks for *programming* cross-platform apps have been proposed [El-+15; Dal+13], but only few *modeling* approaches exist that focus on platform-agnostic representations of mobile apps. Existing commercial platforms provide cross-platform capabilities, but usually limited to source code transformations or partly supported by graphical editors for designing individual views (e.g. [Esp16]). Model-driven software development [SV06] instead utilizes app models as input for a partly or fully automated creation of apps. Various textual domain-specific languages (DSL) [MHS05] such as Mobl, AXIOM, and MD²follow this approach [HV11; JJ15; HM13]. DSLs are suited to cover a well-defined scope, the so-called domain, and model inherent domain concepts on a more abstract level. However, a textual representation provides only minor benefits to non-technical users. Textual modeling is potentially more concise but still feels like programming [ZS09]. Yet, input from stakeholders with strong domain knowledge is essential to ensure the developed software matches their tacit requirements [BKF14].

The Münster App Modeling Language (MAML; pronounced 'mammal') framework aims to alleviate the aforementioned problems. This paper's contributions are twofold: First, a model-driven approach to app development is presented which is also comprehensible for non-technical users. With the help of a graphical domain-specific language, data model, view representation, business logic, and user interactions are defined in a common model from a process perspective. Through model transformations, native business apps can then be generated for multiple platforms without manual programming. Second, to enable such an approach from a technical point of view, an inference mechanism is required that merges multiple partial data models into a global specification. Consequently, the modeler is disburdened from explicit data model specifications that need to be maintained separately and require knowledge about the application as a whole.

The structure of the paper follows these contributions. After presenting related work in Section 26.2, the proposed framework is presented in Section 26.3. Section 26.4 explains how to infer a data model from a set of MAML models. Finally, this report concludes in Section 26.5 and gives an outlook on future work.

26.2 Related Work

The work presented in this paper draws on the scientific fields of cross-platform mobile app generation, domain-specific visual languages, and schema matching. Regarding cross-platform mobile apps, different approaches exist that can be grouped into three major categories according to El-Kassas [El-+15]: Existing source code can be *compiled* from a legacy application or different platform, a runtime or virtual machine can *interpret* a common code base, or app source code can be generated in a *model-driven* way from a textual or graphical representation. With regard to model-driven approaches, a variety of frameworks can be found in academic literature [UB16].

Only few of them, such as Mobl [HV11] and AXIOM [JJ15], set out to cover the full spectrum of structure and runtime behavior of an app; often providing a custom textual DSL for this means. This paper builds on previous work on the MD²framework which focuses on the generation of business apps, i.e., form-based, data-driven apps interacting with back-end systems according to Majchrzak et al. [MEK15]. The input model is specified using a platform-independent, textual DSL [HM13]. After preprocessing, code generators transform it into platform-native source code [ME15; Ern+16]. Despite several development-related improvements in the past years, textual DSLs are generally often perceived as programming to non-technical users [Żył15]. This barrier to a wide-spread usage of textual DSLs motivates the need for further abstraction and visual modeling.

Graphical DSLs or visual programming languages exist for several purposes, including process-related domains such as business process compliance [Knu+13] and data integration [Pen16]. Regarding mobile applications, RAPPT represents a model-driven approach that mixes a graphical DSL for process flows with a textual DSL for programming [Bar+15]; AppInventor encourages novices to create apps by combining building blocks of a visual programming language [Wol11]. Puzzle takes development of mobile applications one step further by providing a visual development environment on the mobile device itself [DP12]. Although all of these approaches aim at simplifying the actual programming tasks for novice developers, they disregard non-technical stakeholders.

In contrast, general purpose modeling notations exist to describe applications and process flows. The Unified Modeling Language (UML) for the domain of software development provides several standards to define the structure and runtime behavior of an application [Obj15b]. One of them, IFML (succeeding WebML), specifies user interactions within a software system [Obj15a]. Cognitive studies have been conducted, e.g. for WebML, and showed that technical modeling notations are often considered as complex to understand for domain experts. Problems such as symbol overload are further aggravated by the combined use of multiple notations in order to describe all behavioral and structural characteristics [Gra+15; Fra+06]. To visualize process flows in general, a variety of modeling notations exist, e.g. BPMN, EPC, or flowcharts [Obj11; van99; Int85]. However, such notations are often superficial regarding technical specificity, and mobile-related aspects are often not included because of their general applicability. Modeling approaches specific to the mobile domain rarely reach beyond user interface modeling. Some approaches explicitly try to incorporate non-technical users, e.g., through collaborative multi-viewpoint modeling [FMM14]. Others use existing modeling notations such as statecharts [Fra+15] or extend them for mobile purposes, e.g. UML to model context in mobile distributed systems [SW07], IFML with mobile-specific elements [BMU14], or BPMN to orchestrate web services [BDF09].

Schema matching and model differentiation are further relevant fields of research with various approaches regarding the identification of common structures in models [Sut+16]. According to the classification by Rahm [RB01], a schema-only and constraint-based approach on element level is required for the inference of a global data model from multiple input models. For the

given problem of partial models, inference can be limited to additive and name-based approaches. Enjo and Iijima presented related work on the UML composition of class diagrams [EI10]. More sophisticated strategies, e.g. relying on ontologies, might also identify modeling inconsistencies beyond strict name-based matching [LGJ07]. An application related to mobile devices, but focused on the visualization of data instead of its manipulation, is MobiMash for graphically creating mashup apps by configuring the representation and orchestration of data services [Cap+12].

In the context of meta modeling, reverse engineering approaches to track meta model evolution deal with similar problems of inferring object structures [Liu+12], and López-Fernández et al. [Lóp+15] presented a related idea in order to derive a common meta model from exemplary model fragments.

The commercial WebRatio Mobile Platform is closest to the work presented in this paper as it can also generate apps from a graphically edited model [Ace+15]. However, it has a high entry barrier for potential non-technical users by relying on the combination of IFML and additional notations. Also, it does not provide modularized app models with extended mechanisms for inferring data structures as presented in Section 26.4. With similar ambitions, companies such as BiznessApps and Bubble promise codeless creation of apps using detailed configurators and web-based user interface editors [Biz16; Bub16]. This work in contrast focuses on a process-centric and platform-agnostic approach as described next, with a higher level of abstraction than simple drag-and-drop configuration tools.

26.3 Münster App Modeling Language

The MAML DSL is built around five main principles:

- Domain expert focus: In contrast to technical specification languages, MAML is designed with a non-technical user in mind; regarding both the actual models and the modeling environment. Therefore, process modelers or domain experts without software development experience should be able to understand, create, and modify models without longsome training.
- Data-driven process: MAML models represent a process perspective on business apps, visualizing the flow of data objects through a sequence of processing steps. Compared to other process modeling notations, the content and structure of these data objects are explicated in the model.
- Modularization: The scope of a model is one *Use Case*, a unit of useful functionality comprising a self-contained set of behaviors and interactions performed by the app user [Obj15b]. To support the domain expert focus, MAML combines data model, process flow, app visualization, and user interactions in a single model. This opposes software engineering patterns such as Model-View-Controller [Gam+95] that separate such aspects for better maintainability of large-scale software.

- Declarative description: Use cases contain platform-agnostic elements describing *what* processing activities are possible on the data objects. However, the concrete representation on a mobile device is not further specified on this abstract level. During the generation phase, sensible defaults for platform-specific appearances are provided.
- Automatic cross-platform app generation: One major design goal of the MAML framework is its capability to create fully-functional software for multiple platforms in order to reach a large amount of users. The graphical model is therefore designed to be interpretable by different code generators without further need for manual programming. As a result, MAML provides the means for codeless development of business apps.

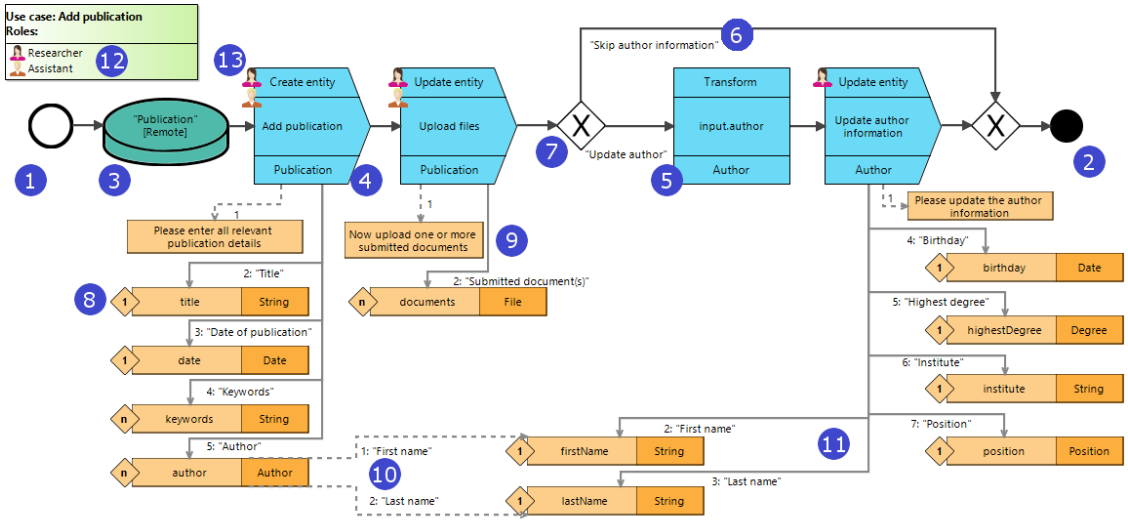


Figure 26.1: Sample MAML Use Case "Add publication"

Figure 26.1 depicts a sample *use case* for adding an item to a publication management system (enriched with numbered circles for reference in the following paragraphs). The model contains a sequence of activities, from a *start event* (1) towards one or several *end events* (2). In the beginning, a *local* or *remote data source* (3) specifies the data type of the manipulated objects. Figure 26.1 depicts the data type "Publication" to be manipulated within the process steps and which is managed by a remote server. The modeler can then choose from a variety of (arrow-shaped) *interaction process elements* (4), for example to *select/create/update/display/delete entities*, show *popup messages*, or access device functionalities such as the *camera* and starting a phone *call*. Due to its declarative nature, the DSL does not indicate the concrete appearance but typically each logical process step may be rendered as one view of the app. Furthermore, *automated process elements* (5) represent invisible processing steps without user interaction, e.g. calling RESTful *web services*, *including* other models for process modularization, or navigating through the object structure (*transform*).

The navigation between connected process steps happens using an automatically created "Continue" button. Alternatively, a distinct denomination can be specified along the *process connectors* (6). To allow for conditional actions, the process flow can be branched out using an *XOR* element (7). The condition can either be triggered automatically by attaching an attribute to the XOR element and evaluating specified expressions, or requires a user decision by providing respective button captions along the process connectors (such as in the example). To sum up, the exemplary use case first creates a new entity of type "Publication", then edits additional data before optionally traversing the object structure to the publication's author and editing his/her data.

The rectangular elements (bottom half of Figure 26.1) represent the data structure which is relevant for a particular process. Every process element needs to specify all (and only those) attributes which are displayed on the screen or utilized in automated processing activities. *Attributes* (8) consist of a cardinality indicator, a name and the respective data type. Besides pre-defined data types such as *String, Integer, Float, PhoneNumber, Location* etc., arbitrary custom types can be defined. Consequently, attributes may be nested over multiple levels in order to further describe the content of such a custom data type. *Labels*, depicted as rectangle without cardinality and type information (9), can be added to display explanatory text on screen, and *computed attributes* (not illustrated) may be used to output calculations on other attributes at runtime. To assign these UI elements to a process step, two types of connectors exist: Dotted arrows (10) represent a *consuming relationship* whereas solid arrows (11) signify a *modifying relationship* regarding the target element. This refers not only to the manifest representation of attribute content displayed either as read-only text or editable input field. The interpretation also applies in a wider sense, e.g. web service calls in which the server "reads" an input parameter and "modifies" information through its response.

Every connector which is connected to an interaction process element also specifies an order of appearance. Additionally, a human-readable representation of the field description is derived from the attribute name unless specified manually (10). To reduce the amount of elements to be modeled, multiple connectors may point to the same UI element from different sources (given their data types match). Alternatively, to avoid wide-spread connections across larger models, UI elements may instead be duplicated to different positions in the model and will automatically be recognized as being the same element (see Section 26.4).

Finally, the MAML DSL supports a multi-role concept. The modeler can specify role names (12) and annotate them to the respective interaction process elements (13). This is particularly useful to describe scenarios in which parts of the process are performed by different people, e.g. approval workflows. If the assigned role changes, the process automatically terminates for the first app user, modified data objects are saved, and the subsequent user is informed about an open workflow instance in his app. The exemplary use case of Figure 26.1, the publication might be added by a researcher or assistant, however, the author information can only be changed by a researcher.

26.4 Data Model Inference Algorithm

The most important advantage of MAML's approach is the renunciation of a global data schema that needs to be modeled and maintained separately. Instead, each process step refers (just) to the attributes which are required (i.e. displayed or edited) within this particular step. As result of the modeling activities, only partial models exist that need to be matched on multiple levels: for each process element individually, for the use case as a whole, and across multiple use cases of the overall app.

26.4.1 Partial Data Model Inference

First, separate data models need to be inferred for each process element. The challenge lies in the unidirectional specification of relationships, i.e. a MAML process element with a given data type "has a" relationship to an *attribute* of a specified type and cardinality but there is mostly no explicit information on the opposite relationship.

Let M denote the set of use case models for which a coherent data model should be inferred. Also, D_m denotes the set of data types within a concrete model $m \in M$. Then,

$$R_m \subset D_m \times String \times D_m \times \mathscr{P}(\mathbb{N}) \times \mathscr{P}(\mathbb{N})$$

denotes the set of *relationship* tuples between two distinct data types. Within such a tuple $r_i = (s_i, rn_i, t_i, sc_i, tc_i)$, the value s_i represents the source data type of a relationship, rn_i the corresponding name, and t_i the target data type. A *cardinality* represents the possible number of values referred to by a relationship [Obj15b], using the notation $i..j = \{i, i+1, ..., j\}$ and n represents infinity. sc_i signifies the source cardinality of the relationship and rc_i the respective target cardinality.

Then, the annotated directed graph $G_m := (D_m, R_m)$ represents the data model of m with data types as vertices and relationships as edges. In particular, the graph may contain multiple edges between the same pair of source and target data types, if the relationship names, the source cardinalities, or the target cardinalities differ. Whereas the former is a valid modeling option (e.g., lectures are held by teachers and attended by students which are both of type "person"), the differing cardinalities can be considered as modeling error (cf. Section 26.4.3).

In addition, let V_m denote the set of primitive types within a concrete model $m \in M$, meaning atomic values which do not contain any relationships to other data types. Then,

$$P_m \subset D_m \times String \times V_m \times \mathscr{P}(\mathbb{N})$$

represents the set of *property* tuples. Accordingly, for a tuple $p_i = (ps_i, pn_i, pt_i, pc_i)$ the value ps_i represents the source data type that contains a property of the primitive type pt_i with the name pn_i and cardinality ps_i.

To apply the above definitions to MAML, a data type is specified either within the data source element, a process element, or as part of an attribute element. For example, Figure 26.2 depicts the visual representations of a remote data source referring to the data type ”Publication” (a), an update entity interaction process element for the same data type (b) as well as an attribute for the data type ”Author” (c). In addition, MAML provides a pre-defined list of primitive types (integer, floating point number, string, location, boolean, date, time, datetime, and file) from which these data types can be composed.

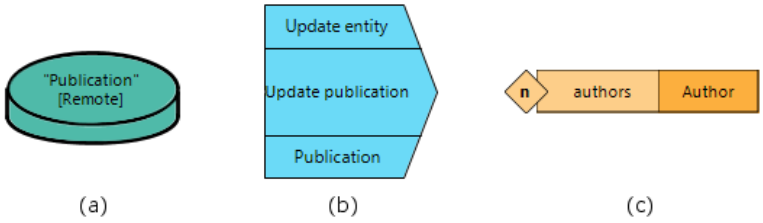


Figure 26.2: Specification Options for Data Types in MAML

In order to identify interrelations between data types, three origins need to be considered (cf. numbered circles in Figure 26.3): First, a relationship may exist between a process element and an attached attribute. Second, a nested attribute adds a relationship to the nesting attribute’s data type. Third, an attribute may be transitively connected to the process element through one or more computed attributes (e.g., (3) in Figure 26.3 representing a ”count” aggregation operator), but still refers to the process element’s data type (unless it has other incoming attribute connectors). MAML does not differentiate between primitive types and data types with regard to their representation in the model. Therefore, the aforementioned options might translate to either a *relationship* or *property* of the originating data type.

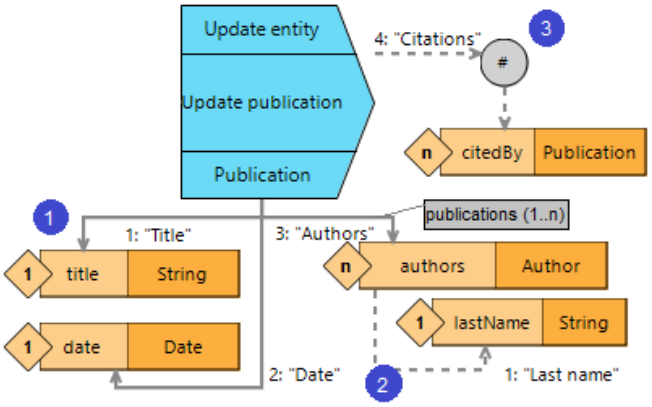


Figure 26.3: Attribute Relation Options in MAML Models

The cardinalities which can initially be assigned to a relationship depend on the type of association. Four main cases can be distinguished:

- Primitive: MAML attributes with primitive data types are trivially converted to single- or multi-valued *properties* of the source data type. For example, the connection (2) in Figure 26.3 translates to the tuple $(Author, "lastName", String, 0..1)$ in P_m [1].
- Unidirectional: Typically, modeled *relationships* are unidirectional and can therefore be modeled only in one direction. Each relationship explicitly specifies a name and cardinality. Because multiple objects of the source data type might reference the same target object, the unspecified source cardinality is unknown and must be interpreted as unrestricted with $0..n$. In the example, the connection (3) translates to the tuple $(Publication, "citedBy", Publication, 0..n, 0..n)$ in R_m.
- Bidirectional: In contrast to the previous case, bidirectional relationships between data types d_1 and d_2 are fully specified in the graphical model and provide both source and target cardinalities. In MAML, this is represented by an additional annotation (containing the name and cardinality for the opposite direction) along the connecting arrow, e.g., the "authors" relationship in Figure 26.3. To capture the navigability in both directions – and particularly the respective attribute names – in the graph, a second relationship is inserted with inverted order of data types and cardinalities.
- Singleton: Relationships originating from singleton data types are a variant of the unidirectional scenario in which the unknown cardinality can be restricted to 0..1 (a maximum of one object can be set) [Gam+95]. In MAML, singleton data types are created when using the singleton data source element within the process flow.

Two models are considered *compatible*, if the combined constraints of both models for data type, name, and cardinality consistency are satisfiable (cf. Section 26.4.3). As an example, Figure 26.4 depicts two compatible MAML models and Listing 3 shows the corresponding graph structure.

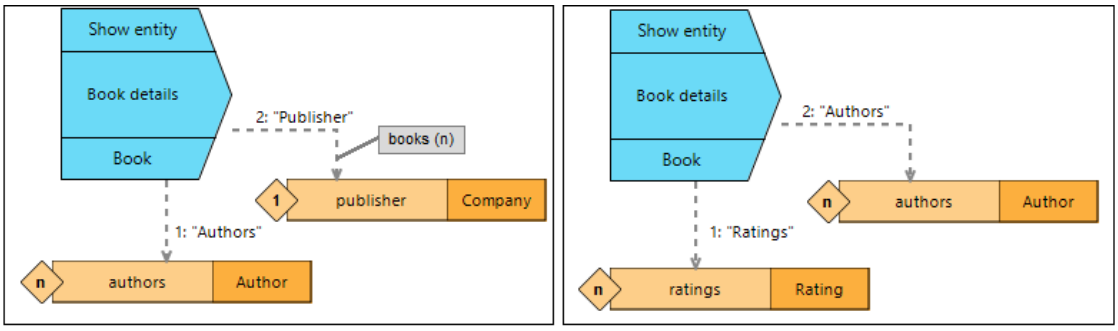


Figure 26.4: Compatible Partial MAML Models

[1] In MAML models, **1** specifies a 0..1 cardinality and **n** refers to $0..n$.

Listing 3 Exemplary data structure for two compatible models

$M = \{m_1, m_2\}$
$D_{m_1} = \{Book, Author, Company\}$ ▷ Data types
$D_{m_2} = \{Book, Rating, Author\}$
$V_{m_1} = V_{m_2} = \emptyset$

$P_{m_1} = P_{m_2} = \emptyset$ ▷ Properties

$r_1 = (Book, "authors", Author, 0..n, 0..n)$ ▷ Relationships
$r_2 = (Book, "publisher", Company, 0..n, 0..1)$
$r_3 = (Company, "books", Book, 0..1, 0..n)$
$R_{m_1} = \{r_1, r_2, r_3\}$

$r_4 = (Book, "ratings", Rating, 0..n, 0..n)$
$r_5 = (Book, "authors", Author, 0..n, 0..n)$
$R_{m_2} = \{r_4, r_5\}$

26.4.2 Merging Partial Data Models

By using an associative and commutative merging operation, all partial models can be merged iteratively or simultaneously into a single global data model G_g before identifying modeling inconsistencies as described in Section 26.4.3.

First, all distinct data types and primitive types can be aggregated from the considered source models directly:

$$D_g = \bigcup_{m \in M} D_m \tag{26.1}$$

$$V_g = \bigcup_{m \in M} V_m \tag{26.2}$$

Second, relationships from each model m are added to R_g if there is yet no relationship between both data types, or (given that source and target data type match) the name or source cardinality, or target cardinality of the relationship differs. Due to the representation of R_m as set of tuples, this boils down to the union of all relationship sets of the source models:

$$R_g = \bigcup_{m \in M} R_m \tag{26.3}$$

Properties of the source models are likewise merged:

$$P_g = \bigcup_{m \in M} P_m \tag{26.4}$$

Applied to the example of Figure 26.4, the relationship r_5 is equivalent to r_1 and therefore ignored. The resulting graph structure is depicted in Listing 4 and the corresponding UML class diagram in Figure 26.5.

Listing 4 Exemplary merged data structure of two compatible models

$D_g = \{Book, Author, Company, Rating\}$
$V_g = \emptyset$

$P_g = \emptyset$
$r_1 = (Book, "authors", Author, 0..n, 0..n)$
$r_2 = (Book, "publisher", Company, 0..n, 0..1)$
$r_3 = (Company, "books", Book, 0..1, 0..n)$
$r_4 = (Book, "ratings", Rating, 0..n, 0..n)$
$R_g = \{r_1, r_2, r_3, r_4\}$

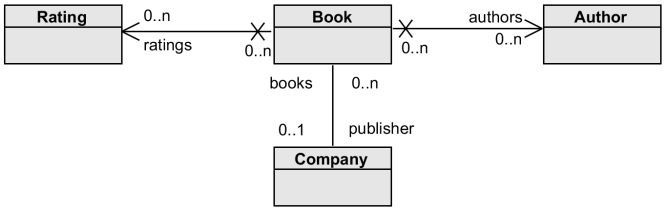


Figure 26.5: UML Class Diagram of the Merged Model

26.4.3 Consolidation and Error Identification

Whereas the previous merging step has aggregated all partial models, the resulting global data model might be invalid. For example, both partial models in Figure 26.6 are valid on their own and can be merged according to equations (26.1) to (26.4). However, they are called *conflicting* because merging those models results in a global model that is semantically invalid. Generally, two types of modeling errors can be observed:

- First, a *type error* exists if any source data type in the graph has two properties or relationships of the same name pointing to different target data types, i.e. not conforming to

$$\forall r_i, r_j \in R_g \mid s_i = s_j, \;\; rn_i = rn_j : \qquad t_i = t_j \tag{26.5}$$

$$\forall p_i, p_j \in P_g \mid ps_i = ps_j, \;\; pn_i = pn_j : \qquad pt_i = pt_j \tag{26.6}$$

- Second, a *name conflict* exists if the same name is assigned more than once to relationships and properties for the same source data type, violating

$$\forall r_i \in R_g, p_j \in P_g \mid s_i = ps_j : \qquad rn_i \neq pn_j \tag{26.7}$$

- Third, a *cardinality conflict* exists if two *relationships* with the same name differ with regard to their cardinalities for any pair of data types, i.e. violating any of

$$\forall r_i, r_j \in R_g \mid s_i = s_j, \;\; rn_i = rn_j, \;\; t_i = t_j : \qquad sc_i = sc_j \tag{26.8}$$

$$\forall r_i, r_j \in R_g \mid s_i = s_j,\ \ rn_i = rn_j,\ \ t_i = t_j : \qquad tc_i = tc_j \tag{26.9}$$

A cardinality conflict also exists if two *properties* with the same name differ with regard to their cardinalities for any pair of primitive types, i.e. not conforming to

$$\forall p_i, p_j \in P_g \mid ps_i = ps_j,\ \ pn_i = pn_j,\ \ pt_i = pt_j : \qquad pc_i = pc_j \tag{26.10}$$

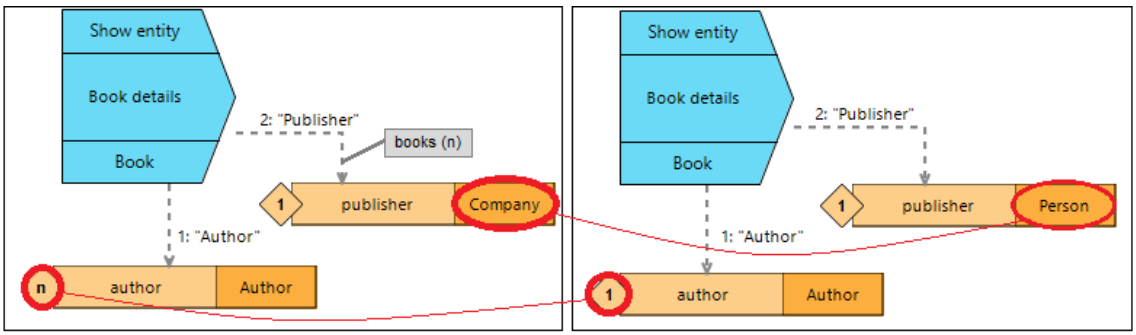


Figure 26.6: Conflicting Partial MAML Models

Type errors and name conflicts according to (26.5) to (26.7) cannot be resolved automatically and need to be corrected by the modeler. For instance in Figure 26.6, the inference mechanism cannot decide whether the ambiguous target data type of the ”publisher” attribute should be set to ”Company” or ”Person”.

In case of cardinality conflicts violating (26.8) to (26.10), the modeler should be warned, but automatic resolution is possible. For each pair of relationships $r_i, r_j \in R_g$ with matching source data type, target data type, and name, the cardinality for each side of the association can be calculated as the intersection between the conflicting cardinalities. The cardinality is calculated accordingly for each pair of *properties* $p_i, p_j \in P_g$ with matching source data type, target data type, and name. For the example of Figure 26.6, the target cardinality $0..n \cap 0..1 = 0..1$ is assigned to the ”author” attribute. Although bidirectional relationships are modeled as two separate tuples, this causes no harm because the resolved cardinality satisfies all constraints.

As a result, the inference algorithm can be applied to partial models of different granularity (first within a single MAML model, then across models) in arbitrary order, and can both serve to validate model correctness and derive a global data model required for software generation and database schema creation.

26.5 Conclusion and Outlook

In this work, the MAML framework was introduced to alleviate the problems of programmer-focused mobile app development. In future, apps can instead be modeled codelessly by domain experts using a declarative graphical DSL. However, as opposed to visual configuration tools or

low-level GUI editors to specify the position of user interface elements on a screen canvas, MAML focuses on a process-centric definition of apps. Using abstract and platform-agnostic process elements, it hence aligns with the business perspective of managing processes and data flows.

In particular, a data model inference mechanism was presented that enables a multi-level aggregation of partial data models into a global data schema. In addition, the combined model allows for real-time validation and consistency checks due to formal constraints on data type and cardinality consistency. This inference mechanism is essential for implementing model-driven techniques that require globally specified data models. MAML therefore achieves the desired balance of abstracting programming-heavy tasks to process flows of moderate complexity while keeping the technical expressiveness required for automatic source code generation.

Ongoing work focuses on an empirical evaluation to support the advantage of MAML over the related technical IFML notation, specifically with regard to its understandability by domain experts. Also, the MAML editor is further developed in order to feed back information from the inferred global data model into the modeling environment and provide the modeler with improved features for naming suggestions and real-time checks within the individual use case models.

Concerning a more general aspect that constitutes future work, applying the prototype to real-world problems might reveal further need for improvements. For example, data flow variations are currently limited to XOR elements due to the sequential proceeding on smartphone displays but might be extended if requested by practitioners. Furthermore, the platform-agnostic principle of MAML allows for its application to mobile devices beyond smartphones and tablets which opens up new possibilities for integrating business apps in everyday work practices. Regarding the emergence of novel devices such as smart watches, interesting questions arise concerning best practices for modeling and implementing apps on such devices with different input/output capabilities and user interaction patterns. Finally, it might be investigated to which extent previously existing process documentation can be reused to simplify app creation. For example, model to model transformations from/to other process modeling notations such as BPMN could be used to convert existing models into MAML use cases and just enrich them with missing pieces of information such as data objects.

References

[Ace+15] Roberto Acerbis et al. “Model-driven Development of Cross-platform Mobile Applications with WebRatio and IFML”. In: *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. MOBILESoft ’15. IEEE Press, 2015, pp. 170–171.

[App07] Apple Inc. *Apple Reinvents the Phone with iPhone*. 2007. URL: http://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html (visited on 09/23/2016).

[Bar+15] Scott Barnett et al. "A Multi-view Framework for Generating Mobile Apps". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 305–306. DOI: 10.1109/VLHCC.2015.7357239.

[BDF09] M. Brambilla, M. Dosmi, and P. Fraternali. "Model-driven engineering of service orchestrations". In: *SERVICES 2009 - 5th 2009 World Congress on Services* (2009). DOI: 10.1109/SERVICES-I.2009.94.

[Biz16] Bizness Apps. *Mobile App Maker | Bizness Apps*. 2016. URL: http://biznessapps.com/ (visited on 09/23/2016).

[BKF14] R. Breu, A. Kuntzmann-Combelles, and M. Felderer. "New Perspectives on Software Quality". In: *IEEE Software* 31.1 (2014), pp. 32–38. DOI: 10.1109/MS.2014.9.

[BMU14] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. "Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End". In: *Lecture Notes in Computer Science* 8640 (2014), pp. 176–191. DOI: 10.1007/978-3-319-10359-4_15.

[Bub16] Bubble Group. *Bubble - Visual Programming*. 2016. URL: https://bubble.is/ (visited on 09/23/2016).

[Cap+12] Cinzia Cappiello et al. "MobiMash: End User Development for Mobile Mashups". In: *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web Companion* (2012), pp. 473–474. DOI: 10.1145/2187980.2188083.

[Dal+13] I. Dalmasso et al. "Survey, comparison and evaluation of cross platform mobile application development tools". In: *2013 9th International Wireless Communications and Mobile Computing Conference, IWCMC 2013* (2013). DOI: 10.1109/IWCMC.2013.6583580.

[DP12] Jose Danado and Fabio Paternò. "Puzzle: A Visual-Based Environment for End User Development in Touch-Based Mobile Phones". In: *Human-Centered Software Engineering: 4th International Conference, HCSE 2012*. Ed. by Marco Winckler, Peter Forbrig, and Regina Bernhaupt. Springer Berlin Heidelberg, 2012, pp. 199–216. DOI: 10.1007/978-3-642-34347-6_12. URL: http://dx.doi.org/10.1007/978-3-642-34347-6_12.

[EI10] Hidekazu Enjo and Junichi Iijima. "Towards Class Diagram Algebra for Composing Data Models". In: *Proceedings of the 2010 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the 9th SoMeT_10*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2010, pp. 112–133.

[El-+15] Wafaa S. El-Kassas et al. "Taxonomy of Cross-Platform Mobile Applications Development Approaches". In: *Ain Shams Engineering Journal* (2015). DOI: 10.1016/j.asej.2015.08.004.

[Ern+16] Jan Ernsting et al. "Refining a Reference Architecture for Model-Driven Business Apps". In: *Proceedings of the 12th International Conference on Web Information Systems and Technologies (WEBIST 2016)* (2016), pp. 307–316. DOI: 10.5220/0005862103070316.

[Esp16] David Esperalta. *DecSoft - App Builder*. 2016. URL: https://www.davidesperalta.com/appbuilder (visited on 09/08/2016).

[FMM14] Mirco Franzago, Henry Muccini, and Ivano Malavolta. "Towards a Collaborative Framework for the Design and Development of Data-intensive Mobile Applications". In: *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. MOBILESoft 2014. ACM, 2014, pp. 58–61. DOI: 10.1145/2593902.2593917.

[Fra+06] R. B. France et al. "Model-Driven Development Using UML 2.0: Promises and Pitfalls". In: *Computer* 39.2 (2006), pp. 59–66. DOI: 10.1109/MC.2006.65.

[Fra+15] Rita Francese et al. "Model-Driven Development for Multi-platform Mobile Applications". In: *Product-Focused Software Process Improvement: 16th International Conference, PROFES 2015*. Ed. by Pekka Abrahamsson et al. Springer Intl. Publishing, 2015, pp. 61–67. DOI: 10.1007/978-3-319-26844-6_5.

[Gam+95] Erich Gamma et al. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995.

[Gra+15] David Granada et al. "Analysing the cognitive effectiveness of the WebML visual notation". In: *Software & Systems Modeling* (2015). DOI: 10.1007/s10270-014-0447-8.

[HM13] H. Heitkötter and T. A. Majchrzak. "Cross-Platform Development of Business Apps with MD^2". In: *Proc. of the 8th Int. Conf. on Design Science at the Intersection of Physical and Virtual Design (DESRIST)*. Vol. 7939. LNBIP. Springer, 2013, pp. 405–411.

[HV11] Zef Hemel and Eelco Visser. "Declaratively Programming the Mobile Web with Mobl". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. ACM, 2011, pp. 695–712. DOI: 10.1145/2048066.2048121.

[Int85] International Organization for Standardization. *ISO 5807:1985*. 1985.

[JJ15] Christopher Jones and Xiaoping Jia. "Using a Domain Specific Language for Lightweight Model-Driven Development". In: *ENASE 2014, CCIS 551*. 2015, pp. 46–62. DOI: 10.1007/978-3-642-23391-3.

[Knu+13] David Knuplesch et al. "Visual Modeling of Business Process Compliance Rules with the Support of Multiple Perspectives". In: *Conceptual Modeling: 32th International Conference, ER 2013*. Ed. by Wilfred Ng, Veda C. Storey, and Juan C. Trujillo. Springer Berlin Heidelberg, 2013, pp. 106–120. DOI: 10.1007/978-3-642-41924-9_10.

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. "DSMDiff: a differentiation tool for domain-specific models". In: *European Journal of Information Systems* 16.4 (2007), pp. 349–361. DOI: 10.1057/palgrave.ejis.3000685.

[Liu+12] Qichao Liu et al. "Application of Metamodel Inference with Large-Scale Metamodels". In: *International Journal of Software and Informatics* 6.2 (2012), pp. 201–231.

[Lóp+15] Jesús J. López-Fernández et al. "Example-driven meta-model development". In: *Software & Systems Modeling* 14.4 (2015), pp. 1323–1347. DOI: 10.1007/s10270-013-0392-y.

[ME15] T. A. Majchrzak and J. Ernsting. "Reengineering an Approach to Model-Driven Development of Business Apps". In: *8th SIGSAND/PLAIS EuroSymposium 2015*. 2015, pp. 15–31. DOI: 10.1007/978-3-319-24366-5_2.

[MEK15] T. A. Majchrzak, J. Ernsting, and H. Kuchen. "Achieving Business Practicability of Model-Driven Cross-Platform Apps". In: *OJIS* 2.2 (2015), pp. 3–14.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[Obj11] Object Management Group. *Business Process Model and Notation, Version 2.0*. 2011.

[Obj15a] Object Management Group. *Interaction Flow Modeling Language, Version 1.0*. 2015.

[Obj15b] Object Management Group. *Unified Modeling Language, Version 2.5*. 2015. URL: http://www.omg.org/spec/UML/2.5.

[Pen16] Pentaho Corp. *Data Integration - Kettle*. 2016. URL: http://pentaho.com/product/data-integration (visited on 09/12/2016).

[RB01] E. Rahm and P. A. Bernstein. "A survey of approaches to automatic schema matching". In: *VLDB Journal* 10.4 (2001), pp. 334–350. DOI: 10.1007/s007780100057.

[Rie16] Christoph Rieger. "A Data Model Inference Algorithm for Schemaless Process Modelling". In: *Working Papers, European Research Center for Information Systems No. 29*. Ed. by Jörg Becker et al. Münster: ERCIS, University of Münster, 2016, pp. 1–17.

[Rv14] Janessa Rivera and Rob van der Meulen. *Gartner Says By 2018, More Than 50 Percent of Users Will Use a Tablet or Smartphone First for All Online Activities*. 2014. URL: http://www.gartner.com/newsroom/id/2939217 (visited on 09/23/2016).

[Sut+16] E. Sutanta et al. "Survey: Models and prototypes of schema matching". In: *International Journal of Electrical and Computer Engineering* 6.3 (2016), pp. 1011–1022. DOI: 10.11591/ijece.v6i3.9789.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Chichester: John Wiley & Sons, 2006.

[SW07] C. Simons and G. Wirtz. "Modeling context in mobile distributed systems with the UML". In: *Journal of Visual Languages and Computing* 18.4 (2007), pp. 420–439. DOI: 10.1016/j.jvlc.2007.07.001.

[UB16] Eric Umuhoza and Marco Brambilla. "Model Driven Development Approaches for Mobile Applications: A Survey". In: *Mobile Web and Intelligent Information Systems: 13th International Conference, MobiWIS 2016*. Ed. by Muhammad Younas et al. Springer Intl. Publishing, 2016, pp. 93–107. DOI: 10.1007/978-3-319-44215-0_8.

[van99] W.M.P. van der Aalst. "Formalization and verification of event-driven process chains". In: *Information and Software Technology* 41.10 (1999), pp. 639–650. DOI: 10.1016/S0950-5849(99)00016-6.

[Wol11] D. Wolber. "App inventor and real-world motivation". In: *SIGCSE'11 - Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (2011). DOI: 10.1145/1953163.1953329.

[ZS09] Uwe Zdun and M. Strembeck. "Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development". In: *Proceedings of 14th European Conference on Pattern Languages of Programs (EuroPLoP 2009)*. 2009, pp. 1–37.

[Żył15] Kamil Żyła. "Perspectives of Simplified Graphical Domain-Specific Languages as Communication Tools in Developing Mobile Systems for Reporting Life-Threatening Situations". In: *Studies in Logic, Grammar and Rhetoric* 43.1 (2015). DOI: 10.1515/slgr-2015-0048.