BIELEFELD UNIVERSITY

DOCTORAL THESIS

MODELING ROBOT CONTROL SYSTEMS IN COMPLIANT INTERACTION WITH THE ENVIRONMENT

Bridging the gap between the envisioned task and the robot's behavior

DENNIS LEROY WIGAND

A doctoral thesis presented for the degree of
Doctor of Engineering (Dr.-Ing.)

in the

Research Institute for Cognition and Robotics
Faculty of Technology

December, 2021

MODELING ROBOT CONTROL SYSTEMS IN COMPLIANT INTERACTION WITH THE ENVIRONMENT

DENNIS LEROY WIGAND

Bridging the gap between the envisioned task and the robot's behavior

A doctoral thesis presented for the degree of
Doctor of Engineering (Dr.-Ing.) at

Research Institute for Cognition and Robotics
Faculty of Technology
Bielefeld University
Inspiration 1
33619 Bielefeld
Germany

DEFENDED AND APPROVED
December 20, 2021

Printed on permanent paper as per ISO 9706.

ABSTRACT

In recent years, robotics applications increasingly rely on physically compliant interaction, which entails the deliberate compensation and exploitation of contact forces. Explicitly considering the compliant interaction between the robot and the environment is essential for successful and safe task execution in contact-rich applications: e.g., snap-fitting electrical clamps, relying on the environment for support (e.g., using a hand-rail), and accommodating for soft materials (e.g., in soft tissue surgery). Developing robotic systems for such applications, requires not only suitable control algorithms, but also a task description that formalizes the compliant interaction as well as other relevant system concerns (e.g., timing). Skill-based approaches are commonly used to create such systems. However, they usually neglect the compliant interaction almost entirely and introduce hidden assumptions for other relevant concerns. This causes a significant gap between the envisioned task and the resulting robot's behavior in terms of explainability and predictability.

To close the gap, this thesis introduces suitable abstractions to model the compliant interaction in the context of a task description as constraints on the robot's behavior. The constraints are based on the physical exchange of forces via (natural) contacts. Using the developed synthesis, the modeled task constraints for compliant interaction are directly transferred into a control system model that uses the Projected Inverse Dynamics Control formalism.

A modularization and composition approach for domain-specific languages is developed to combine the resulting control system model with relevant functional and non-functional robotics concerns, such as the specification of the execution time behavior, which are essential to produce a predictable behavior. The implementation of the conceptual approach allows the modeling of compliant interaction tasks, the synthesis of a suitable robot control system, and the generation of a real-time software system that can be executed in simulation as well as on the real robot. Together, this significantly reduces the aforementioned gap and ensures a behavior that conforms to the modeled task and timing constraints.

Using the developed approach three relevant compliant interaction scenarios from the domains of human-robot interaction and industrial assembly are modeled and executed. The scenarios show the eligibility of the introduced concepts and their ability to scale to different compliant interactions.

“*Every adventure requires a first step.*
— Cheshire Cat
When I get home I shall write a book about this place.
— Alice”

— Lewis Carroll, Alice in Wonderland

ACKNOWLEDGMENTS

This thesis represents not only my work at the keyboard but rather my adventurous academic journey towards a PhD—a milestone in my life. Falling into the rabbit hole was easy, emerging on the other side, however, was a monumental task. I would not have been able to achieve this without the support from the ones around me!

First of all, I would like to express my deepest gratitude to my supervisor and mentor **Dr. Sebastian Wrede** for his guidance and constant support throughout this entire journey. Our discussions and your views had a great impact on the form and shape of this work. I wish to express my sincere appreciation to **Prof. Jochen J. Steil** for the inspirational discussions and advice, especially at the beginning of my PhD and during the CogIMon project. My special thanks are directed to **Prof. Axel Schneider** for accepting my invitation to review this dissertation. I would like to extend my thanks to **Prof. Franz Kummert** for being the chairperson of the examination committee and for his invaluable opinion in the FlexAssembly project. My thanks further extend to **Dr. Marc Hesse** for being the committee’s representative of the scientific staff.

Although a dissertation is an individual work, I strongly believe that research in robotics is teamwork. I am extremely grateful for the opportunity to cross paths with so many remarkable persons. Many thanks to **Dr. Niels Dehio**, **Dr. Pouya Mohammadi**, **Dr. Enrico Mingo Hoffman**, and **Prof. Hsiu-Chin Lin** for the fruitful collaborations and experiments in various research labs. I am indebted to **Prof. Nico Hochgeschwender**, **Dr. Norman Köster**, **Jan Moringen**, and **Dr. Arne Nordmann** for the undeniably helpful heavy meta discussions.

I also thank all former and current colleagues of the CoR-Lab and the Cognitive Systems Engineering group for providing me with such an inspiring and welcoming work environment and to **Stefan Krüger** for keeping the lab up and running. Special thanks go to **Dr. Sugeeth Gopinathan** and **Michael Wojtynek** with whom I started the adventure of doing a PhD and who have always encouraged and supported me. We share great memories, in our office during joyful and inspirational discussions, in the lab during evening coding sessions, and during an unforgettable vacation. Many thanks also to **Dr. Soufian Jebarra**, **Dr. Sebastian Meyer zu Borgsen**, and **Hendrik ter Horst** for the years-long friendship and the horizon-expanding conversations of broader research topics. To everyone who gave me feedback: **Dr. Niels Dehio**, **Dr. Sugeeth Gopinathan**, **Prof. Nico Hochgeschwender**, **Hendrik ter Horst**, **Dr. Norman Köster**, **Dr. Sebastian Meyer zu Borgsen**, **Dr. Pouya Mohammadi**, **Jan Moringen**, **Dr. Arne Nordmann**, **Dr. Jeffrey Queisser**, **Dr. Felix Reinhart**, **Dr. Johannes Wienke**, I really appreciate it.

I am deeply thankful to my "non-scientific" friends and family for their substantial support, providing emotional relief when it is needed the most. In particular, I thank my father **Wolfgang Queisser** for igniting my curiosity for technology. My deepest appreciation to my mother **Petra Wigand** for constantly encouraging and reminding me of what is important in life. To my brother **Maik Joshua Wigand** for all the lightsome and sometimes serious conversations on every imaginable topic there is. I know I can always count on you.

Last but far from least, my love and solid rock **Ellen Kruhöfer**. I cannot begin to express where I would be now without your positive and caring attitude. Thank you for always being by my side.

Finally, to everyone else I was unable to name here personally: Thank you!

CONTENTS

LIST OF ILLUSTRATIONS

FIGURES

NOTATION

MARGIN NOTES

○	Key point	>_	Definition

LANGUAGES AND CONCEPTS

The name of a domain-specific language (DSL) is written in italic with the word "DSL" appended, e.g., *Component DSL*. Whereas the name of a concept is written in a monospaced font, e.g., `ComponentInst`.

Throughout this thesis the terms language and DSL as well as transformation and generation are used synonymous. In contrast to that, a language module explicitly refers to the different aspects of a DSL implementation (see Figure 2.4), excluding the transformations. The implementation of the transformations are contained in a generator module. A DSL module addresses both, the implementation of the language and the transformations.

SYMBOL CONVENTIONS

x	Scalar	$H(\cdot)$	Range space of a matrix
$\mathbf{x}$	Vector	$N(\cdot)$	Null space of a matrix
$\mathbf{M}$	Matrix	$\dot{\mathbf{M}}$	1st derivative of $\mathbf{M}$
$\mathbf{M}^T$	Transpose of $\mathbf{M}$	$\ddot{\mathbf{M}}$	2nd derivative of $\mathbf{M}$
$\mathbf{M}^{-1}$	Inverse of $\mathbf{M}$	$\mathbf{M}^{\#}$	General inverse of $\mathbf{M}$
$\mathbf{M}^{\mathbf{w}+}$	Pseudoinverse inverse of $\mathbf{M}$ (weighted)	$\mathbf{M}^{+}$	Pseudoinverse inverse of $\mathbf{M}$ (Moore-Penrose)

$J_x(q) = J_x$ For brevity the parameter q will be omitted

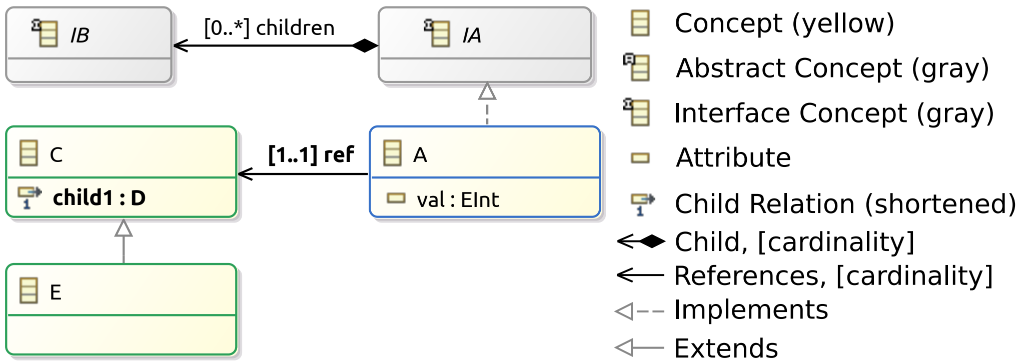
Frames are denotes as $\{\cdot\}$. To avoid a confusion with the notation for an unordered set, an underscore is added to the notation of frames in the text: $\underline{\{\cdot\}}$. The underscore is not used in figures.

ATTRIBUTION OF AUTHORSHIP

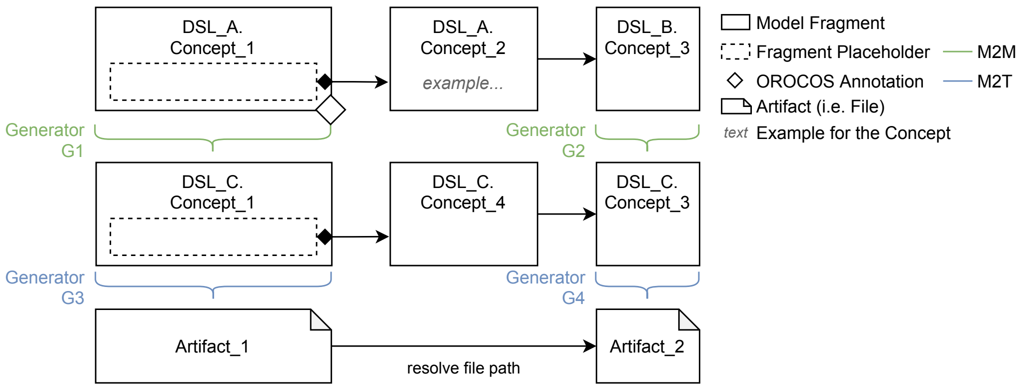
I will speak of myself using *I* to explicitly indicate work originally done by myself alone. To highlight that the results of a collaboration with others are presented, I will use *we*. The respective collaborators are indicated by the co-authors of the publications the results are based on.

HOW TO READ

A Meta-Model The meta-models in this thesis are presented using EMF's Ecore formalism. Concepts are depicted as boxes where the color of the border is used to indicate the DSL they originate from. Relations are indicated as arrows. *Child* and *Reference* relations have a cardinality of [0..1], [1..1], [0..∗], or [1..∗].



A Transformation Pipeline Each row of a transformation pipeline represents the important concepts and relations for the subsequent transformations. The arrows indicate the same relations used in the meta-models. In contrast to references, *Fragment Placeholder* visualize that the linked child concepts are actually contained in the parent concept. Brackets under a concept indicate a transformation. Here, Concept_1 of DSL_A is transformed (green) into DSL_C. Concept_2 is also handled by this transformation, since it is part of the transformed parent concept and has no extra transformation of its own. References can even be resolved after a final transformation (blue) from the model-level into a text-based artifact.



“Computers are good at following instructions, but not at reading your mind.”

— Donald Knuth

1 INTRODUCTION

In recent years an increasing number of robots are entering important sectors of our daily life; from industry (i.e. automotive, food processing and construction work), over robotics service and rehabilitation applications, to agriculture (i.e. farming), disaster (i.e. rescue and hazard), and even space (i.e. mining) applications. While the degree of robotic presence in the individual sectors varies, a clear and increasing trend in robotic applications is noticeable. The robotic applications for e.g., pick and place in machine tending or spray-painting cars that dominated the last decades are designed to provide a comfortable environment for the robots to work in. For a long time, the governing maxim was to avoid collisions at all costs, which reduced the need to physically perceive the environment to a minimum and treated any contact-based interaction with it as an error. The current and envisioned applications in the different sectors, however, differ distinctively from the previous ones. Instead of avoiding physical interactions, contacts are now either tolerated and coped with, or even essential for successful task execution, such as polishing a surface or using a hand-rail as support. Further applications are shown in Figure 1.1.

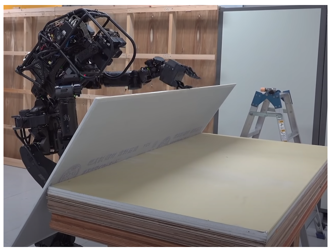
The paradigm shift in these applications can be summarized as the need for the robot to enter a physical and compliant interaction[1] with the environment, where the environment is not specifically tailored to the robot and thus may contain a high degree of uncertainty, especially when involving humans. To achieve a compliant interaction, new requirements for the functional capabilities, the software, as well as the hardware of robots arise, as they need to “feel” the environment, make sense of it, and act compliantly, without hurting a human coworker or damaging the environment [Van+13a].

Currently, there are two general ways to enable the necessary compliance: passive and active compliance. While passive compliance can be achieved through the material elasticity of the robot’s links [Lee+17], it can also be achieved through means of actuation, such as in the case of the BHA [Fes] robot that resembles an elephant trunk that is actuated by air pressure [Que+14].

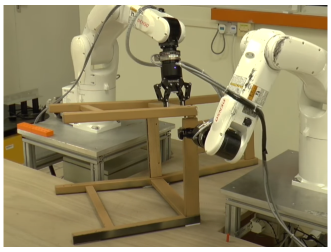
1 This kind of interaction is referred to as “compliant interaction” and is defined in Section 1.1.

(a) Object hand-over [PCW18]

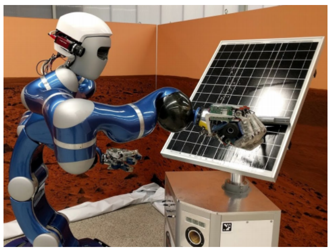
(b) Object manipulation [Cor18]



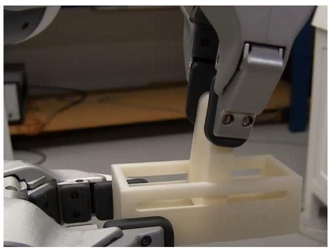
(c) Construction work [AIS18]



(d) Furniture assembly [ZP18]

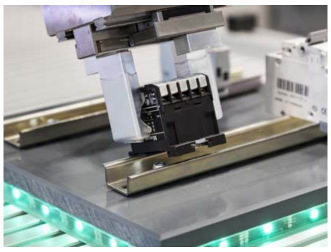


(e) Window cleaning [Lei19]

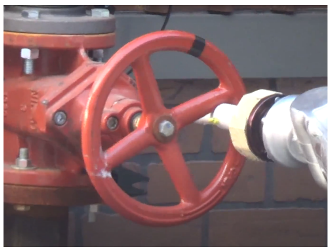


(f) Fine-manipulation [AK16]

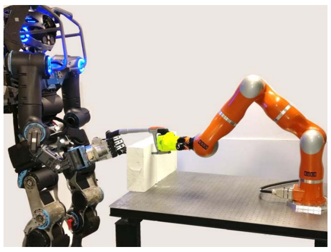
(g) Massaging [Pro19]



(h) Industrial assembly [Näg+19]

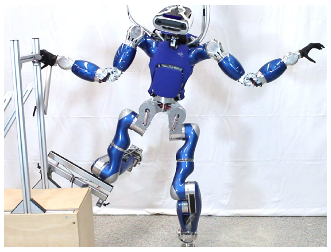


(i) Turning a valve [Lab15]



(j) Collaborative sawing [PA17]

(k) Bi-manual lifting [Cog19]



(l) Environmental support [Hen+17]

Figure 1.1: Robotic applications in compliant interaction with the environment.

Both types offer a very high degree of passive compliance, intrinsic impact resilience [PKM02], and inherent reliability at the cost of significantly increasing the complexity of the required control software [Rad+17]. Series elastic actuators (SEAs) utilize soft elements in series with an electrical motor to benefit from the passive compliance in combination with active and precise torque control [Ama+20]. While this reduces the complexity of control, the compliance parameters of the soft elements are fixed and potentially not suited for a given interaction [Rad+17]. Variable stiffness actuators (VSAs) [Wol+16] actively change the passive stiffness properties to broaden the compliance spectrum. This allows robots to adapt to the requirements of the given interaction by changing between being very soft and gentle to being powerful and fast [MBT19]. In contrast to the compliance that results from soft elements, active compliance is achieved through the control software. By leveraging advanced built-in force-torque sensors to realize highly precise and fast torque control, a naturally compliant behavior can be emulated [Rad+17]. While for active compliance a high control frequency is required, years of established control knowledge exist to aid the development of the control software.

Currently, several robots use compliant actuators, e.g., [Pai+15; Hut+16], which makes them and especially Collaborative Robots (COBOTs), e.g., [Emia; Kuk; Fan; Neu; ABB; Rob] a perfect fit for the emerging applications involving compliant interaction: A COBOT [CWP96], is a robot designed to manipulate objects in direct collaboration with a human [MB17], where the human-machine interaction foremost happens through direct physical contact [KLV09].

However, even with this advanced robotic hardware, capable of compliant interaction, the inevitable challenge remains: *How to design a robotics software system that intelligently initiates and handles the required interactions to successfully realize the new type of emerging applications?*

1.1 ON COMPLIANT INTERACTION

Definition: Compliant Interaction

A compliant interaction (CI) describes every situation in which external forces influence or even define the behavior of a robot. This includes the handling, compensation, and exploitation of physical interactions through contacts, which are natural interfaces for the exchange of forces.

>_ *robot behavior*

Even though COBOTs are technically capable of compliant interaction, most applications are still dominated by pick and place tasks that involve little to no contact at all—as has been the research on robotic manipulation for a long period of time [Lei19]. The majority of such tasks is purely position controlled, which means complete positional freedom is assumed [Mas81] and the performed actions mostly result in discrete instantaneous effects [Lei19]. Pure position control however, cannot cope with situations that involve physical interaction, where the assumed environmental state differs from the real world (e.g., caused by the unavailability of, or the uncertainty in the sensor data). This holds especially true for situations with degrees-of-freedom (DoFs) of no positional freedom, i.e. in which the robot is constrained by the environment and is only able to exert forces and no motions. According to Mason [Mas81], pure position and pure force control can be seen as dual concepts, which are omnipresent in physical interactions, and the historical predominance of position control can be tracked back to the natural cause of applications that involve very little physical contact.

However, describing the CI is particularly important for robotic motion and force control in contact situations, where the robotic system is required to handle predictive as well as unpredictive contacts in a way that successful task execution is feasible. In addition to the passive and compliant handling of contacts, which foremost arises due to the uncertainties related to the robot's internal representation of the physical environment, other situations exist in which an active exploitation of contacts is required to fulfill tasks, such as to perform a guided motion along a surface, to use a hand-rail for support and stability [Koy+08], or to snap-fit an object [Näg+18].

Definition: CI Task

A CI task is in contrast to an application always formulated from the point of view of the robot that performs the task. Such a task involves at least one contact situation and the set of contact transitions that arise during its execution. Here, a contact situation is formed by one or more compliant interactions.

To define what the relevant CIs in an application are depends on the scope of the CI task to be solved. For instance, while the contact of a parallel two-finger gripper with a grasped object practically involves CIs, they are not relevant for the robot that performs a pick and place task with only free-space motions. However, in the case of two multi-DoF manipulators that lift an object, these CIs are crucial to solve the task. Hence, the developer needs to select the CIs relevant to a task from the ones that exist in the application. Transitioning between the contact situations of a CI task, demands a flexible (re)prioritization of control strategies based on the changing set of CIs.

With the increasingly contact-rich tasks that emerge and the impact of CI on the execution of a desired task in mind, it is more important than ever to explicitly consider and describe the compliant interactions with the physical environment to achieve a successful task execution.

1.2 PROBLEM STATEMENT

The design of a robotics software system that is capable of performing a CI task is a challenging undertaking, due to its complex and heterogeneous nature. The designer needs to bridge the gap between an informal representation of an envisioned CI task and the actual architecture of a system that leads to the desired behavior (see Figure 1.2 (a)). Such a representation involves several concerns [Wig+17a; Nor+16b; Sch+10] (e.g., hardware, algorithms, timing) and often contains a mixture of formal, informal, and incomplete descriptions [BLW05; Buc+20] with no clear semantics [Mer17; Bru15]. Concern-overarching semantics (and e.g., constraints) are also rarely considered. Bridging this gap, would require the designer to create a consistent representation that unifies all the different functional and non-functional requirements of a robotic system and their possibly problematic interactions. Additionally, these requirements need to be correctly realized on the source-code level. This potentially involves the use of several existing libraries and frameworks that all come with their own additional requirements and hidden assumptions [Lot18].

component-based architecture ⎯<

concern ⎯<

Currently, there are two problems that significantly prevent the reduction of this gap for CI applications.

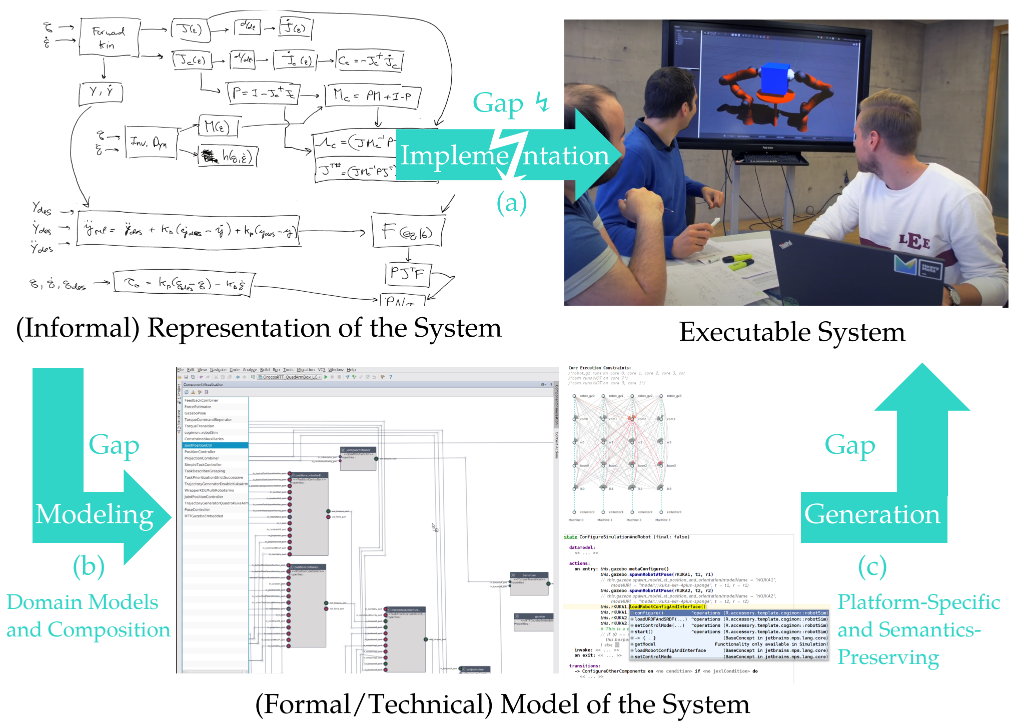

(Informal) Representation of the System
Gap ↯
Implementation
(a)
Executable System
Gap
Modeling
(b)
Domain Models and Composition
Gap
Generation
(c)
Platform-Specific and Semantics-Preserving
(Formal/Technical) Model of the System

Figure 1.2: The gap (a) between the mental representation (a real example [Cog] from the CogIMon [Cog19] project) and the executable system is commonly bridged by a manual implementation, which gives rise to several problems. To reduce the gap, it needs to be subdivided. Modeling is used to capture and compose the individual informal requirements (a). Code generation replaces the manual implementation to ensure the creation of a platform-specific and semantics-preserving robotics software system (c).

PROBLEM 1: LACK OF A UNIFYING ABSTRACTION FOR CI relates to the requirements that are imposed by CI tasks. Individual capabilities of a system, such as the algorithmic computation, the communication or the coordination, are already well covered in the literature [Nor+16a]. Whereas, the physical interaction part of a task description is still often completely neglected, insufficiently considered, or inaccessibly and untraceably hidden [WDW20]. Considering the importance of the physical environment for CIs tasks, the lack of an explicit representation of this aspect is disturbing. In most approaches that do consider CI, the environmental information is either directly entangled with a control component in form of hidden assumptions (e.g., [NWS15]), implicitly modeled as selection matrices [Mas81], or represented as constraints on *feature coordinates* [de +07] (e.g., [Klo+11; Van+13b; Tho+13; Näg+18]). The same holds true for higher-level programming approaches, relying on a form of skill-based technology, such as [Tho+13]. This also includes the programming solutions offered by different COBOT manufacturers, such as [Emib]. It is quite paradox that even COBOT manufacturers that design their robots

>_ *skill*

especially for the interaction in collaboration with humans, do not explicitly specify the physical interaction with the environment and conceptually limit their programming tools to a high-level skill-based approach, where each skill is treated as a configurable black-box for the developers. The introduction of a skill-based abstraction makes sense though, since it enables a larger user base to program robotics applications. However, its lack of representing the requirements imposed by the task-related CIs, either limits the set of designable applications, or results in an increased gap between the envisioned informal representation and the specified requirements. In the latter case, a common workaround is to implicitly encode the constraints imposed by the environment as specific parameterizations of a skill. For instance, to let a robot exert a force against a wall, a Cartesian impedance skill could be configured to target a point inside the wall [Lei19]. Here, the force exerted by the robot, would be partially encoded by the position of the target point and the concrete impedance formulation, hidden by the skill. However, this would require an expert user to have a deep understanding of the individual parameters of the skills and how to exploit them to accurately represent the task at hand, if possible at all. As it might be obvious, this is certainly not a favorable approach, which is very error-prone and depends strongly on the specific realization of the requirements. Additionally, it does not reflect the actual requirements of the CI task, due to the mismatch between the requirements that should and that can be expressed. Hence, the scientific problem at hand is *the lack of formal, unifying, and expressive abstractions to model CIs tasks based on common and scientifically grounded concepts. Thereby overcoming the aforementioned mismatches.*

PROBLEM 2: LACK OF A UNIFYING CONCERN COMPOSABILITY relates to the gap between the different requirements of a CI task as well as between the requirements and their realization in the system design. The ability to specify the requirements of capabilities, such as a CI task description, coordination, control laws, and timing, independently and to realize them in isolation, is not sufficient to achieve a consistent implementation on the source-code level [Bis+10; SLS17]. While the source-code might compile, there is no guarantee that the implementation reflects the semantics of the specified requirements correctly. Unfortunately, the majority of current approaches focuses more on expressing the requirements of a specific capability and too less on their composability [Nor+16a]. Since some capabilities cannot be considered in pure isolation, the required other capabilities are thus partially considered but implicitly hidden. For instance, describing a task using one of the prominent skill-frameworks, does not allow for a composition[2] with e.g., timing requirements to restrict the execution time of a skill or the frequency with which pre- and post-conditions are evaluated. The inability to express certain requirements (e.g., the execution-time behavior) does not contribute to the reduction of the aforementioned gap. However, experiencing the effects of implicit and hidden assumptions is worse, since it might lead to unexpected and dangerous situations, especially in the context of CI. An example for this would be the specific

capability ⸏<
composability ⸏<

2 refers to the combination of parts into a whole for different purposes. See composability.

motion controller or the planning accuracy thresholds that are used to realize a skill and which are hidden from the user. Hidden assumptions are further found in relation to the chosen software and hardware platforms used by a system [Bor+16]. Even if the capabilities can be properly composed, their realization strongly depends on these platforms, which impose additional requirements on the design of the system. Hidden assumptions about specific software frameworks or e.g., the DoFs of a robot, greatly contribute to a decrease of the explainability and verifiability. Thus, preventing the coherent composability of concern-overarching requirements, which leads to a mismatch between the requirements and the resulting behavior. Hence, the scientific problem at hand is *the lack of a modularization and composition approach that supports the heterogeneous functional and non-functional concerns of a robotic system in CI with the environment. Further, establishing a link towards a consistent realization. Thereby overcoming the aforementioned mismatches.*

1.3 RESEARCH APPROACH

To address the previously introduced problems, a research approach based on model-driven engineering (MDE) is chosen in this dissertation. Since 1990, MDE especially in form of domain-specific languages (DSLs) experienced a steady growth in the field of robotics [Nor+16a]. While DSLs are commonly used in domains, such as automotive [Per+12], avionics [Mor15], and embedded systems in general [Ouh+11; AMS07], they are nowadays also frequently used in various robotics (sub-)domains, including robot control [NWS15], behavior or motion planning [Tho+13], and component-based systems [WW19]. In the domain of robot motion control and planning, DSLs are commonly employed on different levels. High-level skill abstractions are used to model a desired behavior, while on a lower level, control architectures are modeled using e.g., Simulink [Mat] or Scilab [Sci]. Considering the recent works on MDE in robotics, MDE proved to increase comparability and understandability as well as to establishing a link from the model to the implementation via model transformation and code generation [Gho10; Nor16; Rin+16; FBC12; Wie+18].

>_ *domain*

>_ *control architecture*

Even though a lot of work exist in the individual domains, the previously introduced problems (see Section 1.2) remain. Hence, in order to create explainable, predictable, and safe robotic systems that are suitable to realize CI tasks, the gap between the informal requirements and the resulting behavior needs to be reduced significantly.

RESEARCH APPROACH FOR PROBLEM 1 The first step to approach the *Lack of a Unifying Abstraction for CI* is to decouple the interaction with the environment from the other aspects of a task description and their (e.g., skill-based) implementation. This is similar to the early evolution of robot control, where the awareness for the explicit modeling of the involved aspects and their individual requirements rose [Nor+16a; HST92; Näg+18]. Abstracting from the mathematical equations, which hold a high amount of knowledge in a strongly condensed manner, made such aspects comparable and more easily understandable. Hence, making the environment explicit would not only foster exchangeability, but would allow the adaptation of a task to different environmental situations.

The second step to address the problem is to establish a link between the CI model and the rest of a task description, down to their realizations as i.e. control tasks. The CI model can then provide a context for the other task-related aspects (e.g., coordination, trajectories, etc.) that ultimately constrains the execution of the task. Referring back to the early evolution of robot control, the importance of such a link, was already recognized in 1981 by Mason [Mas81]. At that time, it was common practice that the abstractions—mostly in form of control block diagrams—had no technical link to the equations or even to the implementation. This was a problem, since without such a link inconsistency and divergence between abstraction and implementation is almost impossible to prevent. To address this problem, Mason introduced the *Task Frame Formalism* (TFF) as an interface to abstract from the low-level control. TFF conceptually plays a major role for the approach presented in Chapter 6. Making the aspects of a task description explicit and composable, would eventually overcome the disadvantage of skill-based approaches that only support strict compositions of skills. Currently, such a composition is usually limited to a sequence of skills, where the successive skill has no knowledge about the previous one. This is due to skills being treated as black boxes. A common and composable representation of the aspects of a task description allows for a more dynamic composition, since the individual aspects can be explicitly composed and verified. By loosely coupling the CI model with aspects, such as trajectories and coordination, the ability to adapt a task to different (modeled) environmental situations as well as to blend between tasks is greatly increased, without creating possibly dangerous and undesired interferences.

In addition to explicitly modeling CI task descriptions, a control architecture needs to be synthesized, which uses a suitable control framework, in order to realize the specified task. In this work Projected Inverse Dynamics Control (PIDC) is used. Apart from the task-related requirements, the control architecture is synthesized from, the model needs to also fulfill the combined requirements of the different robotics system concerns (i.e. functional and non-functional). The composition of the different robotics concerns is addressed by the following research approach.

program synthesis ⦣

RESEARCH APPROACH FOR PROBLEM 2 To overcome the *Lack of a Unifying Concern Composability*, a common composition structure needs to be created that is used by the different concerns of the capabilities, software platforms, and hardware platforms of a robotic system. The aim is to support the composition between all concerns in terms of their requirements and to enable concern-overarching specifications. For instance, combining timing and architectural concerns to specify requirements for the execution order and execution time behavior of a set of control components. The composition of all requirements needs to be reflected in a consistent realization on the source-code level. On the model and code level, the composition structures need to be capable of coping with the evolution of concerns, requirements, and realizations. Since it is obvious that not all the concerns of robotic systems for current as well as for future applications can be explicitly considered today, the composition is required to support incremental extension. For instance, without the explicit consideration of the timing concern, a robot might still be able to move by relying on hidden assumptions in e.g., the used control frameworks. However, by doing so the size of the gap is increased. Instead, by purposefully making the assumptions of the involved concerns, explicit would reduce the gap. Thereby increasing the explainability of this particular part of the resulting system's behavior. Since the requirements of a concern can be rarely considered in isolation, they need to be integrated into the composition structures to achieve a consistent realization with the other requirements. Such a composition then allows to reduce the gap further, by making the interactions between the concerns on the model as well as on the source-code level explicit and their impact on the overall composition explainable. Note that this is in fact a very challenging problem [Völ+13; Sta+16] that is actively researched and not only important for robotics, but rather for every highly heterogeneous domain. As of now, there is no universal solution to that challenge. This kind of composability eventually paves the way for the specification of concern-overarching requirements and thus increases the set of describable robotics applications.

1.4 goal and research questions

On the basis of these considerations the goal of this thesis is to

> *leverage MDE to enrich task descriptions with a suitable abstraction for the compliant interactions and crucial qualitative aspects (e.g., timing), to bridge the gap between the envisioned CI task and the robot's behavior, making it explainable, predictable, and synthesizable.*

To pursue this goal, the following 5 research questions are investigated in the course of this thesis.

rq 1: how to use mde to compose the robotics concerns?

This question aims at finding a way to benefit from the methodology of MDE to handle the complexity that arises from the different heterogeneous robotics concerns that are required to solve CI tasks. These concerns include task descriptions, state-of-the-art control formalisms, component-based system architectures, and aspects related to the chosen software and hardware platforms. Composing the individual robotics concerns while maintaining their modularity, is a non-trivial undertaking, which to the best of my knowledge is not solved at all. This question further investigates how the requirements of the composed concerns on the system or on the resulting robotics behavior can be expressed on the model and on the source-code level, to ensure their compliance in the actual executable software system. Thereby, significantly increasing the explainability and predictability of the robotics behavior.

RQ 2: HOW TO COVER TIMING AS A NON-FUNCTIONAL CONCERN?
The execution time behavior of a robotic system is one of the most important aspects, especially regarding compliant interactions. Defining the timing behavior of a system and enforcing it is crucial for the safety and predictability of a system. This question therefore concerns itself with the extension of the MDE approach of RQ 1, to support additional and future concerns of the robotics domain. In this case, the focus is on the integration of non-functional execution time constraints and the preservation of the resulting qualitative properties during the execution of the robotic system.

RQ 3: WHAT ARE SUITABLE ABSTRACTIONS TO MODEL THE CI IN TASKS?
The previous questions are needed to gain the knowledge to compose the functional and non-functional concerns of robot control systems and to reduce the gap towards the expressed behavior of the robot. Now, this question is concerned with the choice of suitable abstractions with the necessary degree of expressiveness to model compliant interactions with the environment to accurately represent a CI task. Hence, increasing comprehensibility and explainability. In addition to making the CI explicit, a compatibility with the other aspects of a task description needs to be ensured by properly extending the MDE approach of RQ 1.

RQ 4: HOW TO LINK THE TASK TO THE CONTROL SYSTEM?
Closing the gap between the CI task description and the behavior of the robot, depends on the control system that executes the behavior. Thus, this question aims at finding suitable model transformations to synthesize a control system model, which is infused with the task description, to specifically perform the desired behavior. Further, this question investigates the design of a component-based control architecture model, based on Projected Inverse Dynamics Control (PIDC), that is able to realize different CI tasks by being configurable via modeled task descriptions. The system model also needs to provide a common ground that allows the composition with the robotics concerns that are investigated in RQ 1 and RQ 2. By creating a conceptual and technical link between the task description and the control system model, explainability and predictability is increased.

RQ 5: WHAT KINDS OF CI TASKS CAN BE REALIZED?
This question targets the practical application of the developed approach to realize representative CI scenarios to investigate the scalability from an experimental point of view. A conceptual viewpoint is taken by separately discussing the scalability of the modelling approach and the technical control framework, used for realization.

1.5 CONTRIBUTIONS AND OUTLINE

The overall outcome of this thesis is the development and MDE-based implementation of suitable concepts that allow the modeling of CI in terms of defining and constraining the robot's behavior through the exchange of forces via contacts as (natural) interfaces. When considered as part of the task description, those concepts significantly reduce the gap between the abstraction and the actual desired task. Further, the concepts are chosen so that a synthesis of a robot control architecture based on the PIDC framework becomes feasible. Additionally, a MDE-based modularization and composition framework is developed, which allows the consistent modeling of heterogeneous robotics concerns. Via model transformations, the system model that is composed with task-related, functional, and non-functional requirements is generated into an executable software system that realizes a behavior that meets the modeled requirements. In addition to the modeling of CI task descriptions, special emphasis is on the modeling and preservation of execution time constraints. A technical implementation "CoSiMA" of the developed aspects is used for the experimental evaluation of the approach.

>_ *Compliant Simulation and Modeling Architecture (CoSiMA)*

The thesis is outlined in 9 chapters, which cover the introduced RQs: Chapter 1 introduces the research topic, presents the research problem, and defines the relevant RQs of this thesis. Chapter 2 discusses the state-of-the-art in robotics system design for CI tasks. This includes the task description, control formalisms, and software principles. Content-wise, the thesis is presented in three parts. Part I presents CoSiMA, which is a modeling environment that incorporates the conceptual ideas presented throughout this thesis to allow the model-based specification and realization of executable real-time robotic systems for CI. Chapter 3 presents the L3Dim approach, which enables the modeling of the heterogeneous concerns of robot control systems, and its application in CoSiMA. Chapter 4 is concerned with an extension of CoSiMA, to support the definition of timing constraints on the model level as well as their implementation in the real-time execution framework (i.e. OROCOS RTT). Part II sets the focus on researching the domain of CI and its integration into CoSiMA. Chapter 5 is dedicated to the analysis of the CI domain to find the right abstractions to model CI tasks in terms of a task description. Chapter 6 translates the concepts from the previous chapter into an extension of CoSiMA, which allows to practically create valid CI task models. Chapter 7 establishes the link between the task model and the system model via a synthesis mechanism. The synthesis is presented for the PIDC approach. Part III puts this thesis into perspective by evaluating and discussing the applicability of CoSiMA. Chapter 8 presents three representative robotics case studies for CI and compares their execution to the specified models. Chapter 9 gives an overall conclusion and outlook for future work.

The individual contributions are outlined per chapter and part in Table 1.1:

Chapter	RQ	Contribution
Part I CoSiMA		
3	1	The MDE-based modularization and composition framework L3Dim and its concrete instantiation as CoSiMA, which enables the modeling and execution of robotic systems. [Wig+17b; Wig+17a; Wig+18; Nor+16a; Wie+18]
4	2	The modeling and realization of non-functional execution time constraints on the robotic system, integrated as DSL-extension into CoSiMA. [Wig+18; WW19; Moh+18]
Part II Compliant Interaction		
5+6	3	Conceptual and technical realization of the insights gained from the domain analysis on CI as a DSL for CoSiMA to model CI task descriptions. [WDW20; Deh+ss]
7	4	Synthesis of a PIDC-based control system model from a CI task description through model transformations, based on a developed reference architecture. [WDW20]
Part III Perspectives		
8	5	Application of CoSiMA to realize 3 experiments to show the scalability of the approach. [WDW20; Wig+17a; Deh+18; Deh+ss]

Table 1.1: Overview of the contributions per chapter.

1.6 PUBLICATIONS

[WDW20] **Dennis Leroy Wigand**, Niels Dehio, and Sebastian Wrede. "Model-Based Specification of Control Architectures for Compliant Interaction with the Environment." In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 7241–7248. DOI: 10.1109/IROS45743.2020.9340718. *used on: pp. 5, 12, 89, 103, 125, 134, 141, 148*

[Wig+17a] **Dennis Leroy Wigand**, Arne Nordmann, Niels Dehio, Michael Mistry, and Sebastian Wrede. "Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case." In: *Journal of Software Engineering for Robotics* 8.1 (2017), pp. 45–64. ISSN: 2035-3928. DOI: 10.6092/JOSER2017_08_01_p45. *used on: pp. 4, 12, 37, 48, 119, 134, 141, 145*

[Wig+17b] **Dennis Leroy Wigand**, Arne Nordmann, Michael Goerlich, and Sebastian Wrede. "Modularization of Domain-Specific Languages for Extensible Component-Based Robotic Systems." In: *2017 First IEEE International Conference on Robotic Computing (IRC)*. Ed. by IEEE International Conference on Robotic Computing. IEEE, 2017-04, pp. 164–171. DOI: 10.1109/IRC.2017.34. *used on: pp. 12, 37, 60, 119, 134*

[Wig+18] **Dennis Leroy Wigand**, Pouya Mohammadi, Enrico Mingo Hoffman, Nikos G. Tsagarakis, Jochen J. Steil, and Sebastian Wrede. "An Open-Source Architecture for Simulation, Execution and Analysis of Real-Time Robotics Systems." In: *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. Ed. by Mod-

eling IEEE International Conference on Simulation and Programming for Autonomous Robots. Piscataway, NJ: IEEE, 2018, pp. 93–100. doi: 10.1109/SIMPAR.2018.8376277. *used on: pp. 12, 37, 48, 65*

[WW19] **Dennis Leroy Wigand** and Sebastian Wrede. "Model-Driven Scheduling of Real-Time Tasks for Robotics Systems." In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 2019, pp. 46–53. doi: 10.1109/IRC.2019.00016. *used on: pp. 7, 12, 48, 65, 68, 73*

[Deh+18] Niels Dehio, Joshua Smith, **Dennis Leroy Wigand**, Guiyang Xin, Hsiu-Chin Lin, Jochen J. Steil, and Michael Mistry. "Modeling and Control of Multi-Arm and Multi-Leg Robots: Compensating for Object Dynamics During Grasping." In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018-05. doi: 10.1109/icra.2018.8462872. *used on: pp. 12, 22, 94, 98, 134, 141, 145*

[Deh+ss] Niels Dehio, Joshua Smith, **Dennis Leroy Wigand**, Pouya Mohammadi, and Michael Mistryand Jochen J. Steil. "Enabling Impedance-based Physical Human-Multi-Robot Collaboration: Experiments with four torque-controlled Manipulators." In: *The International Journal of Robotics Research* (2021, In Press). *used on: pp. 12, 18, 22, 89, 103, 141, 153*

[Moh+18] Pouya Mohammadi, Milad Malekzadeh, Jindrich Kodl, Albert Mukovskiy, **Dennis Leroy Wigand**, Martin Giese, and Jochen J. Steil. "Real-time Control of Whole-body Robot Motion and Trajectory Generation for Physiotherapeutic Juggling in VR." In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 270–277. doi: 10.1109/IROS.2018.8593632. *used on: pp. 12, 65, 86*

[Nor+16a] Arne Nordmann, Nico Hochgeschwender, **Dennis Leroy Wigand**, and Sebastian Wrede. "A survey on domain-specific modeling and languages in robotics." In: *Journal of Software Engineering in Robotics* 7 (2016), pp. 75–99. doi: 10.6092/JOSER_2016_07_01_P75. *used on: pp. 5–7, 12, 23, 28, 29, 33, 37, 42, 44, 65, 66, 176*

[Wie+18] Johannes Wienke, **Dennis Leroy Wigand**, Norman Koster, and Sebastian Wrede. "Model-Based Performance Testing for Robotics Software Components." In: *The Second IEEE International Conference on Robotic Computing*. Piscataway, NJ: IEEE, 2018, pp. 25–32. doi: 10.1109/IRC.2018.00013. *used on: pp. 7, 12, 37, 86*

“*If I have seen further, it is by standing on the shoulders of giants.*”

— Sir Isaac Newton

2 ENGINEERING COMPLIANT CONTROL SYSTEMS

This chapter gives an overview of the heterogeneous parts that are required to design robotic systems for compliant interaction. The first part covers state-of-the-art task-level programming and suitable control frameworks. The second part introduces component-based software engineering and model-driven engineering principles for robotics. Together, this lays the theoretical foundation for the work presented in this thesis.

Designing a robotic system to perform a task that involves compliant interactions is still a complex endeavor. A system, such as those depicted in Figure 1.1, involves several different concerns and capabilities, including using different sensors and actuators in real time, compensating for uncertainty and noise, and reacting to unexpected situations safely [KSB16]. All, while the system tries to achieve a desired task.

In the course of this chapter, the theoretical background for the work presented throughout this thesis is introduced. It starts with the presentation of state-of-the-art (SotA) approaches for task-level programming, followed by the relevant and prominent control strategies for compliant interaction, and ends with the introduction of suitable software engineering methods to manage the software complexity of a robotic system in terms of reusability and maintainability.

2.1 TASK DESCRIPTION AND TASK-LEVEL PROGRAMMING

The first step in designing a robotic system for compliant interaction is to describe the task which should be accomplished. Without a task, there is no need for a robot to move. The behavior produced by a robot is always but not exclusively depending on the task. According to the literature, a robot's behavior is usually defined by one or more tasks to achieve a particular goal. In several works, tasks describe actions such as Peg-In-Hole [MW01], while skills represent smaller actions that are composed to realize a task, e.g., Move-To and Screw [Tho+13]. However, the other way around can also be found in the literature [Näg+18], where a skill act as a container for multiple tasks. In general, there seems to be no common agreement about the definition of tasks and skills, since in some cases these terms are even used interchangeably. However, what all those approaches have in common is the fact that they provide a kind of task description and aim to make the programming easier, by abstracting from the low-level control.

no consensus in the definition of tasks and skills

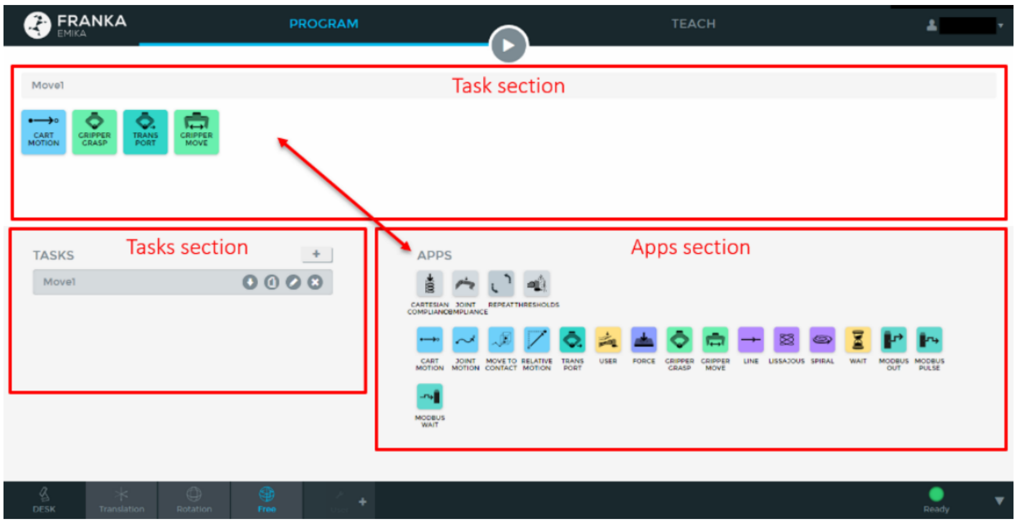


Figure 2.1: Screenshot of Franka Emika Desk [Emib] from [Ion21].

In the recent years a trend in the research community as well as in the industry (i.e. COBOT manufacturers) has emerged, which heavily focuses on task-level programming [Ion21]. Prominent examples from the industry are Franka Emika Desk [Emib] (see Figure 2.1) and Drag&Bot [Dra], which follow a sequential or tree-based skill-oriented programming approach. The interested reader may refer to [Cor+20] for a review of visual robot programming tools. These approaches as well as similar ones from the research community (e.g., [SWW18; MW01]) focus on the simple but intuitive sequencing of skills. In [Ped+16; Sch+18] three different layers are defined (i.e. primitives, skills and tasks) to support the separation of roles and to distinguish the programmer of skills from the operators in the factory that program the high-level tasks. Following a similar distinction, Björkelund et al. [Bjö+11] focus on the definition of skills, using a constraint-based task description, introduced by De Schutter et al. [de +05]. This approach can be traced back to Mason's [Mas81] idea of the Task Frame Formalism (TFF), which was further formalized by Bruyninckx and De Schutter [Bd96].

The TFF is used to independently constrain the translational and rotational DoF of geometric objects, by assigning separate velocity or force control formalisms to the axes of a single task frame (or compliance frame). Constraints are classified into natural constraints enforced by the environment and artificial ones enforced through a task. Figure 2.2 shows the alignment of the end-effector (EEF) tool with a surface (a) as well as the associated TFF description (c). The TFF approach was extended in [de +07; Smi+08] to be applicable to

compliance frame

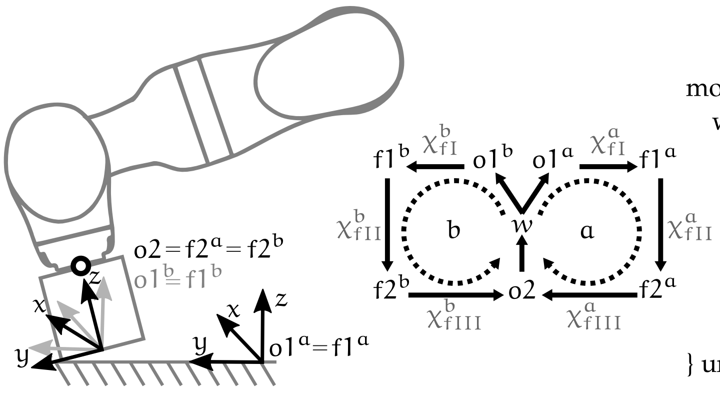
multiple frames, by introducing the concepts of object frames (e.g., o1), feature frames (e.g., f1), and feature coordinates

$$\chi_f = \begin{pmatrix} \chi_{fI}^T & \chi_{fII}^T & \chi_{fIII}^T \end{pmatrix}^T , \tag{2.1}$$

for every feature, to impose constraints on the relative motion between objects [de +07].

TFF as basis for describing the CI

This idea of using constraint-based descriptions of geometric relations between object frames for the task description builds the foundation to describe the compliant interaction in this thesis. Although, TFF models the geometric relationships, it does not specify the dynamic behavior of the relationships and the enforcement of the constraints as part of the task description. Since it's introduction, TFF is used as basis for multiple works (e.g., [Smi+08; Klo+11; Van+13b; Tho+13; Näg+18]).



```
move compliantly {
  with task frame directions
    xt: velocity v₁ mm/sec
    yt: velocity v₂ mm/sec
    zt: force f N
    axt: force 0 Nmm
    ayt: force 0 Nmm
    azt: velocity w₁ rad/sec
} until time > t sec
```

(a) (b) (c)

Figure 2.2: Alignment of the block-shaped robot EEF with a surface. The example (a) and the TFF specification (c) is inspired by Figure 6 in [Bd96]. The corresponding feature frame and coordinate representation (b) is based on Section 5.4 in [de +07].

An exemplary situation which can be described by this framework is shown in Figure 2.2. Here, two relations are constrained. For each relation, a loop between feature and object frames is created. The first feature $(()^a)$ connects the frame $f2^a$, located at the edge of EEF which is rigidly attached to the robot (o2), to $f1^a$ which is located on the surface ($o1^a$). The second feature $(()^b)$ connects the rigidly attached frame $f2^b$ with $f1^b$, which is compliantly coupled to o2 via $o1^b$. Here, the compliant coupling is caused due to the mechanical coupling of a force-torque-sensor. To describe the relative motion for the two features, the following feature coordinates can be chosen:

$$\begin{aligned} \chi_{fI}^a &= (-) & \chi_{fI}^b &= (-) \\ \chi_{fII}^a &= \begin{pmatrix} x^a & y^a & z^a & \phi^a & \theta^a & \psi^a \end{pmatrix}^T & \chi_{fII}^b &= \begin{pmatrix} x^b & y^b & z^b & \phi^b & \theta^b & \psi^b \end{pmatrix}^T . \\ \chi_{fIII}^a &= (-) & \chi_{fIII}^b &= (-) \end{aligned} \tag{2.2}$$

The chosen DoFs can be further constrained to represent artificial and natural constraints using the TFF:

$$\left.\begin{aligned} y_1 &= \dot{x}^a && = y_{1,d} = v_1 \\ y_2 &= \dot{y}^a && = y_{2,d} = v_2 \\ y_3 &= \dot{\psi}^a && = y_{3,d} = w_1 \end{aligned}\right\} \begin{array}{l}\text{velocity constraints}\\ \text{(artificial)}\end{array}$$
$$\left.\begin{aligned} y_4 &= F_z = k_z\, z^b && = y_{4,d} = f \end{aligned}\right\} \begin{array}{l}\text{force constraints}\\ \text{(natural and artificial)}\end{array} \quad , \qquad (2.3)$$
$$\left.\begin{aligned} y_5 &= T_x = k_{ax}\, \phi^b && = y_{5,d} = 0 \\ y_6 &= T_y = k_{ay}\, \theta^b && = y_{6,d} = 0 \end{aligned}\right\} \begin{array}{l}\text{force / compliance constraints}\\ \text{(natural)}\end{array}$$

where y_i denotes output variables and $y_{i,d}$ the corresponding desired values. $\dot{x}^a$, $\dot{y}^a$, and $\dot{\psi}^a$ denote the velocities parallel to the surface and the rotation around the z axis, which are artificially controlled using the set point velocities v_1, v_2, and w_1. When in contact with the surface, y_4 models the natural constraint through the force-torque-sensor coupling (i.e. stiffness k_z, k_{ax}, and k_{ay}) in combination with an artificial one, which is represented by the application of the desired force f. To align the rotational DoFs of the EEF (ϕ^b and θ^b) with the surface, a fully compliant behavior is achieved by artificially choosing $y_{5,d} = y_{6,d} = 0$.

2.2 COMPLIANT INTERACTION CONTROL

To achieve any kind of behavior, including the realization of the previously mentioned constraints, a control signal needs to be generated and send to the robot. While there are numerous methods in the literature to generate a control signal for a robot in the kinematics as well as in the dynamics domain, modeling the dynamics behavior of such a robot in combination with external forces as the first step, seems to be widely agreed upon in the context of compliant interactions [FO16; CFK16; Vd16]. The following equation models the dynamics of a rigid multi-body robotic manipulator with D joints that is subject to (contact) constraints and external forces [Deh+ss]:

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{h} = \boldsymbol{\tau} + \mathbf{J}_c^T \boldsymbol{\lambda}_c + \mathbf{J}_x^T \mathbf{F}_x \;, \qquad (2.4)$$

where $\boldsymbol{\tau} \in \mathbb{R}^D$ is the vector of joint torques, $\mathbf{M} \in \mathbb{R}^{D \times D}$ is the inertia matrix, $\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}} \in \mathbb{R}^D$ are joint positions, velocities, and accelerations, and $\mathbf{h} \in \mathbb{R}^D$ is the vector of generalized gravitational torques and centrifugal, gyroscopic, and coriolis effects. Additionally, $\mathbf{J}_c \in \mathbb{R}^{6\,B \times D}$ denotes the constraint Jacobian that describes B linear independent constraints, which are associated with (virtual) contacts. All contact wrenches applied by the robot to enforce contact constraints, are vertically concatenated and denoted as $\boldsymbol{\lambda}_c \in \mathbb{R}^{6\,B}$. External disturbances—caused by physical interaction—at a (contact) location $\mathbf{x} \in SE(3)$ are modeled as an unknown wrench $\mathbf{F}_x \in \mathbb{R}^6$ and are mapped onto the joint-space using the respective Jacobian for that location $\mathbf{J}_x \in \mathbb{R}^{6 \times D}$.

2.2.1 *Reacting to External Disturbances*

Controlling a robot in a contact-free situation (i.e. $\mathbf{J}_c = 0$), can typically be done in joint space as well as in the operational task space, or a hybrid combination thereof. The relation between a task space wrench $\mathbf{F} \in \mathbb{R}^H$ of an H-dimensional operational point in the task space, a joint torque vector $\boldsymbol{\tau}$, and the desired acceleration $\ddot{\mathbf{x}}$ can be described by the operational-space formulation [Kha87]:

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{h} = \boldsymbol{\tau} \Leftrightarrow \mathbf{J}^T\mathbf{F} = \mathbf{J}^T\boldsymbol{\Lambda}\left(\ddot{\mathbf{x}} - \dot{\mathbf{J}}\dot{\mathbf{q}} + \mathbf{J}\mathbf{M}^{-1}\mathbf{h}\right) \;, \tag{2.5}$$

where $\boldsymbol{\Lambda} = \left(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T\right)^{-1}$ is the operational space inertia matrix. To perform a single control objective $\ddot{\mathbf{q}}$ and $\ddot{\mathbf{x}}$ need to be replaced respectively by e.g., a classical PD control law. For control in compliant interaction with the environment, the external forces need to be explicitly considered as done in Equation 2.4. Following that idea, the external forces are prominently modeled using an impedance control law [Hog85], yielding a (joint space) impedance controller that is capable of compliantly coping with external disturbances:

$$\boldsymbol{\tau} = \mathbf{M}\ddot{\mathbf{q}} + \mathbf{D}_d\dot{\tilde{\mathbf{q}}} + \mathbf{K}_d\tilde{\mathbf{q}} \;, \tag{2.6}$$

with $\tilde{\mathbf{q}} = \mathbf{q}_d - \mathbf{q}$, where $\mathbf{q}_d$ denotes the desired joint angles as the virtual equilibrium point in the joint space, $\mathbf{D}_d$ and $\mathbf{K}_d$ are the desired damping and stiffness matrix respectively.

2.2.2 *Multi-Objective Control*

Depending on the available DoFs, multiple control objectives can be composed to achieve a desired robotic behavior. Approaches that compose control objectives on the same priority level exploit the resulting interferences to achieve the behavior [Mor+13a; DRS15]. This type of prioritization however, does not ensure that any of the control objectives are achieved and might cause unforeseen motions. Instead, to ensure that the primary objective is always achieved, while secondary ones are only achieved when possible, the interferences between the objectives need to be limited. This is done by exploiting the redundancy of the robots.

Robotic manipulators with a small degree of redundancy regarding the overall task DoF, usually perform a primary objective (e.g., drawing a circle in the task space) and a secondary objective that exploits the redundancy to maintaining a particular joint configuration [Emm+13], avoid joint limits [Lie77], or minimize the actuators' generalized forces [HS85]. In contrast, hyper-redundant robots, such as humanoids can employ multiple levels of redundancy to perform various control objectives with decreasing priorities, without interfering with higher priority objectives. Some pioneers in this research area are Nakamura et al. [NH87], Siciliano [Sic90], and Mansard et al. [Man+09]. The latter introduced the notion of a Stack of Tasks (SoT) as a generalized inverse kinematics framework to solve the motion of humanoid robots, which is based on earlier work [SS91]. The SoT approach is capable of combining both previously

>_ *Stack of Tasks (SoT)*

mentioned types of prioritization and can be realized analytically [Deh18] or numerically [Roc+15].

2.2.2.1 *Projection-Based Prioritization*

Approaches that address the SoT analytically usually resort to a projection-based solution. For a redundant manipulator with the task Jacobian $\mathbf{J}$, orthogonal projection for the range-space $\mathbf{H} \in \mathbb{R}^{D \times D}$ and the nullspace $\mathbf{N} \in \mathbb{R}^{D \times D}$ can be defined as

$$\mathbf{H} = \mathbf{J}^{\mathbf{w}+}\mathbf{J} = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1}\mathbf{J} = \mathbf{J}^T(\mathbf{J}^{\mathbf{w}+})^T \tag{2.7}$$

and

$$\mathbf{N} = \mathbf{I} - \mathbf{H} = \mathbf{I} - \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1}\mathbf{J} = \mathbf{I} - \mathbf{J}^T(\mathbf{J}^{\mathbf{w}+})^T \tag{2.8}$$

to project[1] the primary objective $\mathbf{F} \in \mathbb{R}^6$ onto the range-space, while a joint torque vector $\boldsymbol{\tau}_0 \in \mathbb{R}^D$ (in the case of a torque controlled robot) as secondary objective is projected onto the nullspace [Kha87]:

$$\boldsymbol{\tau} = \mathbf{H}\mathbf{J}^T\mathbf{F} + \mathbf{N}\boldsymbol{\tau}_0 \ . \tag{2.9}$$

While such a strict prioritization ensures that the control signal of a control objective at priority level j does not interfere with higher priority levels i if $j > i$, it also means that the lower priority objectives can only be performed if enough DoFs are available in the nullspace. K levels of objectives can be decoupled using a successive or an augmented approach [DOA15]:

$$\boldsymbol{\tau} = \sum_{i=1}^{K} \mathbf{N}_i\boldsymbol{\tau}_i \text{ with } \mathbf{N}_i = \begin{cases} \mathbf{N}_{i-1}^{suc}\left(\mathbf{I} - \mathbf{J}_{i-1}^{\mathbf{w}+}\mathbf{J}_{i-1}\right) & \mid \text{successive} \\ \mathbf{I} - (\mathbf{J}_i^{aug})^{\mathbf{w}+}\mathbf{J}_i^{aug} & \mid \text{augmented} \end{cases}, \mathbf{N}_1 = \mathbf{I} \ , \tag{2.10}$$

where $\mathbf{J}_i^{aug} = \begin{pmatrix} \mathbf{J}_1 & \cdots & \mathbf{J}_{i-1} \end{pmatrix}^T$ denotes the augmented Jacobian for the objective i. To achieve a soft prioritization without decoupling the objectives, a weighting can be introduced in the nullspace projection [LTP15]:

$$\mathbf{N}_i(\alpha_{ij}) = \mathbf{I} - \alpha_{ij}\, \mathbf{J}_i^{\mathbf{w}+}\mathbf{J}_i \ , \tag{2.11}$$

where α_{ij} indicates the priority of objective i with regard to objective j. Alternatively, a set of control objectives can directly be super-imposed using positive scalar priorities ($\lambda_i > 0$) on the signal-level [BK11; Mor+13b; Deh18]:

$$\boldsymbol{\tau} = \sum_{i=1}^{K} \lambda_i\boldsymbol{\tau}_i \ . \tag{2.12}$$

1 $(\mathbf{A})^{\#}$ denotes the general inverse of a matrix A, whereas $(\mathbf{A})^{\mathbf{w}+}$ denotes the pseudoinverse with weighting $\mathbf{w}$. The moore-penrose pseudoinverse uses $\mathbf{w} = \mathbf{I}$ [DOA15].

2.2.2.2 *QP-Based Prioritization*

Similar to the previous approach based on a sequence of pseudoinverses, which ensure that lower level objectives do not influence higher levels [SS91], a sequence of Quadratic Programming (QP) problems can be formulated, where the solution of each problem rests within the nullspace of the higher priority problem. In the QP literature, control objectives are generally represented as QP tasks. A generic QP task ($\mathcal{S}$) and its constraints at priority level $i+1$ can be defined as [Moh20]

Quadratic Programming (QP)

$$\begin{aligned} \mathcal{S}_{i+1} \;&:= \underset{\mathbf{x}\in\mathcal{S}_i}{\operatorname{argmin}}\ \frac{1}{2}\left\|\mathbf{A}_i\mathbf{x}-\mathbf{b}_i-\boldsymbol{\lambda}\mathbf{e}\right\|^2+\frac{1}{2}\left\|\boldsymbol{\omega}\right\|^2+\frac{\rho^2}{2}\left\|\mathbf{x}^2\right\| , \\ &\text{such that } \mathbf{C}_i\mathbf{x}-\boldsymbol{\omega}\leqslant\mathbf{d}_i,\ \boldsymbol{\omega}\in\mathbb{R}^+ \end{aligned} \tag{2.13}$$

where $\mathbf{A}$ is the task matrix, $\mathbf{b}$ is the vector of task space values, and $\mathbf{C}$ and $\mathbf{d}$ are the constraint matrix and the constraint vector respectively. The problem variable is denoted as $\mathbf{x}$, $\mathbf{e}$ is the task space error, $\boldsymbol{\lambda}$ denotes a feedback gain term, ρ is a regularization factor to bound the solution when $\mathbf{A}$ becomes ill conditioned, and $\boldsymbol{\omega}$ is a positive slack variable [KLW11].

Using Equation 2.13, L control objectives can be realized on the same level of priority within the same QP task. This would achieve a behavior similar to Equation 2.12. By using the concept of augmented task matrices (i.e. Jacobians in Equation 2.10), a soft priority relationship between QP tasks can be imposed using relative weights (β_l). The augmented task matrices and error vectors can be written as [Roc+15]

$$\begin{aligned} \mathbf{A}^{\text{aug}} &= \begin{pmatrix}\beta_1\mathbf{A}_1^T & \dots & \beta_n\mathbf{A}_l^T\end{pmatrix}^T \\ \mathbf{b}^{\text{aug}} &= \begin{pmatrix}\beta_1\mathbf{b}_1^T & \dots & \beta_n\mathbf{b}_l^T\end{pmatrix}^T \end{aligned} . \tag{2.14}$$

2.2.3 *Control in Contact Situations*

To control robots in contact situations, PIDC [Agh05; MR11] provides a framework that employs a strict prioritization to allow contact-consistent motion generation without the need for additional (force-torque) sensors. In contrast to other approaches, PIDC prevents the motion control objective $\boldsymbol{\tau}_M \in \mathbb{R}^D$ from affecting the contact and friction constraints, by projecting $\boldsymbol{\tau}_M$ onto the orthogonal nullspace of the contact constraint using

$$\mathbf{P}\in\mathbb{R}^{D\times D} = \mathbf{I}-\mathbf{J}_c^T\left(\mathbf{J}_c^+\right)^T . \tag{2.15}$$

Thus, creating two orthogonal subspaces: the constrained space $(\mathbf{I}-\mathbf{P})$ used for contact wrench control ($\boldsymbol{\tau}_C$) and the unconstrained space ($\mathbf{P}$) used for motion control. For a fully-actuated and stationary robot the total torque command is given by

$$\boldsymbol{\tau} = \mathbf{P}\boldsymbol{\tau}_M+(\mathbf{I}-\mathbf{P})\boldsymbol{\tau}_C . \tag{2.16}$$

The underactuated case can be solved analytically [Deh+ss] or using numeric optimization techniques [AS16]. Considering Equation 2.4, the desired contact wrenches can be realized through

$$(\mathbf{I}-\mathbf{P})\boldsymbol{\tau}_C = (\mathbf{I}-\mathbf{P})\left(\mathbf{h}-\mathbf{J}_x^T\mathbf{F}_x-\mathbf{J}_c^T\lambda_c+\boldsymbol{\epsilon}\right)\ , \tag{2.17}$$

where $\boldsymbol{\epsilon} \in \mathbb{R}^D = \mathbf{M}\mathbf{M}_c^{-1}\left(\mathbf{P}\boldsymbol{\tau}_M - \mathbf{P}\mathbf{h} + \dot{\mathbf{P}}\dot{\mathbf{q}}\right)$ with the (invertible) constraint inertia matrix $\mathbf{M}_c = \mathbf{P}\mathbf{M} + \mathbf{I} - \mathbf{P}$ mitigates the potential generation of accelerations from the motion control objective in the contact space [Lin+18; Deh+18]. For a discussion on mitigation strategies to achieve dynamically consistent motion control, please refer to [DOA15; Deh+ss]. The motion control realized through $\boldsymbol{\tau}_M$ can be performed in the joint space [Agh05] as well as in the operational space [MR11] using Equation 2.5:

$$\mathbf{P}\boldsymbol{\tau}_M = \begin{cases} \mathbf{P}\left(\mathbf{M}\ddot{\mathbf{q}}+\mathbf{h}\right) & \mid \text{joint space} \quad (2.18a)\\ \mathbf{P}\left(\mathbf{J}^T\boldsymbol{\Lambda}_c\left(\ddot{\mathbf{x}}-\dot{\mathbf{J}}\dot{\mathbf{q}}+\mathbf{J}\mathbf{M}_c^{-1}\left(\mathbf{P}\mathbf{h}-\dot{\mathbf{P}}\dot{\mathbf{q}}\right)\right)\right) & \mid \text{operational space} \quad (2.18b)\end{cases}$$

where $\boldsymbol{\Lambda}_c = \left(\mathbf{J}\mathbf{M}_c^{-1}\mathbf{P}\mathbf{J}^T\right)^{-1}$ denotes the constraint task-space inertia matrix. Note that PIDC can be combined with the SoT approach to realize multiple objectives in the subspaces.

The advantage of using a projection-based approach is the possibility to dynamically shape the nullspace projections [DKS18] to adapt to new task requirements. Whereas, transitions between multiple QP-based SoTs is still an active research topic, even though promising approaches exist [Jar+13]. To overcome the shortcomings of projections in handling inequalities [Man+09; KLW11], which are imposed by the friction cone constraints of unilateral contacts, the PIDC can be combined with QP techniques [Lin+18]. This combination benefits from the advantages of both approaches.

2.3 COMPONENT-BASED SOFTWARE ENGINEERING FOR ROBOTICS

systems involve multiple heterogeneous concerns ○

Up to this point, the focus was set on the control capabilities that are necessary for a robotic system in compliant interaction. Even though, the control capabilities are very important, they are also only a subset of the capabilities that an entire robotic system needs to exhibit. Usually, capabilities originate from different heterogeneous domains, ranging from functional aspects, i.e. motion control, vision-based perception, and planning (including coordination) to non-functional aspects, such as safety, reliability, and timing. At a technical level, additional complexity arises from managing the communication between the sensor and actuator drivers, the execution of the control and planning algorithms, as well as the concurrent access to shared resources [Cal+12]. Mastering the combined complexity of the capabilities is imperative to realize real world robotics applications. Eventually, a robotic system translates to a software system with tremendous complexity, which needs to be maintainable, scalable, and reliable. Hence, mastering the robotic system complexity is closely related to managing the software complexity [Sch+09], which faces

capability concerns ○

software and hardware concerns ○

the additional challenges of integrating different software frameworks. This leads to the wide-spread consensus in the robotics community that software engineering principles need to be applied to robotics. Especially, considering the heterogeneous nature of robotics, the principle of separation of concerns reduces the complexity by approaching the problem in a *divide and conquer* manner. While this paves the way for a clear assignment of responsibilities and roles, it also prevents an undesired entanglement of domain aspects and hidden assumptions regarding other concerns. An intended by-product is modularity, which can lead to reusability as long as there are no hidden assumptions present and all relevant properties are explicitly specified.

○ *robotics requires software engineering*

Unfortunately, a lot of robotics software do not yet follow a software development process and often heavily tie software- and hardware-agnostic algorithms to particular software implementations, robots, and application-specific knowledge. This directly hinders scalability and may ultimately prevent the increase of the Technology Readiness Level (TRL),[2] necessary to transition from a laboratory to a relevant validation environment. Furthermore, it also prevents the reuse and adaptation of the system to be applied to even a slightly different application [KSB16]. This is an unacceptable disadvantage, considering the experimental nature of many robotic systems, where individual functional and non-functional parts are quite commonly subject to change. Nowadays, different approaches exist that especially address the challenges of reuse and adaptation in terms of reconfiguration. The most prevalent approaches apply component-based software engineering (CBSE) principles [HC01; CL02; SGM02] to robotics, giving rise to component-based robotic systems (CBRS) [Soe06; Kou16; OSR; Nor+16a; BS09; BS10; Wie18].

>_ *Technology Readiness Level (TRL)*

>_ *component-based software engineering (CBSE)*

>_ *component-based robotic systems (CBRS)*

In general, CBSE arose from the software engineering community with the goal to shift the focus from traditional programming to building software systems by composing self-contained components [Bro+05; Sch+09]. The main idea behind a component-based system is to *divide and conquer*. By splitting up the overall problem into sub-problems, which individually solved, yield the solution to the overall problem upon composition. By separating the component development process from the system development process, major benefits, such as reusability and isolated development of components, are achieved [Sch+09]. This means that components can be reduced to their specification in terms of e.g., interfaces and system-relevant non-functional properties, thus allowing a seamless replacement of component (implementations) as well as an isolated development of the system and the components' internals [Bro+05]. Indeed, the adaptability of the system and the reusability of functionality is significantly dependent on the particular composition and interaction of components [BS10]. For CBSE to live up to its promises of increased reuse, reduced complexity, and hence reduced production costs, it is of upmost importance that the components are actually reusable [Sch+09] (i.e. free of hidden assumptions that would prevent reuse) and that non-functional properties are correctly reflected in the documentation.

>_ *component*

○ *achieving reusability, maintainability, and reconfigurability through modularization and composition*

2 www.nasa.gov/directorates/heo/scan/engineering/technology/txt_accordion1.html

2.3.1 *Separation of Concerns*

separation of concerns (SoC)

SoC reduces the coupling and increases the cohesion

The principle of separation of concerns for CBRSs separates the different aspects of such systems' into five concerns ("5C's") [Pra+09; Bru+13] to decrease their coupling while increasing the cohesion. In contrast to the "4C's" defined in [RE96], the additional concern of *Composition* is emerged from the original *Configuration* concern. The rationale for this is that the composition of individual parts is as important as the modularity itself [VKB14].

2.3.1.1 *Separation of Concerns: C1 Computation*

Computation represents the functional core of the system in form of e.g., components. The concept of a component is widely adopted with varying definitions, depending on the aspect that is emphasized. While Crnkovic et al. [Crn+02] present a minimal definition of the term *component*, based on a review of several definitions, Szyperski et al. give a similar but extended definition:

Definition: Software Component

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."[a] [SGM02].

[a] The definition originated from the Workshop on Component-Oriented Programming at the 1996 European Conference on Object-Oriented Programming (ECOOP).

Even though there exist several definitions for components, there is a common agreement that a component

1. implements domain knowledge [Bru+13] in form of data processing algorithms using data structures and operations [BS10],

2. requires concurrent read and write access to external data sources,

3. needs to provide a specification of non-functional properties as well as a documentation of design aspects [Crn+02] that conform to the paradigms of the framework they are supposed to be composed in.

a component implementation depends on the component model of the underlying software framework

A component designed for the underlying component model of a specific framework (e.g., OROCOS RTT [Soe06]), will be different from a component designed for another framework (e.g., ROS [Qui+09]), since the actual realization of the separation of concerns plays an important role. This also means, that components designed for one framework may not be reusable in another framework, due to the possibly mismatching criteria for reuse (i.e. interfaces), defined by the specific instantiation of the separation of concerns.

2.3.1.2 *Separation of Concerns: C2 Communication*

Communication is concerned with the data exchange of computational components. This includes non-functional aspects, such as time, bandwidth, latency, accuracy, and priority [Bru+13]. Different communication patterns can be employed between components to achieve a desired behavior [KSB16]. The most common communication mechanisms[3] are anchored in object-oriented programming, namely *publish–subscribe* and *client–server*.

2.3.1.3 *Separation of Concerns: C3 Coordination*

Coordination defines the behavioral aspects of a system, by determining the cooperation or competition between components to achieve a desired behavior. This includes managing the concurrent access to shared resources, the application-related behavior coordination, and the timed execution of components to ensure that required data is available when needed. The latter is especially important for real-time constrained robotic systems that require certain response times to produce a stable behavior. The execution semantics of components are also covered by this concern, targeting the individual execution time of a component as well as the overall response time of a defined sense-react chain (see Chapter 4).

>_ *sense-react chain*

A system can be coordinated on different levels. A system-level coordination is able to address component-overarching aspects, while a component-level coordination defines the behavior of a component. There exist numerous coordination approaches [MAC97]. The majority of these approaches, such as state machines [KB12a], behavior trees [Flo+09; Ogr12; Bag+12], and Petri nets [ZM94], define a behavior in a discrete way, by assuming an implicit or explicit state-space.

2.3.1.4 *Separation of Concerns: C4 Configuration*

Configuration is a concern that is twofold. Firstly, it relates to the structural aspect of a CBRS, by defining the involved components and their connections to each other [BS10], while being constrained by the expected communication pattern and the components' interfaces. Secondly, the concern allows the parameterization of components, based on domain knowledge to adapt the system to achieve a desired behavior, necessary for a specific application. Common purposes for configuration are e.g., tuning control gains and safety limits, changing the controlled DoF, defining component interaction policies, and allocating software and hardware resources [Bru+13].

2.3.1.5 *Separation of Concerns: C5 Composition*

Composition is a cross-cutting aspect that is concerned with coupling the previously introduced "C's", which are instead driven by the desire to decouple the design concerns as much as possible. According to Bruyninckx et al. [Bru+13], every composition introduces a trade-off between *composability* and *compositionality*. While the former describes the degree of reusability of a component, the latter takes the system's view point, describing to what degree the behavior resulting from a composition can be inferred through the knowledge over the

3 For more information on the communication mechanisms, please refer to [BS10].

constituent components. According to Brugali and Shakhimardanov [BS10], practical experience in software engineering has demonstrated that effective reuse of components can be achieved

1. if their inter-connections are flexible and their interfaces harmonized enough to accommodate changes in user requirements and the substitution of components;
2. if the components conform to domain-specific composition rules;
3. if no hidden assumptions are present that prevent assessments about the predictability of the system's behavior.

2.3.2 *Separation of Roles*

separation of roles (SoR)

By separating the concerns, different roles can be introduced to distribute the development responsibility per (domain) expertise. These roles extend on the roles defined in RobMoSys' Body of Knowledge.[4] Note, that only the roles relevant for this work are described below:

BEHAVIOR DEVELOPER is responsible for developing a task description for the robot's behavior by introducing task-specific constraints (i.e. motion constraints and coordination constraints). To realize a task description, usually a constraint-conforming orchestration of e.g., skills needs to be developed by this role. While a skill represents a specific functionality, it abstracts from its specific implementation, which is based on an orchestration of components. Hence, ensuring the separation of concerns towards the role of the *Component Supplier*.

COMPONENT SUPPLIER is responsible for developing components that implement the functionalities provided by the *Function Developer*, or that interface with hardware devices managed by the *Robotic Platform Engineer*. Since the components are software components, they are implemented towards a specific software platform, which is in turn managed by the *Software Platform Engineer*. In a non-model-based software development process, these roles are tightly coupled.

PERFORMANCE DESIGNER is responsible for the properties related to the timing of a system. This is a non-trivial task that is often neglected in robotics, but is essential for a stable, predictable, and safe execution of the tasks, particularly in a real-time and safety critical application. The *Performance Designer* defines the execution semantics of a system. This includes a schedule for the execution of the components, involved in the sense-react chains envisioned by the *System Architect* and realized by the *System Builder*. The resulting schedule needs to conform to possible timing-related constraints on the execution that

4 https://robmosys.eu/wiki/general_principles:ecosystem:roles

influence the mapping to a (CPU) core, the execution order, and the activation semantic of components. While several aspects of this role are platform-agnostic, there are other aspects that can only be addressed in close relation to the software and hardware platforms that the system will be executed on.

ROBOTIC PLATFORM ENGINEER is responsible for the hardware used by the system and its integration. This entails the robots, individual sensors and actuators, as well as the processing hardware for the software platform. This role needs to interact with different roles, such as the *Component Supplier* (e.g., hardware drivers) and the *Performance Designer* (e.g., allocation of hardware resources). The rationale for this role, which is not present in the current RobMoSys' approach, is to separate the hardware from the software aspects and to provide a dedicated responsibility to manage the hardware platform independent of the task-specific functionalities.

○ *additional role to explicitly cover the hardware platform concerns*

SOFTWARE PLATFORM ENGINEER is responsible for the software frameworks. This includes e.g., the component execution framework, the middlewares, and other required software artifacts. Apart from managing the software platform, this role is obliged to assist in mapping the system, constructed by the *System Builder* to the associated software platform by providing required, but missing information. This role is also not present in the current RobMoSys' roles. The rationale for this role is to separate the software from the hardware aspects and to provide a dedicated responsibility to manage the software platform independent of the task-specific functionalities.

○ *additional role to explicitly cover the software platform concerns*

SYSTEM BUILDER is responsible for composing the (software) components, which are provided by the *Component Supplier*, into a system according to the blueprint of the reference architecture, constructed by the *System Architect*. The main activity of this role is to select function-wise suitable and interface-wise compatible components that suffice the non-functional constraints regarding e.g., the timing (i.e. sense-react chains), the safety, the performance, and the robustness. Another activity of this role is to provide the resulting system ready for deployment.

2.4 MODEL-DRIVEN ENGINEERING FOR ROBOTICS

Component-based software engineering paradigms only enable the separation of concerns and roles for aspects related to the system's component architecture. A robotic system however, involves highly heterogeneous concerns from various domains, such as motion planning and control, multi-modal perception, machine learning, or interaction design. Such a composition of concerns requires the conceptual and technical combination of the expertise from the different domains, to jointly realize the requirements of a desired robotics application. Even using traditional software engineering (e.g., component-based paradigms), it is too complex and expensive to achieve such a composition while ensuring reuse, flexibility, and adaptability [Sch+10] on the conceptual and technical level. The main reason is that the different artifacts lack interoperability, hence preventing to achieve the aforementioned properties [JZ17].

MDE

Model-driven engineering (MDE) [Sch06; SVC06; ZC06; WHR14; Rod15] has proven to be suitable to address the challenges of similarly complex systems as advanced robotics [Nor+16a], which require high reliability and robustness [Sch+10]. Successful applications have been shown in domains, such as avionics or automotive [vKV00; Sch+10; Ara+21]. In general, MDE describes a software development methodology, which introduces a model level that abstracts from the software artifacts. A model is always an abstraction of something[5] with the focus on purpose-relevant properties; or as Jeff Rothenberg et al. put it:

model

introducing models to abstract from the technical realization

Definition: Model

"Modeling in its broadest sense is the cost-effective use of something in place of something else for some purpose. It allows us to use something that is simpler, safer, or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality." [Rot+89]

With the introduction of models, the responsibility of achieving reuse, interoperability, and the assessment of quality aspects, such as safety, correctness, and performance [Sch+10] is lifted from the source-code level. This way, the modeled aspects can be represented independent of their actual realization, which enables a higher understandability and platform-independence, and reduces the risk of making costly errors even without an actual source-code level realization [Ara+21; Bru15]. Using the concept of code generation, the models can systematically be transformed into different target implementations, achieving a traceable and efficient link to the technical realization of a system. With the introduction of a model level, MDE has the potential to break the cycle of robotics software starting from scratch again and again. Further, paving the way for a cross-fertilization between software engineering and robotics [Sch+09].

MDE already used in robotics

In the field of robotics, MDE approaches primarily focus on describing robotics concerns via conceptual models (i.e. domain models). These models are based on concepts that raise the level of abstraction to focus on properties relevant to the domain while ignoring technical ones [Sch+09]. This leads to an easier understanding and validation, as well as to a lower technical skill requirement for developers to handle the complexity of the robotics system development [Nor+16a]. Further, an increased level of automation, i.e. through model transformation and model interpretation can be achieved that bridges the gap between models and implementation. MDE facilitates the SoC and SoR [Sch+15; Sch+09], hence improving the efficiency and quality of the robotics systems engineering process [Nor+16a].

5 Sometimes called *system*, which does not necessarily refer to a robotic system, though it could.

2.4.1 *Domain-Specific Languages*

A recently trending and quite promising method in the field of MDE is to capture the conceptualized domain knowledge with DSL [Nor+16a]. In addition to the increased level of conceptual abstraction offered by MDE approaches, DSLs provide accompanying notations to the models that are closer to the problem domain than general-purpose languages (GPLs) [Nor16]. Again, increasing understandability and efficiency.

>_ *domain-specific language (DSL)*

Definition: Domain-Specific Language

A DSL is a language[a] of limited expressiveness, focused on a particular domain [Fow10]. It enables the creation of a "program" or "source code"—which is commonly referred to as language model—using the notations of the targeted problem domain. The semantically relevant information expressed by a language model is represented using an abstract syntax as data structure [Völ+13]. The abstract syntax is in turn defined by a meta-model [Com+16]. Hence, a model always conforms to at least one meta-model. It is also possible that a (*reflexive*) model conforms to itself, occupying the roles of the model and meta-model at the same time (see Figure 2.3).

[a] While in general a DSL can be also referred to as programming language, I will only use this term for general-purpose languages in the course of this thesis.

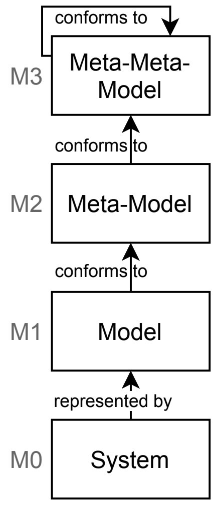


Figure 2.3: Meta-modeling layers [Béz05].

The DSLs used throughout this thesis consist of different aspects to enable the creation of a model (see Figure 2.4): The *structure* aspect of a DSL describes the abstract syntax, i.e. meta-model in form of concepts that act as data structures. These concepts allow a model to represent semantically relevant information in form of a tree or a graph, i.e. abstract syntax tree (AST). In addition to the structural restrictions, further constraints can be introduced via the *constraints* and *type system* aspect, defining the static semantics of a language. In contrast to the abstract syntax, multiple concrete syntaxes can be defined in the *editor* aspect. The concrete syntax is the interface to the language user. It can either be textual, tabular, graphical, or a mixture depending on the purpose and notations of the targeted domain. One optional aspect of a DSL are the *transformations*, defining the execution semantics of a model. By applying a transformation, a source model can be transformed into one or more target models, which conform to the same (homogeneous transformation) or to a different (heterogeneous transformation) meta-model as the source model. The described aspects of a DSL are realized in one or more language modules. A very common approach is to extract the transformations aspect into a separate generator module to increase the SoC. The reader may refer to [Völ+13; Fow10; Com+16] for further information on DSLs and their benefits.

>_ *concrete syntax*

>_ *homogeneous trans.*

>_ *heterogeneous trans.*

>_ *language module*

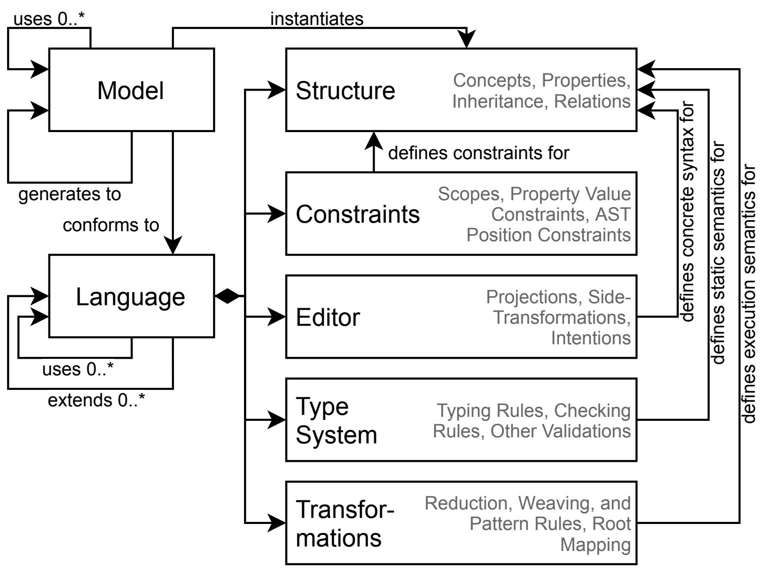

uses 0..*
instantiates
Model
generates to
conforms to
Language
uses 0..*
extends 0..*
Structure
Concepts, Properties, Inheritance, Relations
defines constraints for
Constraints
Scopes, Property Value Constraints, AST Position Constraints
Editor
Projections, Side-Transformations, Intentions
Type System
Typing Rules, Checking Rules, Other Validations
Transfor-mations
Reduction, Weaving, and Pattern Rules, Root Mapping
defines concrete syntax for
defines static semantics for
defines execution semantics for

Figure 2.4: Extension of the language design aspects presented in [Voe15].

2.4.2 *Language Modularization and Composition*

DSLs not only provide an efficient methodology to model the individual concerns of the different involved robotics domains, but also offer the potential to combine them on the model level and to establish a link towards a conforming implementation on the source-code level via model-transformations. This is necessary to enable the design, integration, and verification of entire robotics applications at the model level. However, to achieve this a unifying language modularization and composition (LM&C) approach is required that allows meta-model, model, and transformation composition. Note that this is a very challenging task (see Section 3.1.3). According to the literature, DSLs are commonly modularized and composed along two orthogonal axes (vertical and horizontal) [RMT14], which are not necessarily used together.

language modularization and composition (LM&C)

2.4.2.1 *Vertical Modularization and Composition*

Vertical LM&C as proposed by Object Management Group (OMG) uses the following four model types, each on their unique level of abstraction, to separate platform- and implementation-specific model aspects from the domain-specific requirements of the to be modeled concern [Sie14] (see Figure 2.5). With this, e.g., a component-based control architecture can be exchanged and reused independent of the software platform that is chosen for execution (e.g., ROS).

vertical LM&C is used to separate abstraction levels

Computation-independent models (CIMs) belong to the category of domain models. They are abstractions of "real" entities from a target domain, which could be anything from e.g., timing, kinematics, to hardware, or human-robot interaction requirements.

Platform-independent models (PIMs) in contrast, are closer to an information system's point of view, by describing logical aspects that are independent of a particular technology. PIMs in robotics are commonly used to describe the components involved in a component-based system, how they are inter-connected, and what kind of communication pattern are used, etc.

Platform-specific models (PSMs) belong to the category of implementation models describing aspects or requirements introduced by using specific target technologies or platforms. For instance, a PSM can be used to describe a component-based architecture that explicitly uses ROS communication mechanisms allowing to model valid ROS topics only. Thus, a PSM directly inherits the semantics of the target platform.

Platform-specific implementations (PSIs) are not per se models. The acronym is added by Ramaswamy et al. [RMT14]. I use this type to label the aspects that exist at the border of what is considered outside the modeled system. A PSI can be plain source code, but it can also be a model which is part of another modeling environment.

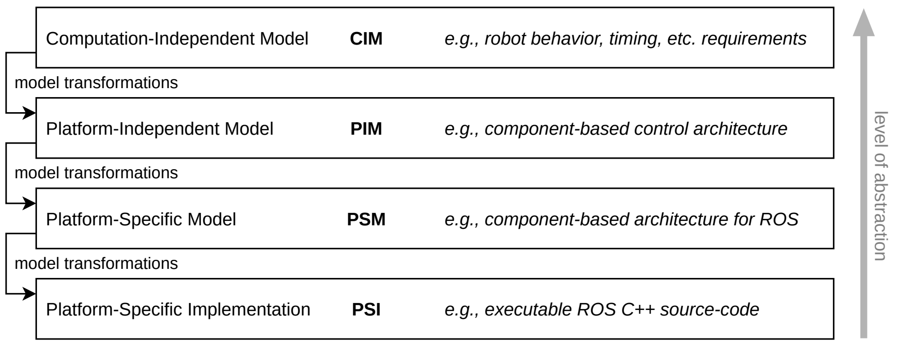

Computation-Independent Model
CIM
e.g., robot behavior, timing, etc. requirements
model transformations
Platform-Independent Model
PIM
e.g., component-based control architecture
model transformations
Platform-Specific Model
PSM
e.g., component-based architecture for ROS
model transformations
Platform-Specific Implementation
PSI
e.g., executable ROS C++ source-code
level of abstraction

Figure 2.5: Vertical LM&C of models, according to [Sie14].

In this vertical LM&C, a certain development process is implicitly dictated. First, the domain-specific requirements of a concern need to be modeled as CIM. Model transformations incrementally decrease the level of abstraction by producing a model based on the previous more abstract model. On each level, the model can be enriched with additional platform- or implementation-specific knowledge. After the final transformation, a software artifact is produced, e.g., an executable ROS-based control system implemented in C++.

○ *this kind of vertical LM&C is applied by several works [Alo+10; Sch+09; Nag+15; Bru+13] to robotics*

2.4.2.2 *Horizontal Modularization and Composition*

Horizontal LM&C is used to separate the different concerns of a model at the same level of abstraction. This enables the introduction of viewpoints [Völ+13] per concern or role (see Section 2.3.2) to manage the complexity by hiding non-relevant parts of the model.

○ *horizontal LM&C is used to subdivide a level of abstraction*

Horizontal LM&C is used in cases where a vertical LM&C would artificially loosen the coupling between naturally connected aspects in favor of modularization, which leads to the detriment of composition. For instance, the *computation* and *communication* concerns of a CBRS are mutually influencing each other. A change of the input parameters of an algorithm needs to be reflected by

○ *modularization does not make sense without an equally important composition* the component's communication interface that is used to retrieve the required data, and vice-versa. Instead, using a vertical separation the influence of the different concerns needs to be achieved beyond model transformation borders, which in the case of the commonly used uni-directional model transformations, is not feasible. The same holds true for the realization of viewpoints of a set of relevant aspects, which are separated by model transformations.

In case of modeling component-based robotic systems, horizontal LM&C is usually inspired by the 5C's (see Section 2.3.1), cf. [Bru+13; Nor16; Alo+10; Dho+12; Sch+09].

2.4.2.3 *Language Composition Mechanisms*

In order to achieve the composition of DSL modules, different conceptual mechanisms can be found in the literature, which are heavily inspired by well-known composition patterns of object-oriented programming [EGR12]. Voelter et al. [Voe11] propose four different types of composition, which I use throughout this thesis:

REFERENCING occurs in the case where a model based on a language references fragments of a model based on another language. The important part here is that the referenced model fragments cannot reside in the same model where the reference originates. In this case, the referencing language directly depends on the referenced one.

model fragment ⧼

REUSE is similar to referencing. However, no direct dependencies between the languages are introduced. This means that only the models are connected through a dependency.

EXTENSION introduces a direct dependency between the extending and the extended (base) language. In contrast to the previous types, the extended and extending model fragments reside in the same model. Thereby requiring composition on the syntactic level as well. Extension can also be used to restrict the scope of the extended language, by e.g., constraining the use of certain concepts and their interaction with other aspects of the language [EGR12].

EMBEDDING combines the advantages of extension and reuse. Hence, syntactic integration of previously unrelated languages can be achieved, while not having to introduce dependencies between the languages or having to modifying either of them. Thereby, rendering embedding a non-invasive kind of composition, which allows the languages to maintain their modularity and enabling heterogeneous model fragments to be composed into the same model. An *annotation* is a mechanism that is a specialization of embedding, which performs syntactic composition, without introducing any kind of dependency between the elements of unrelated models. Annotations can be attached to arbitrary elements of the AST without interfering with the annotated elements, while presenting a seamless combination of the models' concrete syntaxes [Voe11; Rat+12].

2.5 CONCLUSION

This chapter discussed the heterogeneous knowledge that combined is required to create a robotic system, capable of interacting compliantly with the environment. Defining the task or application that needs to be performed is essential to the design of a robotic system. A skill-based approach, or on a lower level constraint-based programming, is commonly used for that purpose. The execution of such tasks is done using control approaches that generate a control signal, which is sent to the robot. Numerous works exist, proposing suitable control approaches that reactively cope with external disturbances and that realize multi-objective control in contact situations. Note that even though the SotA in contact control has advanced beyond the presented approaches, I chose to only rely on already consolidated and mature approaches as foundation in this work. Apart from the domain of control, literature on nearly every individual concern of a robotic system exist. However, their essential composition is alarmingly underrepresented.

Different software principles, i.e. component-based software engineering, SoC, SoR, and MDE, are used by the robotics community to manage the complexity of the system design by separating the concerns, and abstracting from the eventually used software and robotics hardware platforms. DSLs provide an efficient methodology that enables the design, integration, and verification of robotics applications at the model level as well as to establish a link to the source code level via transformations. However, a survey [Nor+16a] conducted by us, showed that also in the increasing field of robotics DSLs, the focus is primarily on the individual self-contained concerns and seldom on their composition. 779 journal, conference, and workshop publications in the field of robotics were surveyed. Following up on the survey, we host the Robotics DSL Zoo,[6] which is a curated online database of surveyed publications. Due to its community-driven nature, it is continuously but irregularly updated.

>_ *Robotics DSL Zoo*

To achieve the interpretability of a system and the traceability between the system and the resulting robot's behavior, the heterogeneous concerns of a robotic system need to be explicitly and interpretably combined on the model level in a way that a consistent and conforming realization on the source code level can be realized. This also includes the actually used software frameworks and robotics hardware, which are integral parts of any real robotic system.

MDE publications that do consider the need and challenges for modularization and composition, however, rarely tailor their LM&C approach towards the robotics domain. In most cases, there is only a vertical SoC between the functional PIMs and the software framework, targeted for execution as PSMs. Aspects such as timing and hardware are often either completely neglected or implicitly hidden in the PIM or PSM. Approaches that do employ horizontal in addition to vertical SoC, seem to limit its application to the PIM layer [RMT14], with the primary focus on the 5C's for component-based systems. Hence, neglecting the component-agnostic concerns on the CIM level, such as a task description, behavior definition, as well as timing and safety requirements.

6 Feel free to contribute: `https://corlab.github.io/dslzoo/contribute.html`.

Part I

COSIMA

This part introduces CoSiMA, which is a composable solution to realize executable real-time robotic systems. CoSiMA uses language modularization and composition mechanisms to achieve a consistent composition of the heterogeneous concerns of robotic systems. It consists of a modeling and an execution part. The modeling part is concerned with the composition of the requirements on different abstraction levels. Whereas, the execution part is based on a real-time capable execution environment and includes a proposed PIDC-based reference architecture for realizing CI tasks.

"I consider nature a vast chemical laboratory in which all kinds of composition and decompositions are formed."

— Antoine Lavoisier

3 MODELING ROBOT CONTROL SYSTEMS

This chapter introduces a language modularization and composition approach that addresses the challenges of domain-specific languages for highly heterogeneous domains, such as robotics. CoSiMA, a concrete instantiation of this approach for the model-driven engineering of component-based robot control systems, is presented. CoSiMA is extended and used throughout this thesis to model the robotics case studies. This chapter closes with an evaluation of the modularization and composition structure of CoSiMA. The chapter is based on [Wig+17b; Wig+17a; Wig+18; Nor+16a; Wie+18].

Advanced robotic systems such as in service, entertainment, or versatile industrial robotics with cognitive and compliant interaction skills require the expertise from highly heterogeneous domains, such as motion planning and control, multi-modal perception, or human-robot interaction, conceptually and technically combined in a coherent design to jointly realize the application-specific requirements that define the different aspects of a robot's behavior and the underlying system. The use of MDE in form of DSLs significantly reduces the complexity by supporting SoC and SoR through the explicit separation of the conceptual requirements (i.e. model level) from the implementation (i.e. source code level) of individual robotics concerns. Model transformations are used to establish a link between the levels (see Section 2.4).

However, to bridge the gap between an envisioned and the actual robot's behavior and to answer RQ 1, a LM&C approach is required that achieves the consistent composition of system-unrelated (e.g., task and behavior) concerns, system-related but platform-independent (e.g., architecture and coordination) concerns, and platform-specific (e.g., OROCOS RTT-related) concerns. Associated model transformations enable the synthesis of a consistent composition on the source code level that realizes the requirements of the modeled concerns, which define the robot's behavior and the system's quality properties, as an executable software artifact. Hence, languages, models, and transformations need to be composable.

>_ *language modularization and composition*

RELATED WORK ON LM&C IN ROBOTICS Ringert et al. [Rin+16] present an approach for language and code generator composition of CBRSs systems. As the majority of related approaches, the authors apply a horizontal separation of structural, coordination, and configuration concerns on the PIM level. A similar separation is used by the Mauve DSL [LDC12], which provides modeling and analysis support for the real-time execution of CBRSs. Both approaches neglect the modeling of software and robot hardware platforms, which are essential parts of a robotic system. As a result, the required robotics-specific knowledge is hidden in the model transformations, which directly transform the PIM into a chosen PSI (i.e. OROCOS RTT), while completely skipping the PSM. V^{3}CMM provides a platform-independent modeling approach for component-based application design. In addition to the horizontal separation on the PIM level, it uses a PSM conform to OMG's vertical separation (see Section 2.4.2.1). The PSM however does not cover any software- or robot hardware-specific aspects. Hochgeschwender et al. [Hoc+13] propose a model-based approach for software deployment in robotics, which allows the explicit modeling of software and hardware platform concerns. While platforms are models in terms of memory, processors, threads, and busses, the robotic hardware (e.g., joints, links, control interfaces) is again not considered. Similarly, the RobotML [Dho+12] DSL is designed to model and deploy CBRSs. It uses a robotics-specific deployment model that covers the target software platform (i.e. OROCOS RTT) and a robot simulator (i.e. Morse). In contrast to Hochgeschwender et al.'s DSL, an overarching composition between the deployment and structural concerns is not possible. Further, the authors provide no details on how the platforms are actually modeled. Both approaches, however, hide software framework-specific knowledge in the code generators. The BRICS component model [BG16] and the BRIDE toolchain [BWV14] provide meta-models and a development process to model CBRSs in a framework-agnostic way. While the approach does not consider the robotic hardware, it provides explicit meta-models to model platform-specific Component-Port-Connector (CPC) systems for different robotics software frameworks (i.e. OROCOS RTT and ROS). Due to the vertical SoC between the PIM and the explicit PSM, model transformations are required to generate a system model that contains platform-specific knowledge from a platform-independent model.

structural concern

focus is predominantly on the (in)dependence on software frameworks

The majority of approaches applies some kind of horizontal SoC for concerns related to the 5C's, while separating the platform-independent aspects from the platform-specific ones. However, the consideration of the software and hardware aspects, appears to vary from being only implicitly considered inside the transformations, over considering only the hardware or the software, to providing explicit modeling support for both aspects. Even though the different publications are concerned with the LM&C of different robotics system aspects, only Ringert et al. describe their LM&C architecture, including the necessity for the composition of transformations, in detail.

LM&C is rarely robotics-specific and explicitly described

3.1 L3DIM: LAYERED 3 DIMENSIONS APPROACH

In the following I introduce the Layered 3 Dimensions (L3Dim) approach which is a LM&C architecture, specifically tailored towards covering the different concerns of a robotic system for CI. In contrast to the majority of related works that use vertical and horizontal SoC with an almost exclusive focus on software-related concerns, L3Dim combines vertical and horizontal SoC based on the natural interactions and couplings of the robotics capabilities-, software-, and hardware-related concerns. Figure 3.1 shows an exemplary instantiation of the approach, which is used to aid the explanation below.

>_ *compliant interaction*

>_ *capability*

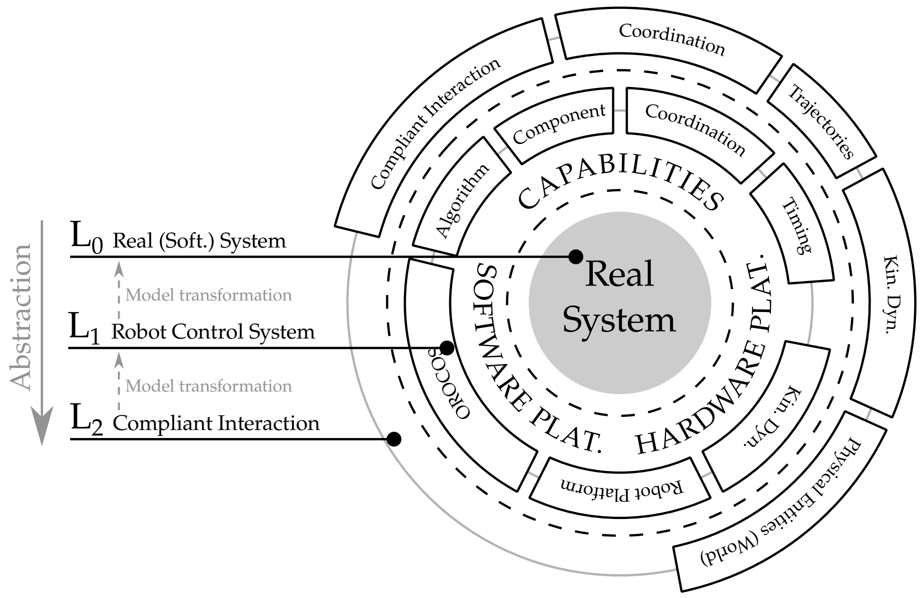


Figure 3.1: An exemplary instantiation of the L3Dim approach with the main focus on modeling a component-based robot control system that performs a specific compliant interaction with the environment. The size of the layers (L_i) is of no relevance. Layer borders are marked as dashed lines. Model transformations are used to cross from a more abstract to a less abstract layer. Layers are vertically separated. However, for readability reasons the vertical axis runs from the center outward. E.g., layer L_1 is on a higher level of abstraction than L_0.

3.1.1 *Vertical Layers*

L3Dim vertically separates concerns by their modeling purpose, to achieve high cohesion of naturally related aspects on the individual layers. To model a CBRS this means that platform-independent (i.e. capabilities) and platform-specific (i.e. software and hardware) concerns are considered to reside on the same level of abstraction, since both kinds of concerns are essential to the very nature of a CBRS, coexisting and mutually influencing each other. For instance, hardware constraints have an impact on the choice of the software frameworks, and the kinematics of a robot influence the choice of a certain capability (e.g., a control algorithm). In contrast to that, a classical vertical SoC would artificially

○ *vertical SoC w.r.t. the modeling purpose to support natural cohesion* decouple and separate the platform-independent and -specific concerns onto different levels of abstraction. Hence, any influence would only happen unidirectional via model transformation, effectively preventing the natural and mutual influence of the concerns. As a result, a PSM is created by a transformation of the PIM, supplied with the required platform-specific information. Analogous to the development process in [BWV14], each change to a PIM system model, would require redoing the transformation into a new PSM as well as supplying the missing platform-specific information again. Instead, L3Dim explicitly supports the bidirectional interaction of the concerns by composing them on the same abstraction level.

no vertical SoC of PIM and PSM

In L3Dim, each layer L_i has its own modeling purpose and contains only the aspects of concerns related to that purpose. As shown in Figure 3.1, three layers are chosen to model CBRS for CI. Note that depending on the chosen level of granularity and the different modeling purposes, more or less layers can be introduced. The important point is that a high cohesion is achieved in the individual layers. The least abstract layer (L_0) represents the real robotic system. Producing an executable software artifact to realize such a system is the overarching purpose of the modeling approach. Hence, the purpose of L_1 is to abstract the real system in form of a component-based system model, involving functional and non-functional capabilities, software, as well as hardware platform aspects to the necessary degree. L_2 is the layer with the highest abstraction. Its purpose is to model the desired robot behavior in physically compliant interaction with the environment. The models in each layer do not know about the modeling purpose of any other layer. Instead, they solely focus on their own purpose. L_2 for instance, models the compliant interaction, but does not care about how the modeled information is used (i.e. to create a robotic system). This ensures high cohesion towards a layer's modeling purpose and low coupling between the individual layers.

Each layer (L_i) is considered as a set of constraints and requirements that the next less abstract layer (L_{i-1}) needs to conform to. As in classical vertical SoC, L3Dim connects the individual layers via unidirectional model transformations. These transformations, are used to realize the set of constraints and requirements by creating and changing model fragments on the next less abstract layer. For instance, the definition of a compliant behavior on L_2 demands a suitable control (e.g., impedance) component to be modeled on L_1.

model transformations decrease the abstraction level and realize constraints on less abstract layers

Starting the modeling process at the highest layer, allows the model transformations to directly realize some of the specified requirements by generating the respective model fragments. Other requirements however, can not be inferred from the more abstract models and need to be manually modeled on the lower layer.

3.1.2 *Horizontal Dimensions*

PIM and PSM concerns coexist on the same vertical layer

L3Dim uses a horizontal separation of platform-specific and platform-agnostic concerns to facilitate their composition by keeping them on the same level of abstraction. However, in related work, a vertical SoC is usually used to separate both types of concerns. The advantage of using a horizontal SoC instead is that it increases the exchangeability through coexistence (see Section 3.4.1.2) and enables the mutual influence of the concerns in form of imposing overarching constraints.

Inspired by the work of Ratiu et al. [Rat+12], L3Dim introduces the concept of horizontal dimensions. Since L3Dim was initially designed to support the modeling of robotics control systems, three orthogonal dimensions were chosen for that purpose, considering the insights gained from the Robot Application Development Process [Bis+10]: *Hardware Platform*, *Software Platform*, and *Capability*. These dimensions are used to categorize orthogonal concerns on the same layer of abstraction to enable their flexible composition based on the composition rules of their associated dimension. This allows the concerns to mutually influence each other and to express dimension-overarching relationships between model fragments.

>_ *dimension*

○ *dimensions maintain concern modularity while enabling explicit composition*

- *Hardware Platform* (platform-specific) dimension: representing any kind of hardware that is involved in a robotic system. This includes robotic platforms and their kinematic structure, robot-specific constraints and interfaces, as well as dynamics, safety constraints, control interfaces, and so on. Explicitly considering the hardware platform has proven to be an essential concern for robotic systems, since it is addressed by various contributions, e.g., [FBC13; RGG94; BSS11; FBC12].

- *Software Platform* (platform-specific) dimension: describing the characteristics of software frameworks (i.e. middlewares) and their requirements for code generation. In contrast to the hardware platforms, this dimension mainly focuses on component frameworks (e.g., OROCOS RTT and ROS [Qui+09]), middlewares (e.g., YARP [MFN06], RSB [WW11]), as well as all other software-related aspects, such as interface protocols (e.g., the Fast Research Interface (FRI)[1]). The majority of the represented software frameworks in this dimension provides requirements and essential information for code generation.

- *Capability* (platform-independent) dimension: defining isolated aspects that are reusable and that can be composed to cover the functional and non-functional concerns of the system. This dimension builds the third pillar of the approach, covering the functional parts of a system that are agnostic to any software or hardware framework. It includes concerns from different (sub)domains, such as coordination, vision, motion generation, etc.

In this work I mainly consider robotic systems that encompass a set of capabilities that are adapted to a set of software frameworks and are deployed on one or more different robots in the real world as well as in simulation.

1 `web.archive.org/web/20180704013224/http://cs.stanford.edu/people/tkr/fri/html`

3.1.3 *Guidelines to Support Language Evolution for LM&C*

language evolution ⎯<

language evolution needs to be considered to achieve an extendable, reusable, and stable composition ○

Especially in the field of robotics, which involves heterogeneous concerns from numerous well-established and newly emergent domains, a LM&C approach needs to explicitly support language evolution to achieve an extendable, reusable, and stable DSL and model composition [Nor+16a; CP10; EGR12; RDN10; Whi+09; Erd+15]. As long as a (sub-)domain is not entirely covered by a DSL, language evolution is imminent, since the chosen abstractions are likely to be extended over time [RDN10; Erd+15]. Even a completely explored domain is usually not set in stone, especially when our understanding of it changes [Whi+09; EGR12]. Hence, the related DSLs need to be changed to incorporate the correct domain abstractions. Other aspects may even be completely removed from the composition in order to be replaced, such as an outdated software platform. The impact of language evolution on the affected DSLs and the global composition can range from tiny to massive. Especially an unpredictable evolution can have devastating effects on a LM&C that is heavily based on reuse, potentially causing a cascade of changes that propagate through the entire composition [Whi+09]. The main challenge for an extendable and reusable LM&C approach is to maintain the modularity of the individual concerns, while minimizing the impact of changes on the composition and between the DSLs. The guidelines (GX) below aid in the development of a LM&C approach with explicit support for language evolution. These guidelines are closely followed by the conceptual and technical realization of L3Dim.

LANGUAGE MODULES FOR COMPOSITION (G1) Modularization of DSLs and generators is the core requirement for composition, to maximize reuse, and to enable distributed development [Dho+12; Kar+09]. A library-based approach offers the possibility to flexibly chose the subset of required DSLs and generators for a particular modeling purpose [HR13].

EXCHANGEABILITY OF HETEROGENEOUS MODULES (G2) A non-invasive and loosely coupled composition of heterogeneous model fragments is required to achieve seamless exchangeability. Meaning that unrelated heterogeneous DSLs are syntactically integrated without the need to introduce dependencies between the languages or having to modifying either of them [Voe11]. For instance, replacing platform-specific model fragments (e.g., OROCOS RTT by ROS), has no impact on the platform-independent fragments [Kar+09].

WELL-DEFINED INTERFACES FOR LM&C (G3) A truly extensible composition can only be achieved through well-defined interfaces [HR13] that are used as extension points in the composition, and to restrict the language evolution to predefined points, making its impact on the composition predictable.

LIMITING THE IMPACT OF LANGUAGE EVOLUTION (G4) To limit the impact of changes on other parts of the composition, a semantic and highly cohesive modularization along the (robotics) concerns needs to be realized. Thereby limiting the propagation of changes to only closely concern-related DSL modules. Hence, making the propagation predictable. Reusing different generator modules to form a transformation pipeline instead of using a single monolithic generator, limits the language evolution to the single modules instead of the entire transformation process [Voe11].

3.2 L3DIM'S MODULARIZATION AND COMPOSITION MECHANISMS

In order to realize the introduced horizontal and vertical SoC of L3Dim, while naturally supporting language evolution, the presented LM&C mechanisms (see Section 2.4.2.3) are used. For the technical implementation of the LM&C mechanisms, JetBrains Meta Programming System (MPS) [MPS] is chosen in this thesis. MPS is a tool specifically designed for the efficient design, technical composition, and reuse of DSLs on the abstract and concrete syntax level. In contrast to other language workbenches [Fow05], MPS achieves projectional editing by decoupling of the AST from the concrete syntax. Thereby supporting the seamless mixing and exchanging of textual, tabular, or other graphical representations [Völ+13].

>_ *projectional*

>_ *abstract syntax tree*

3.2.1 *Composing Horizontal Dimensions*

The capability, software, and hardware platform dimensions separate the concerns so that distributed development and SoR is possible. Capabilities represent concerns that are platform-independent, but cannot produce an executable system without the platform-specific concerns. Concerns from either the hardware or software platform dimension are highly replaceable and depend on the requirements of the robotics application or on the availability of certain software frameworks or hardware parts. Capabilities generally limit the set of applicable software and hardware platforms, while the platforms can define requirements on the capabilities. For instance, the DoF of the chosen robot may influence the controller choice, whereas the software framework may restrict the timing behavior model of the system to non-parallel execution.

L3Dim chooses a non-invasive LM&C that preserves the modularity and independence of the DSL modules in the dimensions (G1), that does not compromise their exchangeability (G2), but that enables the modeling of concern-overarching requirements, constraints, and analyzes. The *annotation* mechanism, presented in Section 2.4.2.3, is used as an interface (see Figure 3.2) to compose the software and hardware platform with the capability dimension (G3), creating an extension point for enriching platform-independent models with platform-specific information without introducing new dependencies.

○ *platform-independent concerns are non-invasively composed with platform-specific concerns, while their modularity is preserved*

Each of the dimensions, contains a language module that acts as the common base for all other language modules in the same dimension. The common bases equip derived languages with the necessary mechanisms and interfaces to support the embedding of other modules. This way, a platform-agnostic capability, such as a component instance representing a control component, can

○ *core languages provide composition interfaces using annotations* be enriched with information regarding the execution environment and the kinematics and dynamics parameters of the to be controlled robot.

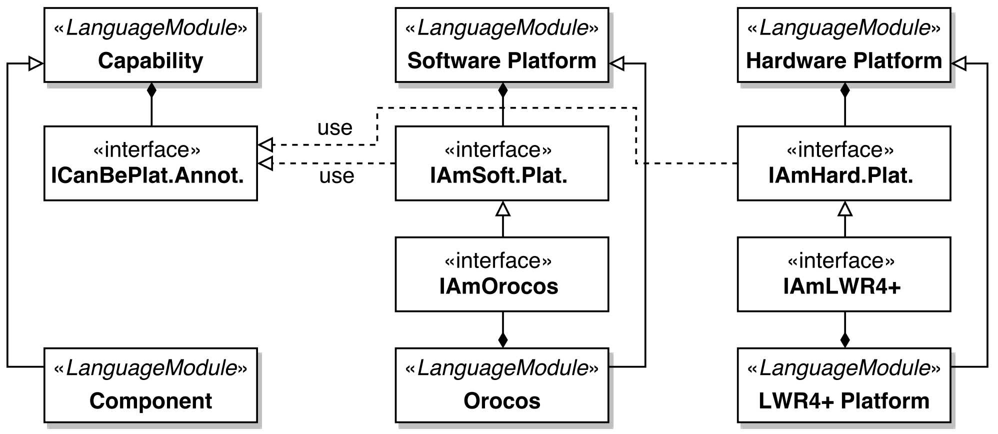

«LanguageModule» Capability
«LanguageModule» Software Platform
«LanguageModule» Hardware Platform
«interface» ICanBePlat.Annot.
use
use
«interface» IAmSoft.Plat.
«interface» IAmHard.Plat.
«interface» IAmOrocos
«interface» IAmLWR4+
«LanguageModule» Component
«LanguageModule» Orocos
«LanguageModule» LWR4+ Platform

Figure 3.2: Visualization of the interfaces used to compose the Capability, Software Platform, and Hardware Platform dimension.

3.2.1.1 *Capability Dimension*

Each DSL in this dimension represents aspects of functional or (non)functional robotics concerns in form of language modules that are derived from the *Capability DSL* as the common base. From this common language, each of the derived DSLs inherits an interface (`ICanBePlatformAnnotated`) that enables the embedding of language modules from the two platform dimensions. This allows platform-specific constraints to be applied onto capabilities, and capabilities to be transformed by generators according to platform-specific needs. Note that no direct dependencies are introduced between capability and platform-specific language modules. For most of the platform-independent concerns, there exist related work in the literature [Nor+16a].

3.2.1.2 *Hardware Platform Dimension*

To represent robotic hardware platforms, the *Hardware Platform DSL* is introduced, which forms the common base in this dimension. It provides the `IAmHardwarePlatform` annotation as an anchor point to force robot-specific constraints onto models from the capability dimension. Each supported hardware platform is represented as a language module that extends the base language and specializes the inherited annotation to match its platform (see `IAmLWR4+` in Figure 3.2). Using this annotation, robot-specific constraints can be imposed on the PIM, which can then be considered on the model level or in the model transformations. This dimension also includes the modeling of sensors (e.g. force-torque sensors), interfaces (e.g. ethernet ports), and computation units (e.g. external workstations).

3.2.1.3 *Software Platform Dimension*

The common base language in this dimension is the *Software Platform DSL*. Analogous to the Hardware Platform dimension, it entails the `IAmSoftwarePlatform` annotation. Each software framework (e.g., middleware, component-based, or control framework) is represented by adapting the inherited annotation from its base language to the respective platform (see `IAmOrocos` in Figure 3.2). Language modules in this dimension can be specially tailored towards extending, restricting, or analyzing capabilities based on software requirements.

In case a software platform requires additional model information (e.g., the period of a component execution) in order to satisfy their software framework requirements, the platform can make use of the demand mechanism. By specializing and implementing the `IDemand` interface, a platform-specific demand can be introduced that mandates to be fulfilled. If there is no DSL in the capability dimension to fulfill the demand, the software platform can provide a model fragment that allows the manual specification of the missing information (G3, G4), until a suitable capability is developed (see Section 4.2).

Note that concerns can also be dimension-overarching. DSL modules for these concerns inherit the interfaces from their primary dimension. They are conceptually and technically realized as adapter languages, which provide DSL aspects, such as meta-models, constraints, and transformations, based on the combination of multiple independent languages, without compromising their independence [Völ+13]. Those languages are also applied to combine multiple concerns within one dimension. Creating such a fine-grained and loosely coupled composition between concerns, results in a DSL dependency graph that is structured along the robotics concerns. Thereby achieving a predictive propagation of changes (G4). For instance, a change in a concern-overarching DSL module does not affect the modules for the individual concerns. Similarly, a change in the e.g., *Component DSL* can only affect other modules related to that specific concern. It cannot affect modules related to different concerns.

>_ *adapter language*

○ *achieves concern- and dimension-overarching composition*

3.2.2 *Linking Vertical Layers*

Model transformations are used to establish a link between models on vertical layers with different modeling purposes. For instance, a CI task model that defines the robot's behavior, imposes constraints on the component-based system model that is used for the generation of the executable software system. Model transformations aid in realizing these constraints, by generating model fragments in the system model, such as suitable control component instances for the CI task.

L3Dim uses multi-staged transformations to iteratively transform a model into a model of the next lower layer until the target software artifact is generated. Inspired by Voelter et al. [Völ+13], multi-staged transformation pipelines

○ *modular transformation pipelines, instead of single monolithic transformations* are formed by combining modular and highly cohesive transformations (G1) related to the different robotics concerns, fostering reuse and exchangeability (G2). This is especially beneficial for the generation of a system for different software platforms (e.g., OROCOS RTT or ROS), since it allows the introduction of an intermediate layer (IL) (gray boxes) for the transformations. As it can be seen in Figure 3.3, the intermediate layer (IL) separates the platform-independent transformations from the platform-specific transformations that produce a software artifact. This way, transformations of platform-independent aspects do not need to consider platform-specific transformations, since they can be reused from the IL languages' generators (backend reuse). Additionally, languages below the IL are implicitly reusing all transformations that happen above the IL. Thus, they only need to provide transformations from the IL languages to their own DSL (frontend reuse).

intermediate layer (IL)

IL enables reuse and exchangeability

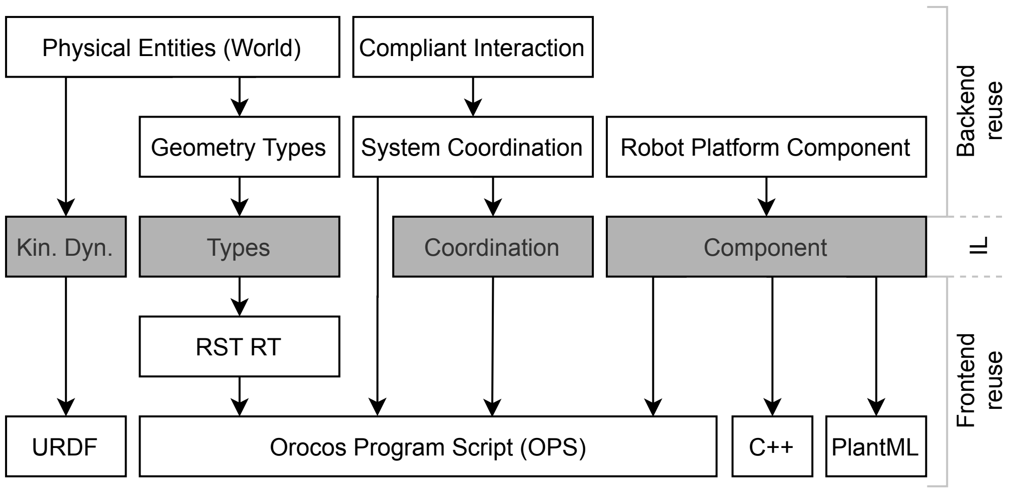


Figure 3.3: Adaption of the multi-staged transformation concepts [Völ+13] to the proposed generator composition: The intermediate layer (IL) that is depicted by the gray boxes, allows for a separation between *backend* and *frontend* language models. Here, backend models are more abstract and technology-independent, while frontend models are very close to the target technologies. Backend models are able to make use of the frontend generators of the IL without being aware of the individual generators. This allows for a flexible exchange of frontend generators to support different generation targets. Here, OROCOS RTT is used as generation target and execution environment. However, it is also possible to ignore the IL and directly generate towards a specific target using a suitable generator. Note, only an excerpt of the existing language modules is displayed in this figure.

well-defined interfaces and platform-specific annotations enable automatic pipeline configuration and limit the propagation of changes

In this approach, multiple transformations are applicable to the same (e.g., system) model, generating e.g., a system for different target platforms. Hence, the high flexibility becomes a disadvantage, requiring the selection of suitable transformations that combined generate the desired target artifact from a specific input model. This problem can be addressed by predefining transformation pipelines, which in turn reduce the flexibility. L3Dim automatically forms the transformation pipelines, using the non-invasive annotations of the platform dimensions and an explicit selection of suitable transformations based on their interface. In L3Dim, each transformation provides a well-defined interface that precisely defines its applicability (G3), based on a valid input model fragment, the to be produced output fragment, and additional constraints on the input and output. Such an interface creates an isolated transformation stage to which language evolution is bound to, preventing its propagation towards other transformations of the multi-staged pipeline (G4).

L3Dim reuses the annotation mechanism for the horizontal dimension composition, to guide the configuration of transformation pipelines:

(PLATFORM) ANNOTATION UNAWARE Transformations that are unaware of annotations can be applied to any suitable model fragment that it is designed for. This rule ensures the default behavior of backend and frontend reuse as described by Voelter et al. [Völ+13].

(PLATFORM) ANNOTATION AWARE Transformations that are aware of annotations are restricted to be applied only to model fragments that the transformation is designed for and that are marked with a specific annotation. The transformation is able to remove, change, or pass on the annotation of the source to the resulting model fragment. Hence, transformation pipelines are initiated by annotated model fragments and iteratively formed, constrained by the previously applied transformations. This case adds the aspect of reconfiguration to transformation reuse, as transformations can use and store additional information in the annotation to change the behavior of annotation-aware transformations.

While the general idea behind annotation-awareness for transformations is applicable to several different aspects, in this work the focus is set on the generation towards different component-based (i.e. OROCOS RTT and ROS) and control (i.e. PIDC and QP-based SoT) frameworks. By selecting the set of transformations that are required to generate OROCOS RTT and ROS systems, e.g., different components of the same system, can be annotated with one of the selected software platforms (see Figure 3.15). Transformations pipelines are then automatically formed to generate the executable code for the respective platform. By deselecting the ROS platform for instance, components cannot be annotated to be realized using ROS anymore. Hence, no code for ROS is generated. This way, multi-staged transformation pipelines are formed naturally, avoiding the need for predefined pipelines and preserving the flexibility that comes with backend and frontend reuse. Further, it is allowed to use different transformations on the same source model fragment to e.g., generate towards different targets. For instance, a model of a component could be generated into its CPP-based OROCOS RTT realization as well as into a documentation of its interface. An exemplary transformation pipeline can be seen in Figure 3.14.

3.3 COMPLIANT SIMULATION AND MODELING ARCHITECTURE (COSIMA)

Compliant Simulation and Modeling Architecture (CoSiMA)

CoSiMA, which is short for "Compliant Simulation and Modeling Architecture", is a concrete instantiation of L3Dim to allow the domain-specific modeling of real-time capable robotic systems for CI tasks and their generation into executable component-based control architectures. The resulting architecture can be simulated as well as executed on the real robotic hardware. In addition to the modeling environment, which is realized using MPS, CoSiMA includes an execution environment as well as a (physics) simulation environment. The execution environment is real-time capable and technically connects the individual software and hardware components. It incorporates well-established technologies, such as OROCOS RTT, ROS, and RSB [WW11]. The simulation environment is used to verify the executed system and to support environmental (e.g., geometric) information during the design of a robotic system. Gazebo [KH04] and Bullet [Cou15] are the currently supported simulators. CoSiMA is published in [Wig+18; WW19; Wig+17a] and has so far been used to control Universal Robot's UR 5, Franka Emica's Panda, Kuka's IIWA, LWR 4+, and Omnirob, as well as IIT's COMAN and COMAN+. Figure 3.4 shows a selection of scenarios realized with CoSiMA by the consortium of CogIMon [Cog19]. Due to the focus on CI, CoSiMA is designed to support the

CoSiMA uses a concrete instantiation of L3Dim to model real-time component-based control system architectures for CI

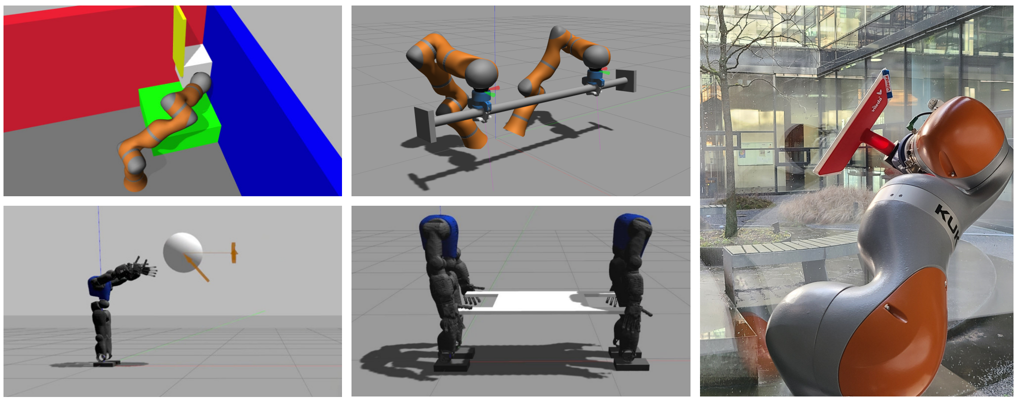


Figure 3.4: Exemplary scenarios realized in CogIMon [Cog19] using CoSiMA. Thanks to the involved project partners. Further scenarios are shown in Chapter 8.

development, simulation, and analysis of real-time-constrained CBRSs, before the deployment on the real robotic platforms (see Chapter 4). For this purpose, CoSiMA supports the fully transparent switching between simulation and the real hardware of robots, ranging from single manipulators to humanoid robots, with minimal effort.

In the following, CoSiMA's modeling environment is described in detail. For further detail on the other environments, please refer to the respective publications. Figure 3.5 gives an overview of the developed DSL modules to model the control architectures of robotic systems. The figure is structured into three parts. The lowest part is concerned with the core languages for the three dimensions of L3Dim and their specializations. The middle part of the figure displays the currently supported domain concerns of a robotic system, associ-

overview of CoSiMA's language modules

ated with a single dimension. The upper part of the figure consists of adapter languages (dashed border) that compose concerns of the same or of different dimensions. This set of DSL modules is used and extended in the following chapters to eventually realize the experimental scenarios in Chapter 8. Note that modules for e.g., domain-specific types, are omitted.

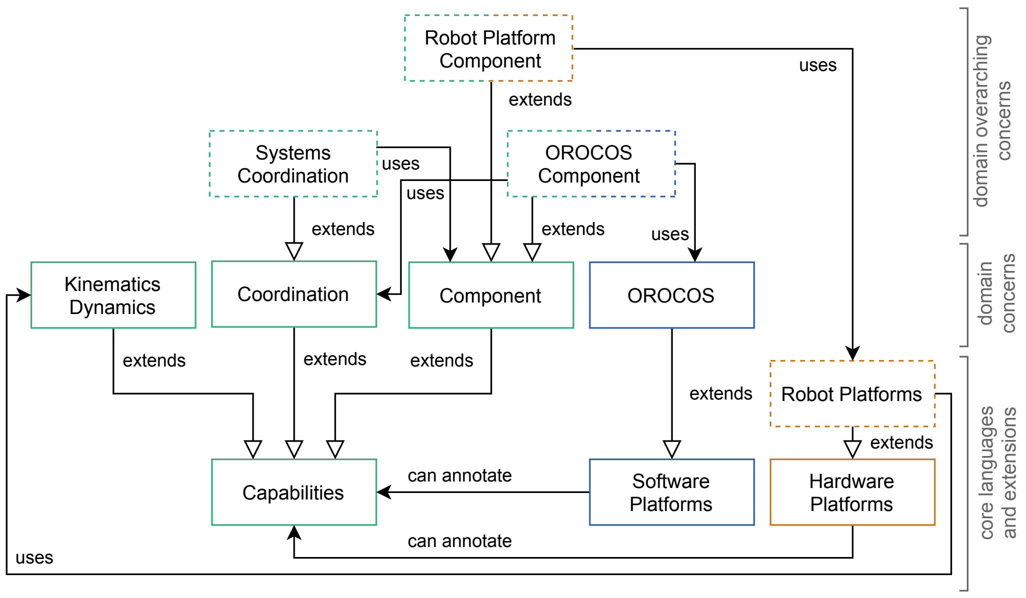


Figure 3.5: Overview of the dependencies of the supported DSL modules to model robotic systems. Modules that only contain domain-specific types are omitted for readability. The DSLs are grouped into three parts: Core languages and their extensions for each dimension, languages covering different domain concerns, and domain overarching languages, which are realized as adapter languages (dashed border). The modules make use of the modularization and composition mechanisms, introduced in Section 2.4.2.3. This set of language modules is used and extended throughout this thesis. Colors indicate dimension affiliation.

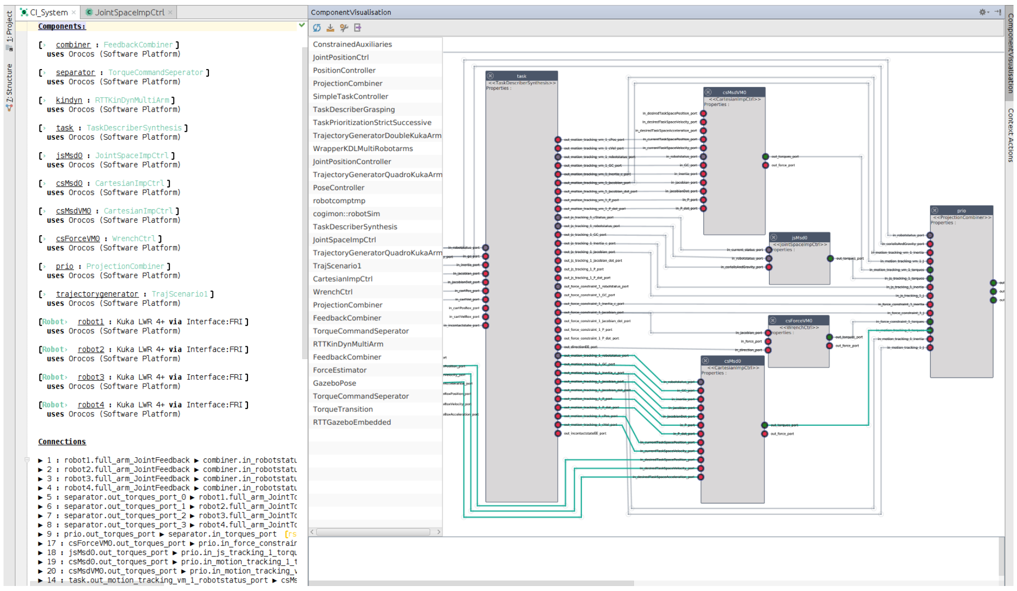
3.3.1 *Platform-Independent Capabilities*

The structural architecture of a CBRS is covered by the *Component DSL* (cf. Section 2.3.1.4). It is developed based on the CPC meta-model, since most robotic systems can conceptually be boiled down to a variant of it [Dho+12; BG16]. The DSL allows the modeling of component interface definitions,[2] their composition into a system, and their interconnection to define the data-flow for communication. In addition to the meta-model, a DSL can introduce constraints on the interaction of the meta-model's concepts (see Figure 2.4). In this case, the *Component DSL* introduces constraints, ensuring that only datatype-wise compatible ports of components can be connected together. The same holds true for the direction of a port. Thus, input ports can be connected to

>_ *Component-Port-Connector*

2 An *Algorithm DSL* is developed in the VeriComp project, extending CoSiMA to explicitly cover the computation concern (cf. Section 2.3.1.1) beyond the component interface.

output ports, while any other combination is forbidden. Figure 3.6 shows an exemplary control architecture modeled using the *Component DSL*.



```
Components:

[›  combiner : FeedbackCombiner ]
  uses Orocos (Software Platform)

[›  separator : TorqueCommandSeperator ]
  uses Orocos (Software Platform)

[›  kindyn : RTTKinDynMultiArm ]
  uses Orocos (Software Platform)

[›  task : TaskDescriberSynthesis ]
  uses Orocos (Software Platform)

[›  jsMsd0 : JointSpaceImpCtrl ]
  uses Orocos (Software Platform)

[›  csMsd0 : CartesianImpCtrl ]
  uses Orocos (Software Platform)

[›  csMsdVM0 : CartesianImpCtrl ]
  uses Orocos (Software Platform)

[›  csForceVM0 : WrenchCtrl ]
  uses Orocos (Software Platform)

[›  prio : ProjectionCombiner ]
  uses Orocos (Software Platform)

[›  trajectorygenerator : TrajScenario1 ]
  uses Orocos (Software Platform)

[Robot›  robot1 : Kuka LWR 4+ via Interface:FRI ]
  uses Orocos (Software Platform)

[Robot›  robot2 : Kuka LWR 4+ via Interface:FRI ]
  uses Orocos (Software Platform)

[Robot›  robot3 : Kuka LWR 4+ via Interface:FRI ]
  uses Orocos (Software Platform)

[Robot›  robot4 : Kuka LWR 4+ via Interface:FRI ]
  uses Orocos (Software Platform)


Connections

▶ 1 : robot1.full_arm_JointFeedback ▶ combiner.in_robotstatu
▶ 2 : robot2.full_arm_JointFeedback ▶ combiner.in_robotstatu
▶ 3 : robot3.full_arm_JointFeedback ▶ combiner.in_robotstatu
▶ 4 : robot4.full_arm_JointFeedback ▶ combiner.in_robotstatu
▶ 5 : separator.out_torques_port_0 ▶ robot1.full_arm_JointTc
▶ 6 : separator.out_torques_port_1 ▶ robot2.full_arm_JointTc
▶ 7 : separator.out_torques_port_2 ▶ robot3.full_arm_JointTc
▶ 8 : separator.out_torques_port_3 ▶ robot4.full_arm_JointTc
▶ 9 : prio.out_torques_port ▶ separator.in_torques_port  [rs
▶ 17 : csForceVM0.out_torques_port ▶ prio.in_force_constrair
▶ 18 : jsMsd0.out_torques_port ▶ prio.in_js_tracking_1_torqu
▶ 19 : csMsd0.out_torques_port ▶ prio.in_motion_tracking_1_t
▶ 20 : csMsdVM0.out_torques_port ▶ prio.in_motion_tracking_v
▶ 14 : task.out_motion_tracking_vm_1_robotstatus_port ▶ csMs
```

```
ConstrainedAuxiliaries
JointPositionCtrl
PositionController
ProjectionCombiner
SimpleTaskController
TaskDescriberGrasping
TaskPrioritizationStrictSuccessive
TrajectoryGeneratorDoubleKukaArm
WrapperKDLMultiRobotarms
JointPositionController
TrajectoryGeneratorQuadroKukaArm
PoseController
robotcomptmp
cogimon::robotSim
TaskDescriberSynthesis
JointSpaceImpCtrl
TrajectoryGeneratorQuadroKukaArm
TrajScenario1
CartesianImpCtrl
WrenchCtrl
ProjectionCombiner
FeedbackCombiner
TorqueCommandSeperator
RTTKinDynMultiArm
FeedbackCombiner
ForceEstimator
GazeboPose
TorqueCommandSeperator
TorqueTransition
RTTGazeboEmbedded
```

Figure 3.6: Concrete textual (left) and graphical (right) syntax of the *Component DSL*.

The coordination concern (cf. Section 2.3.1.3) of a robotic system is addressed by the *Coordination DSL*. It enables the modeling of the behavioral aspects in form of finite state machines (FSMs) on the component- and system-level. While the conceptual foundation of this DSL builds on the SCXML flavor of FSMs [W3C15], it could be realized with any other suitable formalism for coordination, such as behavior trees.

finite state machine (FSM)

The *Systems Coordination DSL* is developed as an adapter language to combine the structural and coordination aspects of the system, by introducing references of model fragments based on the *Component DSL* into models of the *Coordination DSL* (see Figure 3.7). This way, a FSM can access the structural system architecture to change the configuration of a component, turn it on or off, and react to the component's state changes. Without the reference to the *Component DSL*, there is no explicit access to a component. Thus, a FSM could only resort to firing an event, which may or may not be implicitly associated with, and received by, the desired component. The combined meta-model is shown in Figure 3.8.

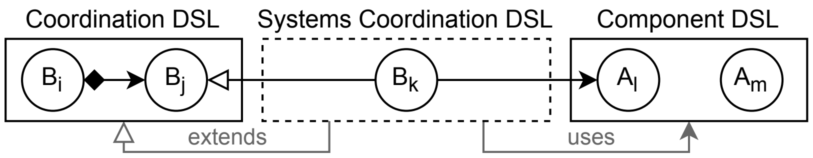


Figure 3.7: Adapter language composition visualized for the *Systems Coordination DSL*. Boxes represent language modules and circles represent concepts. Concepts (i, j, k, l, m) with the same capital letter are based on the same meta-model or an extension of it.

Another platform-independent capability is covered by the *Kinematics Dynamics DSL*. It supports the modeling of kinematic and dynamics entities and constraints (e.g., links and joints), which can be used to define a kinematic chain or tree. It is conceptually based on the Unified Robot Description Format (URDF) [Gar09], which is well-known and widely used. This capability is used by the (hardware) platform-specific *Robot Platform DSL* to model different robots.

Unified Robot Description Format (URDF)

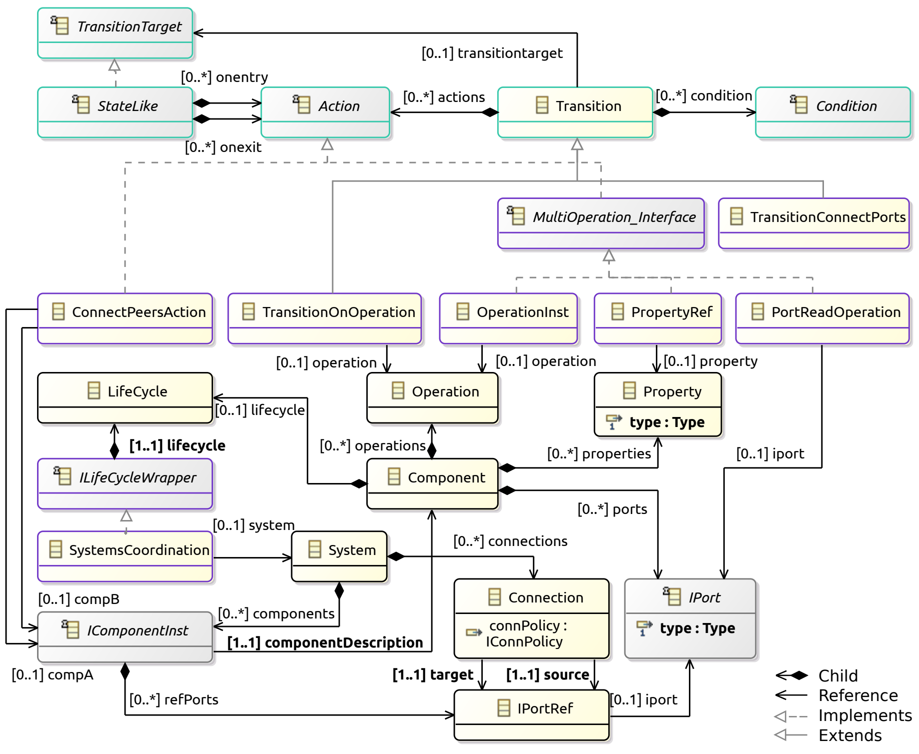
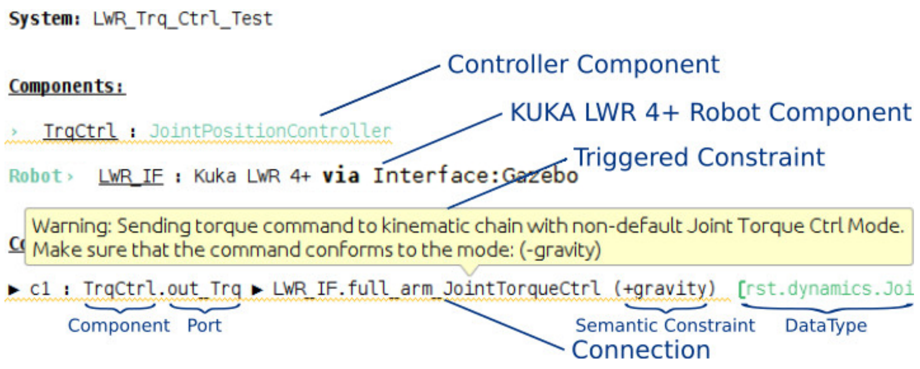


Figure 3.8: Illustration of the meta-model integration between *Coordination DSL* (turquoise), *Component DSL* (black), and *Systems Coordination DSL* (purple). This composition enables the combination of structural and coordination system concerns. An example model can be seen in Figure 3.16.

3.3.2 *Robots as Hardware Platform*

Since a robot is an integral part of a robotic system, it is essential to model the targeted robots to formulate requirements on the desired system architecture and to constrain incompatible aspects.

The *Robot Platform DSL* describes a robot in terms of its physical and computational platform. This also includes the supported control modes of the joints (i.e. position control, velocity, control, and torque control), as well as the interface specification and protocol for communication (e.g., ROBOLLI [Ajo+14] or FRI). The physical appearance of the robot is modeled using the *Kinematics Dynamics DSL*. Currently, there is no need to create a DSL extension of the *Robot Platform DSL* for a specific platform, such as the KUKA LWR4+ or the COMAN. This is because the generic mechanisms of the *Robot Platform DSL*



```
System: LWR_Trq_Ctrl_Test

Components:

› TrqCtrl : JointPositionController

Robot› LWR_IF : Kuka LWR 4+ via Interface:Gazebo

Warning: Sending torque command to kinematic chain with non-default Joint Torque Ctrl Mode.
Make sure that the command conforms to the mode: (-gravity)

▶ c1 : TrqCtrl.out_Trq ▶ LWR_IF.full_arm_JointTorqueCtrl (+gravity) [rst.dynamics.Joi
```

Figure 3.9: Screenshot of a `System` architecture model with a `RobotComponentInst` that allows the access to a robot platform in form of a robot interface, which is connected to a controller `ComponentInst`. In addition to generic data type checks, robot-specific properties can be checked as well, such as the explicit addition or subtraction of the gravity term in the control signal, which is sent to the robot via the interface component. This model is based on the meta-model shown in Figure 3.10.

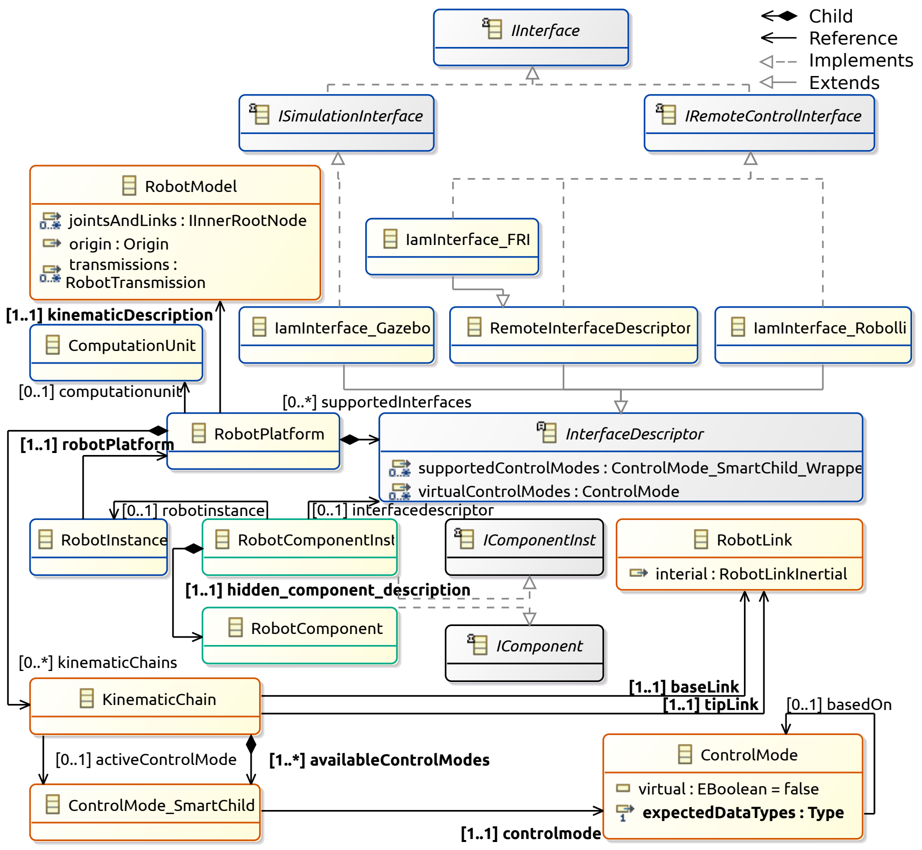


Figure 3.10: Meta-model excerpt combining the *Robot Platform DSL* (blue), the *Kinematics Dynamics DSL* (orange), and the *Component DSL* (black) via the *Robot Platform Component DSL* (green). This allows to model a robot platform, using predefined kinematic structures (`RobotModel`), the supported `KinematicChain`s, relevant `ComputationUnit`, and `InterfaceDescriptor`s. Instances of `RobotPlatform`s can be referenced by `RobotComponentInst`s to be integrated in a `System` of the *Component DSL*.

allow the configuration for a specific robot on the model level. Even if this is sufficient for the scenarios covered in this thesis, the possibility and mechanisms are given to support robot-specific extensions once the need arises.[3]

The *Robot Platform Component DSL* is an adapter language that combines aspects of a robotic platform with the *Component DSL*. This is achieved analogous to the *Systems Coordination DSL* by extending the *Component DSL* and referencing the *Robot Platform DSL*. With this, it is possible to instantiate a robot interface component in a system's component-based architecture model, which references a specific robot platform. Different constraints are then imposed on the system model, which for instance prevents the system to establish a connection between the ports of the interface component and any other component with unsupported data-types, considering the available control modes of the chosen robot (see Figure 3.9). The meta-model is shown in Figure 3.10.

3.3.3 *OROCOS RTT as Software Platform*

OROCOS RTT is chosen as execution environment in CoSiMA for the realization of the robotic scenarios in this thesis. Hence, it needs to be available as software platform on the model level. In the Software Platform dimension, the *OROCOS DSL* is created to contain the `IAmOrocos` annotation, which specializes `IAmSoftwarePlatform` (see Figure 3.11). Once the annotation mechanisms are properly adapted to support the OROCOS RTT platform, further DSLs can be created that enrich capabilities with OROCOS RTT-specific constraints and provide suitable model transformations.

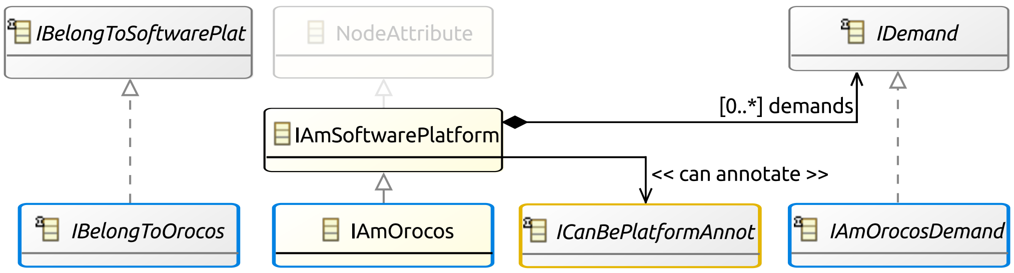


Figure 3.11: Illustration of the concepts supporting OROCOS as a software platform in the composition. While the concepts of the *OROCOS DSL* (blue border) specialize the necessary concepts of the *Software Platform DSL* (gray border), the interface `ICanBePlatformAnnotated` from the Capability DSL (orange border) represents the annotation target for software platforms.

The main purpose of specializing the *Component DSL* is to cover additional structural and behavioral aspects that are key concepts of OROCOS RTT: For instance, so-called activities[4] need to be defined to manage the order and scheduling of OROCOS RTT (C++) components. Since the *Component DSL* does not have any concept that is even remotely related to such an activity (see the black bordered concepts in Figure 3.12), it needs to be included as a

3 For instance to add support for soft robots in the future.
4 For more information see Chapter 4 and `http://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/orocos-components-manual.html#corelib-activities`

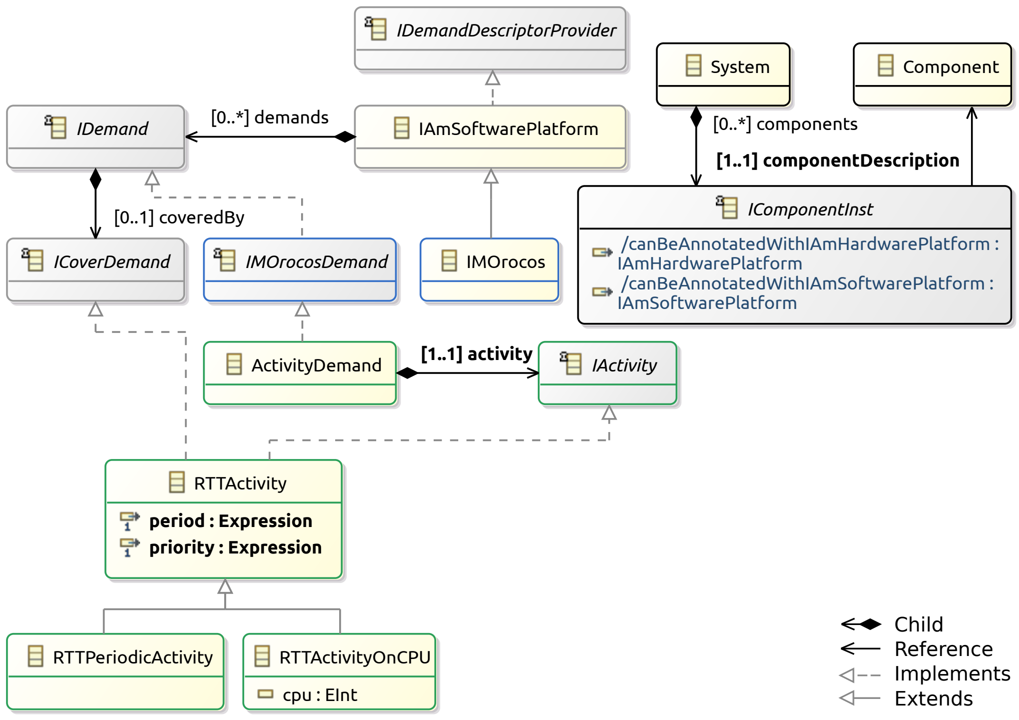


Figure 3.12: Excerpt of the composed meta-model that allows the modeling of required information in form of i.e. an `RTTActivity`, which is realized as a demand of the OROCOS software platform. The *demand* mechanism of the *Software Platform DSL* (gray) is specialized for OROCOS in the *OROCOS DSL* (blue). The *OROCOS Component DSL* (green) uses the `RTTActivity` and `ActivityDemand` to provide a demand for the component modeling capability. Finally, every `IComponentInst` from the *Component DSL* (black) that is annotated with the `IMOrocos` annotation, is responsible to satisfy the associated `ActivityDemand`.

new concept, i.e. `RTTActivity`. Hence, aspects such as activities are treated as mandatory for the specific software platform and, thus are reflected by a demand (i.e. `IAmOrocosDemand`). Each demand utilizes the demand mechanism from the *Software Platform DSL*. In contrast to optional aspects of a software platform, demands are used to express the need for information which is crucial to the creation of a valid model. Here, each `ComponentInst` that is annotated with the OROCOS platform, demands additional information regarding its execution time semantics. The information can be provided using an `RTTActivity` or by platform-agnostic abstractions in form of other domain models (see the *Timing DSL* in Chapter 4). In addition to the structural aspects, the *OROCOS Component DSL* adds an OROCOS RTT specific life cycle to components and their instances by involving concepts from the *Coordination DSL* (see Figure 3.8). This way, a life cycle is represented as a set of states and transitions. Further constraints ensure (1) that all the required states of the life cycle are exposed as operations by the components and (2) that the call order and preconditions of the operations are valid. These operations can then be used by models of the *Systems Coordination DSL* for the orchestration of the system.

3.3.4 *Modular Generator Composition for OROCOS RTT*

With the introduced DSLs one is able to model a component-based robotics control system, constrained by the chosen robotic platform and supplied with the necessary information to execute the system using OROCOS RTT as software platform. To transform the model fragments (gray) into artifacts (blue) for execution, verification, or visualization, different generator modules are developed (see Figure 3.13). These generators contain model-to-model (M2M) and model-to-text (M2T) transformations that are used to arrange generator pipelines for the different system concerns.

○ *overview of CoSiMA's generator modules for OROCOS RTT*

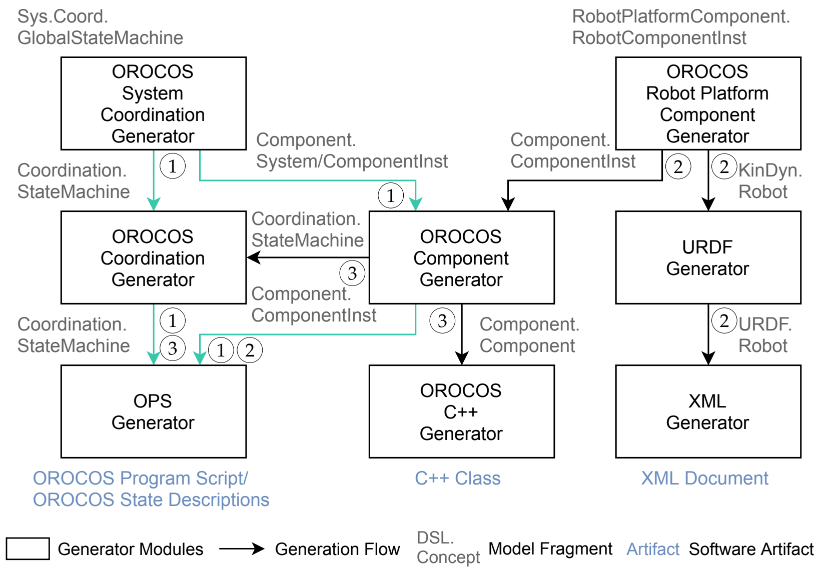


Figure 3.13: Overview of CoSiMA's generator modules for OROCOS RTT. The arrows indicate the pipelines that are automatically formed to produce the desired software artifacts. Arrows with the same number belong to the same pipeline. The pipeline marked in teal is explained in detail below.

The automatically formed generation pipeline, marked in teal in Figure 3.13, is explained in the following. This pipeline generates the structural architecture, i.e. component instances and their connections, as well as a system-level state machine for coordination into an OROCOS Program Script (OPS), which can be executed in an OROCOS RTT environment. Figure 3.14 shows the model fragments that are transformed via multiple transformation stages. The pipeline starts with the *OROCOS System Coordination Generator*, which takes an annotated `GlobalStateMachine` from the *Systems Coordination DSL* as input. Note, other model fragments contained in, or referenced by, the `GlobalStateMachine` are depicted per row via fragment placeholders and tail-less arrows respectively. Additionally, the generator only becomes applicable, if the `GlobalStateMachine` references a `System` that contains at least one `ComponentInst` that is annotated with OROCOS RTT as software platform. Thereby ensuring that at least one OROCOS RTT component is instantiated, while the generation of the `StateMachine` and `Connections` is optional. In the first generation stage (1st row), the *OROCOS System Coordination Generator* uses a transfor-

>_ *OROCOS Program Script (OPS)*

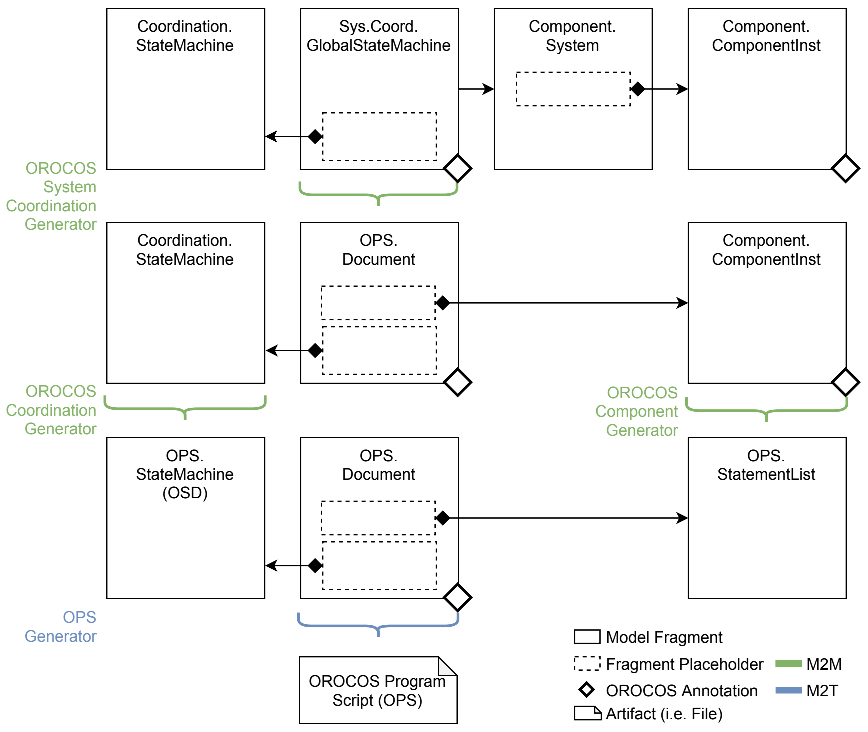


Figure 3.14: A generation pipeline that transforms the static system model and an associated coordination model into an OPS file that can be executed with OROCOS RTT. Downward pointing brackets represent transformation steps, performed by the indicated generator.

the OROCOS State Description (OSD) is a subset of the OPS language to describe FSMs

mation that targets annotated `GlobalStateMachine` model fragments and turns them into a `Document` model of the *OPS DSL*, while passing on the annotation. In the same stage, the `ComponentInst` and `Connection` model fragments are copied into the `Document` model for future transformations, while the `System` is removed. Of course only annotated `ComponentInst`s and `Connection`s that connect OROCOS-annotated `ComponentInst`s are copied. After that first transformation stage, the *OROCOS Coordination Generator* and the *OROCOS Component Generator* are automatically applied to the remaining untransformed fragments (2nd row). While the first generator deems state machine models to be valid inputs if they itself or their parent are annotated, the latter only applies itself to directly annotated component instances. Here, both generators transform model fragments into fragments of the *OPS DSL*. Eventually, the M2M transformation process is completed and the resulting *OPS DSL* model is converted into a text-based OPS file, by reusing M2T transformations from the *OPS Generator* (see Figure 3.3). The OPS file can then be passed into the OROCOS RTT deployer[5] binary for execution.

One note regarding `Connection`s: They are not discussed here since their generation is straight-forward. However, in case a system deploys components with an additional software framework, such as ROS, an explicit generator module needs to provide the correct transformations. Regarding OROCOS and

5 `http://www.orocos.org/stable/documentation/ocl/v2.x/doc-xml/orocos-deployment.html`

ROS, the connections are realized as default topics on the ROS-side, while the OROCOS-side is additionally configured to use a plugin that allows for real-time safe communication between real-time and non-real-time environments.

Two other pipelines are explained in Appendices A.1 and A.2.

3.4 EVALUATION

This section discusses the L3Dim approach and its concrete instantiation in CoSiMA along qualitative and quantitative aspects. The qualitative aspects are analyzed in how far the presented approach addresses the challenges, introduced in Section 3.1.3. Afterwards, a quantitative evaluation of the amount of reuse in the approach is conducted. This is followed by an application of common object-oriented software metrics to analyze the modularization and composition of DSL modules within CoSiMA.

3.4.1 *Qualitative Evaluation of L3Dim and CoSiMA*

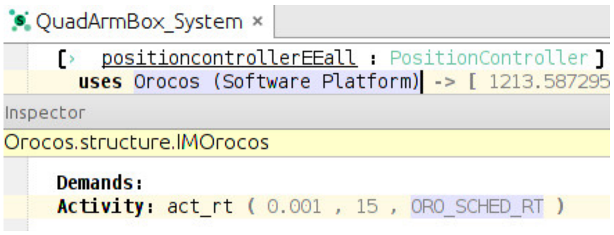
This evaluation is based on the guidelines (GX) to face the challenges of language modularization and composition, as introduced in Section 3.1.3. These guidelines are condensed from the suggestions and lessons learned of the DSL community. Each guideline is discussed separately in how far it is followed by CoSiMA and the general L3Dim approach.

3.4.1.1 *Language Modules for Composition (G1)*

This guideline treats modularization and composition as two equally important sides of a coin. Especially in the context of the robotics domain, modularization does not make considerable sense without composition—and vice versa. The L3Dim approach supports vertical and horizontal modularization to support e.g., distributed development by following the paradigms of SoC and SoR. Flexible composition is enabled through non-invasive composition mechanisms (i.e. annotations and adapter languages) and implicitly formed generator pipelines. L3Dim is designed to import only the required set of language and generator modules for a specific application from a module library. This kind of flexibility allows to decide the exact degree of coverage that is needed for the desired model (i.e. capabilities) and to only generate the artifacts that are needed for the execution with a specific platform (e.g., OROCOS RTT). Using such a library-based modularization is also encouraged by Horst et al. [HR13]. In L3Dim, modules can be extended or reused e.g., to model concern-overarching aspects (see Figures 3.9 and 3.16) or to use the IL to reuse existing generator modules (see Figure 3.13).

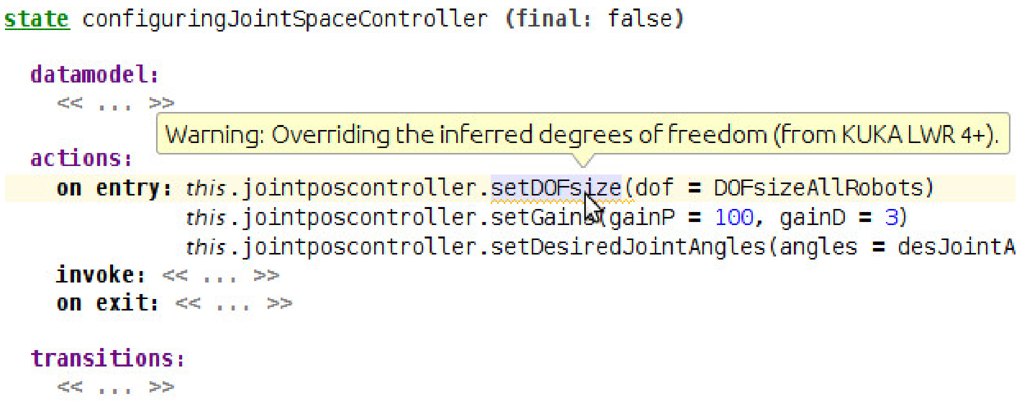
3.4.1.2 *Exchangeability of Heterogeneous Modules (G2)*

Exchangeability is a direct result of the support for SoC and the separation into the three dimensions. Intra- and inter-dimension composition makes use of suitable composition mechanisms (see Section 2.4.2.3), such as adapter languages and annotations respectively (see Figure 3.15), to compose heterogeneous modules without introducing additional dependencies. *Embedding* in form of annotations allows the separation of platform-independent and -specific models, while preserving horizontal composability. This is necessary, because capabilities, software, and hardware platforms are individual aspects of the same model and thus should reside on the same abstraction level (see Section 3.1.2). The fact that moving from a PIM to a PSM does not require model transformations, which are often one-directional or lossy, greatly improves the exchangeability. For instance in L3Dim, replacing the used software platform by another one, only requires to change the affected annotations.


```
QuadArmBox_System ×
  [ positioncontrollerEEall : PositionController ]
    uses Orocos (Software Platform) -> [ 1213.587295
Inspector
Orocos.structure.IMOrocos

  Demands:
  Activity: act_rt ( 0.001 , 15 , ORO_SCHED_RT )
```

Figure 3.15: Screenshot of an activity *demand* to provide the required timing information of a `ComponentInst` for the execution with OROCOS RTT. The model is based on the meta-model seen in Figure 3.12.


```
state configuringJointSpaceController (final: false)

  datamodel:
    << ... >>
                 Warning: Overriding the inferred degrees of freedom (from KUKA LWR 4+).
  actions:
    on entry: this.jointposcontroller.setDOFsize(dof = DOFsizeAllRobots)
              this.jointposcontroller.setGains(gainP = 100, gainD = 3)
              this.jointposcontroller.setDesiredJointAngles(angles = desJointA
    invoke: << ... >>
    on exit: << ... >>

  transitions:
    << ... >>
```

Figure 3.16: Screenshot of a `State` from the *Coordination DSL* that defines calls to `Operations` of a `ComponentInst` from the *Component DSL*. These calls are enabled by a composition in form of the *Systems Coordination DSL* (see Figure 3.8). The user warning is realized through the additional use of constraints defined by the *Robot Platform Component DSL*.

However, in the classical case of vertically separated PIMs and PSMs, the old PSM is discarded and a copy of the original PIM is required to be transformed into a PSM for another software platform, loosing all the modeled elements of the previous PSM. In the worst case, if the original PIM is not available anymore, the PSM needs to be transformed back to the PIM and then generated into the new PSM again. This however, is rarely feasible, because all the transformations need to be designed to be reversible.

Overall, the key to exchangeability are sensitive dependency management and the incorporation of extension points through concrete interfaces and composition mechanisms.

3.4.1.3 *Well-Defined Interfaces for LM&C (G3)*

In L3Dim there are two kinds of interfaces that allow exchangeability, reuse, and guiding the language evolution. The first kind is defined by the specific modularization and composition scheme of the three dimensions (see Section 3.2.1). In this case, *embedding* is used to ensure a non-invasive composition (see Section 2.4.2.3). This way adding a new software platform, following the existing composition mechanisms, is straight-forward and automatically upholds the overall level of exchangeability. By defining clear extension points, as done through the three core dimension languages, the language evolution is restricted to a predefined path. Thus, the propagation of possible changes through the stack of DSL modules are known or can at least be estimated in preface. Such a restriction avoids an undesired evolution, since the evolution can therefore only happen at the predefined extension points (see Section 3.4.1.4).

The second kind of interface is defined by using an IL for language extension and generation. This layer acts as an interface between models to reuse existing generators (see Figure 3.3). The IL defines a particular structure in which generators are designed to fit in. Reusing transformations greatly minimizes duplicated code, maximizes reuse, and increases maintainability.

In L3Dim, language and generator evolution is restricted by the dedicated interfaces and mechanisms, embracing evolution and forcing it to happen in the expected bounds (see Section 3.3.4).

3.4.1.4 *Limiting the Impact of Language Evolution (G4)*

Supporting evolution requires the ability to change elements of a DSL without triggering a cascade of changes, which has an impact on diverse other language modules. At the generator-level, this is achieved by L3Dim through the composition of flexible generator pipelines and the delegation of responsibilities to the IL. For instance, changes in the *Robot Platform Component DSL* would only have an impact on the *OROCOS Robot Platform Component Generator* but not on the complete generation process. This is due to the fact that the generator transforms a model based on a DSL, which is part of the IL (i.e. *Component DSL*), delegating further transformation responsibilities to the IL.

In general, the degree of evolution that needs to be actively considered during the DSL design, depends on its stability [Mar95], which in essence is based on the dependencies between the DSL modules. The stability of the modules in CoSiMA is quantitatively analyzed in Section 3.4.2.2. Modules with a high number of afferent couplings (i.e. dependent modules), pose a higher risk for a change to affect several other modules. Therefore, those modules need to consider and guide language evolution as described previously. An example in CoSiMA is the software platforms that use the introduced *demand* mecha-

>_ *afferent coupling*

nism (see Section 3.2.1.3) to provide a concrete extension point for aspects that cannot be modeled using a dedicated DSL at the moment (see Figure 3.15). This allows for an incremental evolution, supporting the introduction of new DSLs into CoSiMA to cover previously unmodeled[6] but required concerns.

Several afferent couplings urge the language designer to prevent unforeseen changes as much as possible. This is why, if possible, this kind of language modules should be based on mature formalisms that are not likely to experience (impactful) changes, except at the predefined extension points. In CoSiMA for instance, the *Component DSL* is thus based on the well-established CPC scheme, the *Coordination DSL* is based on FSM, and the *Kinematics Dynamics DSL* relies on the widely-used URDF scheme.

DSL modules that do not carry a responsibility for other modules (i.e. low number of afferent couplings), have more flexibility regarding their evolution. This is due to the fact, that the consequences of a change are not as grave, since they do not influence numerous other modules. Modules in that position can be used to cover currently not well-understood or not completely covered domains, which are thus prone to frequent or heavy changes. Through this kind of flexibility, incremental development is enabled.

3.4.2 *Quantitative Evaluation of CoSiMA*

This part of the evaluation aims at quantifying the reuse and stability of CoSiMA's language composition. First, the actually reused elements are analyzed and compared with the additional effort that would occur by not reusing parts of existing DSLs. Second, an analysis is conducted that applies an object-oriented design quality metric to evaluate the stability of the DSLs in CoSiMA.

3.4.2.1 *Reuse and Effort Analysis*

To gain insights on how the reusability, achieved through the modularization and composition of CoSiMA, has a positive impact on the required development effort, four platform-specific and capability-overarching DSLs are analyzed in Table 3.1. As it can be seen, extending upon the *Component DSL* to support a specific software platform (i.e. NAOqi[7] and OROCOS) requires the development of significantly fewer concepts, compared to not reusing existing DSL modules. While the *OROCOS Component DSL* saves 54.5 % extra effort through reuse, the *Robot Platform Component DSL* saves a total of 416.7 %. Hence, without reuse, 4.2 times more concepts would need to be implemented to allow the modeling of the robot hardware in combination with the component-based structure of a CBRS.

Through the introduction of dimensions by L3Dim, a semantic modularization along the (sub-)domains is incentivized. This allows the elements of the languages to be reduced to ones that are essential to the language's purpose. Necessary elements from other (sub-)domains are reused from the respective

6 These aspects may indeed be modeled, but not with a dedicated DSL.
7 For details on the NAOqi extension, please refer to [Wig+17b].

languages. This way, a high cohesion is achieved, and the maintainability is increased due to the limited scope of the languages and the reduced set of involved elements. By increasing the degree of reuse, the development effort of supporting e.g., another software framework is significantly reduced (see Table 3.1).

1)	**NAOqi Comp.**			**OROCOS Comp.**			**Robot P. Comp.**		**Sys. Coord.**	
2)	Comp.	NAOqi	Soft. P.	Comp.	OROCOS	Soft. P.	Comp.	Robot P.	Comp.	Coord.
3)	16.0 %	100.0 %	40.0 %	8.0 %	33.3 %	20.0 %	32.0 %	47.4 %	28.0 %	20.9 %
4)	19.6 %			10.3 %			33.8 %		24.7 %	
5)	+23.9 %			+54.5 %			+416.7 %		+76.7 %	
6)	0.14 : 1			0.42 : 1			2.89 : 1		0.55 : 1	

1) Adapter language module. 2) Reused language modules.
3) Percentage of reused concepts from the set of available concepts per language module.
4) Percentage of reused concepts from the total set of available concepts.
5) Additional implementation effort of concepts in percent in case of no reuse.
6) Ratio of reused to created concepts in an adapter language module.

Table 3.1: Language reuse and development effort of adapter languages: e.g., the *Robot Platform Component DSL* reuses 32 % of all the concepts in the *Component DSL*, and 33.8 % of the concepts from all the reused DSLs combined. With a ratio of 2.89 : 1, significantly more concepts are reused than newly created. Without reuse, 416.7 % ≊ 4.2 times more concepts would be required to be implemented. Thereby, significantly increasing the development effort.

3.4.2.2 *Stability Analysis*

The semantic modularization of DSLs into dimensions along (sub-)domains and concerns, leads to several benefits, such as cohesion, maintainability, and a reduced development effort. However, it also leads to a fine-grained modularization that results in an increased number of DSLs. Together with the high degree of reuse in CoSiMA, couplings in form of dependencies are naturally introduced between the DSLs. Even though the dependencies are reduced to a minimum (e.g., non-invasive composition), their number cannot be reduced to zero. Language evolution potentially causes cascades of changes to propagate through the entire composition. Depending on the strength of the coupling between the DSLs, language evolution needs to be considered and handled to different degrees. The stability metric, introduced by Martin [Mar95], offers a way to determine the required degree for each DSL and to act as a guide towards a stable basis for composition. Martin proposes two characteristics to classify a (DSL) module: *Independence* and *Responsibility*. Independent modules do not have dependencies. Responsible modules are heavily depended upon by other modules. Modules that are both independent and responsible, do not need to and should not change. To calculate the stability of a module, afferent couplings (Ca), i.e. dependencies from other modules, and efferent couplings (Ce), i.e. dependencies on other modules, are put in relation as the instability index

>_ *efferent coupling*

$$I = \frac{Ce}{Ca + Ce}\,, \tag{3.1}$$

which ranges from 0 (stable) to 1 (instable). An index of 0 does not necessarily mean that the module is stable, but that it needs to be stable, which means that it needs to especially consider language evolution. In contrast to that, an index of 1 does not mean that a module is instable, but that changes do not have a significant impact on other modules. Applying this metric to the DSLs of CoSiMA leads to the findings displayed in Table 3.2: The core DSLs of the three dimensions are the basis of the language composition and the root of the dependency graph. Thus, they need to be stable, even though the *Hardware Platform DSL* is not yet depended on as much, which explains the rather high index. Languages that cover the core concerns of CBRS modeling, e.g., *Component DSL*, *Coordination DSL*, *Timing DSL*, need to be stable as well. This is achieved by being grounded in well-known and accepted formalisms (see Section 3.4.1.4) that are very unlikely to change. Semi-stable modules, such as the *OROCOS DSL* and the *Robot Platform DSL*, are prone to changes and potentially have an impact on other modules. While the *OROCOS DSL* reduces this potential by mainly specializing inherited interfaces and mechanisms, the *Robot Platform DSL* remains a potential source of fluctuation for the *Robot Platform Component DSL*. However, since this language has a high instability index, it is very flexible in adapting to changes. For this reason, it is desirable to not only use stable modules. If all parts of a system would be entirely stable, there would be no room left for changes [Mar95], thus preventing evolution. Hence, all languages that are still work in progress or build on a non-established conceptual base are chosen to be instable (high index). This way, changes are prevented from rippling through the DSLs, while being able to adapt and extend the current state of composition.

		Ca	Ce	I
Chapter 3	**Capability**	11	0	0.000
	Software Platform	12	5	0.294
	Hardware Platform	1	2	0.667
	Component	111	1	0.009
	Coordination	25	1	0.038
	Systems Coordination	2	53	0.964
	Kinematics Dynamics	148	1	0.007
	OROCOS	1	4	0.800
	OROCOS Component	1	6	0.857
	Robot Platforms	14	8	0.364
	Robot Platform Component	0	72	1.000
Chapter 4	**Timing**	11	5	0.313
	Timing Component	0	22	1.000
	Timing OROCOS Component	0	4	1.000
Chapter 6	**Physical Entities (World)**	6	5	0.455
	Compliant Interaction	2	84	0.977
	Control Frameworks	0	3	1.000

Table 3.2: Stability [Mar95] analysis of CoSiMA's DSL modules. Based on afferent couplings (Ca) and efferent couplings (Ce), the instability of a module is determined. The metric ranges from 0 (stable) to 1 (instable).

The visualized dependencies in Figures 3.5 and 3.17 show that the stability of the composition allows an isolation of DSL (sub-)graphs per dimension or concern without impacting DSLs that are closer to the root of the dependency graph. For instance, all DSLs related to OROCOS can flexibly be removed from the composition and exchanged by DSLs for another framework. The same goes for concerns, such as timing. The advantage is a flexible composition that can be optimized for the specific modeling use case.

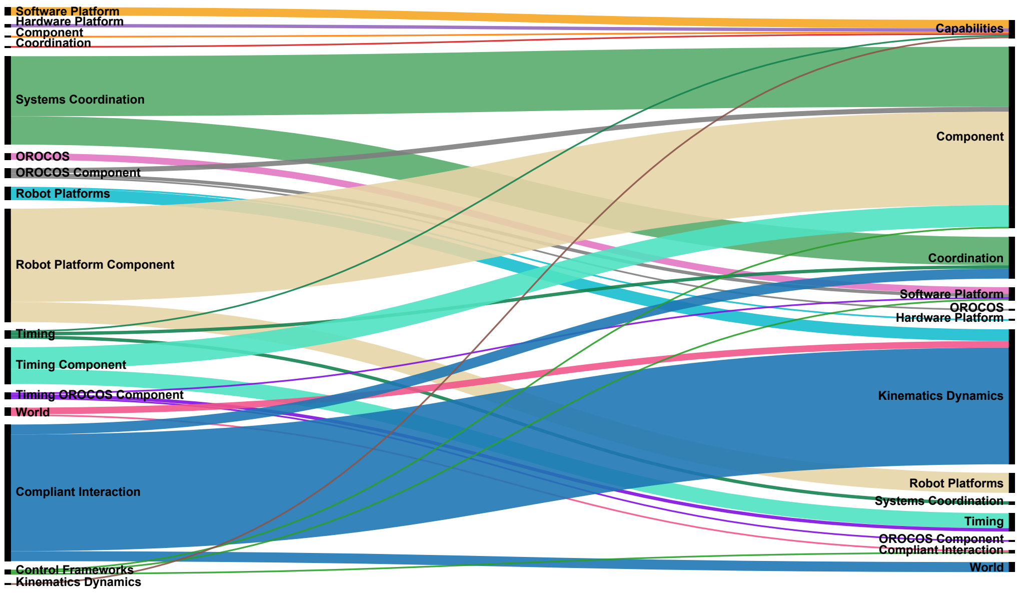


Figure 3.17: Visualization of the DSL dependencies in CoSiMA. A DSL on the left depends on DSLs on the right.

3.5 CONCLUSION

In this chapter, the application of MDE for robotics is motivated due to the complexity and heterogeneity of the robotics domain. While the application of MDE for comparably complex domains such as avionics and automotive is considered best practice, it is unfortunately not yet the default approach in robotics. Most of the approaches that do employ MDE, focus on particular sets of subdomains and do not propose a general methodology to address (component-based) robotics system modeling in an extensible, reusable, and modular way.

Hence, this chapter introduces the L3Dim approach, which aims at particularly addressing these aspects. It incorporates best practices of software design, such as SoC, SoR, and vertical as well as horizontal composition of DSLs, tailored towards robotics. This chapter further proposes guidelines that are based on related work and suggestions from the community to addresses the challenges of language modularization and composition. Along these guidelines, the proposed L3Dim approach introduces a three-dimensional model-based composition approach for developing component-based robot control systems, which is specifically designed to conceptually cover heterogeneous concerns of the robotics domain, e.g., component-based systems, coordination, and motion control. Apart from modeling platform-independent capabilities, L3Dim specifically considers software and hardware platform modeling as separate dimensions. Together with a truly modular and flexible multi-staged code generator composition to support multiple heterogeneous generation targets, these three dimensions form the pillars of L3Dim. The MPS-based modeling part of CoSiMA is presented, which is based on the L3Dim approach and features a set of language and generator modules, enabling the modeling of coordinated component-based robotic systems, and the generation towards OROCOS RTT as execution environment. CoSiMA is evaluated with special focus on reuse and language evolution, including extensibility. The qualitative and quantitative evaluations show that the correct use of L3Dim greatly improves modularity and reuse, while at the same time supporting the successful interplay of concerns from the different system aspects, i.e. hardware, software, and capabilities.

The presented set of modules build the foundation for the extensions in Chapters 4, 6 and 7. Further extensions are developed in cooperation with colleagues from other institutes, such as in the RobMoSys [Rob16] ITPs "VeriComp" and "CMCI". While the extensions made in VeriComp add the modeling and verification of the internal algorithmic behavior of a component as an additional capability, CMCI [CMC20] provides an abstraction to model QP-specific solver aspects, targeting the generation towards QP software frameworks, i.e. OpenSoT [Roc+15], instead of PIDC. The mentioned extensions are not in the scope of this thesis and are therefore not discussed here.

As of writing this thesis, CoSiMA has already managed to be used in several publications and is recognized by related works at the border between robotics and software engineering [Iun+20; Bar19; SH20; de +21].

"*The most reliable way to forecast the future is to try to understand the present.*"

— John Naisbitt

4 TIMING MODELING IN COSIMA

This chapter covers the modeling of non-functional timing properties as additional capability, which is added as an extension to CoSiMA. The modeled properties are used as constraints to synthesize a schedule that is executed with the proposed execution semantics, realized in OROCOS RTT. This chapter is based on [Wig+18; WW19; Moh+18].

With the current rise of collaborative robots entering various fields of the industry, considering and ensuring safety as well as reliability becomes more and more important [Lot+16]. When deploying a robotic system in a sensitive environment, especially if it includes collaborative interactions with humans, a faulty behavior could lead to a hazard, endangering the robot, the environment, or even worse a human being. Thus, it is mandatory to provide guarantees on the system's behavior [Gob+16; Cho+13]. While this is already necessary for non-real-time applications, it is even more mandatory for real-time CBRSs [Car12]. In order to be able to give guarantees for such a system that acts dynamically in the physical world, a recent trend is to apply model-driven engineering techniques to specify and analyze the system and its behavior [RMT14]. However, investigating non-functional properties such as the real-time execution behavior has not yet been a major concern in robotics, since current research tends to address mainly functional capabilities [Nor+16a]. In contrast to that, other domains such as avionics [Mor15], automotive [Per+12], and embedded systems in general [Ouh+11; AMS07], are aware that the modeling of the execution time behavior (e.g., reaction times) is a crucial aspect in the development process [Arn00]. With that in mind, this chapter introduces an extension to CoSiMA that integrates the modeling and analysis of the execution time behavior for robotic systems along the L3Dim approach. The extension allows the identification and correction of potential timing or performance problems early on, thereby leading to a significant reduction of the development costs and person-hours for maintenance [Kra10; Boh+09; Car12].

timing properties and execution time behavior are underrepresented concerns in robotics system modeling

RELATED WORK ON TIMING MODELING FOR CBRS There are currently very few publications in the field of robotics that consider timing specification and analysis an integral part of the development process, and even less that actually model timing aspects for analysis and code generation of robotic systems. These findings are supported by the survey of DSLs in robotics [Nor+16a] as well as by the Robotics DSL Zoo [Nor+16b], suggesting that the modeling of execution time aspects are an underrepresented concern, compared to other concerns in robotics that are already well covered. In domain-specific approaches such as RobotML [Dho+12] or BCM [Bru+13], timing-related aspects are often entirely hidden in the components or framework, and sometimes even in the hardware (interfaces) [NBB16]. Nevertheless, there are approaches in the literature that do explicitly formalize non-functional timing requirements (e.g., [COU16]). The majority of these approaches, however, only covers a single aspect e.g., the worst-case execution time (WCET) [RMT14], the worst-case response time (WCRT) [Gob+16; Lot+16; Cho+13; Kra10; Boh+09; Car12], or the clock- and precedence-semantics [Mal08; AMS07]. Each aspect alone, is not enough to sufficiently cover the timing behavior of a real-time CBRS. Fortunately, there are a few publications that consider an adequate amount of aspects [Lot18] and use them for extensive model-based analyzes and to facilitate the integration of existing analyzers and visualizations into the workflow via model-transformation [Lot+16]. While those approaches use the model information for analyses and visualization, they do not use the model to generate code that realizes an execution conform to the modeled timing constraints. To the best of my knowledge, there is currently no approach that covers the timing aspects essential for (real-time) robotic systems, and combines them with model-based design-time support, analyzes, and executable code generation.

4.1 DOMAIN ANALYSIS

To ensure a desired system behavior, timing constraints need to be enforced on the execution of the individual components. In the literature, this is commonly approached by applying the existing work on real-time systems (RTSs) to component-based systems. A RTS is usually abstracted by a set of schedulable activities[1] $\mathcal{A}$. Multiple instances[2] ($a_{i,j}$) of the activities ($a_i \in \mathcal{A}$) are grouped into a schedule $\mathcal{S}$.

4.1.1 *Timing Characteristics for Component-Based Robotic Systems*

Each activity is defined by a set of timing-related characteristics (see Figure 4.1): The first is the *release date* ($r_{i,j}$), marking the earliest time the j^{th} instance of an activity (a_i) can be executed. The actual starting time is defined by $s_{i,j} \in [r_{i,j}, d_{i,j} - \min(dur_{i,j})]$, where $dur_{i,j}$ denotes the execution time and $d_{i,j}$ marks the *deadline* of the j^{th} instance. Consequently, the *completion date* is

1 Sometimes called *tasks* in this context.
2 Sometimes called *jobs* or *executions*.

represented by $c_{i,j} \in [r_{i,j} + \min(dur_{i,j}), d_{i,j}]$. The time from the release to the completion date defines the *response time* ($R_{i,j}$), which needs to be less or equal to the *relative deadline* (D_i) of the activity (a_i). In general, the WCRT is defined as $WCRT_i = \max_{\forall j}(R_{i,j})$. Under sequential circumstances, the WCRT can be treated as the WCET [BTv08]. The WCRT also includes the system-level overheads and preemption times [Die+17]. In this thesis however, non-preemptive scheduling is assumed. There exist numerous approaches to determine the WCET in the literature. However, doing so is not a trivial task, especially in the preemptive multi-processor case [CBG01]. Still, the most common approaches can be sorted into static analyses [CP00] and dynamic analyses [RS04]. Another important characteristic relates to the *activation pattern* of an activity. A *periodic* activation demands the release dates to be fixed to a time interval T_i:

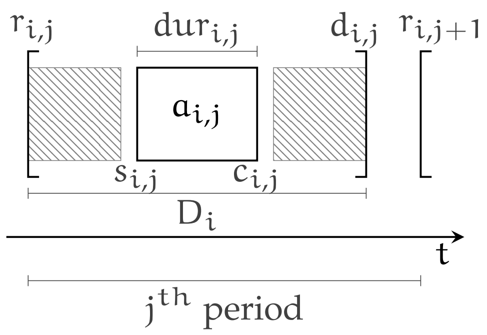


Figure 4.1: Timing characteristics of a real-time activity execution. The hatched areas represent the earliest and the latest possible execution of the activity instance $a_{i,j}$.

$$r_{i,j} = r_{i,1} + (j-1)T_i \ . \qquad (4.1)$$

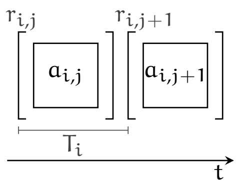


Figure 4.2: Periodic activation.

The *sporadic* activation pattern allows for a more flexible release date, based on a minimum separation time T_i:

$$r_{i,j} \geqslant r_{i,j-1} + T_i \ . \qquad (4.2)$$

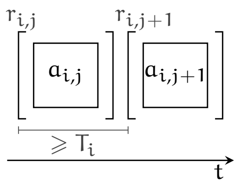


Figure 4.3: Sporadic activation.

Aperiodic activities may arise at any instant and are often triggered by an external event:

$$r_{i,j} > r_{i,j-1} \ . \qquad (4.3)$$

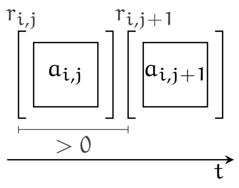


Figure 4.4: Aperiodic activation.

In the following the timing aspects of a RTS are grounded into the context of a CBRS by mapping each activity to a component instance. This means that the activation pattern and the WCET are considered per component instance. More important than the individual worst-case execution and response times, is the worst-case end-to-end response time (WCE2ERT) [MM06][3] for the system's execution. A robotic system that interacts with the environment, naturally senses information from the environment and acts to influence it. In the

3 Sometimes referred to as *whole system response time* [Die+17].

same manner, a CBRS forms a control cycle by chaining multiple components together, starting with the sensed information and responding with a respective action [Lot+15]. In the following, I refer to such a sequence of components as sense-react chain (SRC) [WW19]. An example is shown in Figure 4.5.

sense-react chain

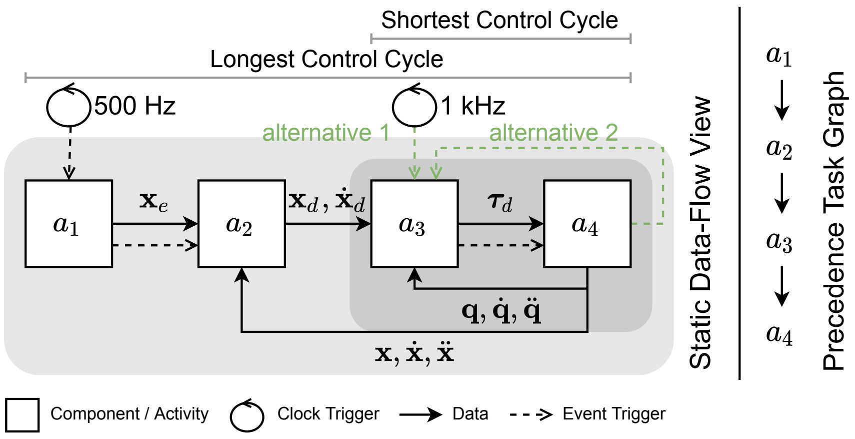


Figure 4.5: Static view of an exemplary closed-loop control system. Two mixed SRCs are shown. The shorter one starts with a_3 and ends with a_4. While a_4 gets event-triggered upon completion of the previous activity, a_3 is clock-triggered with a frequency of 1 kHz. This configuration mixes data and trigger chains. Alternatively (green), instead of being clock-triggered, the next iteration of the SRC could directly be started upon completion of a_4. This would turn the type of the SRC into a pure trigger chain. The longer SRC involves all four activities, but is activated with only half the frequency of the faster SRC. This means that for the calculation of the E2ERT, the iteration of the faster SRC counts that begins after the completion of a_2. This data-flow dependency is also captured in the precedence sequence on the right, which is however not sufficient to capture the activities' activation patterns. Note that here activity and component is used synonym.

The time from sensing to acting of a SRC represents the end-to-end response time (E2ERT), which consequentially leads to the WCE2ERT being the largest possible response time of a SRC. To describe the execution semantics of such a chain, it can be classified into a *trigger-chain*, a *data-chain*, or a mixture of both [MMS12b]. A *trigger-chain* is defined by having only one triggering source per SRC. In terms of activities, this means that the first activity in the chain is triggered by e.g., a clock, an event, or an interrupt, while subsequent activities are triggered upon completion of the previous activity. In contrast to that, a *data-chain* uses independent triggers for each activity. Depending on the type of SRC, the calculation of the WCE2ERT needs to be approached differently [MMS12b]. Figure 4.5 shows a mixed chain that brings in aspects from the *data-chain* type, by composing multiple *trigger-chain* type control loops with different triggers. Since low-level control loops are mainly data-flow driven, the execution order of the involved activities are implicitly defined and constrained by their data-flow. Hence, this data-dependency can be specified with a Precedence Task Graph (PTG) [But11b] by introducing precedence constraints between activities [But11a]. Using a PTG it can be determined which parts of the system can be executed in parallel and which

precedence constraint

activities need to wait for others to finish, e.g., because they depend on previously processed results from another activity. However, specifying a PTG is not sufficient to cover the semantics of *data-chains*, since in this case the execution order cannot be completely deduced from the data-flow. For instance, to ensure a stable control behavior of the robot, the components need to be executed with different frequencies rather than in the exact sequence that is shown in Figure 4.5. A direct and potentially fatal consequence is that components process old input data, which frequently occurs in robotics [MMS12a; Ben+09]. Thus, in addition to a PTG, the independent triggers for the (sub)chains need to be modeled as well.

Once the execution semantics of a system are defined, a suitable schedule needs to be found. In light of the numerous works on scheduling that exist in the literature [Bru+16], a schedule basically represents an assignment of activities to processing cores over time [Ouh13], defining the points in time when activities start and end, such that the three major types of constraints are satisfied [Dej16]: *time-related*, *resource*, and *objective function* constraints.

TIME-RELATED CONSTRAINTS Time-related constraints can be expressed in form of the same precedence constraints as used by PTGs. An activity a_i precedes another activity a_j, if it is completed before the other one starts:

$$c_i \leqslant s_j \; . \tag{4.4}$$

Additionally, transition-time constraints can be used to constrain the timing gaps between activities. Since a low-level control loop should usually run as fast as possible, this type of constraint will not be explained further.

RESOURCE CONSTRAINTS Resource constraints are used to define the access to a resource (m), shared by a set of activities (Ω_m). While there are various different kinds of resource models in the literature, exclusively *unary resources* [Vil04] are assumed in the following. A *unary resource* can only be accessed by one activity at any point in time. Thus, a disjunction of precedence constraints is introduced by this kind of constraint:

$$\underset{i \neq j}{\forall} a_i, a_j \in \Omega_m : c_i \leqslant s_j \vee c_j \leqslant s_i \; . \tag{4.5}$$

Since processing cores are commonly represented as resources, *core affinity* constraints are used to define the processing cores, an activity can be executed on. Hence, either adding or removing the activity from Ω_m for the respective processor resource m.

OBJECTIVE FUNCTION CONSTRAINTS Different constraints can be formulated on the optimization of a schedule. A common objective function constraint is to minimize the *makespan* of a schedule. The *makespan* defines the width of the time window in which the schedule exists. Minimizing it means minimizing the maximum completion date of all activities:

$$\text{minimize} \max_{a_i \in \mathcal{S}}(e_i) \, , \tag{4.6}$$

where $\mathcal{S}$ denotes the set of activities, involved in the schedule. Other objective function constraints might concern minimizing the amount of used resources or the earliness and tardiness of the activities [Ouh13].

4.1.2 *Separation of Roles and Concerns*

To cover the aforementioned timing concerns of a CBRS, their responsibilities need to be distributed among the developer roles (see Section 2.3.2). In particular, the *Component Supplier*, the *System Builder*, and the *Performance Designer* need to be addressed.

COMPONENT SUPPLIER The Component Supplier is naturally interested in determining general execution properties, such as the WCET/WCRT and the memory consumption, in order to provide this information alongside the associated components. Especially, the WCET of a component is later used as ground-truth for validation and analyses, conducted by other roles. Suitable visualizations for the concerns of this role found in the literature are histograms or classical one-dimensional plots.

SYSTEM BUILDER The responsibility of the System Builder is to create a system and its data-flow by drawing on the components provided by the Component Supplier. From those components, SRCs need to be created. This task includes defining precedence constraints in form of a PTG, specifying the trigger types of the SRCs, and constraining data items on how far an input can date back to be still considered valid for processing. Together with the Behavior Developer, the System Builder defines the WCE2ERT for the system at which it is still capable of performing the desired behavior in a stable manner. This optionally includes the specification of the maximum time / minimal frequency that different control cycles (i.e. sub-SRCs) are allowed to be updated, in order to keep the robot stable. In Figure 4.5, the shortest and faster control cycle keeps the robot stable using low-level commands, while the longest and lower cycle executes higher-level commands.

PERFORMANCE DESIGNER The Performance Designer takes care of realizing the system planned by the System Builder in terms of activities that are assigned to processing cores. Here, the challenge is not only to make the system schedulable, but also to meet the different requirements, specified by the other roles. Therefore, this role has very different concerns and thus requires other support as compared to the previously mentioned roles. The main concern of this role is to create a suitable schedule. This involves not only all the previously defined timing-related constraints, but also the capabilities of the targeted hardware platform. The number of available processor cores influences the degree of parallelization and thus the actual WCE2ERT. This is important, because the main objective of the Performance Designer is to avoid a violation of the WCE2ERT. In the case of a violation, the system is considered to be not schedulable and another iteration of the development process is required, triggering the System Builder to optimize the system and the PTG. This, hopefully enabling the Performance Designer to eventually find a schedulable solution. Depending on the trigger type of the SRCs, suitable activation patterns need to be chosen for the activities [Lot+15]. While these new requirements seen individually do not seem to be overly expensive, the entire set of requirements however, yields a highly complex optimization problem that requires concern-specific views in terms of analyses and visualizations to support this role: The actual system's execution and its data-flow need to be compared to the specified SRCs. A visualization in form of e.g., a timing diagram with data-flow information is suitable for this purpose.

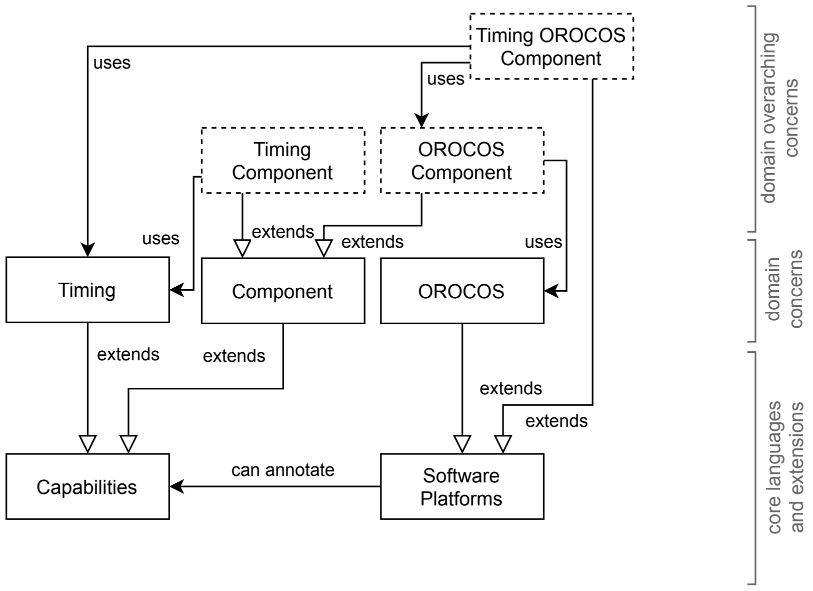


Figure 4.6: Illustration based on an excerpt of Figure 3.5, showing the integration of the timing-related DSL modules into the existing module composition of CoSiMA.

4.2 MODELING OF TIMING CONSTRAINTS

The findings from the domain analysis are realized in three DSL modules. These modules extend CoSiMA to support the modeling of constraints on the execution time behavior of a (real-time) CBRS. Following the L3Dim approach, the timing-related modules are seamlessly integrated into the existing composition structure of CoSiMA. Figure 4.6 shows the integration using an excerpt of the extended Figure 3.5. While the general timing concepts, such as the WCET, the WCE2ERT, and the SRCs are captured in the *Timing DSL* as new modeling capability, the concepts needed for the capability- and dimension-overarching integration are realized in the *Timing Component DSL* and in the *Timing OROCOS Component DSL* respectively. Figure 4.7 shows the composition on the concept-level along the involved DSL modules, depicted in Figure 4.6. As identified in the domain analysis, the different concerns are distributed among different roles to avoid overwhelming a single role with various different tasks. With the same mindset, the modeling responsibilities are split up as well, relieving the individual developers and offering the possibility for a modular workflow.

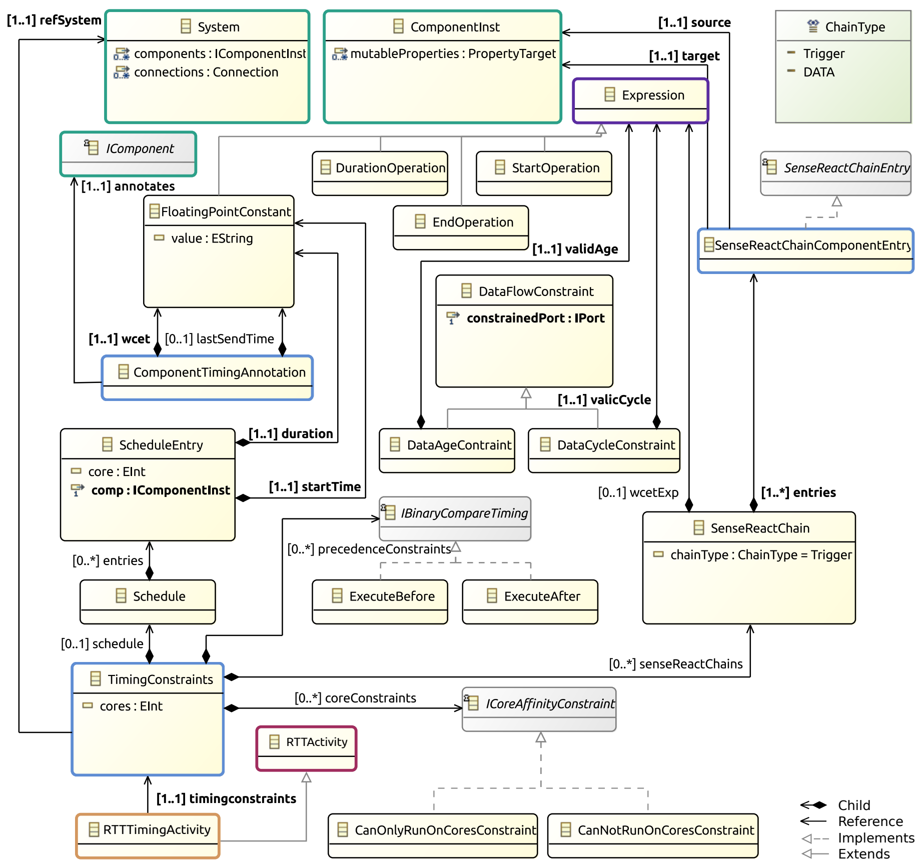
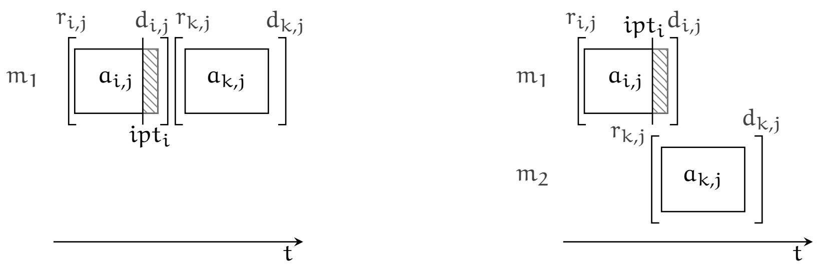


Figure 4.7: *Timing DSL* (gray) meta-model including dependencies to concepts of related language modules from CoSiMA, namely, *Component DSL* (green), *Timing Component DSL* (blue),*Timing OROCOS Component DSL* (orange), *OROCOS Component DSL* (red), and MPS' *BaseLanguage DSL* (purple).

4.2.1 *Worst-Case Execution Time Modeling*

component-level aspects of the timing concern

The findings of the domain analysis show that for the verification of the temporal behavior of each task, the WCET is usually combined with the WCRT to take into account the interference of other tasks executing on the same processing core [Lau+14]. While the WCRT is heavily concerned with the scheduling-overhead, the WCET is more appropriate to cover the execution time of a task itself. Thus, the concept of the WCET occurs to be an essential part of a timing analysis, especially for non-preemptive scheduling with unary resources as used in CoSiMA (see Section 4.3). The *Timing DSL* contains the `WCET` concept, which is put into the context of component-based systems by using the `ComponentTimingAnnotation` from the *Timing Component DSL*, connecting the timing with the component modeling capability. The annotation can be added to an `IComponent` model fragment, allowing the Component Supplier to enrich the component description with the WCET as well as the independent processing time (IPT) [WW19] (see Figure 4.9). In contrast to the WCET, the IPT provides information regarding the time it takes until a component enters a phase where none of the other components in a SRC is dependent on the data-flow or processing of that component (see Figure 4.8). This information is optional, but when provided it can be used to improve the schedulability of a CBRS. This is because, it decouples the execution of components that have data-flow precedence constraints, from the completion of the preceding component execution. In general, if component a_k is dependent on the data-flow information of component a_i, $a_{k,j}$ needs to wait for the completion of $a_{i,j}$. Using the idea of IPT, $a_{k,j}$ can be executed in parallel to $a_{i,j}$ once the data-flow constraints are met, even though the execution of $a_{i,j}$ has not yet finished. Using solely unary resources (i.e. processing cores), the IPT only has an effect if multiple resources are available. Otherwise, the subsequent components need to wait until their predecessor is fully completed. Note that determining the IPT is however not less challenging than the WCET.



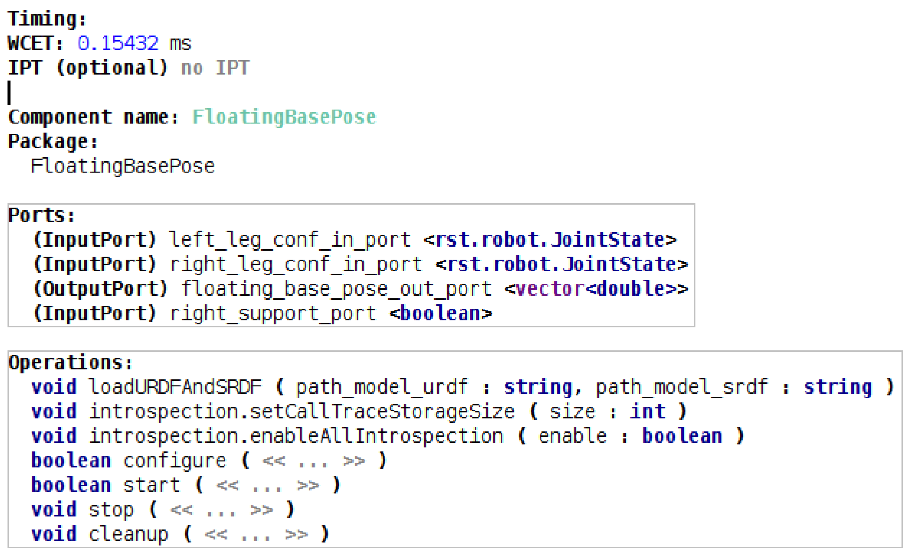
(a) Single unary processing core (b) Multiple unary processing cores

Figure 4.8: Comparing the influence of the IPT on the E2ERT of two activities (a_i, a_k), scheduled on a single or multiple unary processing cores (m_1, m_2).

4.2.2 *Worst-Case End-To-End Response Time and Sense-React Chain Modeling*

system-level aspects of the timing concern

The main task of the System Builder in the context of timing modeling is to sort the `ComponentInsts` involved in the system into one or more SRCs. A `SenseReachChain` in general is a sequence of `SenseReactChainEntrys`, which represent executable units. Both concepts are provided by the *Timing DSL*. In the realm of component-based systems, the SRC resembles a sequence of component executions using `SenseReactChainComponentEntrys` from the *Timing Component DSL* that reference `ComponentInsts` from the *Component DSL*. Each sequence can either be classified as a trigger-chain or a data-chain (see `ChainType`). The type generally depends on the activation patterns associated with the individual SRCs (see Equations 4.1 to 4.3). A mixture can be achieved by



```
Timing:
WCET: 0.15432 ms
IPT (optional) no IPT
|
Component name: FloatingBasePose
Package:
  FloatingBasePose

Ports:
  (InputPort) left_leg_conf_in_port <rst.robot.JointState>
  (InputPort) right_leg_conf_in_port <rst.robot.JointState>
  (OutputPort) floating_base_pose_out_port <vector<double>>
  (InputPort) right_support_port <boolean>

Operations:
  void loadURDFAndSRDF ( path_model_urdf : string, path_model_srdf : string )
  void introspection.setCallTraceStorageSize ( size : int )
  void introspection.enableAllIntrospection ( enable : boolean )
  boolean configure ( << ... >> )
  boolean start ( << ... >> )
  void stop ( << ... >> )
  void cleanup ( << ... >> )
```

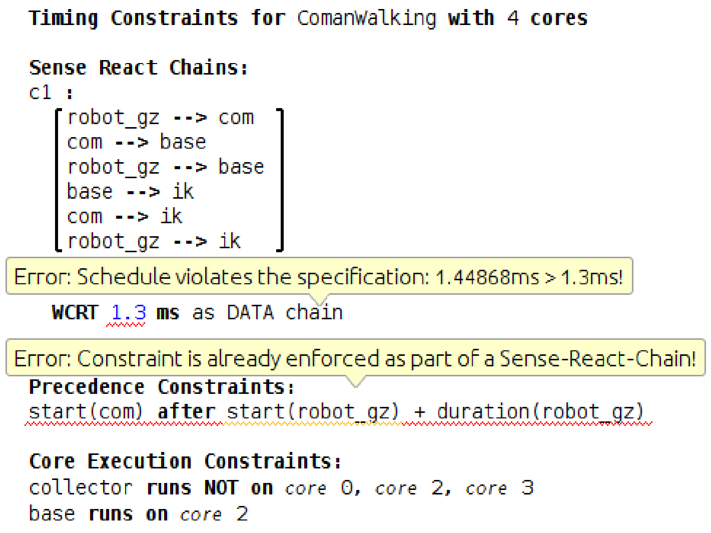
Figure 4.9: Excerpt from the model of the FloatingBasePose component, including the interface in terms of ports and operations, as well as additional timing information e.g., the WCET.

composing multiple chains. In addition to that, data-flow constraints need to be specified in form of precedence constraints [But11a] or a PTG using the binary timing relations `ExecuteBefore` and `ExecuteAfter` (see Equation 4.4). With this, SRCs can be modeled and the Performance Designer can be tasked to find a schedule that meets the specification (see Section 4.3). However, not every schedule will result in the desired behavior of the modeled robotic system. According to the domain analysis, in a lot of scenarios, it is crucial to specify the WCE2ERT that a SRC is allowed to have in order to ensure a stable control behavior of the system and robot [MMS12b]. Hence, when specifying a SRC the System Builder needs to provide the maximal desired E2ERT, which is then used to check if such a chain can be executed in the specified amount of time. As it can be seen in Figure 4.10, model-checks can be applied. For instance, the duplication of precedence constraints, already implied by the sense-react chain, can be detected.

4.2.3 *Core Affinity and Schedule Modeling*

To model an executable schedule that meets the specified SRC of the System Builder, the per-component timing information from the Component Supplier needs to be involved as well. This is the non-trivial responsibility of the Performance Designer, who has the option to define additional core affinity constraints, to restrict which component should or should not be executed on which cores (see Section 4.1.1). This decision may be influenced by application-specific knowledge. Core affinity constraints are modeled with the `ICoreAffinityConstraint` concept offered by the *Timing DSL*. The outcome of the Performance Designer's task is a `TimingConstraints` model that contains a valid `Schedule` (see Figure 4.7). The *Timing Component DSL* provides the ability to link a `TimingConstraints` model to a `System` model. The link to a system is used to supply the necessary (timing) information to map the platform-independent schedule to the execution semantics of a e.g., software platform, with which the system will be deployed and executed. In case of OROCOS RTT, the `ActivityDemand` from the *OROCOS Component DSL* is satisfied by instantiating a `RTTTimingActivity` from the *Timing OROCOS Component DSL* that delegates the responsibility to the modeled `TimingConstraints` for the respective `System` model.

Now even though the composed model is sufficient to generate an executable system with explicitly defined execution behavior, a mapping between the modeled and the actual execution semantics of the chosen software platform needs to exist. However, defining the execution of a component in terms of period and priority as it is offered by OROCOS RTT, is not sufficient to express all aspects of which the *Timing DSL* is capable of modeling. Therefore, an extension to the execution semantics of OROCOS RTT is presented in Section 4.3.2.



```
Timing Constraints for ComanWalking with 4 cores

Sense React Chains:
c1 :
  [robot_gz --> com  ]
  [com --> base      ]
  [robot_gz --> base ]
  [base --> ik       ]
  [com --> ik        ]
  [robot_gz --> ik   ]
Error: Schedule violates the specification: 1.44868ms > 1.3ms!
  WCRT 1.3 ms as DATA chain

Error: Constraint is already enforced as part of a Sense-React-Chain!
Precedence Constraints:
start(com) after start(robot_gz) + duration(robot_gz)

Core Execution Constraints:
collector runs NOT on core 0, core 2, core 3
base runs on core 2
```

Figure 4.10: Excerpt from a system's timing model, including the defined sense-react chain as well as precedence and core constraints. Furthermore, a set of timing-related model checks are shown.

4.3 SYNTHESIS OF AN EXECUTABLE SCHEDULE

The previously introduced DSL modules allow the specification of timing constraints on a modeled system as well as the manual specification of a schedule. However, creating a schedule that meets all the specified requirements is still a tough task for the Performance Designer, especially if the data-flow dependencies in the system are complex and involve a great amount of components. To support the Performance Designer in performing this task, a homogeneous transformation is used that enriches an existing model with additional information. The aim of the transformation is to use the model information (e.g., `SenseReactChain`) provided by other roles to synthesize a suitable `Schedule`.

The challenge of creating such a transformation is addressed in the following: In general, finding a schedule can be formulated as a scheduling problem, where n jobs need to be mapped to m resources. According to the domain analysis, *resource*, *objective function*, and *time-related* constraints are the three major types of constraints that have an impact on the resulting schedule (see Section 4.1.1). For this work the problem formulation of a Flexible Job Shop Scheduling Problem (FJSSP) [CK16] is used. Let $J = \{J_1, \ldots, J_n\}$ be a set of n jobs and $M = \{M_1, \ldots, M_m\}$ be a set of m available machines (i.e. resources). Regarding the resource constraints, the machines (i.e. processing cores) are considered to be unary (see Equation 4.5) and there is no predefined constraint that associates a specific job (i.e. activity) with a specific machine. Instead, a job can be mapped to any machine, as long as this mapping does not violate any other constraint. Further, each job J_i is characterized by a set of r non-preemptive operations:

machines refer to processing cores

$$O_i = \{O_{i,1}, \ldots, O_{i,r}\} \ , r \leqslant m \ . \tag{4.7}$$

Each set of operations represents the potential executions of the associated job on the chosen machines. The elements in the sets O are influenced by the modeled core affinity constraints: `CanOnlyRunOnCoresConstraint` and `CanNotRunOnCoresConstraint`. Additionally, the execution time of the operations inside a set O_i is considered to be the same:

$$\forall k : e_{i,k} = e_i \ . \tag{4.8}$$

This means that there is no difference in execution time using different machines. The objective function for this particular problem is to minimize the makespan (see Equation 4.6), which will result in the shortest E2ERT of the SRC that is schedulable. The time-related constraints are covered in form of the precedence constraints (see Equation 4.4) that are modeled in a `SenseReactChain`. Figure 4.11 shows an exemplary graph visualization of the precedence constraints, where the nodes represent the elements of O.

To solve the FJSSP it can be turned into a Constraint Satisfaction Problem (CSP), for which multiple solvers exist. A benchmark of current state-of-the-art solvers, including Google's CP-SAT,[4] is presented in [DT19]. A CSP is a combinatorial optimization problem. It is formally defined by a triplet $\mathrm{CSP}(V, D, C)$, where V defines a set of decision variables, D represents a set of domains (i.e. possible values for the variables), and C defines a set of constraints on the assignments of values to variables [Dej16]. In order to represent the previously defined FJSSP, V, D, and C are chosen as follows:

VARIABLES V: For each operation $O_{i,k}$ two variables are defined. The starting time $s_{i,k}$ and the completion date $c_{i,k}$. The variables are set into relation according to $s_{i,k} + D_i = c_{i,k}$, where D_i is the (worst-case) execution time (i.e. the relative deadline) or (if provided) the IPT.

4 `https://developers.google.com/optimization`

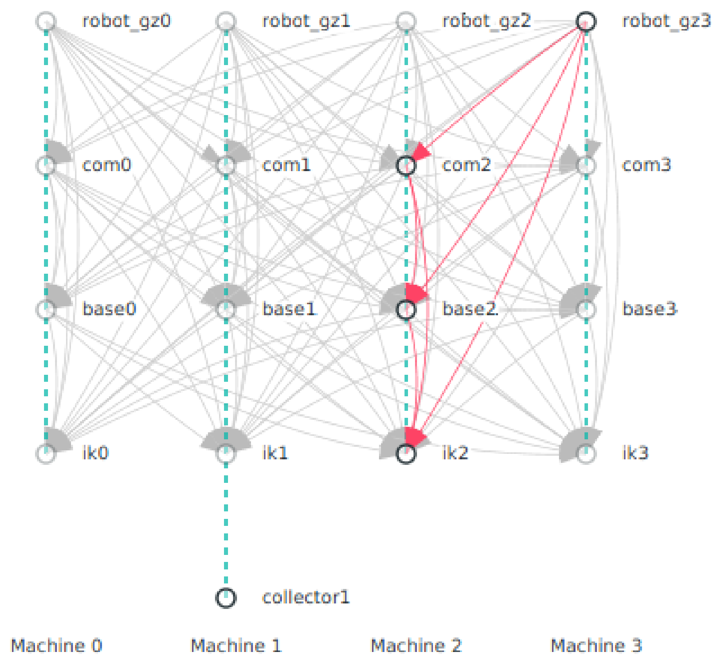


Figure 4.11: Screenshot of the interactive visualization in MPS for the possible component executions (nodes) on each processing core (machine), derived from the modeled timing constraints. Arrow-headed and dotted lines indicate precedence and machine constraints respectively. The red lines indicate the precedence constraints associated with the component executions that are part of the resulting schedule.

DOMAINS $D(X)$: The domains for the variables are defined as $D(s_{i,k}) = [0, horizon]$ and $D(c_{i,k}) = [0, horizon]$, where the horizon marks the end of the valid time window for the schedule. In this case it equals the specified WCE2ERT of the SRC.

CONSTRAINTS C: There are disjunctive and conjunctive constraints. The first kind is used to formulate resource constraints: $\forall O_{i,k}, O_{q,k}\ i \neq q : s_{i,k} + D_i \leqslant s_{q,k} \vee s_{q,k} + D_q \leqslant s_{i,k}$ ensures that a machine can only process one operation at a time. Whereas, the latter kind is used to express precedence constraints: $\forall i, q$ where i precedes $q \wedge i \neq q : s_{i,k} + e \leqslant S_{q,p}$,

$$\text{where } e = \begin{cases} \min(ipt_i, D_i) & \text{if } k \neq p \text{ and } ipt_i \text{ exists} \\ D_i & \text{else} \end{cases}.$$

This constraint ensures that precedence is not violated and operations executed on different machines can make use of the IPT if provided. This allows to start the execution of another operation even though the previous execution is not yet completed, but the necessary data-flow information exchange is already completed. Since each operation of a job represents an execution alternative for a specific machine, only one operation of each job is allowed to be contained in the final schedule: $\forall i \dot{\exists} k : s_{i,k} \in S$, where S denotes the resulting schedule, consisting of a set of starting times for the chosen operations.

4.3.1 *Synthesis Using a Homogeneous Model Transformation*

A homogeneous transformation is created to enrich the existing model with a suitable schedule that respects the modeled timing constraints. To this end, the transformation first translates the existing model into a CSP that is fed to e.g., Google's CP-SAT solver. Second, the output of the solver is then lifted back in form of a schedule into the model. A graphical visualization of a schedule model in CoSiMA can be seen in Figure 4.12.

implementation concepts shown in the meta-model in Figure 4.7

In detail, the amount of available cores is extracted from the *cores*-property of the `TimingConstraints`. The involved `ComponentInst`s are inferred through the reference to the `System` and the individual WCETs and IPTs are collected from the `Component` description properties of the instances. The conjunctive constraints are extracted from the precedence constraints of the `SenseReactChain`s and the additional ones, defined in the `TimingConstraints`. In contrast, disjunctive constraints are solely derived from the core execution constraints of the `TimingConstraints`. Eventually, the set O (see Equation 4.7) is formed using the modeled information, leading to the definition of the set of the CSP constraints (C), by drawing on the extracted conjunctive and disjunctive constraints. To lift the solved schedule into the existing model, a `Schedule` model fragment is created, which is populated with `ScheduleEntries`, associating each `ComponentInst` with a core for execution, the start time, and the estimated duration based on the provided WCET.

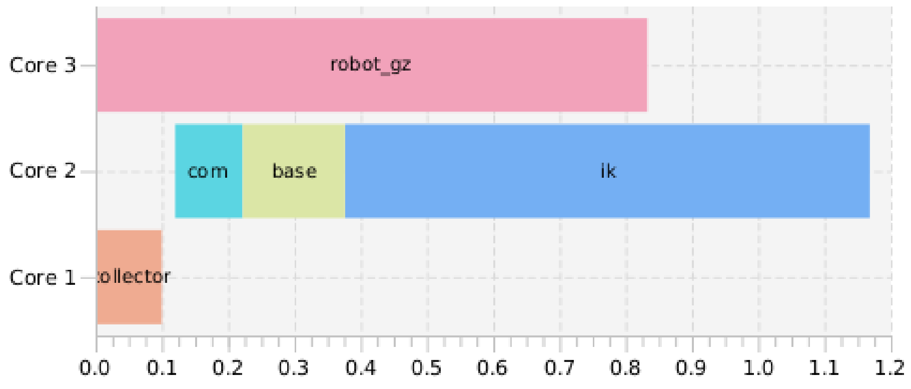


Figure 4.12: Screenshot of the lifted schedule based on the constraints displayed in Figure 4.11 lifted back into MPS.

Once a schedule is found, it can either be an optimal schedule or a feasible one, which means that the solver did not find an optimum. In both cases, however, the schedule meets the specified requirements. The WCE2ERT of the SRC can also be treated as a weak constraint for the solver, to get an initial idea of the schedule, or if the WCE2ERT is unknown. In this case, the found schedule will still be lifted into the modeling environment, but indicated with an error (see Figure 4.10). This is useful for the Performance Designer and ultimately for the System Builder as well, since the defined SRC cannot be scheduled in the desired amount of time. In such a case, the development process needs another iteration, including the System Builder to optimize the architecture and SRCs, to eventually enable the Performance Designer to find a valid schedule.

4.3.2 *Execution Semantics of the Schedule*

To execute a synthesized schedule, the chosen execution environment needs to support appropriate execution semantics. Unfortunately, the semantics of OROCOS RTT are not expressive enough to realize the synthesized schedules. OROCOS RTT uses the concept of activities, where each component needs to be assigned to an activity, which in turn executes the assigned components in sequence on a particular core. An OROCOS RTT activity is mainly defined by its activation, which is either periodic or aperiodic, and by its priority. To close the gap between the modeled schedule and the capabilities of the execution environment, I extended the execution semantics of OROCOS RTT to provide a target for the schedule generation from CoSiMA.

4.3.2.1 *CoSiMA's Core Scheduler*

Being limited to the execution semantics of OROCOS RTT, results in a multitude of schedules that can be modeled, but that cannot be mapped to or executed with OROCOS RTT. In fact, to produce a stable robot behavior, it is according to [Mor15] in most cases not mandatory to ensure that all communications and computations are completed within a fixed time interval. Instead, the flow-preservation [Tal+04] property is sufficient for the execution semantics. Flow-preservation guarantees that the data-flow between two components is preserved even if the execution times vary. Hence, as long as the WCE2ERT of a SRC does not exceed a defined maximum, the system is able to produce a stable behavior while also compensating for varying execution times of the components.

The flow-preserving execution semantics are implemented as an extension for OROCOS RTT that provides an API, which directly integrates into the OROCOS RTT scripting language (i.e. OPS). Thereby, facilitating the generation of schedules. In contrast to the approach proposed by [Mor15], a non-invasive strategy is used, which avoids changes to the OROCOS RTT framework itself. To realize a desired schedule on the technical level, *Core Schedulers* (implemented as OROCOS RTT *TaskContexts*) are introduced per involved processing core. Components that are scheduled on the same processing core are linked to the same *Core Scheduler*. The execution of each component can be conditioned by different constraints (e.g., precedence constraints). A scheduler sequentially triggers the execution of the linked components in a provided execution order, while respecting the precedence constraints. To coordinate multiple schedulers, one is selected to be the *Coordinator* that initiates a new iteration of the schedule. The *Coordinator* is triggered conform to the trigger of the sense-react chain, which is either time-based (i.e. periodic activation), or data-based, representing an activation based on an (external) event. Note, to start a new iteration as fast as possible, the event can also be the completion signal of the last component in the SRC.

As seen in Figure 4.13, all Core Schedulers have four phases:

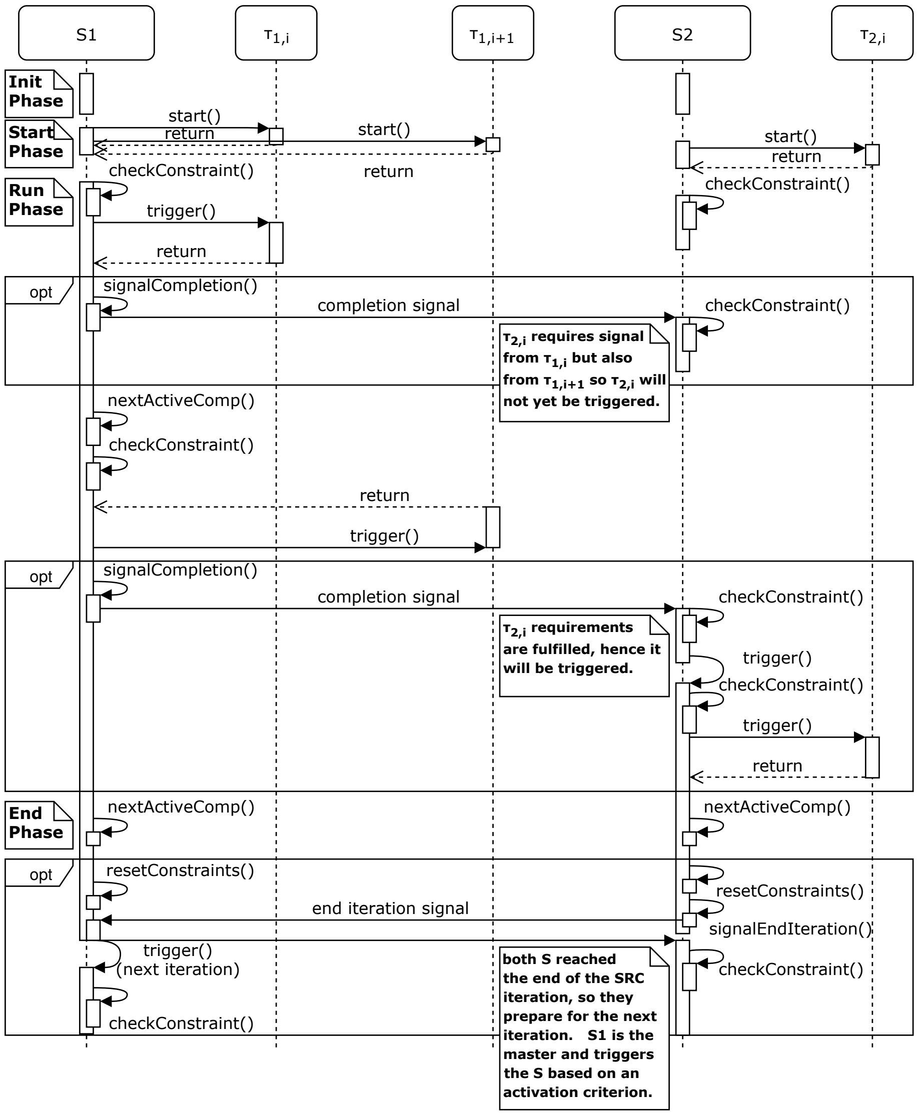


Figure 4.13: This sequence diagram shows an exemplary execution of two Core Scheduler (S1 and S2) with their associated TaskContexts ($\tau_{S,i}$, where S is the index of the associated Core Scheduler and i is the index in the execution order). In this execution, all four phases are visualized. $\tau_{1,i}$ has no constraints and is the first task to be executed. $\tau_{2,j}$ needs the completion signal from $\tau_{1,i}$ and $\tau_{1,i+1}$ in order to execute, so it waits until it is woken by a signal. After all tasks are completed, the *coordinator* (S1) schedules the next iteration and notifies the other Core Scheduler (S2).

INIT. PHASE In the first phase, all constraints for the linked components are initialized and the associated ports for inter-core signaling are created. Further, each component gets assigned to a *Core Scheduler*, and sorted according to the provided execution order. Finally, the scheduler that is responsible for the first component in the SRC, becomes the *Coordinator* that coordinates all other schedulers.

START PHASE Before the schedulers are ready to enter the run phase, they verify that all required ports are connected ensuring that inter-core signaling is enabled. Afterwards, the run phase is entered automatically.

RUN PHASE This phase starts with checking the precedence constraints for the currently active component in the execution order. If not all constraints are fulfilled, the *Core Scheduler* yields until woken by a signal and then reevaluates the constraints. Instead, if all constraints are satisfied, the execution of the currently active component is triggered. Once a component completed its execution, an *execution-completed* signal is fired to notify the other schedulers. Eventually, the next component in the execution order becomes the currently active component.

END PHASE Once one iteration of the execution order has finished, the end phase is entered and the schedulers reset all constraints and the first component becomes the currently active one again. Every *Core Scheduler* waits for the *coordinator* to trigger the next iteration of the SRC. The following iterations omit the init and start phase.

A schedule containing components that make use of the concept of IPT, will result in the generation of an additional constraint for each precedence-constrained successor on another *Core Scheduler*. Instead of waiting for the *execution-completed* signal, the successor components can already start with their parallel execution as soon as the IPT signal is received. This allows to realize a more optimized schedule in particular cases, where the preceding component uses a lot of time for computations that are irrelevant for the precedence-constrained successors.

4.3.2.2 *Transformation from Model to Execution Environment*

The *Timing OROCOS Component DSL* provides a M2T generator, which takes the modeled `Schedule` and the SRC and translates them into a set of OPS statements that relate to the API exposed by the extended execution semantics (i.e. OPS). For each core on which a component should be executed, a Core Scheduler is instantiated, except if a core would only contain one component that is also the trigger of the sense-react chain. In this case, no Core Scheduler is used for a particular core to prevent execution overhead. All components in the schedule are sorted into their associated Core Scheduler together with the precedence constraints defined in the SRC. Due to the fact that the extended execution semantics use a flow-preservation approach, the concrete start times of the component executions are not relevant. Instead, the overall trigger type of the SRC is used to determine the *coordinator* Core Scheduler and how it initiates a new control cycle (e.g., time-based or event-based). A detailed explanation of the transformation pipelines is given in Appendix A.3.

4.4 EVALUATION

The work presented in this chapter is evaluated from two different points of view: First, the integration of the created language and generator modules into the composition structure of CoSiMA is investigated. Second, the vertical application of CoSiMA shows the modeling of a robotic case study and the analysis of the system's execution in terms of explainability and predictability.

4.4.1 *Integration into CoSiMA's Composition Structure*

As described in detail in Section 4.2, the timing-specific concepts are realized in one language module: *Timing DSL*. Two adapter language modules are required to achieve a seamless integration with the existing CoSiMA DSL stack, namely *Timing Component DSL* and *Timing OROCOS Component DSL*. Note that the integration is completely non-invasive on the language (i.e. meta-model) as well as on the model level. Hence, there is no need to change existing language modules and the model fragments related to the timing concern can be dynamically added or removed from the rest of the model without invalidating other concerns. This is achieved through the use of annotations to link *Component DSL* with *Timing DSL* model fragments. The final transformation of the *Timing OROCOS Component DSL* hooks automatically into the generation pipeline of the *OROCOS Component DSL* generator, since both target the generation of OPSs statements (see Appendix A.3). Therefore, no additional generators need to be created. Instead, front-end and back-end reuse of already existing generators is performed.

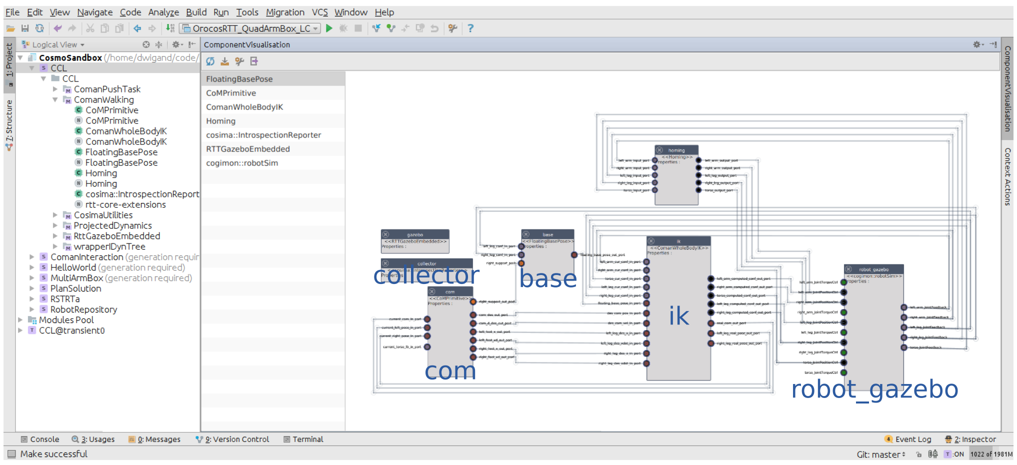


Figure 4.14: Screenshot of the system model showing the components used in the case study and their data-flow connections.

4.4.2 *Case Study: Bipedal Walking*

As a case study for CoSiMA's timing modeling, the control of the humanoid robot COMAN (see Figure 4.15) is investigated.[5] In this experiment, COMAN is tasked with walking on a straight line using a *Zero Moment Point*-based [Kaj+03] approach. For tasks such as bipedal walking, the correct time-wise execution is of upmost importance. Thus, modeling a system for such tasks, ignoring the timing concerns may lead to an executable system, however, with an unpredictable behavior, e.g., a falling robot. The system itself is modeled with the DSLs presented in Section 3.3. A screenshot of the system model in CoSiMA showing the involved components and their data-flow connections can be seen in Figure 4.14.

floating base (*base*) is responsible to compute the position and orientation of the robot's floating base. Its inputs are the current robot configuration as well as the active support foot.

com primitive (*com*) provides reference trajectories for the center of mass (CoM) and feet (swing/support) of the robot to the inverse kinematics component (Whole Body IK). The other output of this component is an indicator for the current swing foot (left/right) which is needed by the Floating Base component.

whole body ik (*ik*) solves the inverse kinematics of the humanoid robot. Given a desired Cartesian velocity, current body configuration, null-space choice, and pose of the floating base, it solves the IK using a closed loop inverse kinematic (CLIK) method and sends the results to the robot.

redundancy resolution (*rr*) provides a null-space joint motion behavior that influences the Whole Body IK. For this particular case, we try to find joint values that keep the robot's torso near its upright configuration.

robot interface (*robot_gazebo*) resembles the interface to the robot. It provides the robot's feedback in terms of joint position, velocity and torques to the components and forwards control commands to the robot.

The software platform OROCOS requires a definition of activities for each component. Without the timing model and the new execution semantics, the OROCOS RTT activities need to be chosen manually, which is very error-prone. The system model is eventually transformed into an executable system for OROCOS RTT using the generator pipelines, described in Section 3.3.4. The resulting unstable behavior can be seen in Figure 4.15 (a)-(d).

A timing-analysis with CoSiMA's introspection tools reveals that the order of execution of the components results in *com* being executed after *ik*. This leads to *com* sending its data to a later iteration of *ik*, meaning that *ik* operates on "old" data, causing the robot to fall (see Figure 4.16).

5 Special thanks to my colleagues from CogIMon for providing the functional algorithms for this scenario as well as for creating the motivation for this work in the first place.

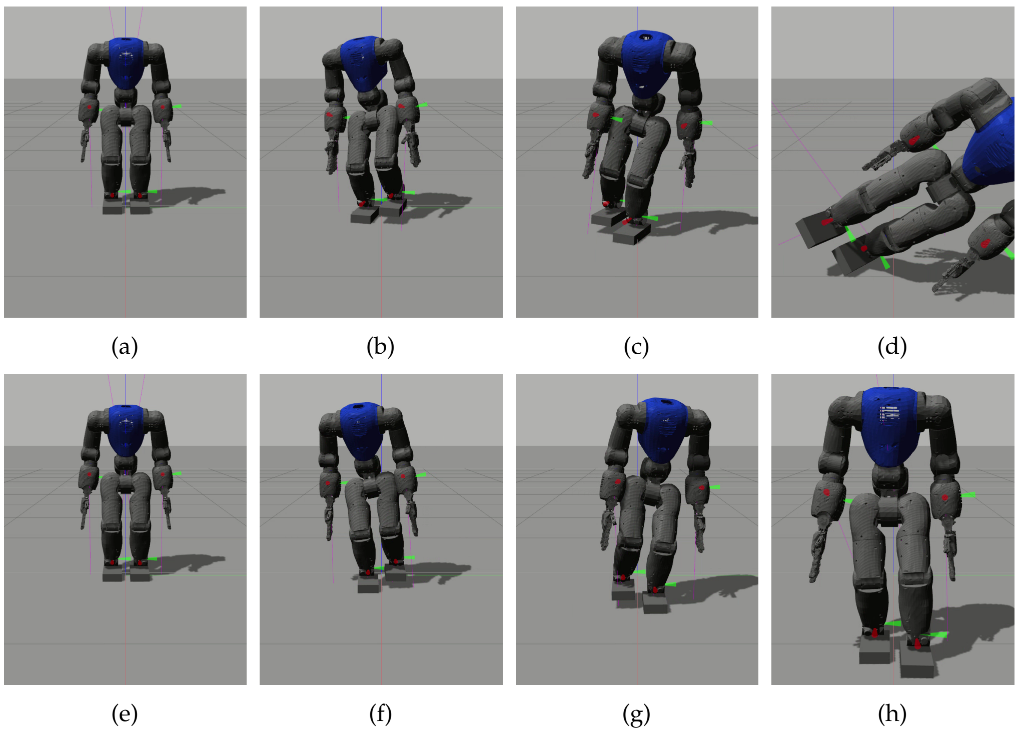


Figure 4.15: Simulation of the walking COMAN. Images (a)-(d) show an unstable behavior that leads to the robot falling down, while (e)-(h) show the expected walking sequence.

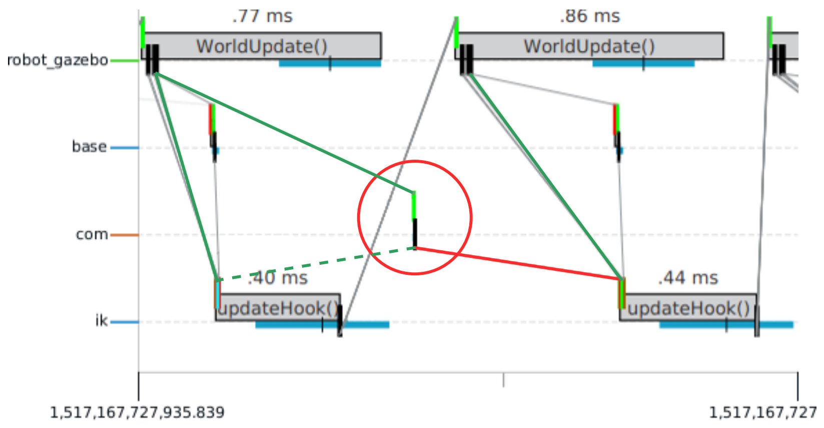


Figure 4.16: CoSiMA's introspection tool to analyze the execution behavior, shows a faulty execution cycle. Short vertical lines represent output (black) and input ports with (green $\widehat{=}$ new, cyan $\widehat{=}$ old, red $\widehat{=}$ no) data. The green lines indicate the correct data-flow, whereas the red line indicates the actual but wrong data transfer. The red circle marks the occurrence of the problem. The correct execution introspection can be seen in Figure 4.17.

To approach this problem, a suitable schedule needs to be created, ensuring that the data is received when it is needed. Therefore, the timing model that can be seen in Figure 4.10 is created. Here, the main focus is on the data-

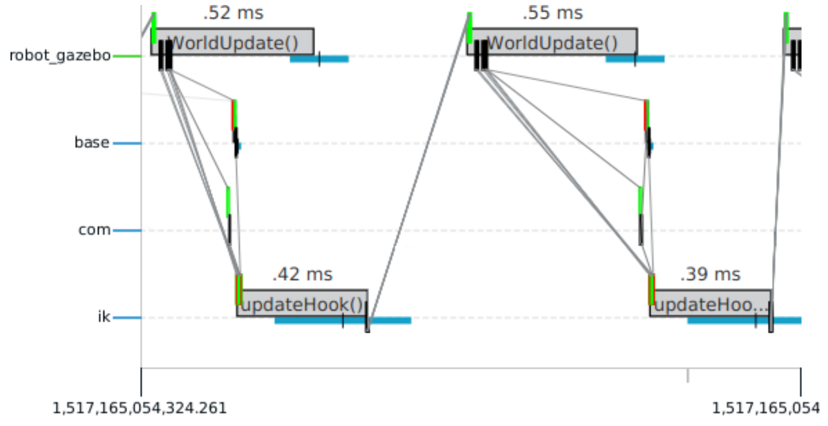


Figure 4.17: CoSiMA's introspection tool to analyze the execution behavior, showing a correct execution cycle. The color coding next to the component names, indicate the (Core Scheduler) thread they belong to.

dependencies. From the model, the schedule in Figure 4.12 is synthesized. Note that the WCET and IPT of the involved components are determined using a sample-based approach for this experiment, since this is not in the focus of this work. Please refer to [Lot+16] for more details on that topic. Additionally, the schedule is transformed to be interpreted by CoSiMA's Core Scheduler. In this experiment, only one Core Scheduler is instantiated on core 2. It takes care of scheduling *com*, *base*, and *ik* in sequence. To avoid over-head as described in Section 4.3.2.1, the robot interface is the only component that is running on core 3 without a Core Scheduler. In this particular case, this is possible, because the robot interface component represents the beginning of the SRC. The resulting and correct robot behavior can be seen in Figure 4.15 (e)-(h). An introspection of the system's execution behavior in Figure 4.17 shows that the

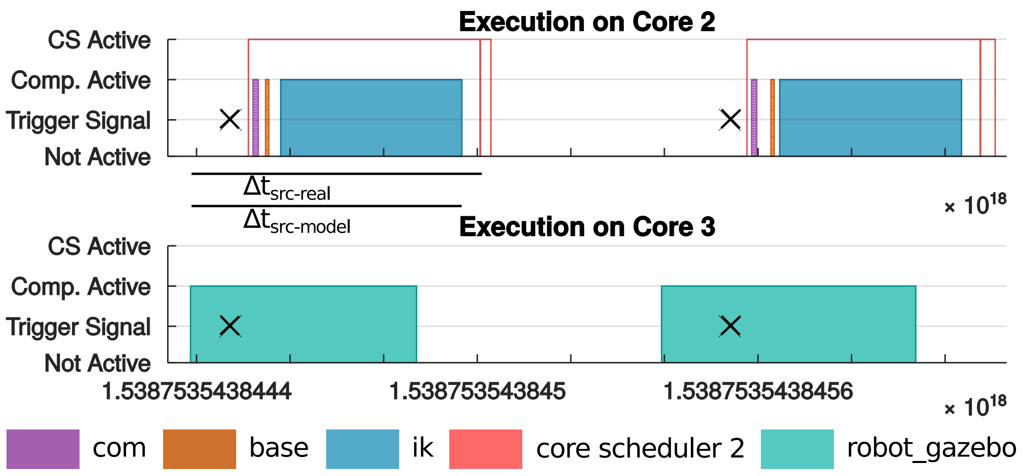


Figure 4.18: This figure shows a visualization based on collected execution samples of the system. It can be seen that the execution conforms to the schedule in a flow-preserving manner. The x indicates the firing of the IPT event, which is needed by the com component in order to execute. $\Delta t_{src-model}$ corresponds to the WCRT of the modeled SRC, while $\Delta t_{src-real}$ also includes the execution-overhead of the Core Scheduler.

data-flow complies to the precedence constraints of the SRC. The visualization of the collected execution samples from the Core Scheduler and the system's components in Figure 4.18 shows that the execution behavior conforms to the synthesized schedule in a flow-preserving manner. We did a similar scenario modified for the physio-therapeutic juggling with patients in [Moh+18].

4.5 CONCLUSION

This chapter presented a DSL extension to CoSiMA that allows the modeling of the non-functional execution timing behavior of component-based robotic systems. Making this non-functional aspect explicit, increases the explainability and opens up the possibility for verification. Through the domain analysis that was conducted, it became clear that constraints, such as precedence and core constraints, are essential aspects for modeling the timing behavior. The concepts for the meta-models are chosen in a way to allow a straight-forward generation to formulate a FJSSP, which in turn is used to synthesize a schedule that conforms to the modeled constraints. In order to execute and analyze the schedule, the execution semantics of OROCOS RTT are extended to support the property of flow-preservation, which ensures data freshness. Thus, adding another layer of explainability and reliability to the execution of such a system. With the presented approach the flow, execution order, and machine (core) constraints are preserved from the model to the execution level. The full support of additional constraints and execution patterns of SRCs still remain a challenge. For instance, more than one time-trigger is currently not supported within one SRC. Further, defining constraints for alternating execution cycles can currently not intuitively be modeled. Instead, the SRC needs to be unrolled to span the representative execution cycles for the desired behavior. In case the gap between the modeled and the execution time behavior is required to be closed beyond the flow-preservation criterion, additional challenges arise that include the proper estimation of WCET in multi-core systems as well as the right consideration of the impact of the memory bandwidth on the systems' execution.

data freshness ⎯<

Even though the synthesis of such an executable schedule is a very useful application, showing the richness of the timing model, the primary aim of this chapter is to showcase the domain-overarching integration of CoSiMA, which in this case is used to create a seamless combination of non-functional and functional aspects of CBRSs. Inspired by this, we further conducted research on model-based performance testing for robotics software components in [Wie+18].

Part II

COMPLIANT INTERACTION

This part sets the focus on researching the domain of CI and its composable integration into the modeling structure of CoSiMA. Chapter 5 is dedicated to the analysis of the domain to find the right abstractions to model CI tasks in terms of a task description. Chapter 6 translates the concepts from the previous chapter into an extension of CoSiMA, which allows to practically create valid CI task models. Chapter 7 establishes the link between the task model and the system model via a synthesis mechanism. The synthesis is presented for the projected inverse dynamics control approach.

“ *Look deep into nature, and then you will understand everything better.* ”

— Albert Einstein

5 CI DOMAIN ANALYSIS AND MODELING

This chapter provides insights on the very nature of environmental interaction and its relation to a task description. These insights are essential to consolidate the concepts, necessary to enable the modeling of compliant interaction tasks. This chapter is based on [WDW20; Deh+ss].

Describing the task that a robot is asked to perform, is generally the first step towards creating a robotics application. Without a task, there is no need for a robot to actually move. The behavior exhibited by a robot is always, but not exclusively, depending on the task. However, a task does not fully define the behavior, it rather defines requirements that bound the behavior. Hence, it controls the flexibility (in the action space) that the robot has to perform the task. Describing a task as e.g., *move to frame 1* results in an underspecification that gives the robot a lot of flexibility. Therefore, it would be totally fine for a redundant robot to arrive at that frame with a different and randomly chosen redundancy resolution every time. Colliding with objects during the execution would also be completely valid, as long as the task does not state otherwise. Especially for compliant interactions with e.g., humans, the gap between the desired task and the exhibited behavior needs to be closed to avoid dangerous situations. Nowadays, in most skill-based approaches this flexibility is reduced by additional constraints. These constraints however, are hidden inside the skills and their realizations, and are not explicitly represented on the task level [BS09; Sto09; SG07]. For applications that try to avoid contacts with the environment as much as possible, it might be sensible to request a collision-free path per default for every task. However, it is generally never a good idea to introduce hidden assumptions, since they prevent traceability and predictability. Especially, for applications that rely on compliant interactions with the environment, the relation between the task and the behavior needs to not only be free of hidden assumptions, but also clearly controllable. This requires a way to explicitly describe the interaction with the environment, which is composable with the other aspects that form a task description (see RQ 3).

5.1 TASK DESCRIPTION FOR COMPLIANT INTERACTION

no consensus about the abstraction level of a task

While there is a widespread agreement in the robotics community that a task description is essential, there is no common agreement about the individual aspects that form a task and how they are represented. A similar situation can be found in the literature regarding the terminology of "tasks" and "skills". Both terms are used to describe performable actions. According to Scioni et al. [Sci16], the two words are even used interchangeably sometimes. From a global point of view, there is also no distinctive separation between what counts as a task, a skill, or a skill primitive (see Table 5.1). Within the context

Publication	Mission	Task	(Sub) Skill	(Skill, Motion, Device) Primitive	(Elemental) Action
[SLS17]	Do order picking for order 45	Grasp blue cup	Grasp object with constraint	-	-
[Näg+18]	-	-	Screw, Push, Idle	-	-
[Tho+13]	-	TakeAndPlugObject	GraspObject	-	MoveTo
[MW01]	-	Peg-In-Hole, Screwing	-	MoveTo	-
[Klo+11]	-	-	Alignment	Linear PTP	-
[Buc+14]	-	-	-	-	Grasping, Peg-In-Hole, Screwing
[Sch+18] [Ped+16]	-	Assemble rotor	Pick Object	Move PTP	-
[SWW18]	-	-	Pick and Place, Peg-In-Hole, Screwing	-	-
[Bjö+11]	-	-	SnapFit	-	-

Table 5.1: Different interpretations of the terminology of skill-based approaches. A clear example is the activity of screwing, which is addressed as task, skill, and action by different publications.

of the individual publications, however, there is a clear hierarchical separation between the terms. This separation is sometimes so strong that each level appears as a black box that hides essential information, related to its realization and composability (e.g., [Tho+13]). To overcome the negative impact on composability and interpretability, other authors, such as Nägele et al. [Näg+18], propose to soften the hierarchy. By only using e.g., skills and explicitly providing the relevant information for self-composition, the level of abstraction can be flexibly increased. Inspired by this composition-friendly mindset, I will use the term "task" as core concept for the approach introduced in Chapter 6, which will act as a synonym for skills and actions.

Independent of the terminology, there exist individual aspects that are required to create a task description. The most prominently mentioned ones related to compliant interaction with the environment are (see Figure 5.1):

TRAJECTORIES allow the definition of task-specific motions that provides setpoints to the control system. Trajectories can be position-, velocity-, or force-based, either in joint space as well as in task space [Klo+11; Näg+18; SWW18].

COORDINATION is realized in the majority of skill-based approached by hierarchical statecharts [Van+13b; NWS15; Tho+13; Klo+11; Näg+18]. Other approaches use guarded sequencing based on pre and post conditions [MW01; Buc+14; Sch+18; Ped+16; Bjö+11].

MONITORS manage the evaluation of specified conditions. If a condition is met, the monitor triggers an event. While pre and post conditions are evaluated discretely, stop conditions [Näg+18] and guards are continuously evaluated [Ped+16; Sch+18; Tho+13; MW01].

COMPLIANT INTERACTION needs to be considered in the task description, especially considering CI tasks [Ped+13; Sci+16] (see Figure 1.1). In the majority of approaches, this aspect is rather a world description than a description of the environmental interaction, limited to the manipulated object, the used robot, or a set of feature coordinates [de +07]. Even though this aspect is heavily underrepresented, the community agrees on it being a part of the task description [Sci+16; Van+13b; Bor+16]. In the following, the interaction refers to the exchange of physical forces.

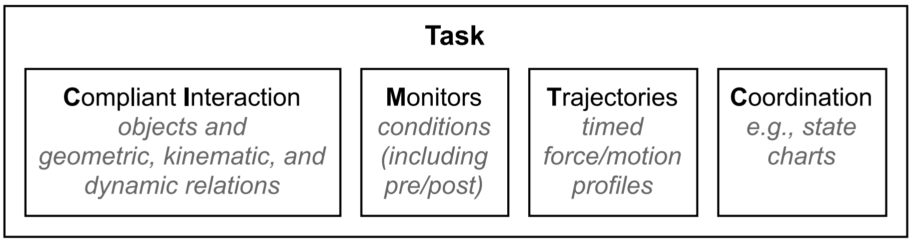

Task
Compliant Interaction
objects and geometric, kinematic, and dynamic relations
Monitors
conditions (including pre/post)
Trajectories
timed force/motion profiles
Coordination
e.g., state charts

Figure 5.1: The individual aspects that form a task description, extracted and condensed from the literature.

While there is mostly a clear separation between the monitors, trajectories, and coordination, the physical CI is often entangled in these aspects. To benefit from the separation of concerns in terms of reusability and scalability, the interaction needs to be decoupled and made explicit. Petersen et al. [Ped+16], for instance, considers the environment to exist outside the task and the skill. I propose that the interaction is considered as a context for the task, which is shared between different tasks. This way, each task is able to influence the interaction for itself and by being a context for subsequent or parallel active tasks. For instance, if the interaction is separate but composable with a trajectory, it is easy to exchange the trajectory.

In the following, the focus lies on describing the physical interaction. The other aspects of the task description are equally important, but sufficiently covered by multiple other publications. Those aspects are not addressed in detail, except for their relation to the CI with the environment.

5.2 CONTACT-BASED INTERACTIONS

a description of the environment alone is not sufficient

As previously mentioned, approaches that consider compliant interactions with the environment, model the environment to different degrees, using different abstractions. The most natural abstractions utilize geometrical objects and relations [Ad14] to describe the state of the environment. Kresse and Beetz [KB12b] for instance, use a symbolic approach to introduce geometric constraints between physical entities (e.g., *constraint keep-over spatula oven*). This idea is even carried over to recent approaches. Migimatsu et al. [MB20] use a symbolic representation for object-centric pre and post conditions of black box actions (i.e. *pick*, *place*, and *push*) to predict and define the environmental interaction. This, however, rather models discrete environmental states than the actual interaction.

a description of the interaction with the environment is necessary

To describe the compliant interaction with the environment, I argue that the focus needs to be extended from the environmental state and the geometric relations of the entities to their kinematic and dynamic interaction. An overview of the conceptual aspects, presented in the following, is given in Figure 5.2 below:

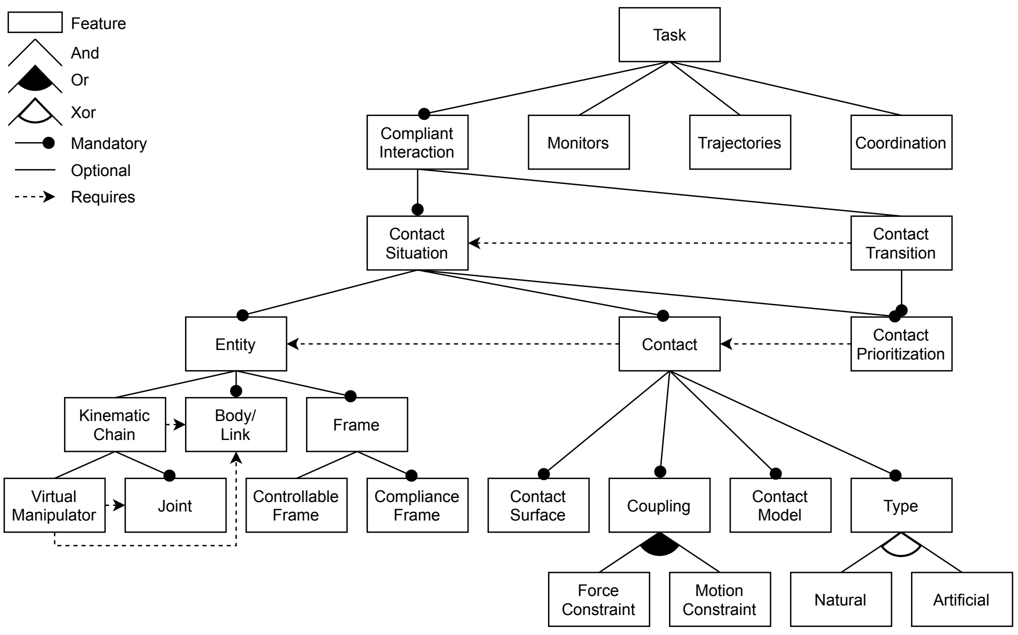


Figure 5.2: Feature diagram showing the relations of the conceptual aspects for the CI task description, discussed in this chapter.

5.2.1 *Compliant Interaction Modeling via Contacts*

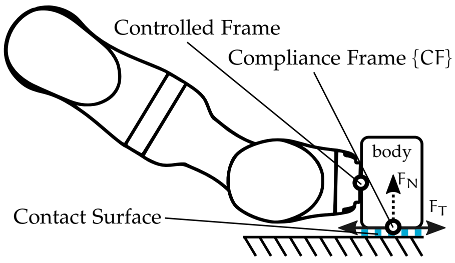
In a multitude of publications [SB18; LRS11; EWS03; CFD19; EB15; KSP08; Len+13], a contact can be viewed as the natural interface through which (physical) interaction is possible. A contact couples two physical entities, i.e. geometric (rigid) bodies. We use the term couplings to describe the interactions between the entities, which are characterized by the forces that can be trans-

coupling

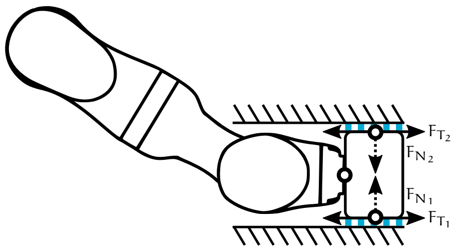
mitted through the contact and the allowed relative motion of the contacting entities. A coupling can be influenced by properties, such as the contact surface geometry and the material properties, i.e. friction and deformation [LRS11; SWW]. Contacts and their couplings can be expressed in different spaces (i.e. task and joint space), depending on the type of their contacting geometries. The behavior of a robot in a contact situation can be described by a composition of real (i.e. natural) and virtual (i.e. artificial) contacts (see Section 2.1). Virtual contacts employ virtual couplings to impose an artificial behavior, which is not imposed by the real environment. An example is the tracking of a trajectory as explained in Section 5.2.2.3.

5.2.2 *Expressing Contact Couplings via Constraints*

Couplings can be represented by a mixture of equality and inequality constraints, depending on the contact situation [PB93]. Unilateral constraints for instance may prevent one of the coupled entities from moving against the contact normal and penetrating the other coupled entity [KLB08]. Based on the works of Mason [Mas81], Raibert et al. [RC81], Featherstone et al. [FTK99], and Bruyninckx et al. [Bd96], constraints can be either "natural" or "artificial". This classification also determines the type of the coupling and the respective contact. In contrast to a natural constraint which is enforced by the environment, an artificial constraint is used to realize an interaction that is desired by a (high-level) task.



(a) Unilateral



(b) Bilateral

Figure 5.3: Rigid body contact constraints. (a) shows a unilateral contact. $-F_N$ denotes the contact force along the normal direction, while F_T represents tangential contact forces, which can be caused by e.g., friction. (b) shows a combination of two contacts, resulting in a bilateral contact constraint. Here, the controlled body is constrained in one (a) or two (b) directions. The constrained DoF depend on the contact surfaces (e.g., `f-f` see Section 5.2.3).

Considering the contact situation in Figure 5.3a, the coupling between the EEF entity (i.e. body) of the robot arm and the ground can be represented by the following types of constraints:

5.2.2.1 *Rigid Contact Constraint*

A contact divides the task space into a wrench and a motion subspace at the contact point. Contacts are generally unilateral (see Figure 5.3a). However, a contact constraint can artificially assume a bilateral contact (see Figure 5.3b), to not only ensure that the contacting bodies are not penetrating each other, but also that the contact is not broken. Assuming a rigid-body contact model, the relative wrench and velocity of the coupled entities are bound to a zero Cartesian velocity and acceleration at the contact point [Deh+18; FTK99] by

$$\mathbf{J}_c^i\,\dot{\mathbf{q}} = 0 \wedge \mathbf{J}_c^i\,\ddot{\mathbf{q}} + \dot{\mathbf{J}}_c^i\,\dot{\mathbf{q}} = 0\,, \tag{5.1}$$

where $\mathbf{J}_c^i \in \mathbb{R}^{6\times D}$ describes the constrained Jacobian associated with the ith contact point, and D the number of joints (i.e. DoFs). However, considering that Figure 5.3a depicts only a natural and unilateral contact, Equation 5.1 needs to be extended by

$$\boldsymbol{\lambda}_{f,n} \geqslant 0\,, \tag{5.2}$$

ensuring that an artificial contact force[1] $\boldsymbol{\lambda}_{f,n}$ can only be applied in the contact normal direction (n) to allow pushing and prevent pulling. To determine the separation into subspaces, a contact constraint refers to the contact surface (see Section 5.2.3). The separation can be based on individual wrench and motion directions (cf. TFF) or on the space spanned by a composition of different contact surfaces.

Task Frame Formalism

5.2.2.2 *Force Constraint*

A force constraint can be used alone or in combination with a contact constraint to define the wrench exerted in the wrench subspace. To apply a pressure into a contact, a direct force control approach can be used (see Equation 2.17) for instance. Further, total compliance in a particular direction can be realized by ensuring the contact wrench to be zero:

$$\boldsymbol{\lambda}_{f,n} = 0 \wedge \boldsymbol{\lambda}_{t,n} = 0\,. \tag{5.3}$$

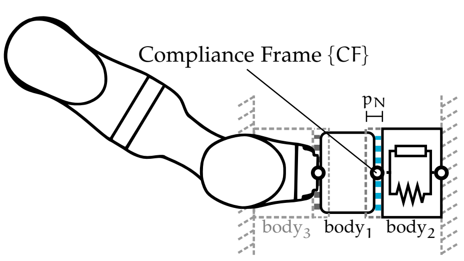
Total compliance makes the robot's motion behavior solely dependent on external perturbations in the selected direction (see Figure 6.18).

5.2.2.3 *Motion Constraint*

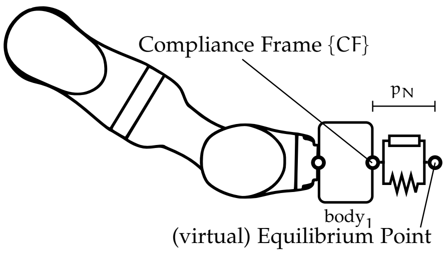
A different contact situation is depicted in Figure 5.4a. Instead of a rigid contact model, a soft contact model based on a mass-spring-damper (MSD) model [KLB08] is used to describe the interaction of the EEF's body being trapped between two elastic bodies. Figure 5.4b shows a bilateral motion constraint that is used to artificially resemble the model shown in Figure 5.4a. To achieve the same behavior, the two MSD models need to be converted into a single corresponding one.

Using a MSD model is only one way to represent a motion constraint. However, for compliant interactions it is desirable to support a coupling that when

1 The subscripts $(\ )_f$ and $(\ )_t$ refer to the contact forces and torques of the contact wrench $\boldsymbol{\lambda}$.



(a) Compliant Bilateral Contact



(b) Virtual Mass-Spring-Damper

Figure 5.4: The compliant coupling of the contact situation in (a) motivates the modeling of an artificial coupling based on a mass-spring-damper model (b) to represent a common motion constraint. The conversion from (a) to (b) requires a transformation of the two MSD models (a) into a single MSD model (b). The mass for the models is often chosen to be the EEF's mass, although inertia shaping could be employed as well.

perturbed, makes the Compliance Frame (CF) virtually behave as a physical MSD system. To achieve this behavior, typically a second-order linear relationship between position and force is introduced via an impedance controller [Vd16]. The advantage of an impedance controller is that it is especially suited to cope with unknown forces that result from unmodeled dynamics and interactions with e.g., humans. It also ensures a stable response to anticipated or unstructured disturbances [Hog85]. The external perturbation $\mathbf{F}_x$ of a Cartesian controller is given by:

in figures, frames are denotes as { · }, see Notation on page xv

$$\mathbf{F}_x = \mathbf{\Lambda}_d \ddot{\tilde{\mathbf{x}}} + \mathbf{D}_d \dot{\tilde{\mathbf{x}}} + \mathbf{K}_d \tilde{\mathbf{x}} \,, \tag{5.4}$$

where $\tilde{\mathbf{x}} = \mathbf{x} - \mathbf{x}_d$ and $\mathbf{x}_d$ is the virtual equilibrium point, $\mathbf{\Lambda}_d$ is the desired inertia, $\mathbf{D}_d$ is the desired damping, and $\mathbf{K}_d$ is the desired stiffness matrix. While the equation depicts a task space formulation, it can be formulated for the joint space analogously. An overview of impedance control variants can be found in [Sch+19].

5.2.3 *Contact Surface Constraints*

The contact surface—even a virtual one—has a direct impact on the constraints that realize the behavior of a coupling. While constraints can be formulated independently, the behavior is a result of their composition. A surface geometry naturally adds additional constraints to a coupling. These constraints can be derived from topological contact primitives. A common approach is to categorize topological contact primitives based on the contacting geometries [XJ01]. In contrast to low-level primitives such as the point contact [LMT84; Don85] or surface contact notation [PK08; KKK16], which can also be considered as multiple point contacts at the corners of the contact surface [LRS11], Xiao and Ji [XJ01] present 10 different higher-level primitives, which are based on

relation between contact surfaces and constraints combinations of point, edge, and surface contacts: principal contacts (PC)[2] $\mathtt{pc}_i = (\mathtt{a}\text{-}\mathtt{b}),\ \mathtt{a},\mathtt{b} \in \{\mathbf{v}\text{ertex}, \mathbf{e}\text{dge}, \mathbf{f}\text{ace}\}$. A point contact for instance is represented by (`v-v`), (`v-e`), or (`v-f`). The symmetrical cases are omitted for brevity. The constraints specific to this type of contact would restrict the bodies from locally moving against the contact normal, while at the same time enabling the bodies to freely move along the remaining non-restricted DoFs [KLB08]. By combining the PCs with the classification of motion constraints, introduced by Morrow and Khosla [MK97], which aim at providing specific geometric interpretations for different classes of constrained DoFs, the constrained and free DoFs for a specific contact situation can be derived. For instance, if no contact is established, the situation would be referred to by Morrow and Khosla as "Free (unconstrained) motion", which belongs to the class of **aaa**, meaning that the translation of and rotation about x, y, and z are completely free. Hence, all six DoFs can be used for pure motion control. In case of a contact described by (`v-f`), the situation would be classified as **aac**, meaning that only the translational component of the z direction is constrained, resulting in five free DoFs and one constrained DoF. Table 5.2 shows the combination of contact types and constraint classes introduced by [XJ01] and [MK97] respectively.

Contact Type	Constraint Class	DoF
-	$(1\,\vert\,1)\ (1\,\vert\,1)\ (1\,\vert\,1) \equiv \text{aaa}$	6
`f-f`	$(1\,\vert\,0)\ (1\,\vert\,0)\ (0\,\vert\,1) \equiv \text{bbc}$	3
`f-e/e-f`	$(1\,\vert\,1)\ (1\,\vert\,0)\ (0\,\vert\,1) \equiv \text{abc}$	4
`f-v/v-f`	$(1\,\vert\,1)\ (1\,\vert\,1)\ (0\,\vert\,1) \equiv \text{aac}$	5
`e-e(-c)`	$(1\,\vert\,1)\ (1\,\vert\,1)\ (0\,\vert\,1) \equiv \text{aac}$	5
`e-v/v-e`	$(1\,\vert\,1)\ (0\,\vert\,1)\ (0\,\vert\,1) \equiv \text{acc}$	4
`v-v`	$(0\,\vert\,1)\ (0\,\vert\,1)\ (0\,\vert\,1) \equiv \text{ccc}$	3
`e-e(-t)`	$(1\,\vert\,0)\ (0\,\vert\,1)\ (0\,\vert\,1) \equiv \text{bcc}$	3

Table 5.2: This table shows the different types of contacts with the associated constraint classes and the resulting DoF for the non-constrained space, using the notations of [XJ01; MK97].

Each contact surface geometry relates to a set of natural constraints, which can be combined with further artificial constraints that are not bound to any physical relation, but are purely chosen to express a desired behavior. On the one hand, the constraints of a coupling together define the (virtual) contacting surface that can be represented in terms of PCs. On the other, using TFF or a similar approach, it is also possible to derive the set of constraints from the PCs, since one or more constraints can be used to represent the constrained directions imposed by the PCs. Table 5.2 forms the basis for the conversion between contact surface primitives and the DoFs that are targeted by the constraints.

2 The first element of the pair represents a contact surface associated with the robot, while the other one represents a surface on an object.

5.2.4 *Kinematic Chains and Joint Constraints*

Constraints can also be used to describe multi-body entities. A robot arm for instance is formed by a series of (single-body) entities (i.e. links) that are only allowed to move in a certain way. How the links are kinematically chained together depends on the used mechanism (i.e. joint). A joint can be considered analogous to a contact, which couples two or more[3] links. Prismatic and revolute are very common types of joints, defining the DoFs that the coupled links are allowed to move with respect to each other. Using Table 5.2, the different types of joints can be represented as motion constraints by translating the DoFs into constraint classes for contact surfaces. In addition to the general DoFs that can be used for motion, joints are constrained by mechanical limits that further restrict the motion in the defined DoFs. While most of the geometric and kinematic constraints are formulated in the Cartesian space, joint constraints are usually formulated in the joint space.

5.2.5 *Virtual Manipulator Constraints*

○ *a frame or body can only be controlled when connected to a source of forces*

In the physical world, an entity can only be moved through forces. If an entity is not able to produce the necessary forces to move itself, it needs to rely on other external forces. While natural constraints are realized by the physical environment, artificial ones can only be realized by a source of controllable forces. In case of a robot arm, the last link or EEF can only be controlled, because the individual joints represent natural constraints that can be artificially controlled. Hence, an artificial motion constraint that acts on a controllable entity (i.e. an EEF) can be realized. This in turn means that any entity that is in contact with a controllable entity, becomes controllable as well to a certain degree. The degree depends on the type of contact that is established. Following that idea, the task specification can be made object-centric, instead of being solely robot- or EEF-centric. An object-centric approach allows to specify a task independent of how the constraints are realized eventually [Sha+20]. Figure 5.5 shows a box that is tasked with moving towards the edge of the table to brush off

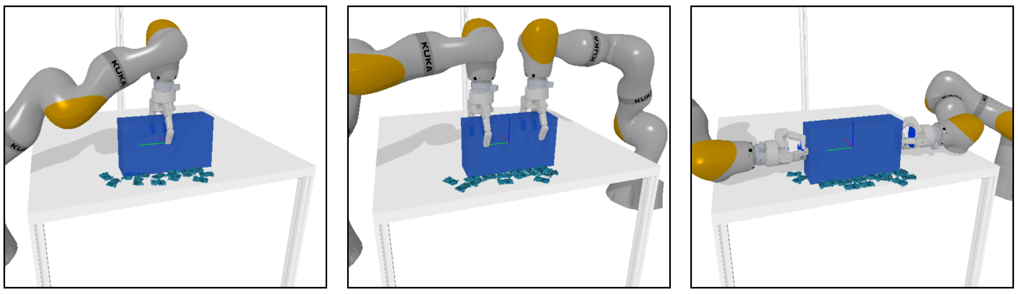

(a) Static Single Grasp (b) Static Dual Grasp (c) Dynamic Dual Grasp

Figure 5.5: Different ways to turn the box into a virtual manipulator in order to wipe the small objects off the table.

3 In case of a kinematic tree, which is used to represent a humanoid robot for instance.

the small objects. The box itself is not controllable. Only when a contact with a controllable entity is established, then the box becomes controllable and the artificial motion constraint for moving forward can be realized. In Figure 5.5a, the box is grasped by an actuated kinematic chain (i.e. a robot arm), turning the uncontrollable box into a Virtual Manipulator (VM) [LDC18; Has+05]. As it can be seen in Figure 5.5b, creating a VM is not limited to one kinematic chain, instead multiple chains can be combined as a closed kinematic tree as long as the internal forces are accounted for by employing a *force closure* or even a *form closure* using a suitable grasp. While the grasps in the first two images are considered rigid and fixed, the grasp in Figure 5.5c is very dynamic and relies on a constant application of forces to maintain the contact while moving.

The VM is created based on the *grasp matrix* $\mathbf{G}$ that relates the manipulated object twist to the contact twists of B manipulators [Deh+18]:

$$\mathbf{G} = \begin{bmatrix}\mathbf{G}_i \dots \mathbf{G}_B\end{bmatrix} \in \mathbb{R}^{6\times 6B}, \ \mathbf{G}_i = \begin{bmatrix}\mathbf{R}_i & \mathbf{0} \\ \mathbb{S}(\mathbf{r}_i) & \mathbf{R}_i\end{bmatrix} \in \mathbb{R}^{6\times 6}, \tag{5.5}$$

where $\mathbf{R}_i$ represents the rotation matrix of the ith contact frame. $\mathbf{r}_i$ is the distance between the ith contact position and the object's Center-of-Mass (CoM). $\mathbb{S}(\mathbf{r})$ is the skew-symmetric matrix. The geometry of the (virtually) manipulated object needs to be encoded into $\mathbf{G}$. To control the internal forces, as needed in Figure 5.5c, the nullspace projection of $\mathbf{G}$ can be used. The resulting contact wrench does not produce any net wrench, i.e. $\mathbf{G}\mathbf{F}_c = \mathbf{0}$. Since the motion of the box should not be affected by the forces that maintain the grasp, only the internal wrench is allowed to be controlled by

$$\mathbf{J}_{int} \in \mathbb{R}^{6B\times D} = \left(\mathbf{I} - \mathbf{G}^T(\mathbf{G}^+)^T\right) \begin{bmatrix}\mathbf{J}_1 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \mathbf{J}_B\end{bmatrix}, \tag{5.6}$$

where $\mathbf{J}_i$ represents the ith manipulator Jacobian and D the combined number of joints. For more detail, including the compensation for object dynamics, please refer to [Deh+18; Lin+18; Deh18].

5.3 CONTACT SITUATIONS AND TRANSITIONS

Contacts that happen at the same point in time, can be grouped together in a Contact Situation (CS). Such a situation represents one element in a high-level contact state space to describe an interval in time or rather a discrete state of the environment [XJ01]. While a CS can be modeled explicitly during the design phase of a system, it can also be formed implicitly during runtime by the set of contacts that exist at that point in time.

5.3.1 *Prioritizing Contacts and Constraints*

Within a single contact situations it is essential to define the importance of the individual constraints. For instance, the behavior of hyper-redundant robots, such as humanoids, in multi-contact situations, is generally defined by a multitude of constraints [Aud+14; PK08; PK05; Yun+99]. However, not all the constraints can be satisfied in every situation. Hence, through prioritization a set of constraints are defined that need to be satisfied at any cost, while others become optional. As shown in Figure 1.1l, the humanoid robot uses different contacts with the environment to support itself while raising a leg. In this case, the constraints related to a stable stance are more important than the (motion) constraints that achieve reaching a target position with the leg. Considering a simpler task, such as sanding a part, it is more important to ensure that the same force is exerted into the contact direction than it is to perfectly follow the sanding trajectory. Applying more or less pressure would result in uneven sanding, while deviations from the motion trajectory can be easily fixed.

There are numerous works on prioritization of control tasks to achieve a desired behavior. A very prominent one is the stack-of-tasks approach (see Section 2.2.2). It supports two types of prioritization, i.e. strict and soft, which are conceptually lifted to manage the prioritization of constraints: The idea of a strict prioritization is that lower priority constraints do not influence the execution of the higher level constraint. In contrast to that, a soft prioritization introduces a weighting between the constraints, which allows the constraints to influence each other, but also does not guarantee the correct execution of the constraints. Both, projection-based and QP-based control frameworks provide the means to realize the two prioritization mechanisms [DOA15; Hof+17a; Roc+15].

5.3.2 *Contact Transitions*

Several manipulation tasks require the establishing and breaking of contacts. In order to successfully accomplish such tasks, the transition from non-contact (free-space motion) to contact (constrained-space motion) and vice versa is mandatory [FBJ06; JC92]. This is why the physical interaction aspect of a task description contains one or more CSs. Thus, allowing the transitioning between different contacts and non-contact states, by traversing from one CS to another. Such a transition is realized by deactivating and activating sets of contacts together with the constraints that describe their couplings. Each CS can have guards that restrict or allow the traversal based on predefined conditions. Those guards can be uni- or bi-directional. The former type allows entering a CS (e.g., when a contact is established), while the latter actively triggers a transition to another CS (e.g., when a contact breaks). These conditions can be based on the internal high-level task coordination state or on sensor data.

○ *the separation into free and constrained spaces orthogonally decouples motions and forces, based on the duality between force and motion at a contact [KLB08]*

In order to realize a transition between contact states and situations, different approaches exist, covering the estimation and identification of the active contacts as well as for the detection of the transition between contact states.

In [Mir+18] a confidence-based approach is presented to transition between controllers upon contact in a discrete way:

$$u(x, x^e, \varphi) = \begin{cases} u_1(x, x^e, \varphi) & \text{if } P(\|\hat{x} - \hat{x}^e\| \leqslant \delta, \bar{\theta}) \leqslant \delta_p \\ u_2(x, x^e, \varphi) & \text{if } P(\|\hat{x} - \hat{x}^e\| \leqslant \delta, \bar{\theta}) > \delta_p \end{cases} , \tag{5.7}$$

where $u_1(\cdot)$ and $u_2(\cdot)$ are control controllers that are (de)activated based on the confidence δ_p for the distance between the system and the agent (e.g., the robots' EEF and the desired position). Here, each controller implements force and motion constraints to realize a CS.

A means to not only (de)activate controllers but enable a transition on the prioritization-level of the individual control tasks that form a controller, is offered by the dynamically consistent Generalized Hierarchical Control (DynGHC) [DS19] approach. DynGHC allows to continuously reprioritize stacks of control tasks and to even switch between strict and soft hierarchies. The prioritization of K control tasks are encoded in the matrix

each control task is assumed to produce a joint torque control signal

$$\boldsymbol{\Psi} = \begin{bmatrix} \alpha_{1,1} & & \alpha_{1,K} \\ & \ddots & \\ \alpha_{K,1} & & \alpha_{K,K} \end{bmatrix} \in \mathbb{R}^{K \times K} , \tag{5.8}$$

where the priority for each pair of control tasks $< i, j >$ is represented as a scalar value $\alpha_{i,j} \in \{0, 1\}$ for a strict or a soft $\alpha_{i,j} \in (0, 1)$ hierarchy. The entries $\alpha_{i,i} \in [0, 1]$ define the (de)activation of a task in the hierarchy. The priorities related to a control task $\boldsymbol{\tau}_i$ are rearranged into a diagonal matrix $\mathbf{A}_i$ that is used to create the shaped projector $\mathbf{N}_{\mathbf{A}_i}$. The final control law is given by

$$\boldsymbol{\tau} = \sum_{i=1}^{K} \mathbf{N}_{\mathbf{A}_i} \boldsymbol{\tau}_i . \tag{5.9}$$

By smoothly adjusting α, the prioritization of control tasks can be changed, which with regard to the satisfaction of constraints means that a smooth traversal between contact constraints and thus contact situations is possible.

5.4 CONCLUSION

In this chapter, a domain analysis was conducted towards answering RQ 3 "What are suitable abstractions to model the CI in tasks?". First, the aspects that are essential to a task description for CI were investigated and the need for an explicit and in-depth consideration of the compliant interaction was motivated. Second, the different kinds of abstractions used to describe individual parts of the compliant interactions with the environment that can be found in the literature are investigated. The outcome of this chapter is the insight that a suitable abstraction for CIs is based on the physical entities that enter a compliant interaction through the natural interface of contacts. Contacts impose force and motion constraints, expressed in the geometry, kinematics, and dynamics domain, on the motion behavior exhibited by a robot. By choosing an abstraction that is grounded in the task-space, the description of the CI becomes intuitive and explainable. Grouping the contacts into contact situations, allows to define the transitions between situations that are essential to establishing and breaking a contact. A prioritization of the constraints is used to impose a desired (artificial) behavior. The same can be done by the other task-related aspects, such as trajectories and coordination, which further constrain the interaction in an artificial way to produce a desired behavior. Hence, it is important that the description of the CI part of the task is composable with the other aspects of the task description. The insights of this chapter are consolidated as a conceptual base for the domain-specific modeling of CI tasks, presented in Chapter 6.

“ *Abstraction is one of the greatest visionary tools ever invented by human beings to imagine, decipher, and depict the world.* ”

— Jerry Saltz

6 CI LANGUAGE DESIGN AND (META-)MODELING

This chapter presents the transfer of the insights from the domain analysis into a DSL that allows the modeling of CI tasks as an extension of CoSiMA. This chapter is based on [WDW20; Deh+ss].

As stated in Chapter 1, the realization of emerging robotics applications that involve CIs, becomes feasible with the rise of hardware that leverages precise torque control and built-in force-torque sensors, capable of entering a compliant interaction (e.g., COBOTs). Correlating with this rise is the continuous search for software development approaches for designing executable task descriptions for the emerging contact-rich applications. In particular this is shown in Figure 6.1 by the sudden but sustained increase in DSL publications, providing MDE approaches related to the robotics task and skill modeling. Choosing modeling over programming is proven to cope with the challenges of similarly complex domains, such as automotive and aerospace [AGD18].

>_ *compliant interaction*

This chapter presents an answer to RQ 3 in form of a DSL as an extension to CoSiMA, which allows the modeling of robotics tasks with particular focus on CI. To this end, the insights gained in Chapter 5 are transformed into concepts and meta-models for the modeling of CI tasks based on contact situations in which interactions occur through a set of contacts. To model the effect of the contacts on the behavior of the contacting physical entities, different kinds of force and motion constraints are modeled. The constraints are expressed in the geometry, kinematics, or dynamics domain. In contrast to other publications, this approach makes no distinction between tasks, skills, or skill primitives, since the interaction of the scenario is abstracted by a common concept: the involved contacts. Thus, creating a common representation to express, for example, a peg-in-hole task or a screwing motion. Despite the focus on CI, other task-related aspects, such as coordination or trajectories, further constrain the behavior of the robot. Hence, the approach presented in this chapter, must ensure the composability with those aspects on a concept and language (i.e. DSL) level.

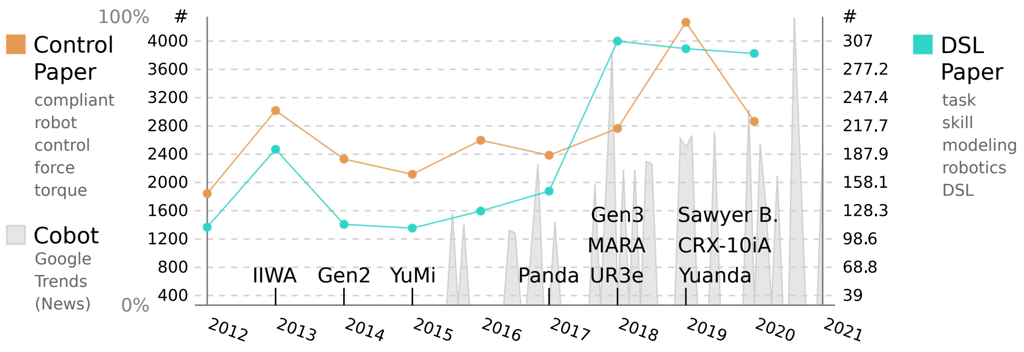


Figure 6.1: Keyword-based trend analysis of control frameworks and task-related DSL publications. The two axes show the number of publications, collected using [Dim]. The used keywords are presented in the legend. A Google-based [Goo] interest search for the keyword *cobot* is overlaid in gray, which spans the range from zero to the maximum recorded interest in percent. Further, the release dates of prominent COBOTs are added.

6.1 RELATED WORK

Several DSLs in the robotics domain were developed over the last years, which utilize different levels of abstraction. Approaches with a high level of abstraction, usually choose a skill-based representation to increase the ease of use and to lower the required expert knowledge. One of those approaches is LightRocks [Tho+13], a DSL for skill-based robot programming, which uses Statecharts to coordinate the skills depending on pre and post conditions. The authors state that LightRocks allows generating code "while hiding controller specific implementation details". This perfectly highlights the wide-spread problem of DSLs that only allow modeling on a high abstraction level. Here, the abstraction is raised so far that even the lowest supported level, represented by LightRocks' Elemental Actions, hides the actual implementation of the robot's behavior in black boxes (e.g., *Move Close*, *MoveTo Grasp*, and *Move Back*). Hiding e.g. robot-specific aspects, such as how the redundancy is handled, in favor of the usability increases the gap between the envisioned and the resulting behavior and limits the general applicability. Another downside of a black box approach is being limited to very trivial composition mechanisms, which is why the majority of skill-based solutions only support the sequencing of skills. A solution to the problem of composition and orchestration is proposed by Nägele et al. [Näg+18; Näg+19]. The core concept of their DSL is built on a prototype-based inheritance for skills, which breaks a skill down to a constraint-based representation. This representation enables more sophisticated composition patterns, such as the parallel execution and prioritization of skills.

The concept of describing a desired robotics task in terms of constraints on relative motions between points of interest is adopted by different approaches on the lower end of the spectrum of abstraction. Klotzbücher et al. [Klo+11] present a DSL that allows to define low level motion and force constraints, which are based on TFF. The constraint models are platform-agnostic.

Task Frame Formalism

Multi-staged model transformations are used to compose the constraints with hardware- and software-specific aspects. In contrast to LightRocks, this enables the explicit switching of the used robot, causing minimal changes to the constraint models. Aertbeliën et al. [Ad14] introduce a task specification DSL (eTaSL), which lifts the general idea of TFF to *feature variables* and represents geometric operations between rigid bodies (i.e. scalar, vectors, transformation matrices, etc.) as expression graphs on the geometry and kinematics level. While geometric relations between objects can be modeled, eTaSL does not explicitly consider the robot, the task, and the environment, since all are essentially modeled as feature variables, expression graphs, and position or velocity level constraints. Expression graphs are used to model the error function of a constraint explicitly. For task space constraints however, the resulting behavior is implicitly defined by a first order linear system and only represented on the model level by a time constant. eTaSL can model equality as well as inequality constraints, which can be used to describe joint limits. Bartels et al. [BKB13] present a DSL to model geometric position level constraints between objects. Compared to eTaSL, a higher level of abstraction is used by explicitly modeling and visualizing the robot and the geometric environment. While the static relations between objects are modeled, the dynamic realization (i.e. how a constraint is realized) is not considered on the model level. Vanthienen et al. [Van+13b] introduce a DSL for iTaSC [Smi+08], whose level of abstraction lies in between the previous approaches. The DSL explicitly models the robots and the geometric objects in the world. Additionally, it uses feature coordinates to constrain the relative motions between the objects, which is similar to eTaSL. However, the constraints can be prioritized on multiple levels of prioritization, while eTaSL only supports up to two. The satisfaction of constraints and thus the definition of the resulting behavior is implicitly referenced by the selected controller. Compared to eTaSL, different controllers for the realization can be chosen on the model level, but their semantic relation to the resulting behavior is not made explicit. For instance, joint positions are hardcoded in the controller without a link to the available joints in the virtual kinematic chain (VKC) that is used.

Table 6.1 shows a comparison of the mentioned approaches in terms of different aspects, which act as inspiration for the CI task modeling in CoSiMA. It can be seen that Vanthienen et al. [Van+13b] already address most of the aspects, but lack a complete formalization of the meta-model as well as suitable tool support. Even though the approach explicitly considers the physical environment, including the robot, it does not consider contacts or contact surfaces. The same holds true for almost all other related work. Further, it can be seen that the approaches are mainly designed to support single, or individually controlled robots. To the best of my knowledge, none of the mentioned approaches support dynamically closed VKCs and are limited to kinematic interaction. A dynamically closed VKC allows controlling e.g., two robots that create a closed VKC through maintaining a rigid or soft contact, forming a single robot, where even dynamics level constraints can be imposed on the combined robots.

DSL	CoSiMA	[Klo+11]	[Tho+13]	[Van+13b]	[BKB13]	[Ad14]	[Näg+19]
Explicit Environment	✓	✗	✗	✓	✓	✗	✗
Separated Interaction	✓	✗	✗	✓	✗	✓	✗
Prioritization Levels	Multiple	✗	✗	Multiple	✗	1-(2)	Multiple
Constraint Behavior	✓	✗	✗	✓	✗	✓	✗
Task Transitions	✓	✓	✓	✓	✗	✗	✓
Formalization	MM	(MM)	Math	(MM)	✗	✗	✗
Tool Support	✓	✗	✓	✗	✓	✗	✗
Abstraction Level	1 | 2 |	1 | |	| | 3	1 | 2 |	| 2 |	1 | |	| | 3
Dynamically Closed VKC	✓	✗	✗	✗	✗	✗	✗
Contacts and Contact Surfaces	✓	✗	✗	✗	✗	✗	✗

Table 6.1: Comparison of CoSiMA's CI task modeling with relevant and related DSLs. The compared aspects refer to the modeling and not to the functional capabilities of the underlying execution frameworks. A DSL is formalized via a meta-model **MM** or mathematically **Math**. Presentations of only small meta-model excerpts are marked with **(MM)**. Abstraction levels range from low-level control (1), over behavior constraints (2), to high-level actions (3).

6.2 MODELING CONTACT-BASED COMPLIANT INTERACTIONS

In this section, I introduce the CI task modeling with CoSiMA along the window wiping example shown in Figure 3.4 (right). The aim of the wiping task is to establish and maintain a contact with the window, while following a predefined trajectory to clean the window's glass.

Every necessary modeling step is presented using the relevant concrete syntax and the underlying concepts in form of meta-model excerpts. The general idea behind this approach is to model the physical and virtual entities in the environment and to explicitly constrain their interaction using contact constraints (i.e. `ICIConstraint`s) to form discrete `ContactSituation`s that can be transitioned between in a contact state graph that is decoupled from, but loosely composable with, the other task-related aspects.

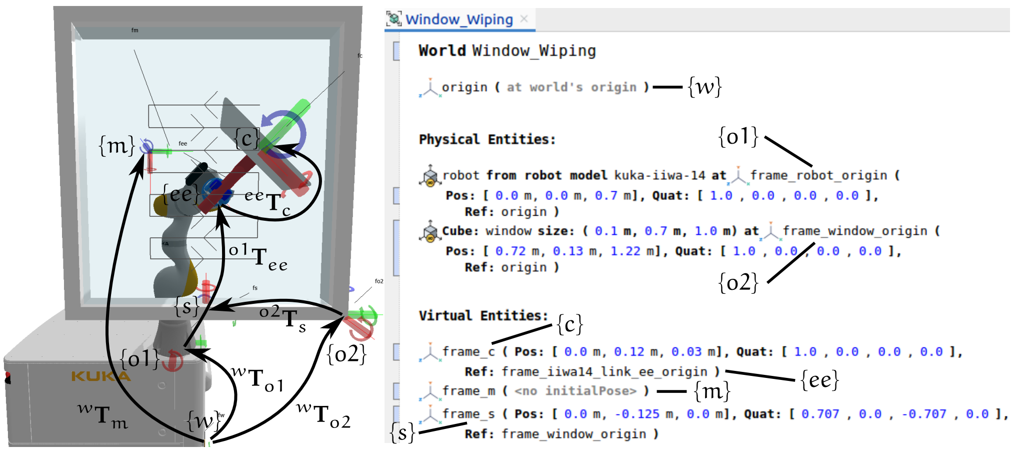
implementation concepts of a constraint and a contact situation, see Figure 6.14

6.2.1 *Physical Environment Model*

The first step towards defining a task description that models a desired robotic behavior in (physical) contact situations, requires the specification of the relevant physical entities (i.e. bodies) and virtual points of interest (i.e. frames). While bodies provide geometric, kinematic, as well as dynamic information, the compliant interaction is modeled on the frames.

Figure 6.2 shows the specification of an environmental state using the `World` concept. In this example, the `World` contains the window as a single `Body`. The robot is modeled using a `MultiBodyFromRobotModel`, which references a `Robot-Model` defined with concepts from the *Kinematics Dynamics DSL*. It includes the mobile platform, the robot manipulator, as well as the rigidly attached window cleaner. The virtual entities in the `World` model, are chosen to represent key points for the actual task. The `NamedFrame` $\underline{\{m\}}$ defines a frame, which is directly expressed in the world frame $\underline{\{w\}}$ to be externally controlled to realize

in the text, frames are marked with an underline $\underline{\{\cdot\}}$ to avoid confusion, see Notation on page xv



```
World Window_Wiping

origin ( at world's origin )

Physical Entities:

robot from robot model kuka-iiwa-14 at frame_robot_origin (
  Pos: [ 0.0 m, 0.0 m, 0.7 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0 ],
    Ref: origin )
Cube: window size: ( 0.1 m, 0.7 m, 1.0 m) at frame_window_origin (
  Pos: [ 0.72 m, 0.13 m, 1.22 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0 ],
    Ref: origin )

Virtual Entities:

frame_c ( Pos: [ 0.0 m, 0.12 m, 0.03 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0 ],
    Ref: frame_iiwa14_link_ee_origin )
frame_m ( <no initialPose> )
frame_s ( Pos: [ 0.0 m, -0.125 m, 0.0 m], Quat: [ 0.707 , 0.0 , -0.707 , 0.0 ],
    Ref: frame_window_origin )
```

Figure 6.2: Environment model with physical and virtual entities (i.e. objects and frames). A screenshot (right) of the model in MPS, and of the objects and frames (left), directly generated from the model into the simulator is shown. For readability, frame names and transformation arrows are overlaid. The line on the window represents a commanded motion trajectory.

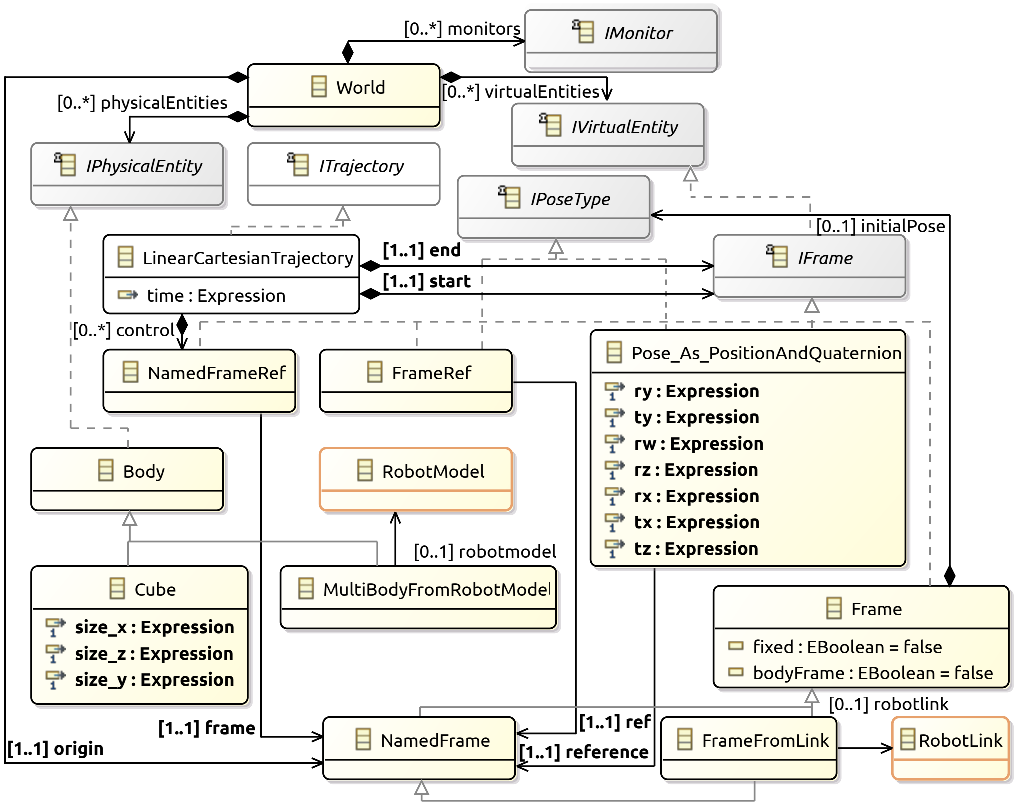


Figure 6.3: Excerpt of the *World DSL* meta-model that allows the modeling of physical and virtual entities (see Figure 6.2). The core concepts are `Frame` and `Body`. Concepts from the *Kinematics Dynamics DSL* (orange) are reused by reference. Other aspects of a task description, i.e. `ITrajectory` (white) are defined based on the same core concepts (i.e. `Frame`).

a trajectory. The NamedFrame {c} represents the compliance frame at which the contact with the surface of the window will be established and with respect to which the motion of the robot will be controlled. The frame is expressed with reference to the EEF link (i.e. FrameFromLink) of the chosen robot (i.e. RobotLink). All the relevant concepts to model the physical and virtual entities of an environmental world state can be seen in Figure 6.3.

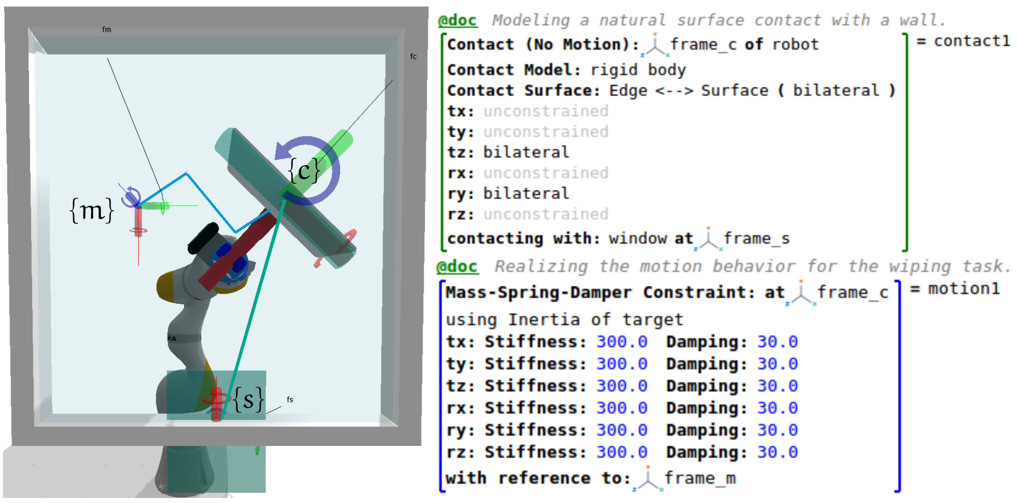
Those concepts act as an extension point for more sophisticated concepts that involve multiple or completely new domains. The MultiBodyFromRobotModel for instance extends the Body concept, to allow the composition with the *Kinematics Dynamics DSL* by referencing a RobotModel concept. Since the *Kinematics Dynamics DSL* is based on the URDF formalism, it offers the possibility to relate single rigid bodies to one another by introducing different kinds of RobotJoints. Thereby, allowing to model multi-body entities such as a robot manipulator in form of KinematicChains, including joint limits.

6.2.2 *Modeling Contacts and Constraints*

ICIConstraint is the implementation concept from which all constraint concepts are derived, see Figure 6.5.

the type is defined using the CType concept

To define the interaction between physical or virtual entities, different kinds of constraints (i.e. derived from the ICIConstraint) are used to restrict their relative motion (see Figure 6.4). In the case of defining the interaction between two physical bodies in a KinematicChain, a robot joint represents one or more natural constraints that act on features (e.g., frames) defined on the involved bodies (e.g., robot links) (see Section 5.2.4). CoSiMA uses a common representation based on artificial and natural ICIConstraints to model the compliant interaction between entities (see Figure 6.5). To this end, the following three different types of constraints can be arbitrarily composed to express an interaction.



```
@doc Modeling a natural surface contact with a wall.
Contact (No Motion): frame_c of robot                    = contact1
Contact Model: rigid body
Contact Surface: Edge <--> Surface ( bilateral )
tx: unconstrained
ty: unconstrained
tz: bilateral
rx: unconstrained
ry: bilateral
rz: unconstrained
contacting with: window at frame_s
@doc Realizing the motion behavior for the wiping task.
Mass-Spring-Damper Constraint: at frame_c          = motion1
using Inertia of target
tx: Stiffness: 300.0  Damping: 30.0
ty: Stiffness: 300.0  Damping: 30.0
tz: Stiffness: 300.0  Damping: 30.0
rx: Stiffness: 300.0  Damping: 30.0
ry: Stiffness: 300.0  Damping: 30.0
rz: Stiffness: 300.0  Damping: 30.0
with reference to: frame_m
```

Figure 6.4: The screenshot of the DSL (right) shows a contact constraint and a mass-spring-damper constraint acting on the compliance frame {c}, describing the compliant interaction towards target frames {s} at the window surface and {m} for the tracking of a desired motion trajectory. The screenshot of the simulator shows the visualization of the specified constraints (left).

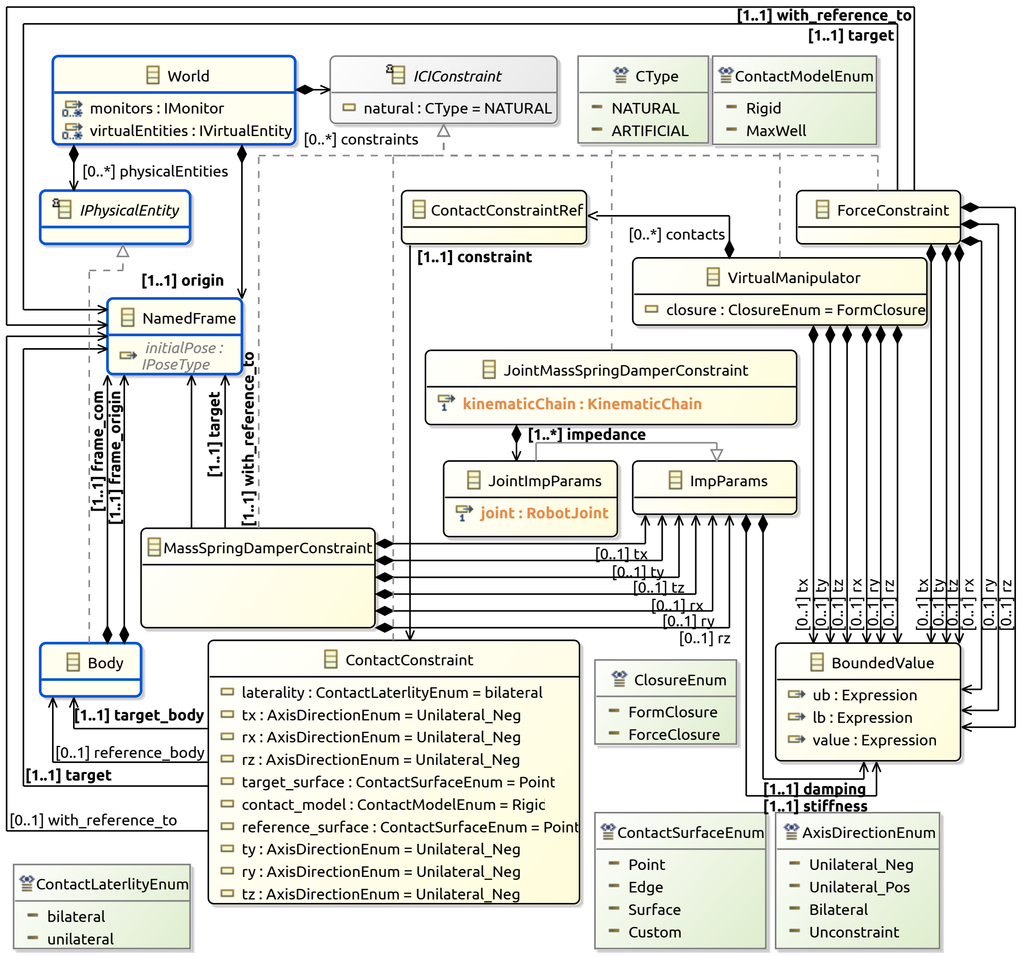


Figure 6.5: Excerpt of the *Compliant Interaction DSL* meta model, showing the constraint concepts and their link to the concepts `Frame` and `Body`.

6.2.2.1 *Contact Constraint*

Based on the insights from the domain analysis a contact is modeled between two `Body`s (see Section 5.2.2.1), where the combination of the target and reference surface defines the `ContactConstraint` (see Section 5.2.3). The constraint represents the restrictions imposed by the surface on the individual translational and rotational DoFs. Table 5.2 realizes a conversion between the DoFs and principal contacts (PCs). PCs increase the explainability of the contact surface by adding a semantic meaning to describe the contact constraint based on combinations of **v**ertex, **e**dge, and **f**ace primitives. Therefore, providing a clear interpretation of the DoFs in form of well-known mechanisms, such as a ball joint, a slider, or a hinge. Further restrictions can be expressed over the individual DoF to give the user more control. This may also include combinations that cannot be interpreted by a PC. Additionally, a contact can impose unilateral and bilateral constraints. This choice however, has a direct impact on the set of software control frameworks that are capable of realizing such a constraint.

○ *presented in Section 5.2.3*

In the wiping example, a contact is established between the compliance `Frame` {c} of the window cleaner and the `Frame` {s} at the window's glass, during the wiping action. As it can be seen in Figure 6.4, the contact constraint uses a rigid contact model, constraining the motion bilaterally using the PC (`e-f`). This prevents any translational motion in the Z (tz) direction and any rotational motion in the Y (ry) direction w.r.t. the surface of the glass. This also means, that wrenches can be applied in the constrained directions. For instance, without any artificially applied torques in Y, the window cleaner would adapt to a potential tilting of the glass surface.

6.2.2.2 *Mass Spring Damper Constraint*

The second type of constraint are motion constraints such as the `MassSpring-DamperConstraint`. These kinds of constraints are usually artificial, e.g., describing the compliant tracking of a trajectory. Depending on the respective space in which such a constraint is expressed, a `MassSpringDamperConstraint` concept or a `JointMassSpringDamperConstraint` concept can be used. In the first case, impedance parameters are defined for one or more relative Cartesian motion directions between the involved `Frame`s. In the second case, the constraint is expressed in the joint space instead of the Cartesian space. This is done, because it is more intuitive and widely adopted than to express joint constraints or limits in the Cartesian space.

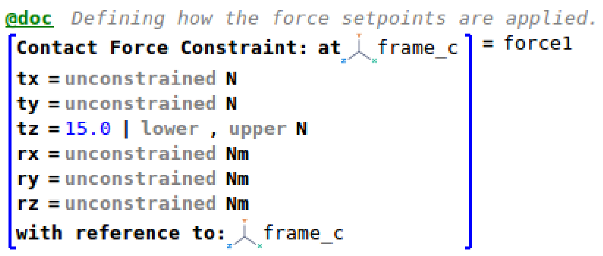
mass-spring-damper <

In case of the wiping example, a MSD constraint is used that connects the motion `Frame` {m} to the compliance `Frame` {c}, imposing an impedance behavior with the equilibrium point at {m} (see Figure 6.4). Hence, a coupling is created that allows the tracking of a motion guided by {m}. This motion can for instance be provided by a trajectory generator. Further information regarding the behavior of the constraint is described in the domain analysis (see Section 5.2.2.3).

6.2.2.3 *Force Constraint*

An artificial contact `ForceConstraint` can be used in a direction in which a motion is not possible. This is usually the case in contact situations. Such a constraint can be used in addition to a `ContactConstraint` to apply a wrench in one or more Cartesian DoF. The wrench is applied at the target `Frame` expressed in a reference `Frame`. Similar to a MSD constraint, the to be constrained DoFs can be individually chosen. The `BoundedValue` concept is used to offer the possibility to specify lower and upper bounds. This way, inequality and unilateral constraints can be modeled. Using the `ForceConstraint` in addition to an artificial `ContactConstraint` enables the specification of a totally compliant behavior as described in Section 5.2.2.2. A single `ForceConstraint` can only achieve total compliance if no other motion constraint is present.

Figure 6.6 shows the artificial contact `ForceConstraint` that is used in addition to the other constraints in the wiping example to enforce a contact force in the Z (tz) direction of the `Frame` {c}, which is expressed in the same frame.

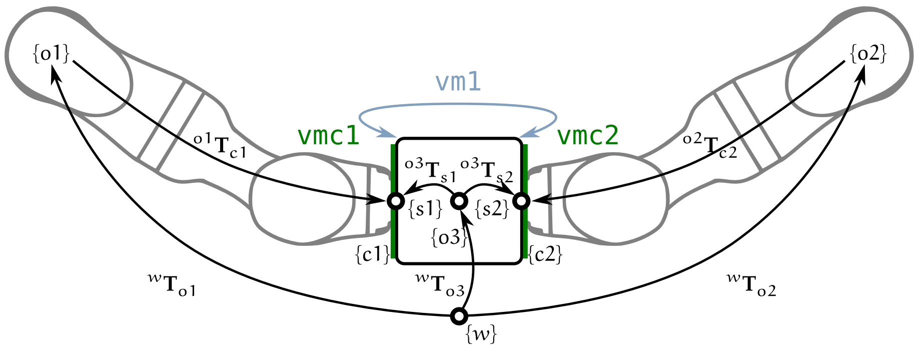


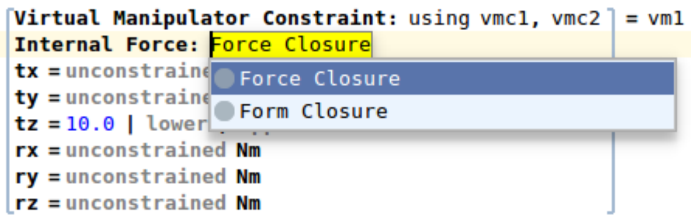
```
@doc Defining how the force setpoints are applied.
Contact Force Constraint: at frame_c  = force1
tx = unconstrained N
ty = unconstrained N
tz = 15.0 | lower , upper N
rx = unconstrained Nm
ry = unconstrained Nm
rz = unconstrained Nm
with reference to: frame_c
```

Figure 6.6: The screenshot of the DSL shows a force constraint that constrains the force exerted into the contact normal direction.

6.2.3 *Virtual Manipulator Constraint with Internal Wrench Handling*

Natural constraints represent physically existing relationships, while artificially imposed constraints need a source of controllable forces to be realized. As discussed in Section 5.2.5, a `Body` can only be controlled with respect to a `Frame`, if it is in contact with another controllable `Body` to ensure the transmission of forces. A robot's EEF link, for instance, can only be controlled if it is part of a controllable kinematic chain, because the EEF itself does not contain any controllable sources of forces. The DoFs with which the body can be controlled depends on the type of contact. In CoSiMA, everything that is turned into a controllable body via a non-rigid attachment to another controllable body is called a Virtual Manipulator (VM). This includes grasped or pushed bodies for instance. Since the EEF is rigidly connected through a physical joint to the rest of the robot's kinematic chain, it becomes a controllable physical manipulator that can be used to turn previously uncontrollable bodies into controllable ones. For instance, a body that is pushed by an EEF, can create a virtual manipulator as long as the contact is maintained. However, the available DoFs of the manipulator depend on the kind of contact and are usually unilateral, since the robot cannot pull the pushed body. To overcome this, multiple controllable bodies can be used to create contacts with an object, such as a box (see Figure 6.7), to form a closed kinematic chain. Hence, turning the box into a `VirtualManipulator`. Again, the box can only be controlled as long as the referenced `ContactConstraint`s forming the chain are maintained. Using a form closure avoids the need to maintain the contact through artificial forces. In contrast, a force closure requires the artificial application of internal wrenches. As it can be seen in Figure 6.7, a force closure is created due to the controlled EEFs squeezing the box. This means that a certain force needs to be applied into the contact to maintain the closed kinematic chain. Physically closing a kinematic chain offers the possibility to treat the involved robots as one single robot and also enables the distribution of the necessary torques (to move the box) over both robots on the dynamics level. Modeling a virtual manipulator in the wiping example is not necessary, since the window cleaner is screwed to the EEF. Instead, an example using a VM is presented in Chapter 8. The meta-model for modeling the CI constraints is shown in Figure 6.5.





```
Virtual Manipulator Constraint: using vmc1, vmc2 = vm1
Internal Force: Force Closure
tx =unconstraine
ty =unconstraine
tz =10.0 | lower
rx =unconstrained Nm
ry =unconstrained Nm
rz =unconstrained Nm
```

Figure 6.7: The definition of a `VirtualManipulator` is shown. Depending on the involved contact constraints (e.g., *vmc*1 and *vmc*2) and the type of closure (i.e. grasp), the internal force needs to be artificially controlled to maintain the virtual kinematic chain on the dynamics level.

6.2.4 *Modeling Constraint Prioritization Structures*

The prioritization of constraints is necessary, since a robot cannot guarantee an arbitrary amount of constraints in any given situation. In fact, this strongly depends on the constraints, the (sub)space they are acting on, the robot, and the current state of the world. In CoSiMA however, overconstraining the robot's task and joint space is not prevented, since conflicting constraints may be used to create (desirable) interferences. In order to ensure the satisfaction of essential constraints, such as maintaining a contact or keeping the center-of-mass of a robot inside the support polygon, constraints need to be prioritized. For this, the redundancy of the controlled robot can be exploited. As mentioned in Section 5.3.1, CoSiMA supports multiple levels of strict prioritization as well as (intra-level) soft prioritization.

For each `ContactSituation`, a `CompliantControlArchitecture`[1] concept is instantiated, which contains a prioritization structure (PS) per `KinematicChain` used in the respective CS. The `ICIConstraint`s of a `ContactSituation` are separated by the kinematic chain and are reflected in the associated PS.

Contact Situation ⎯<

A PS forms a tree that comprises a set of `ControlTask`s (vertices) and `ICCARelation`s (edges) to impose different types of prioritization. The beginning of a PS is marked by an `Entry_Relation` (see Figure 6.8a), which provides the context for the control tasks and relations in terms of the controllable joints represented by a `KinematicChain` K and the `Frame`s that can be targeted by the respective type of control task. Figure 6.8 shows the concrete syntax of the supported `ICCAEntry` concepts.

1 `CompliantControlArchitecture` is abbreviated as `CCA`, e.g., `InterfaceCCARelations` and `CompliantInteraction` as `CI`, e.g., `InterfaceCompliantInteractionConstraint`.

A `CompliantControlArchitecture` can be transformed into an executable robotics software architecture, for a chosen software control framework (see Chapter 7). The prioritization concepts are presented below and the associated meta-model is shown in Figure 6.9.

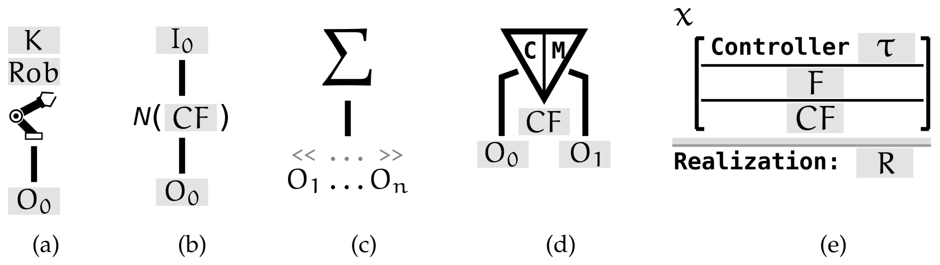


Figure 6.8: Concrete syntax of the concepts to model a prioritization structure: (a) `Entry_Relation`, (b) `Nullspace_Relation`, (c) `WeightedSum_Relation`, (d) `MotionForceSubSpace_Relation`, and (e) `ControlTask`. I and O denote links to other `ICCAEntry`s. CF, K, τ, F, and R denote references to the controllable target frame, the kinematic chain, the identifier of the control task, the control formalism, and the platform-dependent realization of a formalism respectively.

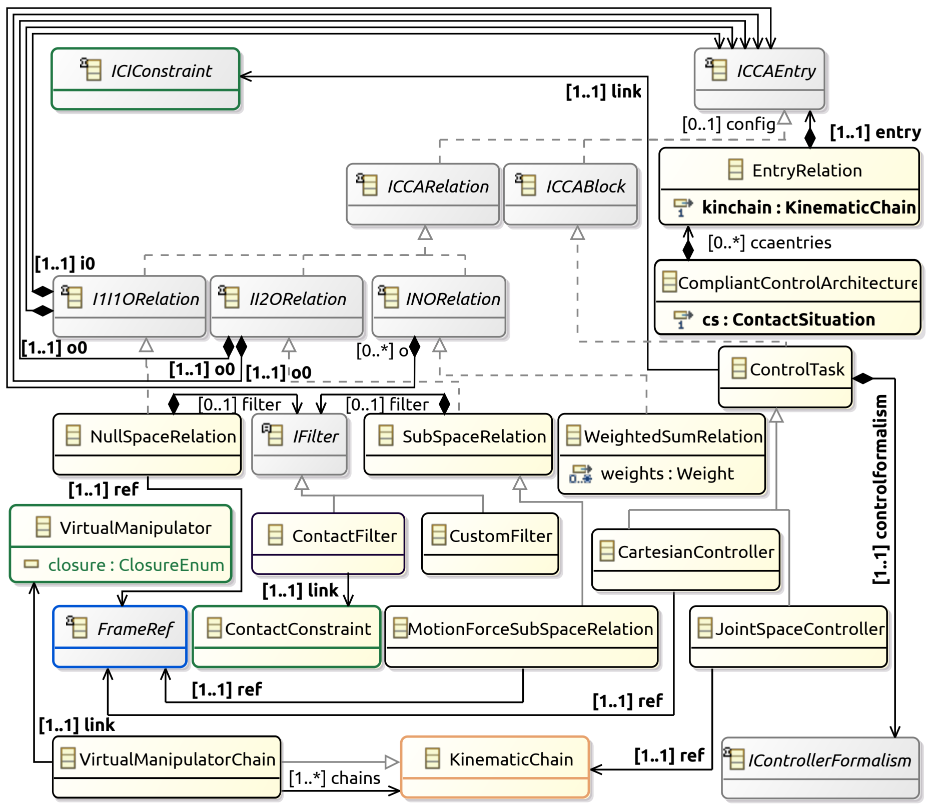


Figure 6.9: Excerpt of the *Compliant Interaction DSL* meta-model containing the prioritization concepts. Each `ControlTask` references an `ICIConstraint` that it realizes via a suitable control formalism. An exception is the `ContactConstraint` which is realized via a subspace relation, providing a `ContactFilter`. The color coding indicates that this meta-model reuses concepts from other meta-models.

6.2.4.1 *Control Task*

A ControlTask (see Figure 6.8e) represents a control formalism F (IController-Formalism) applied to a referenced controllable Frame CF; both depend on the space, the ControlTask is expressed in: task space (CartesianController) or joint space (JointSpaceController). In the case of a joint space control task, the controllable entity (CF) represents a subset of joints from the Kinematic-Chain, which is given by the context of the encapsulating PS, instead of a frame. The used formalism itself is not directly tied to a specific implementation of a control equation. Instead, each formalism can be realized by a set of control implementations. The set of suitable implementations depends on the chosen software control framework that is globally annotated to a CI model (see Figure 6.16). Note that all implementations associated with the same formalism produce a control behavior that conforms to the general definition of the formalism (see Figure 6.10).

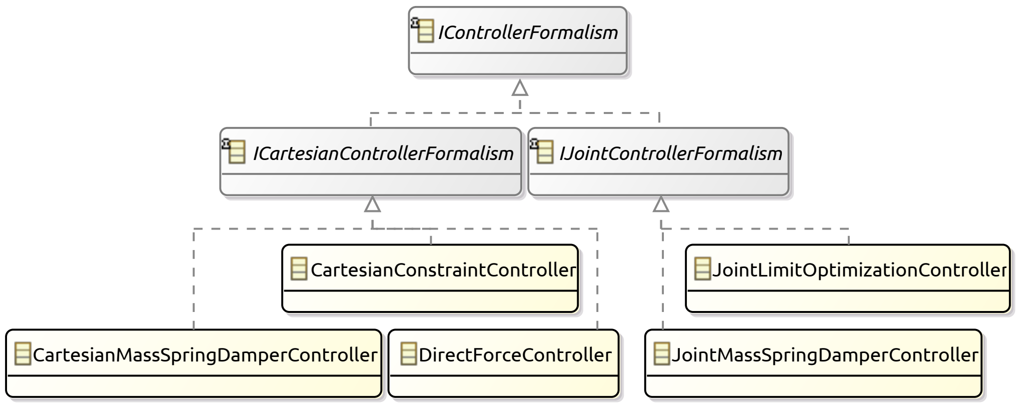

IControllerFormalism
ICartesianControllerFormalism
IJointControllerFormalism
CartesianConstraintController
JointLimitOptimizationController
CartesianMassSpringDamperController
DirectForceController
JointMassSpringDamperController

Figure 6.10: This meta-model excerpt shows the supported controller formalisms. Depending on the abstract concepts they are derived from, it is determined which kind of constraint can be represented by the formalism.

The link between a control task and the constraint it has to realize is established via a direct reference. Each ControlTask can reference one constraint of any type, except for a ContactConstraint, which is a special case explained in Section 6.2.4.4. In fact, a control task maps the referenced constraint to a suitable formalism for realization and embeds the constraint into a prioritization structure. The referenced constraint can then be exploited by the M2M transformations of the synthesis in Chapter 7 to parameterize the associated formalism. Thereby, also configuring the software implementation of the actual control components.

model-to-model ⊸

6.2.4.2 *Strict Prioritization*

The Nullspace_Relation concept (see Figure 6.8b) is used to model a strict prioritization between two ControlTasks, based on the literature presented in Section 2.2.2. Let e be an instance of this concept. Semantically the Nullspace_-Relation enforces that the control signal of the ControlTask $e.O_0$ is not interfering with the signal of $e.I_0$ if the target frame of $e.I_0$ is also chosen for the relation itself: $e.\mathrm{CF} = e.\mathrm{I}_0.\mathrm{CF}$. That also means that the second task can only be accurately performed if enough DoF are available. Otherwise, it is performed at best, without interfering with the primary task. A `CustomFilter` can be used to further shape the nullspace. However, in contrast to the `MotionForceSubSpace_Relation`, the `Nullspace_Relation` does not enforce the active control of the created subspace. A similar specification is possible in the Math of Task (MoT) [Hof+17a] DSL. In general the realization of the relation is independent of a specific formulation. Both, a PIDC- and a QP-based realization is able to support this kind of prioritization.

6.2.4.3 *Soft Prioritization*

A soft prioritization is represented by the `WeightedSum_Relation` concept e (see Figure 6.8c). Each `ControlTask` $e.\mathrm{O}_i$ linked by this relation is on the same prioritization level (see Section 2.2.2). Scalar weighting factors $e.w_i$ are used to superimpose the weighted set of control tasks $e.\mathrm{O}_i$ [BK11; Mor+13a]. All weighting factors in the same relation need to sum up to 1. Analogous to the `Nullspace_Relation`, the `WeightedSum_Relation` is formalism-agnostic and can be realized by PIDC- and QP-based frameworks alike. In addition to its role as a prioritization relation, the `WeightedSum_Relation` can also be structurally referred to as a control task, thus allowing the same positioning in the tree structure of a PS (as a vertex).

6.2.4.4 *Motion-Force Subspace Relation*

The `MotionForceSubSpace_Relation` concept e (see Figure 6.8d) divides the task space into separate subspaces (e.g., constrained “force” and unconstrained “motion” space), to avoid the interference of control tasks. As described in Section 2.2.3, this is commonly used to represent a rigid body contact at which the unconstrained space refers to the motion subspace, while the constrained space represents the contact wrench space. In the first space, movement is possible while the exertion of forces is not. Whereas in the latter space, it is the other way around. In contrast to the `Nullspace_Relation` that uses a `CustomFilter`, this relation uses a `ContactFilter` to split the space into two subspaces, each individually controlled by a `ControlTask`. A `ContactFilter` further establishes a link to the `ContactConstraint` that is realized by this PS (sub)tree. The default behavior of this relation prioritizes the wrench space ($e.\mathrm{O}_0$) strictly higher than the motion space ($e.\mathrm{O}_1$). The rationale behind this is that maintaining a contact is often more important than the accurate execution of a motion trajectory; especially for polishing tasks [ZB20]. This also means that each `ContactConstraint` requires the instantiation of a respective `MotionForceSubSpace_Relation`. In general the separation of the subspaces can be freely chosen[2] at any controllable frame.

A `MotionForceSubSpace_Relation` is a special `ICCARelation`, that divides the task space and realizes a behavior based on the chosen contact model. The resulting subspaces can be further constrained by `ICIConstraint`s and other

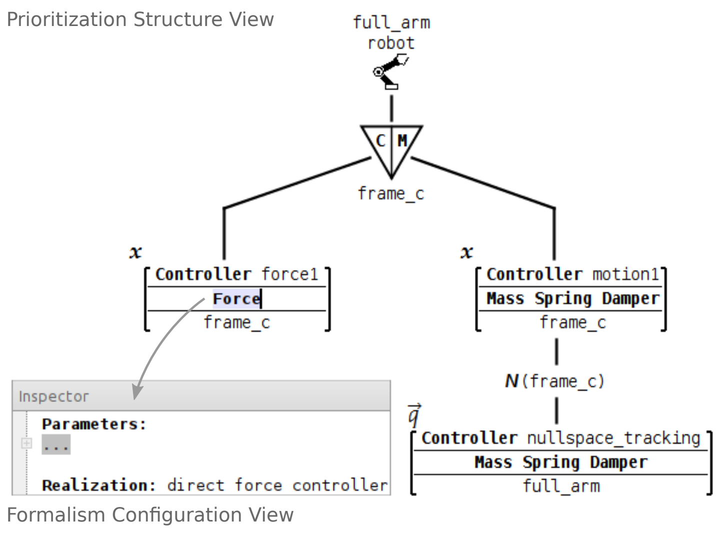
2 as long as the property of orthogonality is ensured.

ICCARelations. In addition to providing a prioritization structure for other control tasks, the MotionForceSubSpace_Relation also takes on the role of a control task itself. This is similar to a WeightedSum_Relation, which is also not conceptually derived from a ControlTask, but which is positioned as a control task (i.e. vertex) in the PS.

6.2.4.5 *Composing Prioritization Relations*

Having a look at the PS for the contact situations of the wiping example during the wiping motion, shown in Figure 6.11, one can see that an Entry_Relation is used as root node, referencing the same robot instance as in the World model. During the wiping motion there is a contact established that is modeled by a MotionForceSubSpace_Relation to separate the control of the contact forces from the wiping motion. Since for this wiping task, it is more important to maintain the contact than to accurately follow the trajectory, the default behavior of this relation is perfectly suited. The contact ForceConstraint used in the example is represented by the ControlTask ("force1") acting on the frame {c}. The chosen formalism is of the type DirectForceController, which describes a feed forward force. In the motion space, a ControlTask ("motion1") that uses a CartesianMassSpringDamperController formalism is followed by a joint space ControlTask ("nullspace_tracking") that operates in the nullspace of the Cartesian control task. While the Cartesian control task acts on the frame {c}, the joint space control task acts on the kinematic chain of the selected robot. Using the respective FormalismMappings (see Figure 6.16), the control tasks have an explicit link to the constraints (i.e. ICIConstraints) that define the control task configuration (e.g., impedance parameters, controlled DoF, etc.).

establishing this link is essential for the synthesis in Chapter 7



Parameters:
...
Realization: direct force controller

Figure 6.11: Screenshot of the concrete graphical syntax for the prioritization of the control tasks, which are linked to the specified constraints in Figure 6.4.

As it can be seen in Figure 6.9, the structural composition of a PS is given by the relation interfaces: I1I1ORelation, II2ORelation, and INORelation. Each interface defines structural restrictions to ensure that compositions which cannot be realized are directly ruled out: *correct-by-design*. Every derived relation, such as the `Nullspace_Relation`, `WeightedSum_Relation`, and `MotionForce-SubSpace_Relation`, is based on one of those interfaces and thus inherits the structural restrictions. Note that the remaining compositions are syntactically valid, but they do not have to produce a meaningful behavior.

Figure 6.12 uses a simplified syntax to show a set of structurally valid PSs. To make the user aware of potential problematic compositions, the user is notified if control tasks operate on the same target frame as a prioritization relation that is closer to the root node in the respective branch of the PS tree. Such a notification would be issued in Figure 6.12a, as well as in Figure 6.12b if the lower `ControlTask` or one of the branches of the `WeightedSum_Relation` targets the same frame as the `Nullspace_Relation`. Figure 6.12c potentially only makes sense if the filters of both `MotionForceSubSpace_Relation`s are different. In Figure 6.12d it cannot be ensured that the `ContactConstraint` represented by the `MotionForceSubSpace_Relation`s can be properly enforced, since it becomes weighted in the `WeightedSum_Relation`.

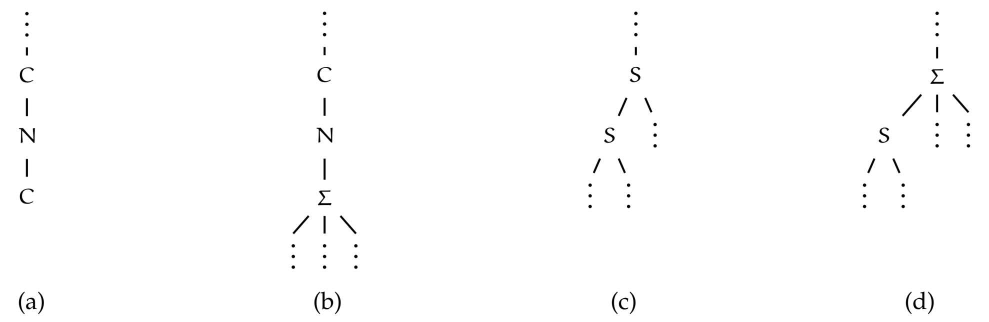


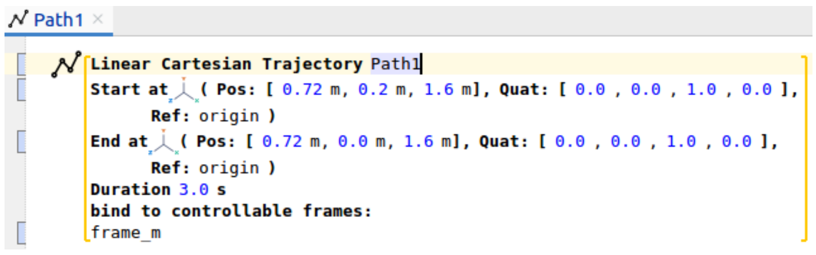
Figure 6.12: Different valid PSs. C, N, Σ, S denote a `ControlTask`, a `Nullspace_Relation`, a `WeightedSum_Relation`, and a `MotionForceSubSpace_Relation` respectively. Omitted structural elements are denoted by three vertical dots.

6.3 CI COMPOSITION AND OTHER TASK ASPECTS

Up until this point, the presented modeling approach was mainly focused on one aspect of the task description, i.e. specifying the compliant environmental interaction. Even though the focus of this thesis is not to present new solutions to model the other aspects of a task description, the composition between the CI and the other models is still considered in the following.

CI and trajectory composition

As discussed in the domain analysis (see Chapter 5), trajectories do not directly belong to the compliant interaction model. Trajectory models are used in combination with CI models to complement a task description. In CoSiMA, trajectories are represented as a path with additional timing constraints on the execution. Thus, a Cartesian motion trajectory uses a geometric path, while a force or velocity trajectory uses a force or velocity profile respectively. A trajectory generator then interprets the trajectory model and provides set points to a control task. On a conceptual level to compose the trajectory model with the constraints that are realized by the control tasks which receive the setpoints of the trajectory, the `Frame` concept is used as a common representation. Hence, a very loose and exchangeable coupling between the trajectory and CI aspects of a task description can be achieved. Such a trajectory model for the wiping example can be seen in Figure 6.13.



```
Path1
Linear Cartesian Trajectory Path1
Start at ( Pos: [ 0.72 m, 0.2 m, 1.6 m], Quat: [ 0.0 , 0.0 , 1.0 , 0.0 ],
        Ref: origin )
End at ( Pos: [ 0.72 m, 0.0 m, 1.6 m], Quat: [ 0.0 , 0.0 , 1.0 , 0.0 ],
        Ref: origin )
Duration 3.0 s
bind to controllable frames:
frame_m
```

Figure 6.13: A model of a linear Cartesian motion trajectory for the wiping example, acting on the `Frame` {m}, which is constrained by a mass-spring-damper constraint in the CI model.

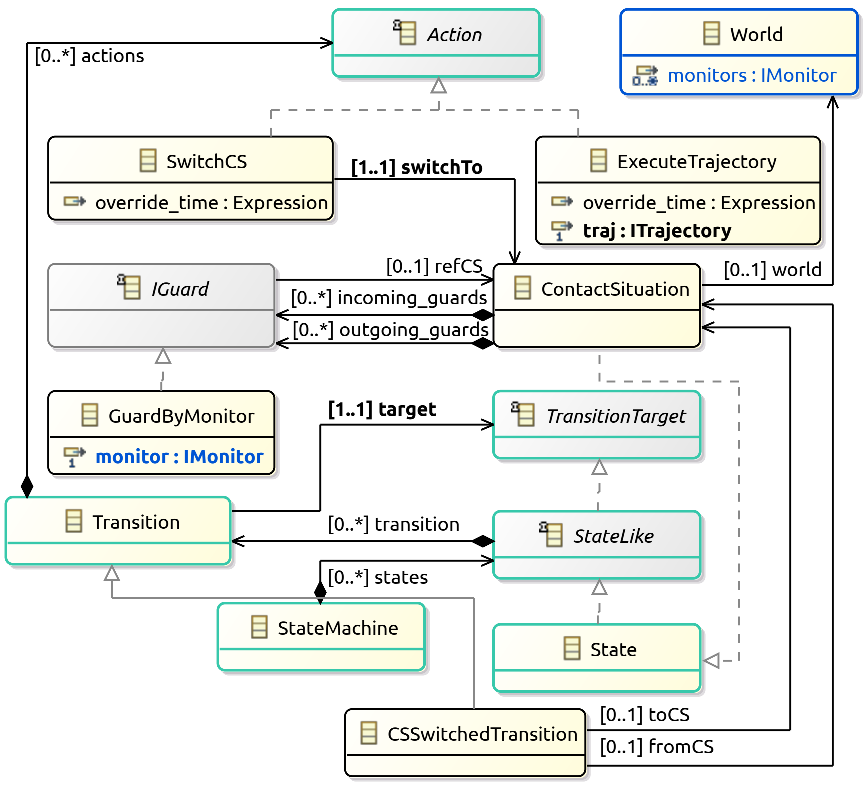


Figure 6.14: Excerpt of the meta-model that shows the relevant concepts for the loosely coupled composition of the CI model and task coordination. The color coded concepts are reused from the world meta-model (blue) and the *Coordination DSL* (green).

CI and monitor composition ○ The aspect of monitors to observe an external or internal signal perceived by a system is widely used and covered by numerous publications concerned with task specifications. Monitors are commonly used in conjunction with the task coordination aspect to specify conditions for state transitions. As discussed in the domain analysis, the majority of relevant approaches use some kind of state-chart-based coordination mechanism for the task coordination. In CoSiMA, this coordination is modeled in the *Coordination DSL* based on the well-known SCXML [W3C15] formalism (see Section 3.3 and [Wig+17a; Wig+17b; Nor16]). Approaches that consider the CI to a relevant degree, have a tight coupling, where the state of the interaction is governed by the global task coordination. In contrast to that, CoSiMA strictly separates the state of the CI from the task coordination, but allows modeling `IMonitor`s and CI-specific coordination `Action`s to achieve a loose coupling. This way, the environmental state can change independent of the task coordination state. Analogous to the coordination, the CI state space is modeled based on an extension of the *Coordination DSL* (see Figure 6.14), where each state is represented by a `ContactSituation`, forming a contact situation graph. Per default, this graph can be freely transitioned. This means that each state is connected to every other state by a `Transition`. Using `IGuard`s, these default transitions can be constrained: One possibility is to use monitors to e.g., react to perceived changes in the environment. The DSL extension also provides additional `Action`s related to the CI state switching for the task coordination. This allows the artificial changing of internal CI states.

○ *loose coupling of CI and task coordination*

6.4 DISCUSSION

In this section, the integration of the CI modeling approach into CoSiMA is discussed. Further, it is compared to the closely related approaches identified in Section 6.1. This is followed by a discussion of the benefits that arise from the decoupling of the task-related aspects.

6.4.1 *Integration into CoSiMA's Composition Structure*

The introduced concepts in this chapter aid the purpose to model a CI task description. By using the L3Dim composition mechanisms, the concepts can be integrated into the existing DSL composition of CoSiMA in form of three new DSLs: *World DSL*, *Compliant Interaction DSL*, and *Control Frameworks DSL*. Even though the DSLs in the composition target different modeling purposes (see Figure 3.1), their model fragments coexist in a unified and rich model.

Figure 6.15 shows the integration of the three CI-related DSLs into CoSiMA. While the *World DSL* reuses concepts from the *Kinematics Dynamics DSL*, the *Compliant Interaction DSL* strongly relies on the concepts from the *World DSL* for the modeling of contact constraints. It inherits the state machine concepts from the *Coordination DSL* to model the contact state graph. Hence, it also inherits the L3Dim mechanism of the *Capability DSL* to be platform-annotatable. This is important for the flexible modeling and synthesis. That way, a platform-independent CI task description can be annotated using the *Control Frameworks DSL* with a software platform specialization for control frameworks, such as PIDC or OpenSoT. This way, additional constraints and restrictions can be enforced on the model depending on the requirements and functionalities of the chosen software control framework. The OpenSoT annotation for

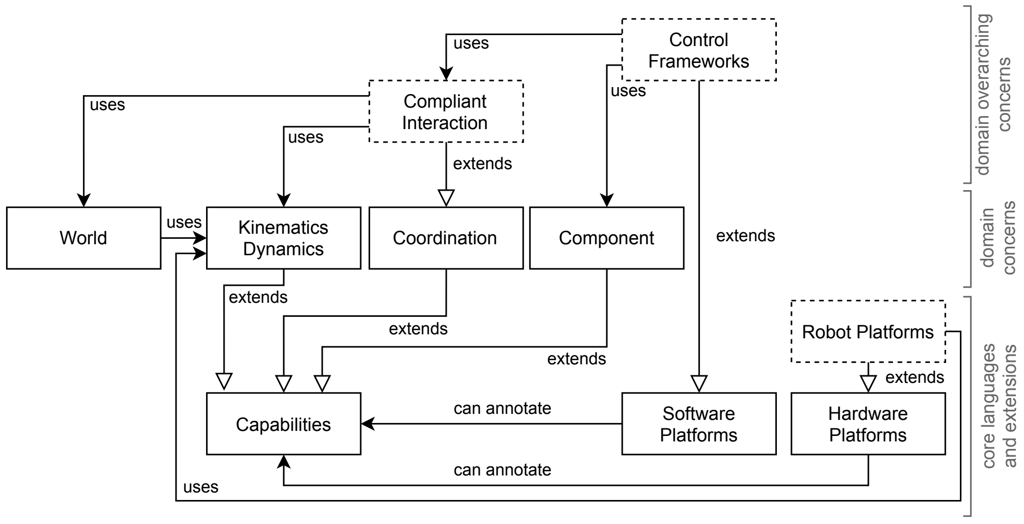
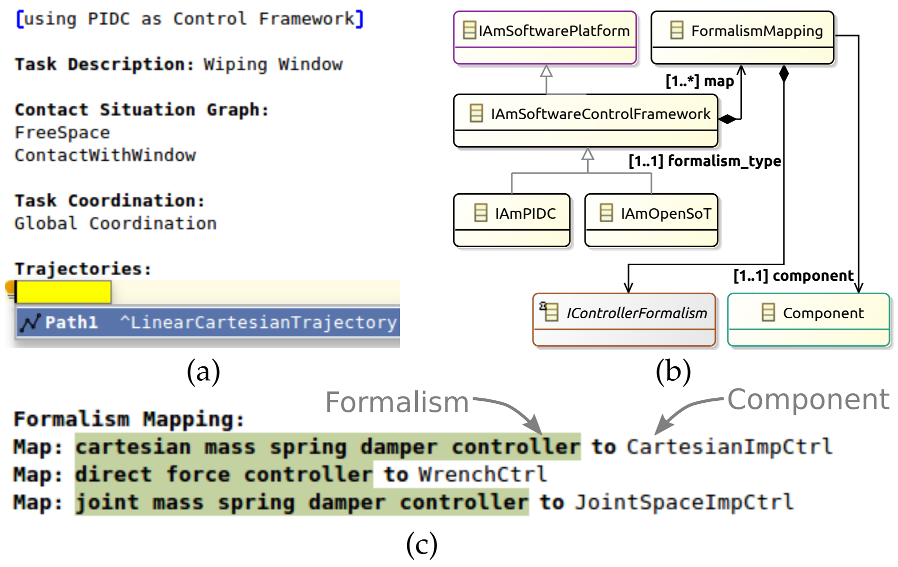


Figure 6.15: Integration of the CI DSLs into the existing composition of CoSiMA.

instance can further reduce the set of valid PSs to avoid structures such as shown in Figure 6.17. The respective annotation is provided by the *Control Frameworks DSL* for every supported control framework. The annotation uses the `FormalismMapping` concept to establish the mapping between framework-independent `IControllerFormalism`s and `Component`s from the *Component DSL*. Note, the mapping is relevant for the synthesis in Chapter 7.



```
[using PIDC as Control Framework]

Task Description: Wiping Window

Contact Situation Graph:
FreeSpace
ContactWithWindow

Task Coordination:
Global Coordination

Trajectories:
Path1  ^LinearCartesianTrajectory
```

(a)

(b)

```
Formalism Mapping:
Map: cartesian mass spring damper controller to CartesianImpCtrl
Map: direct force controller to WrenchCtrl
Map: joint mass spring damper controller to JointSpaceImpCtrl
```

(c)

Figure 6.16: Annotating a software control framework (e.g., PIDC) to the platform-independent CI task description model using the *Control Frameworks DSL*. (a) shows the annotation, (c) shows the mapping between formalisms and components, and (b) shows the relevant meta-model concepts. Color coded concepts are reused from other DSLs: *Software Platform DSL* (purple), *Compliant Interaction DSL* (brown), and *Component DSL* (green).

Figure 6.16 shows the annotation and mapping model for the wiping example. The concept of a `Task` is technically used as a root container to combine the different task-related aspects identified in Section 5.1, such as the contact state graph, a global task coordination, and the trajectories. Monitors are not referenced as first class model elements, since they are modeled inside the contact situations. The `Task` concept is the root element in the AST for the task description modeling and defines the annotation target for a software control framework.

>_ *abstract syntax tree*

6.4.2 *Comparison with Related Approaches*

As previously discussed and shown in Table 6.1, eTaSL and the DSL of iTaSC are closely related to the presented CI modeling approach of CoSiMA. One key difference is that CoSiMA supports closing a VKC on the dynamics level. This means that constraints on the joint configuration can be modeled over the entire closed VKC. Further, compensation forces for an unactuated body (e.g. a grasped box) can be distributed and optimized over the entire closed VKC as well.

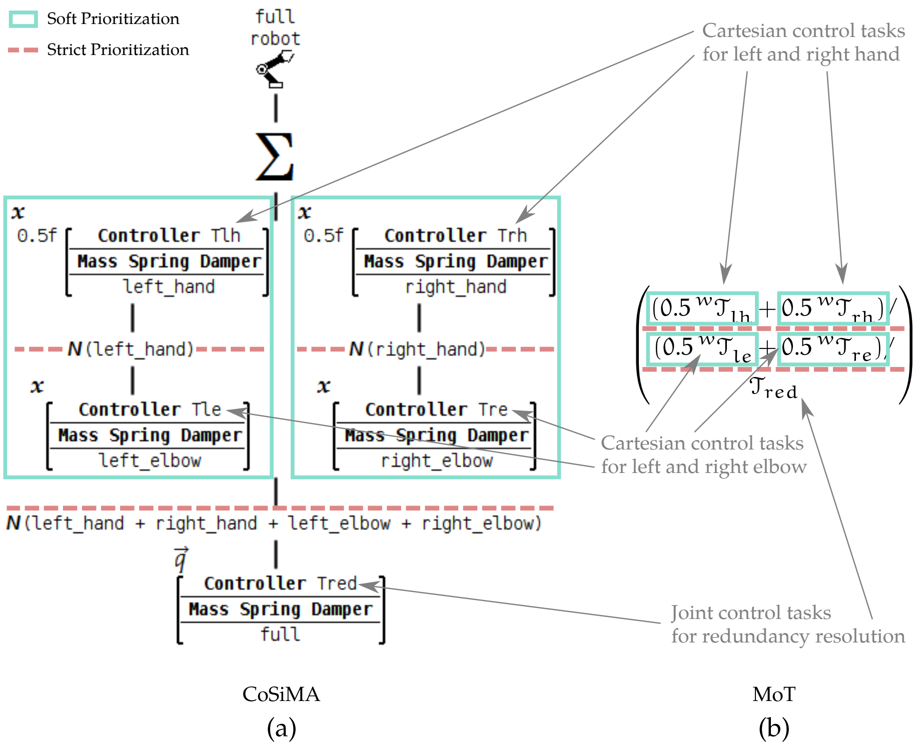


Figure 6.17: The PS shown in (a) cannot be modeled using MoT. (b) shows a composition of soft and strict prioritization structures that is as similar as it can be achieved by MoT. However, it still does not match with (a).

In terms of the prioritization of constraints or control tasks, approaches such as eTaSL and iTaSC allow strict and soft prioritization—as does CoSiMA. However, the priority level and weighting factor are directly modeled in the individual configurations of the tasks by the other approaches. Math of Task

○ *the prioritization modeling in CoSiMA is equally or even more expressive compared to related approaches*

(MoT) [HT21; Hof+17b] features a similar approach. It is a DSL to define stacks of QP-based control tasks. Instead of separating the information for prioritization among the different control task configurations, MoT uses a unified way to model the prioritization, which is similar to CoSiMA's concept of PSs. From a pure modeling perspective, eTaSL, iTaSC, and MoT are limiting their expressiveness by solely relying on a priority and weighting factor per control task. For instance, CoSiMA allows to model a strict prioritization in a (sub)branch of a soft prioritization (see Figure 6.17). To the best of my knowledge, this is not possible with the other approaches. In MoT the reason is tied to it being designed in close relation to the implementation of the soft and strict prioritization mechanisms in the OpenSoT framework, which solves sequential QPs. Combining and weighting multiple sequential QP problems (i.e. SoTs) is currently not possible in OpenSoT. Further, in CoSiMA the `MotionForceSubSpace_Relation` is introduced to structurally support the modeling of control tasks applied to different subspaces. A similar concept can be found in MoT, called subtasks. Since CoSiMA and MoT are conceptually very close, MoT provides a suitable generation target for the realization of CI tasks modeled in CoSiMA to be executed using QPs via OpenSoT in addition to a realization using PIDC.

6.4.3 *Decoupling Coordination and CI*

CoSiMA realizes the proposed loose coupling of the different task-related aspects (see Section 6.3). A great advantage comes from the decoupling of the coordination from the CI modeling. Both aspects are often deeply entangled in related skill frameworks. This means that for instance a specific contact situation is only considered in the context of a specific skill. The skill that is executed next has no knowledge about the previous contact situation. This leads to an unnecessary discretization of the robot's motions. In CoSiMA, the general task coordination operates independent of the contact state graph and its transitions. This enables the following two types of behaviors:

contact situations can change independent of the trajectory or coordination state

COMPLIANT INTERACTION CHANGES Assuming the case a robot transitions from a free space state without contacts to a contact situation where it is physically guided along one direction (see Figure 6.18). By loosely coupling the task coordination state space and the CI contact state graph, the execution of a trajectory (i.e. Traj 1) does not need to be split into discrete motion events. In the majority of skill frameworks, one skill would cover the free space motion, whereas a subsequent one would cover the rest of the motion, while being in contact with the guide. In CoSiMA, such a separation is not necessary. The entire motion can be executed independent of the contact situation, since the active contact state acts as a context for the motion execution. Hence, once in contact (i.e. CS1), the trajectory would lose its impact in the guided direction and regain it once the robot enters free space again. Of course, the coordination and contact state graph are notified every time a state change happens.

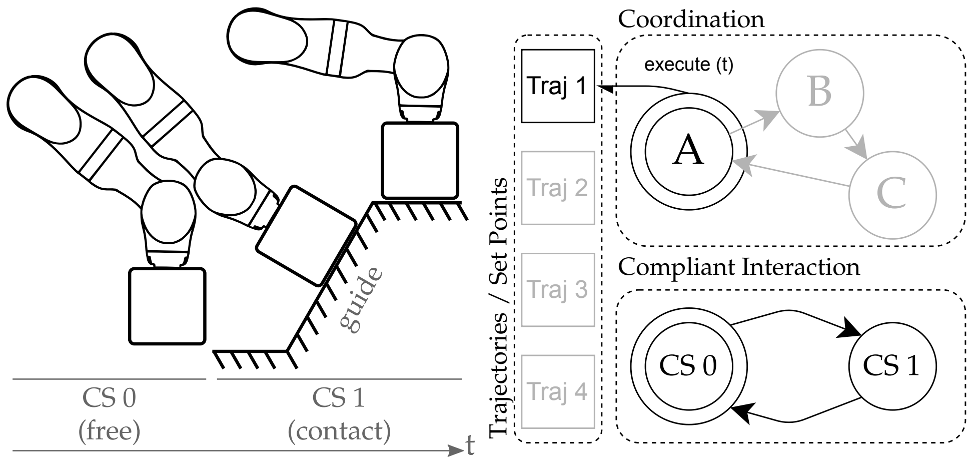


Figure 6.18: The robot follows a trajectory while being in the coordination state A. During the execution, the contact situations state changes and a transition is triggered from CS0 to CS1 in the contact state graph. There is no need for the system to stop and change to another skill. Hence, a smooth execution is performed also during changes in the contact situations.

TASK COORDINATION CHANGES Assuming a scenario in which it is vital to maintain a specific contact situation throughout the whole execution of a scenario (see Figure 6.19). In conventional skill frameworks, each skill would require an internal representation of the contact state, because the CI is not shared among skills, but is often directly encoded. This means that if the first skill establishes a contact situation, the following skills need to know this in preface. In CoSiMA, a contact situation can be used as a context for the entire task coordination. This way, a contact state can be upheld without being specifically encoded into different coordination states.

○ *trajectories and coordination states can change independent of the contact situation*

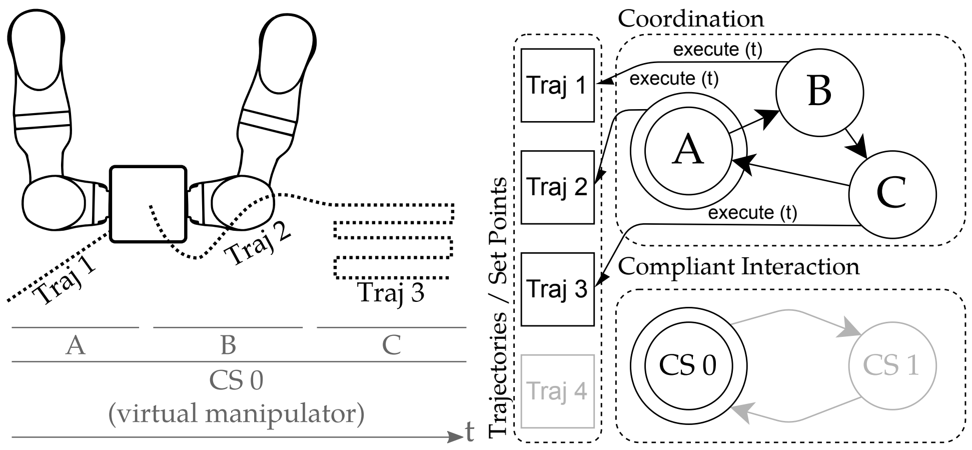


Figure 6.19: The contact situation CS0 creates a virtual manipulator. Even during state changes in the general task coordination, the closed VKC is maintained. This is due to the contact state acting as context for the entire coordination aspect instead of being limited to a single skill.

6.5 CONCLUSION

While the previous chapter conducted a domain analysis to cover RQ 3 in terms of finding a conceptual base to approach the description of CI tasks, this chapter answers the second part of RQ 3: How to turn the conceptual base into a set of domain-specific meta-models? To this end, the presented meta-models were developed. They allow the technical modeling of CI tasks, while also being fully integrated into CoSiMA as DSLs. Hence, they reuse existing model elements from e.g., the robotic platform and the system coordination concerns and provide the model input for the synthesis in Chapter 7.

These meta-models allow the creation of CI models using an explainable abstraction, grounded in physical constraints from the geometry, kinematics, and dynamics domain. Additionally, a prioritization of the constraints can be modeled to define the influence of constraints on each other and to handle potential overconstrainedness on the task space DoF.

By choosing the CI abstractions to be composable with other task-related aspects such as trajectories or coordination, every task that is modeled in this way can be boiled down to a common set of abstractions. Technically, this common conceptual base allows (1) the decomposition of a task model into its different aspects, (2) the individual comparability of these aspects, (3) and a flexible (re)composition. Thereby, overcoming the currently predominant approach of treating a task or skill as a black box, and opening up the possibility for task-overarching constraints and verifications. Hence, further providing the required information to smoothly blend between tasks during the execution.

In contrast to the exemplary model fragments that are used to aid the explanation of the underlying concepts in this chapter, a more sophisticated application of the meta-models can be seen in Chapter 8.

“*Nothing is lost, nothing is created,*
everything is transformed.”

— Antoine Lavoisier

7 CI ARCHITECTURE SYNTHESIS

This chapter presents the synthesis of an executable control architecture for a robotic system that realizes a modeled CI task. The synthesis contains two generator pipelines, used to transform the CI model into (1) a component-based control architecture model and into (2) a graph-based representation of the compliant interaction to configure the architecture to the task. The resulting models are generated into an executable OROCOS RTT-based PIDC software architecture. The chapter is based on [WDW20].

In Chapter 3 the transformation from a component-based robotic system model to an actual executable software system was presented. To overcome the implicitness of the task definition in such a system, Chapter 6 presented a DSL extension to CoSiMA, which allows for the explicit modeling of a task description with particular focus on the CI. This chapter links the task to the system model by synthesizing a component-based control architecture for the robotic system, based on the modeled CI task. Two generators are developed and integrated into CoSiMA that use four different transformations to generate the control architecture for the chosen control framework (i.e. PIDC) and to configure it with the modeled CI task.

>_ *control architecture*

First, a reference architecture is introduced that acts as a blueprint, which is populated with component models for the PIDC framework. While the blueprint itself is framework-independent, its concrete instantiation depends on the chosen framework (i.e. PIDC). To reuse the existing generator pipelines of CoSiMA, the component models are assumed to represent existing implementations based on the chosen software platform (i.e. OROCOS RTT).

The second step is the introduction of a set of transformations that synthesize an instantiation of the reference architecture for the chosen control framework, based on the CI task model. This directly addresses RQ 4. The resulting artifacts of the transformations are a component-based control architecture in form of a `System` model and a configuration artifact that holds the required task knowledge (i.e. the contact state graph, control formalism parameters, and monitors). A task-level coordination needs to be provided if an active switching of contact situations is desired rather than a passive switching, which is solely based on sensory data. Note that trajectories and the task-level coordination are excluded from the presented synthesis and need to be manually supplied by the *Behavior Developer* for now.

The last step addresses the OROCOS RTT- and PIDC-specific interpretation of the configuration artifact by the synthesized control architecture.

7.1 COSIMA'S REFERENCE ARCHITECTURE FOR CI

We developed a suitable reference architecture for CI, based on the experiences gained in the CogIMon [Cog19] and CMCI [CMC20] projects, that is generally control framework independent. The structural view of the reference architecture is shown in Figure 7.1 below.

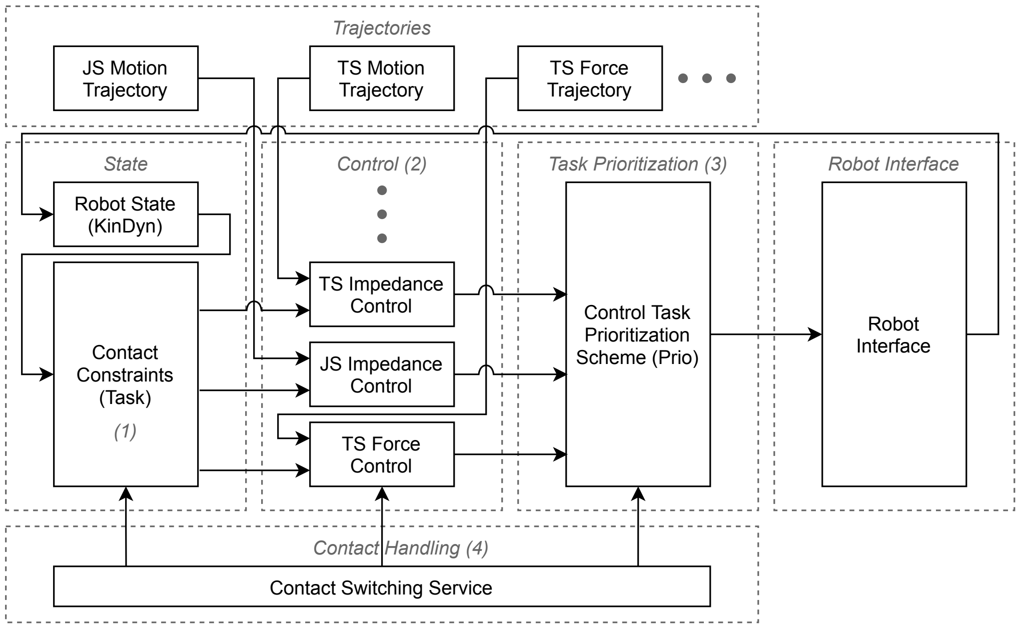


Figure 7.1: Structural view of the reference architecture that is instantiated with the synthesized knowledge from the contact situations.

In general, there are four major parts that define configuration points in the framework-independent architecture, namely the control components, the task component, the prioritization component, and the contact (situation) switching service. The interface to command a robot, the kinematics and dynamics component providing several state-related robot quantities (e.g., Jacobian, Cartesian Pose, etc.), and the trajectory generators are necessary for the actual execution of a CI task. However, they are not directly in the focus of this work and are thus ignored by the synthesis. The components relevant for the synthesis are described in the following:

TASK COMPONENT (1) The task component provides the inputs for the control components in the associated task (Cartesian or joint) space. This means that the data (e.g., Jacobians) is also modified according to the subspace the controller is tasked to operate in. The (sensory) data relevant for the monitors and guards of contact state transitions are also automatically provided by this component. Currently, the link to the task-level coordination needs to be manually performed by the *System Builder*.

CONTROL COMPONENTS (2) Each control component represents an instantiation of the `Component` concept from the *Component DSL*, which is mapped to a suitable control formalism by the software control framework annotation (i.e. `IAmPIDC` see Section 6.4.1). A control component receives its input from the task component. Depending on the type of controller, different quantities are provided through the task component.

PRIORITIZATION COMPONENT (3) The prioritization component receives the control signals of all control components and superimposes them, based on the active contact situation in the contact state graph, into a single control signal that is sent to the robot.

CONTACT SWITCHING SERVICE (4) This service provides an interface for the task coordination to switch between contact situations. When a transition is triggered or finished, the task, the prioritization, and the control components are notified. During a transition the task component shapes the data it provides according to the requirements of the new contact situation, while the prioritization component is triggered to blend from the current PS to the new one. To evaluate transitions and guards, the service receives the monitor data provided by the task component and forwards it to the task coordination.

>_ *prioritization structure*

7.2 SYNTHESIS OF THE SYSTEM'S CONFIGURATION

The aim of this synthesis is to instantiate the reference architecture in form of a `System` model with the set of control components and their specific configurations, required to ensure the constraints that are modeled in the contact situations of the CI task. The synthesized system model can afterwards be enriched with further system-related concerns (e.g., timing). Eventually it is transformed into an executable software system (see Figure 7.3).

○ *the* `System` *concept belongs to the* Component DSL

Two generators are developed to achieve this. The *CI Config Generator* uses the following multi-staged M2M transformations A, B, and C to transform the modeled PSs, that prioritize the constraints of the CSs, into a configuration model. This model contains the information that cannot be expressed in the component-based structure of the control architecture, i.e. the prioritization and parameterization of the control components. In contrast, the *CI System Generator*, explained in Section 7.3, uses the multi-staged M2M transformations A and D to generate the system's control architecture model (using the `System` concept) based on the chosen control framework (i.e. PIDC) and the reference architecture.

>_ *model-to-model*

>_ *Contact Situation*

TRANSFORMATION A The aim of this transformation is to generate a set of unified prioritization graphs U from the prioritization structures associated with the modeled CSs. In terms of prioritization, each CS is represented by a `CompliantControlArchitecture` concept instance CCA_i that contains one or more `Entry_Relation`s that define the roots of the PSs (see Section 6.2.4). The following definition of a graph is used throughout this chapter:

Definition: Graph

A graph G is defined by a 5-tuple $(V, E, \sigma_v, \sigma_e, \Sigma)$, where

- $V = V(G)$ is the set of vertices of the graph G,
- $E = E(G) \subseteq V \times V$ is the set of edges of G,
- each edge $e \in E$ is represented as an ordered pair of vertices,
- the label functions $\sigma_v : V \to \Sigma$ and $\sigma_e : E \to \Sigma$ map vertices and edges to labels of the set Σ.

transformation step A.1 in Figure 7.2 ○

To achieve this aim, each PS gets transformed into a graph $\hat{G}_{i,K_r}$, where K_r is the kinematic chain referenced by the `Entry_Relation` of the PS. Every control task sharing the same controllable frame CF and formalism F is mapped to the same vertex τ_l[1] in the graph. The relations, i.e. `Nullspace_Relation` ("*N*") and `MotionForceSubSpace_Relation` ("*S*"), introduced in Section 6.2.4, are mapped to edges that connect two associated control tasks and are directed towards the higher prioritized one. The label $\sigma_e((\tau_u, \tau_v))$ for the edge (τ_u, τ_v) indicates the type (i.e. "*N*", "Σ", or "*S*") and CF of the relation. In contrast to the other relations, the `WeightedSum_Relation` ("Σ") cannot be directly mapped to an edge. Instead, inspired by Equation 5.8, the relation is transformed into multiple weighted Σ-edges between the involved control tasks. Further, if the `Entry_Relation` of a PS references a virtual manipulator as K_r that uses a `ForceClosure`, a new vertex needs to be added to $\hat{G}_{i,K_r}$. This vertex represents the control task that controls the internal forces to maintain the virtual manipulator. An additional *S*-edge (i.e. S_{vm_r}) represents the relation between the motion and internal force subspaces. The generation of the vertex and edge for the virtual manipulator is described in Algorithm 7.1.

Since all control tasks will be instantiated in the reference architecture as control components, all of them need to be considered (for prioritization) in every CS. However, if a control task is not part of a particular CS, it needs to be disabled once that CS is active. Hence, the resulting graphs $\hat{G} = \{\hat{G}_{i,K_r}\}$ are combined into the unified graphs $U = \{U_{K_r}\}$, grouped by the associated kinematic chain K_r of the source PS, where

transformation step A.2 in Figure 7.2 ○

$$U_{K_r} = \bigcup_i \hat{G}_{i,K_r} \,. \tag{7.1}$$

1 Vertices are indicated by τ, because they refer to control tasks.

Algorithm 7.1 Transformation A.1: Virtual Manipulator

```
Input: A graph $\hat{G}_{i,K_r}$ where $K_r$ refers to a virtually closed kinematic chain
1: if the associated VirtualManipulator constraint uses ForceClosure then
   Introduce a control task to control the internal force of the VM:
2:     find the vertex $v_{root} \in V(\hat{G}_{i,K_r})$ without an outgoing edge,
        i.e.$\neg(\exists u \in V(\hat{G}_{i,K_r}) : (v_{root}, u) \in E(\hat{G}_{i,K_r}))$
3:     let $v_{vm}$ be a vertex
4:     $\sigma_v(v_{vm}) \leftarrow$ *unique_name*("ifctrl")        ▷ generates e.g., ifctrl1, ifctrl2, ...
5:     $V(\hat{G}_{i,K_r}) \leftarrow V(\hat{G}_{i,K_r}) \cup v_{vm}$
6:     $E(\hat{G}_{i,K_r}) \leftarrow E(\hat{G}_{i,K_r}) \cup (v_{root}, v_{vm})$
7:     $\sigma_e((v_{root}, v_{vm})) \leftarrow "S_{vm}"_r$        ▷ e.g., $S_{vm_{1,2}}$
8: end if
Output: $\hat{G}_{i,K_r} \in \hat{G}$
```

>_ *see Definition 7.1 for the graph definition (i.e. V, E, σ_e, σ_v)*

○ *the colored step produces the colored edges in Figure 7.2*

TRANSFORMATION B The aim of this transformation is to derive the prioritization information of the control tasks, the control formalism parameters, and the subspace filters for all involved kinematic chains per contact situation CCA_i. By applying Algorithm 7.2, the graphs $\{U_{K_r}\}$ and $\{\hat{G}_{i,K_r}\}$ together yield the prioritization graphs $\{G_{i,K_r}\}$. For each contact situation, the respective graph G_{i,K_r} then contains the prioritization information of the control tasks $V(G_{i,K_r})$ for the kinematic chain K_r.

○ *transformation step B.1 in Figure 7.2*

Inspired by the insights gained from the domain analysis in Section 5.3, the priority of a pair of tasks $<\tau_u, \tau_v>$ is defined by the direction and label of the connecting edges and is expressed as a scalar value $\alpha_{u,v} \in \mathbb{R}^+$. *N*-edges express a strict prioritization hierarchy by choosing $\alpha_{u,v} \in \{0, 1\}$. Whereas a Σ-edge expresses a soft hierarchy by choosing $\alpha_{u,v} \in [0, 1]$. The (de)activation of a task in the hierarchy is specified by the diagonal entries $\alpha_{u,u} \in [0, 1]$. Eventually, each G_{i,K_r} is expressed in form of a prioritization matrix (cf. Equation 5.8) by calculating its adjacency matrix:

○ *transformation step B.2*

$$\boldsymbol{\Psi}_{i,K_r} = (\alpha_{uv}) \in \mathbb{R}^{k \times k} = \mathrm{Adj}(G_{i,K_r}) \,, \tag{7.2}$$

where k denotes the number of control tasks in the graph.

The advantage of deriving the prioritization graphs of the PSs using a unified graph representation is the possibility to blend between contact situations (e.g., $CCA_a \rightarrow CCA_b$) by blending between the prioritization graphs (e.g., $G_{a,K_r} \rightarrow G_{b,K_r}$). Given CoSiMA's PIDC-based control framework implementation that employs a mixture-of-controllers approach [Deh18], a dynamic reprioritization of control tasks is supported at runtime. This means that by interpolating $\{\boldsymbol{\Psi}_{current}\}$ and $\{\boldsymbol{\Psi}_{new}\}$ for all K_r, the synthesized robotic system is able to achieve a continuous transition from one contact situation to another.

An exemplary application of the transformations A and B can be seen in Figure 7.2.

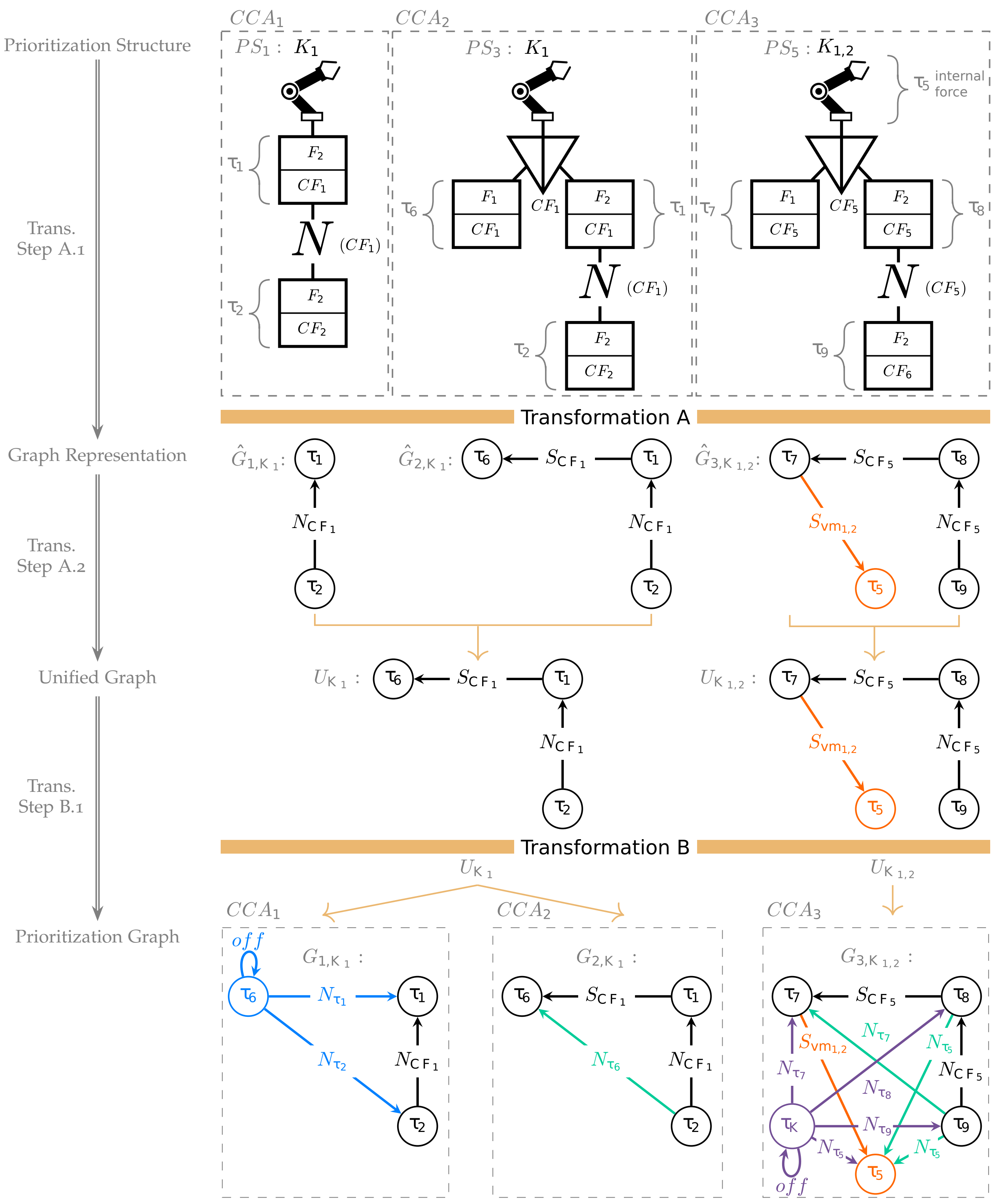


Figure 7.2: Multi-staged M2M transformations for the generation of prioritization graphs from the PSs of the CI models (i.e. `CompliantControlArchitecture`s). Control tasks are denoted as τ. Transformation step A.1 translates each PS into a graph and step A.2 combines them into unified graphs. The final prioritization graphs are created by B.1 based on the unified graphs. Colored edges are described in Algorithm 7.2.

Algorithm 7.2 Transformation B.1: Prioritization Graphs

```
Input: The graph sets $\hat{G}$ and $\{U_{K_r}\}$ from the previous transformation
1: for each $\hat{G}_{i,K_r} \in \hat{G}$ do
     1) Make implicit nullspace relations explicit:
2:     $E_p \leftarrow \{(v,u) \mid \forall v,u \in V(\hat{G}_{i,K_r}) : path(v,u) \wedge \neg path^{direct}(v,u)\}$
3:     $\sigma_e((v,u)) \leftarrow "N"_{\sigma_v(u)} \mid \forall v,u : (v,u) \in E_p$
4:     $E(G_{i,K_r}) \leftarrow E(\hat{G}_{i,K_r}) \cup E_p$
     2) Deactivate irrelevant vertices (i.e. $\notin G_{i,K_r}$) and turn them into subordinates of the relevant vertices:
5:     $V'' \leftarrow V(U_{K_r}) \setminus V(G_{i,K_r})$
6:     $E'' \leftarrow \{(v,v) \mid v \in V''\}$
7:     $\sigma_e(e) \leftarrow "off" \mid \forall e \in E''$
8:     $\sigma_e((v,u) \in E'') \leftarrow "N"_{\sigma_v(u)} \mid \forall v,u : v \in V'' \wedge u \in V(G_{i,K_r})$
9:     $V(G_{i,K_r}) \leftarrow V(G_{i,K_r}) \cup V''$
10:    $E(G_{i,K_r}) \leftarrow E(G_{i,K_r}) \cup E'' \cup \{(v,u) \in E(U_{K_r}) \mid \forall v,u \in V''\}$
     3) Introduce a deactivated vertex for the virtually closed kinematic chain as subordinate to all other vertices:
11:    if $K_r$ is a virtually closed kinematic chain then
12:        let $v_m$ be a vertex
13:        $\sigma_v(v_m) \leftarrow "\tau_K"$
14:        $E''' \leftarrow \{(v_m,u) \mid u \in V(G_{i,K_r})\}$
15:        $\sigma_e(e) \leftarrow "N"_{\sigma_v(u)} \mid e = (v_m,u) \in E'''$
16:        $V(G_{i,K_r}) \leftarrow V(G_{i,K_r}) \cup v_m$
17:        $E(G_{i,K_r}) \leftarrow E(G_{i,K_r}) \cup E'''$
18:    end if
19: end for
Output: $G = \{G_{i,K_r}\}$
```

>_ *see Definition 7.1 for the graph definition (i.e. V, E, σ_e, σ_v)*

○ *the three colored steps produce the colored edges in Figure 7.2*

○ *transformation C*

TRANSFORMATION C The aim of this transformation is to derive (1) the parameterization of the control tasks and (2) the filters that define the subspaces from the `VirtualManipulator`s and `ContactConstraint`s, associated with the vertices and edges in $\{G_{i,K_r}\}$ per `ContactSituation`. Each vertex represents a `ControlTask` concept instance in a PS that is defined by its `IControllerFormalism` and realizes a specific `ICIConstraint` (see Section 6.2). Formalisms have different parameters that need to be configured to achieve a certain control behavior. For instance, any mass-spring-damper-based formalism needs their stiffness and damping parameters to be set. These parameters are derived from the reference of the `ControlTask` to the `ICIConstraint`. Due to the parameters being specific to a certain formalism, the parameters can currently only be derived from the pairs of `ICIConstraint` and `IControllerFormalism`, defined in the `FormalismMapping`s (see Figure 6.16). The `MassSpringDamperConstraint` and `JointMassSpringDamperConstraint`, for instance, are realized by a `CartesianMassSpringDamperController` as well as a `JointMassSpringDamperController` respectively. In this case, the stiffness and damping parameters for every active DoF are persisted per contact situation.

The second kind of derived information is related to the (sub)spaces the control tasks operate in. First, the controllable frame CF is determined for each control task in $\{G_{i,K_r}\}$. Second, the DoFs a control task operates in are derived from the DoFs constrained by the e.g., `MassSpringDamperConstraint` or `ForceConstraint`. Third, each control task that is connected by the arrow's head of an edge with a subspace relation label (i.e. S_{CF_p}) is employed in the constraint space of a contact, while tasks connected by the arrow's tail are employed in the unconstrained space. The respective filter is then derived from the contact surface of the `ContactConstraint` referenced via the `MotionForceSubSpace_Relation`, represented by the edge. In the case of a VM, the internal forces are handled analogously by retrieving the subspace filter from the associated `VirtualManipulator` constraint. Since the PIDC implementation used throughout this thesis currently only supports bilateral constraints, the definition of any other type is avoided directly on the model level, when choosing PIDC. Therefore, each filter f can be represented as a matrix $\Gamma^f = (\gamma^f_{jl}) \in \mathbb{R}^{d \times d}$, where

Virtual Manipulator

```
---
Config:
  - Robot: kuka_iiwa_14
    JointDof: 7
    ControlTasks:
      - Name: $\tau_0$
          ControllableFrame: {Type: frame, Name: frame_c, Dof: d}
      ...
    ContactSituations:
      - Name: CS0
        CustomFilters:
          - $\tau_0$: filter0
          ...
        SubspaceFilters:
          - $\tau_0$: filter_c_0
          ...
          - $\tau_k$: filter_m_0
        Params:
          - $\tau_0$:
              - stiffness: [300.0, 300.0, 300.0, 300.0, 300.0, 300.0]
              - damping: [3.0, 3.0, 3.0, 3.0, 3.0, 3.0]
          ...
        PrioritizationMatrix: [
          [$\tau_0$, ... , $\tau_k$], [$\alpha_{0,0}$, ... , $\alpha_{0,k}$], ... , [$\alpha_{k,0}$, ... , $\alpha_{k,k}$]
        ]
      ...
  ...
Filters:
  - Filter: filter_c_0
    Type: ComponentWise
    Data: [[$\gamma^{c0}_{0,0}$, ... , $\gamma^{c0}_{d,0}$], ... , [$\gamma^{c0}_{0,d}$, ... , $\gamma^{c0}_{d,d}$]]
  - Filter: filter_m_0
    Type: ComponentWise
    Data: [[$\gamma^{m0}_{0,0}$, ... , $\gamma^{m0}_{d,0}$], ... , [$\gamma^{m0}_{0,d}$, ... , $\gamma^{m0}_{d,d}$]]
  ...
```

Listing 7.1: Example of a YAML-based configuration artifact. The math notation is chosen to visually indicate the control tasks (orange), the elements of the prioritization matrix (magenta), and the elements of the subspace filter matrices (blue). Monitors are excluded for brevity.

d denotes the DoFs of the (task)space that is filtered. The individual directions can be component-wise constrained with $\gamma_{j,j} = 1$, while unconstrained directions are defined by $\gamma_{j,j} = 0$. Note that it is the responsibility of the task component to interpret the matrix to produce the concrete task-related quantities (see Section 7.3.2).

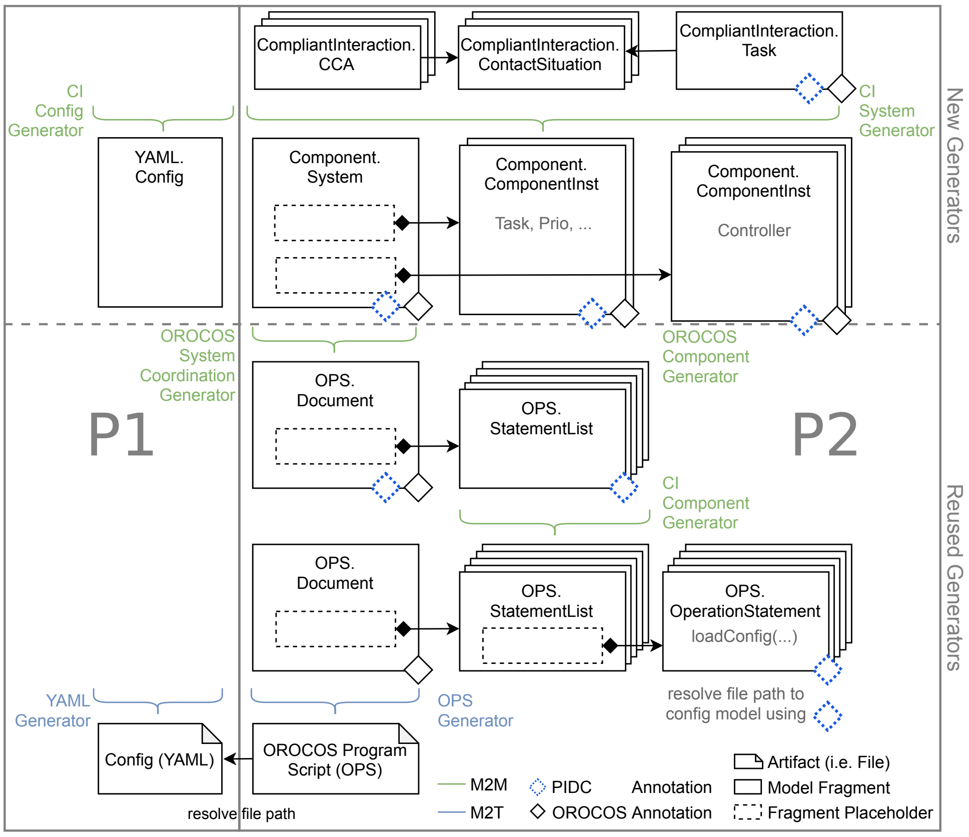


Figure 7.3: Model-to-model transformation pipelines (P1 and P2) showing the vertical synthesis of an executable PIDC-based OROCOS RTT control system from the CI task model. The PIDC annotation stores a reference to the `Task`, which allows to resolve the link to the configuration model and eventually the configuration artifact (e.g. YAML file). The *CI Config Generator* and the *CI System Generator* implement the synthesis transformations, while the *CI Component Generator* uses the PIDC annotation to resolve the link to the configuration artifact in form of OROCOS operations that are woven into the existing OPS model.

After the application of the transformations A, B, and C, the control tasks (orange), the prioritization matrices $\{\Psi_{i,K_r}\}$ (magenta), the formalism parameters (green), and the subspace filters $\{\Gamma^f\}$ (blue) are persisted in one configuration artifact via the *CI Config Generator*. For the experiments in Chapter 8 a YAML[2] representation is chosen as the concrete syntax of the artifact. An example of such a representation is shown in Listing 7.1.

○ *the colors refer to Listing 7.1*

Figure 7.3 shows an overview of the transformation pipelines formed by the *CI Config Generator*, the *CI System Generator* (explained in Section 7.3 below), and a set of reused generators to synthesize an executable control system that

2 https://yaml.org/spec/1.2/spec.html

uses PIDC as control framework and OROCOS RTT as execution environment. The *CI Config Generator* uses the transformations A, B, and C that are presented in this section, to form a transformation pipeline (see P1 in Figure 7.3) that generates a YAML-based configuration artifact from the set of $\{CCA_i\}$. This artifact is used to configure the software system to the CI task. Note that in order to exclusively allow the generation of models that can actually be realized by the chosen control framework implementation, only rigid contact models are currently supported.

7.3 SYNTHESIS OF THE SYSTEM'S PIDC ARCHITECTURE

This section introduces the transformation pipeline (see P2 in Figure 7.3) that is formed by the *CI System Generator* by drawing on the unified graphs $\mathcal{U}$ from transformation A to generate a component-based control architecture for the PIDC framework based on the reference architecture. To this end, a new transformation D generates a `System` model using the *Component DSL* by instantiating the control `Component`s along with the PIDC-specific variants of the other non-control `Component`s of the reference architecture. This transformation leverages insights from previous work, conducted in [WDW20; Wig+17a; Wig+17b; Deh+18; Deh18].

transformation D

i.e. kinematics and dynamics, task, prioritization, robot interface, and contact switching service components in Figure 7.1

In the following, the PIDC-specific implementations for the instantiated components of the reference architecture are described:

7.3.1 *Control Components*

In the reference architecture for PIDC, all control tasks produce a joint torque signal. Tasks that share the same formalism F are executed in the same control component. To know which `Component` needs to be instantiated as a `ComponentInst` in the `System` model to realize the formalism of a specific control task $\tau \in \mathcal{U}$, the `FormalismMapping`s (see Section 6.4.1) defined in the PIDC annotation (see Figure 7.3) is used. All the necessary inputs for each control task are stacked and processed simultaneously. Eventually the resulting torque control signal for each task is stacked and passed to the prioritization component. This mechanism allows to reduce the amount of instantiated control components. The *CartesianImpCtrl* component (in Figure 8.14) for instance combines Equation 2.18b and Equation 5.4 to form the control law defined as:

$$\boldsymbol{\tau}_m = \mathbf{P}\mathbf{J}_x^T \left[\mathbf{h}_c + \boldsymbol{\Lambda}_c \ddot{\mathbf{x}}_d - \mathbf{D}_d \dot{\tilde{\mathbf{x}}} - \mathbf{K}_d \tilde{\mathbf{x}}\right] \ , \tag{7.3}$$

where $\mathbf{h}_c = \boldsymbol{\Lambda}_c \mathbf{J}_x \mathbf{M}_c^{-1}(\mathbf{P}\mathbf{h} - \dot{\mathbf{P}}\dot{\mathbf{q}}) - \boldsymbol{\Lambda}_c \dot{\mathbf{J}}_x \dot{\mathbf{q}}$ denotes the operational space gravitational, centrifugal, and coriolis effects. The parameters $\mathbf{K}_d$ and $\mathbf{D}_d$ are exposed to be set by the task component, according to the values defined for the currently active contact situation in the generated configuration artifact (i.e. YAML file). The control law for the *JointSpaceImpCtrl* component (in Figure 8.14) is designed analogously in the joint space. To apply a desired wrench $\boldsymbol{\lambda}$ in a particular direction (e.g., contact direction) expressed in the controllable frame, the *WrenchCtrl* component (in Figure 8.14) implements its control law as follows:

$$\boldsymbol{\tau}_c = (\mathbf{I} - \mathbf{P})\mathbf{J}_c^T \boldsymbol{\lambda} \,. \tag{7.4}$$

7.3.2 *Task Component*

The main quantities that the PIDC-specific task component provides are a task matrix (i.e. Jacobian $\mathbf{J}$), a projection matrix $\mathbf{P}$ for a potential subspace projection, a constraint consistent joint-space inertia matrix $\mathbf{M}_c$, their derivatives, and compensation quantities for gravitational and coriolis effects. All quantities are expressed in the world frame W with relation to the controllable frame CF. A `CustomFilter` is realized as modification of the Jacobian. In the case of a `ContactFilter` (i.e. subspace filter), two selection matrices are formed based on the DoFs of all contact constraints associated with the same controllable frame: ${}^{CF}\mathbf{S}_f$ for the constraint space and ${}^{CF}\mathbf{S}_m = \mathbf{I} - {}^{CF}\mathbf{S}_f$ for the unconstrained space. The respective constraint (${}^{W}\mathbf{J}_c$) and unconstrained (${}^{W}\mathbf{J}_x$) Jacobians are denoted as[3]

see Figure 6.9 for the meta-model of the concepts

$$ {}^{W}\mathbf{J}_c = {}^{W}\mathbf{R}_{CF}\ {}^{CF}\mathbf{S}_f\ {}^{CF}\mathbf{R}_W\ {}^{W}\mathbf{J} \,, \tag{7.5}$$
$$ {}^{W}\mathbf{J}_x = {}^{W}\mathbf{R}_{CF}\ {}^{CF}\mathbf{S}_m\ {}^{CF}\mathbf{R}_W\ {}^{W}\mathbf{J} \,, \tag{7.6}$$

where ${}^{W}\mathbf{R}_{CF}$ is the transformation between the controllable and the world frame. In the case the respective control task is not employed in a subspace (i.e. no subspace filter is used), the control task is fully employed in the complete and unconstrained space (${}^{CF}\mathbf{S}_m = \mathbf{I}$), resulting in $\mathbf{P} = \mathbf{I}$. While the subspace projection for control tasks in the unconstrained space is denoted as

$$\mathbf{P} = \mathbf{I} - {}^{W}\mathbf{J}_c{}^{+}\ {}^{W}\mathbf{J}_c \text{ , cf. Equation 2.15 ,} \tag{7.7}$$

$\mathbf{I} - \mathbf{P}$ defines the projection for control tasks in the constraint space. In addition to that, the constraint consistent joint-space inertia matrix is defined as

$$\mathbf{M}_c = \mathbf{P}\mathbf{M} + \mathbf{I} - \mathbf{P} \,. \tag{7.8}$$

For each closed VKC (i.e. VM), the Jacobian for the task, constraint, and internal wrench, based on Equation 5.6, are provided analogous to non-VM manipulators.

>_ *virtual kinematic chain*

Figure 7.4 shows the internal distribution and stacking of the quantities in one specific instance of a configured task component. The input quantities provided by the robot state in the reference architecture (see Figure 7.1) is separated by the involved kinematic chains. In the case a VM is present, the quantities from the associated chains are stacked and forwarded to the VM chain. In the next step, all chains forward their quantities to the individual control tasks, where the filters and other modifications are applied. In the last

3 The superscript ${}^{W}(\ )$ and the target frame CF may be added or neglected for readability.

step, the modified quantities are stacked per control formalism to create the right output for the instantiated control components in the system.

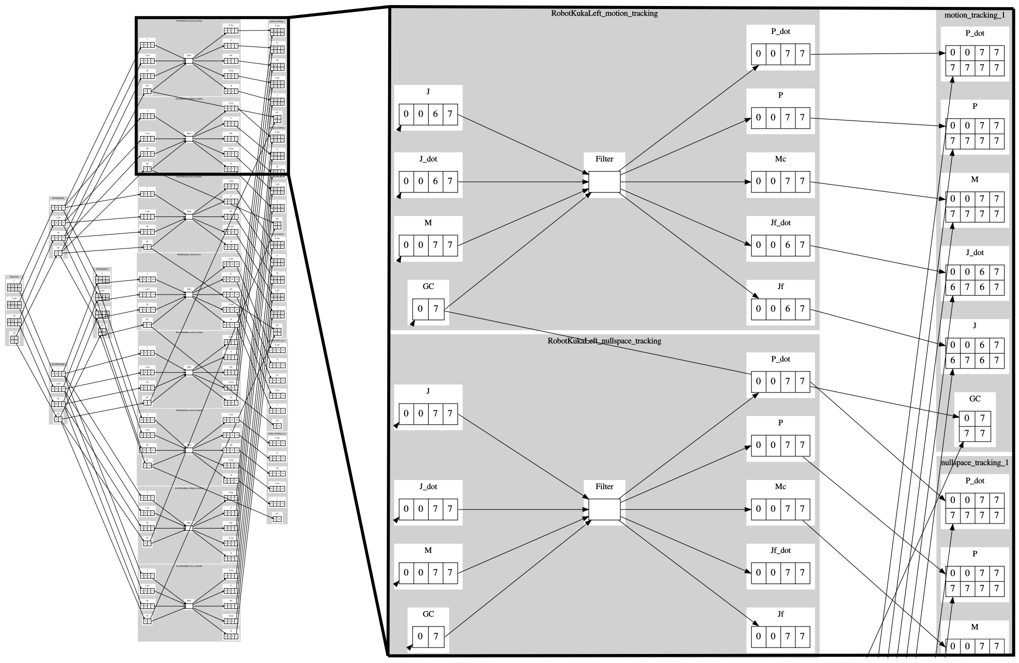


Figure 7.4: Example introspection of a configured task component. The right part shows a zoomed-in region of the graph on the left. The received quantities are separated per kinematic chain and distributed per control task, modified, and then stacked per control formalism to be sent to the instantiated control components. The position and dimension of a quantity in a stack is indicated by the numbers in the respective grid row.

7.3.3 *Prioritization Component*

The PIDC-specific prioritization component uses a continuous prioritization approach [Deh18] to change the prioritization of control tasks by smoothly blending between the PSs, interpolating $\boldsymbol{\Psi}_{\mathtt{current}}$ and $\boldsymbol{\Psi}_{\mathtt{new}}$. The task component is configured with the synthesized PS graphs (i.e. G_{i,K_r}) for each CS that are represented per kinematic chain K_r as a matrix $\boldsymbol{\Psi}_{i,K_r} = \mathrm{Adj}(G_{i,K_r})$ (see Equation 7.2) in the configuration artifact. Here, $\mathrm{Adj}(\cdot)$ denotes the adjacency matrix of a graph. This means that upon a trigger, a smooth traversal of the contact state graph from one CS to another is possible. In the special case of using a VM, the resulting control signal of the prioritized control tasks, associated with the kinematic chains that form the VM, is inserted as a virtual control task ($\boldsymbol{\tau}_K$) into the prioritization graph of the VM (see purple step in Algorithm 7.2). This enables the smooth blending between a closed VKC and the individual use of the involved robots.

7.3.4 *Contact Switching Service*

The contact switching service requires the information of the individual prioritization graphs $\{G_{i,K_r}\}$ to command a contact situation transition. Upon an internal or external trigger (i.e. from the task coordination), the task and prioritization components are simultaneously commanded to execute a smooth transition. Analogously, the control components are commanded to change the parameterization of their formalisms accordingly. By using the developed PIDC implementation, the prioritization matrices $\{\boldsymbol{\Psi}_{i,K_r}\}$ can either be transitioned instantaneously or linearly using a scalar transition factor.

The OROCOS RTT-specific implementation of this service loads the required information from the generated YAML configuration artifact. In addition to that, the service is able to load a separately modeled task-level coordination in form of an OSD to trigger transitions based on the coordination.

>_ *OROCOS State Description*

7.4 CONCLUSION

This chapter presented the synthesis of an executable software control system from a modeled CI task. A (control) framework independent reference architecture is introduced, which acts as a blueprint to be adapted and configured to realize a modeled CI task. The reference architecture together with the task model define the input for the presented generator pipelines. One part of the developed model transformations populates the reference architecture with the required control components for the chosen software execution framework (i.e. OROCOS RTT) and the control framework (i.e. PIDC). Another part of the transformations generates a configuration artifact that is independent of the component-based architecture. The artifact is used to infuse the CI task specific knowledge into the individual components of the system to provide the task-related quantities. To this end, the PIDC-specific implementation of the components for the execution with OROCOS RTT is configured with a framework-specific interpretation of the information persisted in the configuration artifact. In addition to the transformations introduced in this chapter, the synthesis pipeline reuses already existing transformations to generate an executable artifact of the system model for OROCOS RTT in form of an OPS.

This chapter addresses RQ 4 “How to link the task to the control system?” by synthesizing a control framework-specific architecture as part of the system model that is tailored to the modeled CI task via a generated configuration artifact. RQ 1 “How to use MDE to compose the robotics concerns?” is addressed by the developed generator modules (i.e. *CI System Generator* and *CI Config Generator*) which are integrated into CoSiMA’s generator composition to form new transformation pipelines. By reusing the *Component DSL* to express the structural model of the control architecture, CoSiMA’s LM&C mechanisms are also reused to allow the composition with other system-related concerns. Together, a link is established from the task description (on the L_2 *Compliant Interaction* level) over the control architecture model (on the L_1 *Robot Control System* level) to the executable robotic system (on the L_0 *Real Soft. System* level in Figure 3.1). This link is used to enforce the task-related constraints via the control architecture on the robot's behavior. As a result, the explainability and predictability is significantly increased, leading to a reduction of the gap between the envisioned task and the robot's behavior (see Section 1.2).

Part III

PERSPECTIVES

This part takes a global perspective on CoSiMA. The following Chapter 8 on page 141 gives an experimental evaluation of three representative CI applications, modeled and executed in CoSiMA. Based on these experiments an overview of CI tasks that can be modeled and realized is presented to answer RQ 5. Chapter 9 on page 159 is the last chapter of this thesis. It summarizes the advantages and limitations of the presented approach. This is followed by an outlook of possible future work and extensions to CoSiMA.

“ *It doesn’t make a difference how beautiful your guess is. It doesn’t make a difference how smart you are, who made the guess, or what his name is. If it disagrees with experiment, it’s wrong.* ”

— Richard P. Feynman

8 EVALUATION

This chapter showcases three different scenarios as case studies that are modeled and executed with CoSiMA. The focus of each of the three scenarios is on the realization of different characteristics, such as the use of multiple CSs, a VM, and the loose coupling between the task aspects (i.e. CI, trajectories, and coordination). Followed by an investigation of the effects of the different tasks on the synthesized control architecture, this chapter closes with a discussion on the currently supported CI constraints and CoSiMA’s scalability to other control and execution frameworks. This chapter is based on [WDW20; Wig+17a; Deh+18; Deh+ss].

The previous chapters introduced the modeling and generation of robotic systems for CI tasks using CoSiMA. This chapter chooses three different scenarios as case studies, which are entirely realized with CoSiMA, to answer RQ 5. Each case study focuses on a relevant characteristic of CI tasks:

1. Tetra-Arm Object Handling: *multiple constraints forming a VM*.
2. Dual-Arm Yoga Mat Rolling: *transitioning between multiple CSs*.
3. Single-Arm Clamp Assembly: *interplay between CI and task coord*.

The scenarios are realized using PIDC and OROCOS RTT as execution environment. They are executed in simulation as well as on the real robot using a generated system with the same synthesized control architecture, where only the interface component for the robot is exchanged (i.e. sim or real). In the following, each scenario displays its coordination states using images from the actual execution on the real hardware, while the modeled CSs are visualized in the corresponding simulated environment.

8.1 TETRA-ARM OBJECT HANDLING

This first scenario is introduced to showcase the ability of CoSiMA to model and simultaneously control multiple robots to perform a task together. The experimental setup can be seen in Figure 8.1. The focus of this case study is on the VM that is formed by four KUKA LWR 4+[1] to manipulate a 9.2 kg octagonal prism. Our aim was to create (1) an impedance behavior between the prism and a virtual frame that follows a (circular) path as well as (2) a compliant behavior that allows human interaction. To achieve both kinds of

Virtual Manipulator

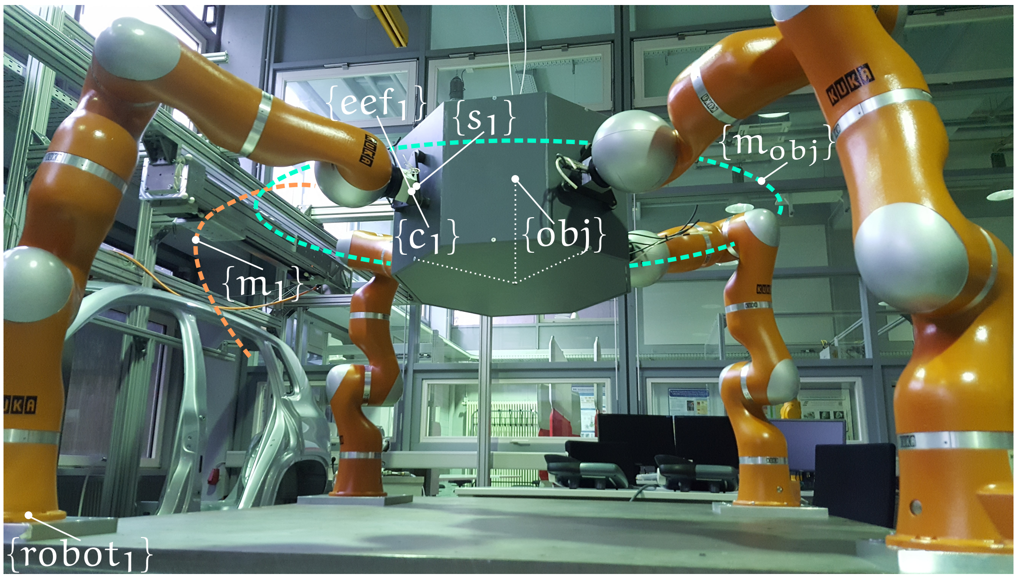


Figure 8.1: The university of Braunschweig, Edinburgh, and Bielefeld collaborated to set up the tetra-arm scenario at the institute of robotics and process control. Bielefeld and Braunschweig each provided two KUKA LWR 4+ for the experiment. The setup is overlaid with the relevant frames and the individual (orange, robot$_1$) and VM (green) trajectories.

interaction, a VM is created in an object-centric way to turn the octagonal prism into a controllable entity. Thereby, combining and treating the individual robots as a single one. The VM is able to distribute the applied forces over the combined DoFs and to compensate for the modeled weight of the object (see Figure 8.4b).

Contact Situation

The scenario is divided into 3 CSs, which are switched by the associated state transition in the task coordination (see Figure 8.2). To model the CSs, a world model involving the physical and virtual entities (i.e. robots, objects, and frames) needs to be specified first (see Figure A.5). The relevant frames for the prism object and for one of the four robots are depicted in Figure 8.1 along with the trajectories, which are indicated in orange and green. As a second step, the constraints for each CS are modeled using the previously defined physical and virtual entities. CS1 describes a single MSD constraint between the controllable frame $\underline{\{c_i\}}$ at the EEF of each robot and the respective frame $\underline{\{m_i\}}$ that follows an individual trajectory (orange) to approach the desired contact location on the object's surface $\underline{\{s_i\}}$. This CS is used as context for the initial state of the task coordination (*Individual Approach*). Once each robot has reached its contact point, a transition to CS2 is triggered by the coordination. CS2 models the octagonal object as a VM, by introducing bilateral surface-to-surface contact

mass-spring-damper

in the text, frames are marked with an underline $\underline{\{\cdot\}}$ to avoid confusion, see Notation on page xv

1 The scenario was jointly achieved by the universities of Edinburgh, Braunschweig, Bielefeld.

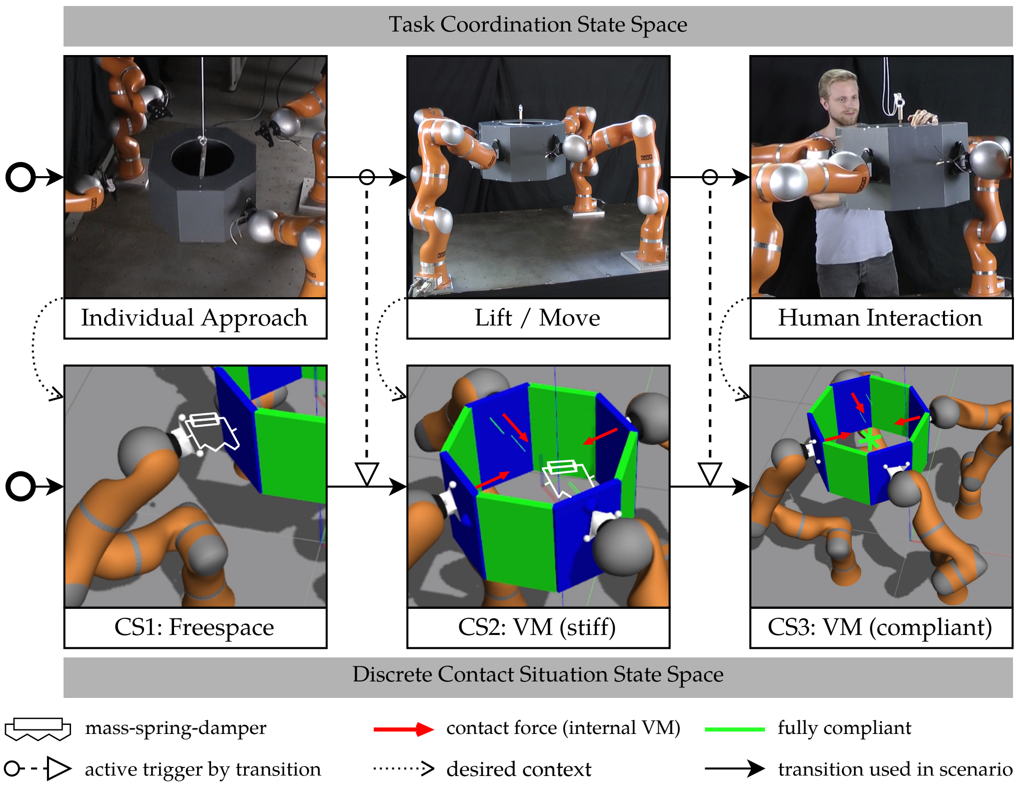

Task Coordination State Space
Individual Approach
Lift / Move
Human Interaction
CS1: Freespace
CS2: VM (stiff)
CS3: VM (compliant)
Discrete Contact Situation State Space
mass-spring-damper
contact force (internal VM)
fully compliant
active trigger by transition
desired context
transition used in scenario

Figure 8.2: The states of the task coordination (top row) relate to the contact situations (bottom row), in which they are executed (curved dashed arrows). Here, contact state transitions are triggered by a transition in the coordination (round-tail arrow). The main constraints in each CS are visually overlaid for readability.

constraints at the four contact points (i.e. $\underline{\{s_1\}}$, $\underline{\{s_2\}}$, $\underline{\{s_3\}}$, $\underline{\{s_4\}}$), which constrains the translational Z and the rotational X as well as Y direction of the robots' controllable frames ($\underline{\{c_i\}}$). Triangular metal plates with slip-resistant rubber domes at each corner are used as EEFs to establish a force-closure grasp. Thus, an active control of the internal forces is required to maintain a rigid connection. A VM constraint is modeled, based on the four contact constraints, which approximates the octagonal prism by a cube. Of course more complex geometries can be described using the URDF-based abstractions supported in CoSiMA. Here, the modeled VM constraint also defines a heuristically chosen contact force that is applied as internal force towards the center of all contacts. With this constraint the object's center $\underline{\{obj\}}$ becomes controllable. For CS2 a MSD constraint is modeled between $\underline{\{obj\}}$ and a virtual frame $\underline{\{m_{obj}\}}$, enforcing the robotic system to realize an impedance behavior while compensating for the objects weight and inertial properties of the entire virtual manipulator. Parallel to switching from CS1 to CS2, the coordination state is changed to *Lift / Move*, which moves $\underline{\{m_{obj}\}}$ along a circular trajectory (green). Note, the modeled constraints for CS2 can be seen in Figure A.6. Upon changing the coordination state to *Human Interaction*, CS3 is activated. CS3 is very similar to CS2. Instead of the MSD constraint, a force[2] constraint is modeled for the 6 Cartesian directions of the VM and zero forces are commanded. As a result, the robotic system's behavior changes from following a circular trajectory to solely compensating for the modeled weight of the object. This means that the system is not capable to withstand any external forces (except for gravity) and the object can be compliantly moved by the human.

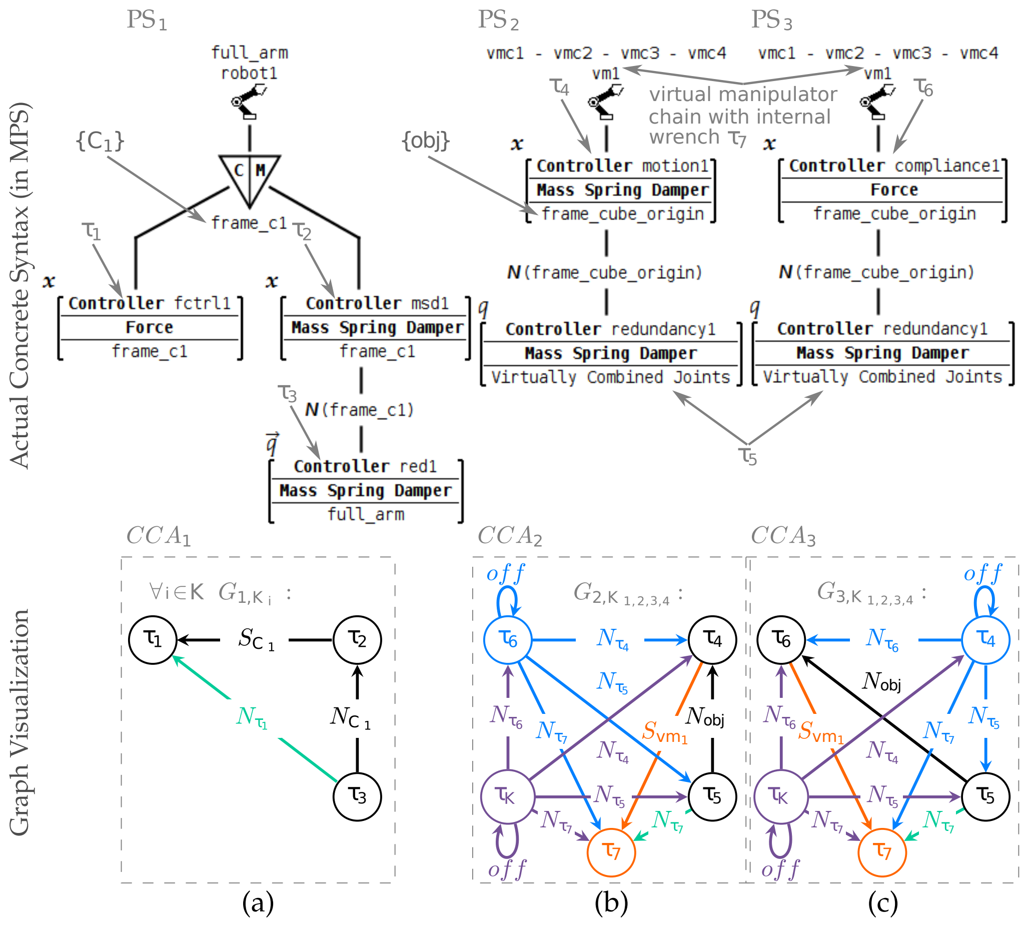


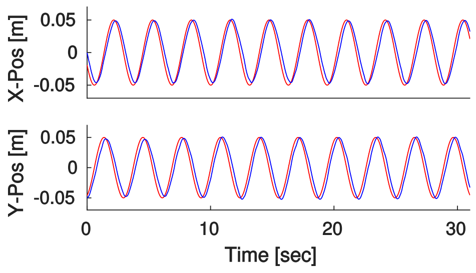
Figure 8.3: Overview of the prioritization models. (a) shows the individual constraint prioritization for each of the 4 robots. (b) employs a Cartesian MSD control task on the VM and a lower prioritized joint-space redundancy resolution control task on the combined joints of the VM. (c) is similar to (b) except for the Cartesian force control task to achieve total compliance (including gravity compensation).

To properly define the behavior in a contact situation, the modeled constraints need to be prioritized. Figure 8.3 shows the prioritization structures needed to switch between individually controlled manipulators and the creation of the VM. The first PS in the top row (a) shows the prioritization for a single robotic manipulator (here: robot1) in contact with the octagonal object. In this case, maintaining the contact force (τ_1) is higher prioritized than the Cartesian motion control (τ_2). On the lowest priority level, an additional joint-space MSD is realized to handle the redundancy resolution (τ_3). In contrast to

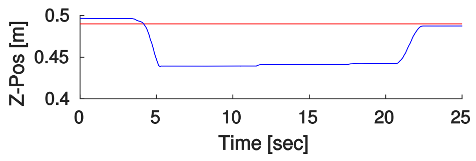
prioritization structure

2 Note, a contact constraint is not needed, since the Cartesian space does not need to be divided.

(a), (b) and (c) define the prioritization based on the combined kinematic chain of the VM (vm_1) that is formed by the modeled contact constraints (vmc_i). The bottom row of Figure 8.3 visualizes the prioritization graphs, which are synthesized from the PSs. Switching between the individually controlled robots (a) to the VM (b or c) is done by introducing the result of the first PS into the second or third PS (denoted as τ_K). Thus, allowing a smooth transition back and forth.



(a) Circular motion tracking



(b) Additional weight compensation

Figure 8.4: Both plots show the desired position (red) plotted against the actual position (blue) of the VM (i.e. octagonal object). (a) and (b) refer to CS2 in Figure 8.2. While (a) shows the tracking of a circular motion, (b) shows the adaptation to an online change in the object's mass. We published both figures in [Deh+18].

Figure 8.4 shows the actual behavior of the robotic system in terms of (a) tracking the circular trajectory by the VM and (b) compensating for the modeled weight of the octagonal object in the world model. Since the tracking and compensation error in both cases is very small, it proves that the modeled VM was properly established. To the best of my knowledge, this is the first experiment, treating more than three industrial manipulators in a cooperative manner for dexterous object manipulation using a force-closed grasp. More detail on the execution of this scenario can be found in [Wig+17a; Deh+18].[3]

8.2 DUAL-ARM YOGA MAT ROLLING

The bi-manual rolling of a yoga mat is chosen to investigate the transition between several CSs, where a task coordination state can be executed in different CSs, depending on predefined conditions. A photo of the experimental setup can be seen in Figure 8.5. In contrast to the first case study, there is no one-to-one mapping between the CSs and the coordination states. Depending on the evaluation of predefined conditions by the task coordination, using the information provided by the modeled monitors (see Figure 8.7), the CSs are switched and used as a context for the execution of the currently active coordination state.

Figure 8.6 depicts the task coordination and the CSs. Rolling the yoga mat a single time requires sequencing the *(Re)Grasp Left*, *(Re)Grasp Right*, and *Rotate Forward* states. Depending on the size of the yoga mat, the optional state *Pull*

3 A video of the tetra-arm experiment can be found at `https://youtu.be/HPNOChspXcM`

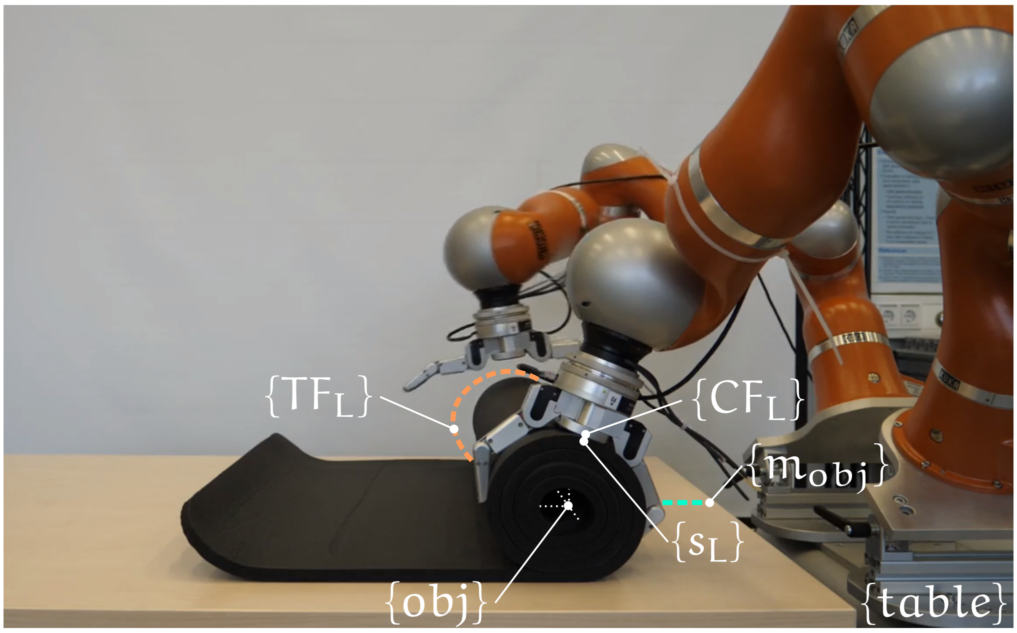


Figure 8.5: Dual-arm yoga mat rolling experiment, overlaid with the relevant frames and the individual (orange) and VM (green) trajectories. The two KUKA LWR 4+ robots each use a Barrett BH8-282 gripper as EEF, which includes a F/T sensor in the wrist.

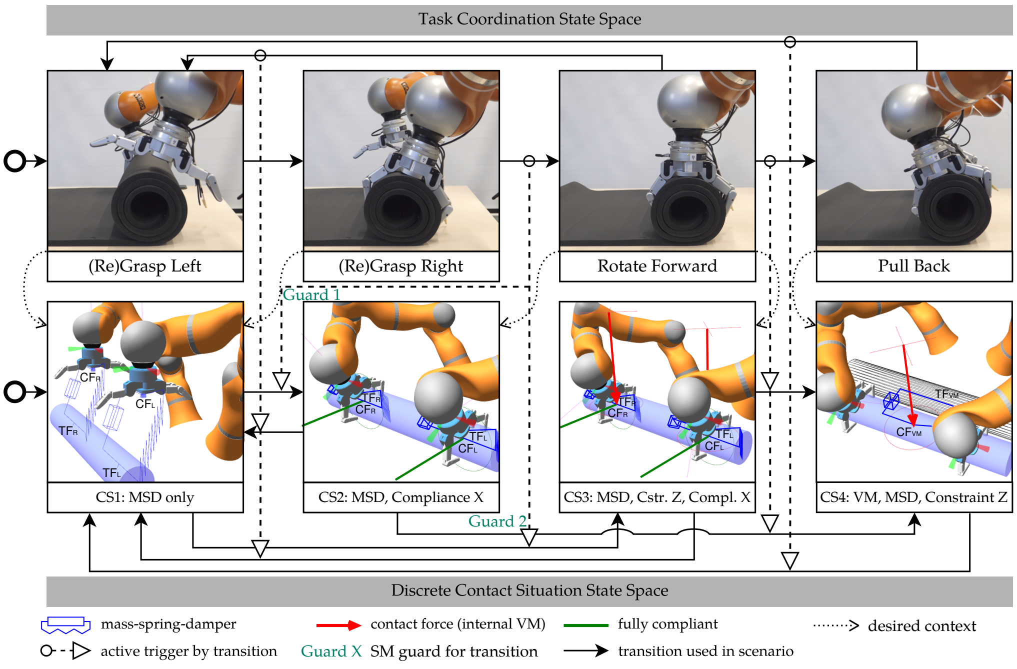
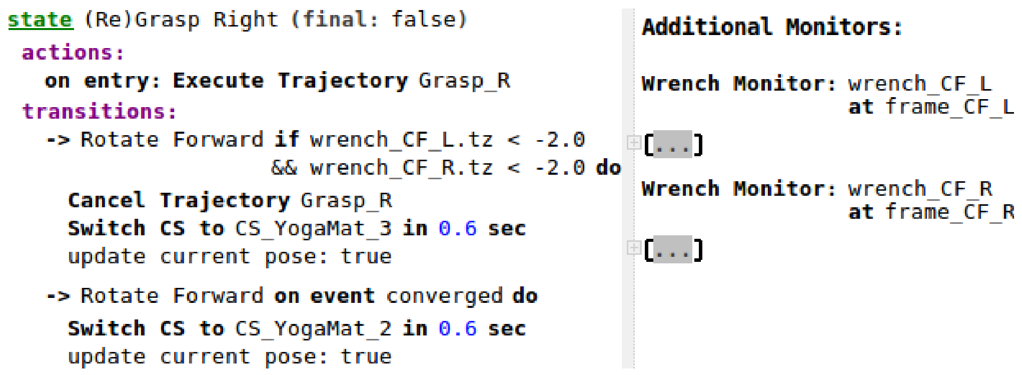

Task Coordination State Space
(Re)Grasp Left
(Re)Grasp Right
Rotate Forward
Pull Back
Guard 1
Guard 2
CF_R
CF_L
TF_R
TF_L
TF_VM
CF_VM
CS1: MSD only
CS2: MSD, Compliance X
CS3: MSD, Cstr. Z, Compl. X
CS4: VM, MSD, Constraint Z
Discrete Contact Situation State Space
mass-spring-damper
contact force (internal VM)
fully compliant
desired context
active trigger by transition
Guard X SM guard for transition
transition used in scenario

Figure 8.6: The states of the task coordination (top row) relate to the contact situations (bottom row), in which they are executed (curved dashed arrows). Here, contact state transitions are triggered by a transition in the coordination (round-tail arrow). The main constraints in each CS are visually overlaid for readability.

Back needs to be used to avoid violating the limits of the robots' workspace. The scenario begins with the left hand approaching and grasping the mat, which is initially pre-rolled a single time. This state is always executed in CS1, which models a MSD constraint between each hand $\underline{\{CF_i\}}$ and the target frames $\underline{\{TF_i\}}$. Depending on the state, a trajectory is executed (e.g., $\mathtt{Grasp_i}$) that moves the target frame to the minimal distance from the table, where the fingers can be safely closed. Predefined wrench monitors are used to abort the trajectory and stop the movement of the hand once a certain resistance (i.e. 2N), caused by a contact with the mat, is perceived. Once the left hand is in contact, the coordination is changed to *(Re)Grasp Right*, which performs the same behavior for the right hand. Once the two hands are in contact with the mat, a transition is triggered to *Rotate Forward*. Depending on the perceived wrench by the F/T sensors, the contact situation is switched to CS2 or CS3. If the mat is not yet thick enough, CS2 is chosen to prevent the hands from pressing into the table. Otherwise, CS3 is chosen to compress the mat and compensate for the increasing radius while rolling. In both cases a contact constraint at frame $\underline{\{CF_i\}}$ is modeled that constrains the rolling direction (i.e. X) to ensure a compliant forward motion guided by the mat. This prevents the commanded joint torques to exceed their safety limits, due to the friction forces between the material of the mat and the surface of the table. In addition to that, CS3 also constrains the Z direction and models a force constraint that applies a constant force to compress the mat. An excerpt of the data collected from the experiment is shown in Figure 8.8. For illustrative reasons, a segment is chosen where the mat is thick enough to show CS3. Once the forward rotation is finished, the new position of the robots, caused by the compliant forward motion, is used to update the current position for the next motion command. The next state *Pull Back* is used to avoid reaching the limits of the robots' workspace. In CS4, a VM for both manipulators is modeled that reflects the geometry of a cylinder, creating the controllable frame $\underline{\{obj\}}$. While the mat is pulled back using a MSD constraint ($\underline{\{m_{obj}\}}$), a contact constraint in the world's Z direction prevents the VM's trajectory to lift the mat. A small force ($\boldsymbol{\lambda}_{f,z} = -2$ N) is applied to ensure staying in contact and to compensate for disturbances that arise from dragging on an uneven surface.

>_ *mass-spring-damper*


```
state (Re)Grasp Right (final: false)
 actions:
   on entry: Execute Trajectory Grasp_R
 transitions:
   -> Rotate Forward if wrench_CF_L.tz < -2.0
                     && wrench_CF_R.tz < -2.0 do
     Cancel Trajectory Grasp_R
     Switch CS to CS_YogaMat_3 in 0.6 sec
     update current pose: true

   -> Rotate Forward on event converged do
     Switch CS to CS_YogaMat_2 in 0.6 sec
     update current pose: true
```
```
Additional Monitors:

Wrench Monitor: wrench_CF_L
                at frame_CF_L
[...]
Wrench Monitor: wrench_CF_R
                at frame_CF_R
[...]
```

Figure 8.7: A state machine coordination model (left) is able to use predefined monitors (right) to evaluate conditions for coordination state transitions that act as guards. Once the respective condition is met and the transition is triggered, the actions defined below are executed (e.g., switching the contact situation).

Having a look at Figure 8.8, it can be seen that the resulting behavior matches the modeled one. Further, it can be seen that a smooth traversal between the modeled contact situations is possible. Hence, the modeling of CI tasks that involve multiple CSs to achieve their goal is supported. By leveraging combinations of motion and contact constraints, the interaction with the environment can be treated to the task's advantage. In this case study, a desired compliance is modeled, which enables the manipulator to adapt to the radius of the mat that increases with every rolling cycle. This scenario also shows that CSs and coordination states are loosely coupled, since a CS can be considered as a context in which a coordination state or a trajectory is executed. Hence, reusability of CSs and coordination states can be achieved. More details on this experiment can be found in [WDW20].[4]

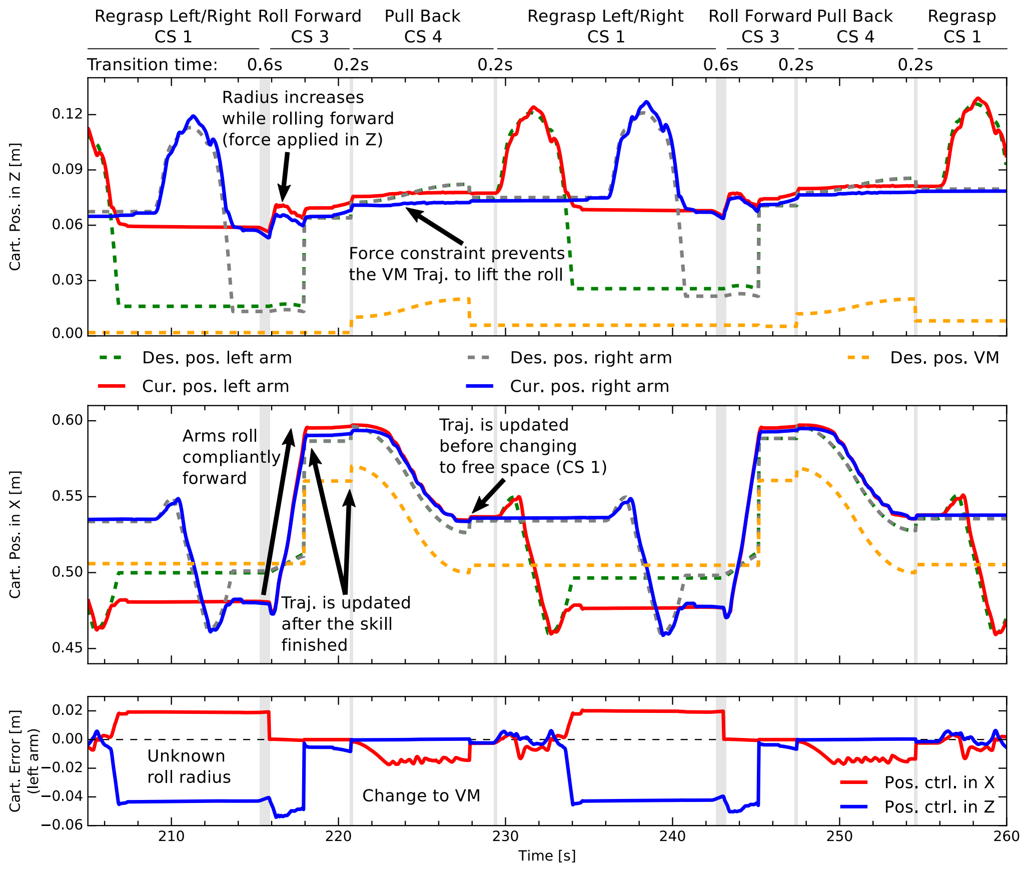


Figure 8.8: Recorded data of the yoga mat rolling experiment. Textual annotations are added to interpret the data. The top row indicates the active task coordination state and the row below the active contact situation. Note that the bottom plot has a different legend compared to the plots above.

4 A video of the dual-arm experiment can be found at https://youtu.be/c2shW903Eo4

8.3 SINGLE-ARM CLAMP ASSEMBLY

The third scenario targets the terminal assembly domain by realizing the single-arm snap-fitting of electrical clamps onto a rail. A photo of the experimental setup can be seen in Figure 8.9.

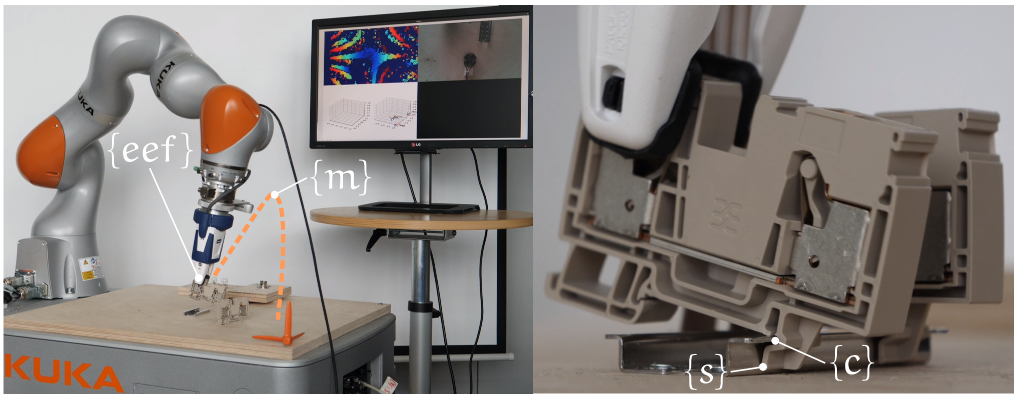


Figure 8.9: Single-arm clamp assembly experiment, overlaid with the relevant frames. An exemplary trajectory is indicated in orange. The scenario relies on the internal F/T estimation of the KUKA IIWA14 for the assembly step and on an Intel RealSense D415 in combination with a CNN-based detection of the clamps to calculate the grasping point.

The previous case study was focused on the loose coupling between the CSs and the task coordination, and how a CS can be activated by the coordination to achieve a desired task. The aim of this experiment is to investigate the benefits of separating both aspects further. Instead of enforcing a CS by the task coordination, the activation of a CS is here only based on the actual state of the interaction with the environment.

In this case study the CSs are modeled to represent the compliant interactions that are likely to occur during the snap-fitting process. As it can be seen in Figure 8.10, there are no triggers from the coordination state transitions to the CS transitions (indicated by an arrow with round tail and triangular head). Instead, the triggers are inverted. This means that the coordination is notified when a CS is switched. Every assembly of a clamp begins in free-space (i.e. CS1). The *Approach Table* state asks a non-RT Convolutional Neural Network (CNN)-based component to detect a clamp on the table and to provide a valid grasping point to a path planning component. The resulting trajectory is then executed using the modeled MSD constraint in CS1 to grasp and lift the clamp, before approaching a location on the table, which is close to the rail. Upon sensing a contact with the table (see Figure 8.12) the CS is changed to CS2. Thus, ensuring the contact with the table using a constant force of 3.8N. The next coordination state *Approach Rail* drags the clamp towards the rail. Upon reaching a contact with the rail (cf. Guard2) CS3 is activated, which makes the Z direction completely compliant, allowing the edge of the rail to align the position of the clamp. In the Y direction, the contact with the rail is maintained by modeling a contact constraint with a contact force of 10N. To assemble the clamp, the *Snap-Fit Clamp* state rotates the clamp until it snaps onto the rail.

>_ *Convolutional Neural Network (CNN)*

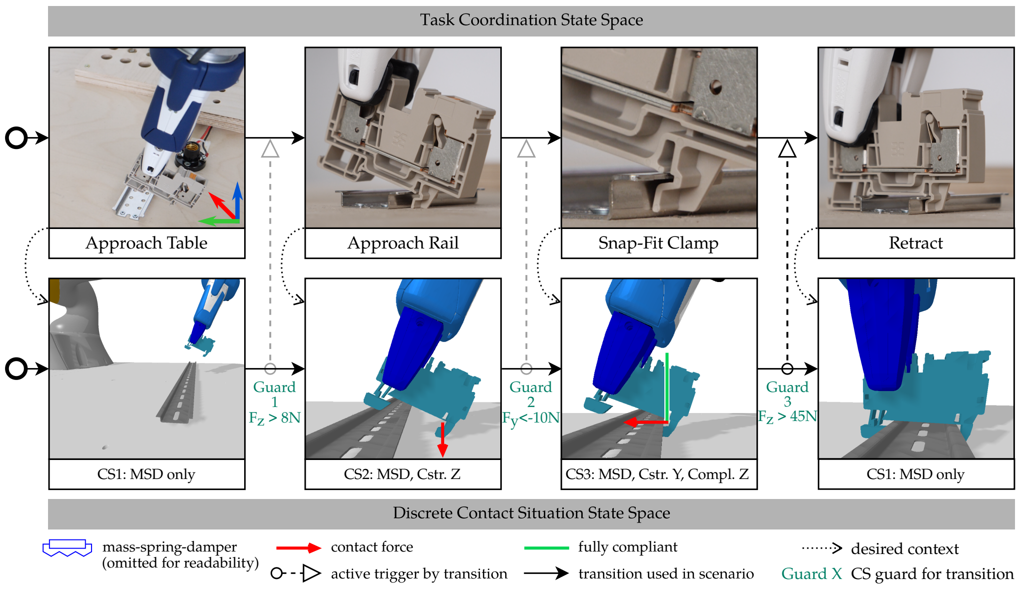

Task Coordination State Space
Approach Table
Approach Rail
Snap-Fit Clamp
Retract
Guard 1 $F_z > 8N$
Guard 2 $F_y < -10N$
Guard 3 $F_z > 45N$
CS1: MSD only
CS2: MSD, Cstr. Z
CS3: MSD, Cstr. Y, Compl. Z
CS1: MSD only
Discrete Contact Situation State Space
mass-spring-damper (omitted for readability)
contact force
fully compliant
desired context
active trigger by transition
transition used in scenario
Guard X CS guard for transition

Figure 8.10: The states of the task coordination (top row) relate to the contact situations (bottom row), in which they are executed (curved dashed arrows). Here, contact state transitions are triggered by a transition in the coordination (round-tail arrow). The main constraints in each CS are visually overlaid for readability. The coordinate system is shown at the top left.

After the coordination is notified by the change from CS3 back to CS1 due to a predefined sensed force in the Z direction that detects the snap (cf. Guard3), the coordination state is changed to *Retract*. This state lifts the arm again and initiates the assembly of the next clamp.

disadvantage of implicitly modeling the task using only motion trajectories

One approach is to realize the task of assembling a clamp without modeling the different CSs. In that case, the challenge of describing the task is shifted to the motion trajectory, which is tracked using an impedance behavior. To achieve the desired contact forces to align the clamp with the rail and to perform the snap-fit motion, the trajectory needs to provide suitable set-points. These set-points have to be chosen so that the displacement, caused by a physical contact, results in the desired contact forces (see Figure 8.11, top row). However, this is not a favorable approach, since the successful execution heavily depends on the correct positioning of the set-points. The set-points in turn depend heavily on the correctness of the perceived locations of the clamp and the rail. Thus, offering only a minimal tolerance for uncertainties, which might lead to an unwanted behavior, giving rise to safety-critical forces. With this approach it is not possible to prioritize the maintaining of a contact over the execution of a motion. Further, the developer needs to have an in-depth understanding of the used impedance behavior to design the required trajectories that achieve the desired contact forces and motions to solve the task. Hence, making this a challenge that does not scale well to complex scenarios with multiple contacts.

advantage of explicitly modeling the task using contacts

In contrast to that, each CS in this case study is modeled with a guard that enables the automatic switching of CSs based on only i.e. sensed data (see Figure 8.12). Thereby, achieving a decoupling of the coordination and the motion

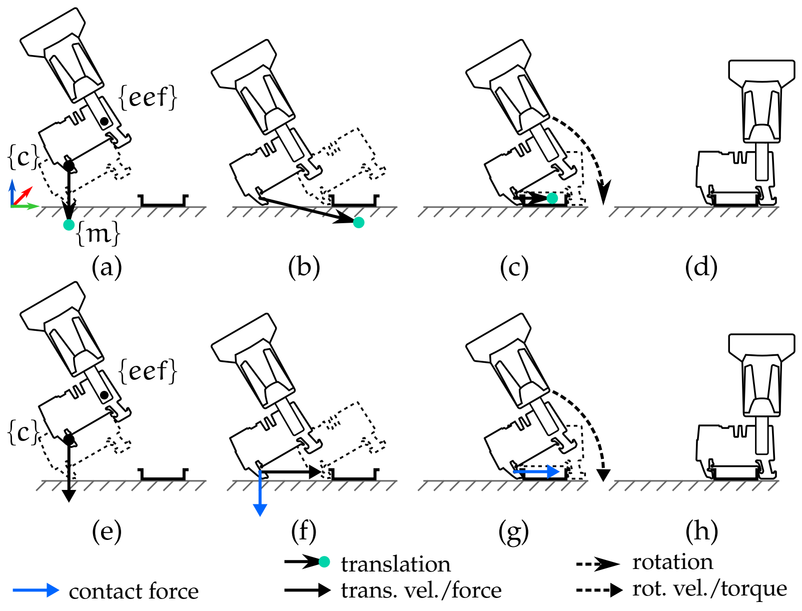
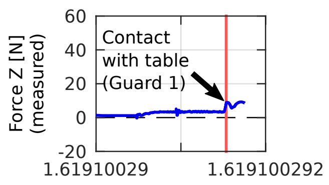
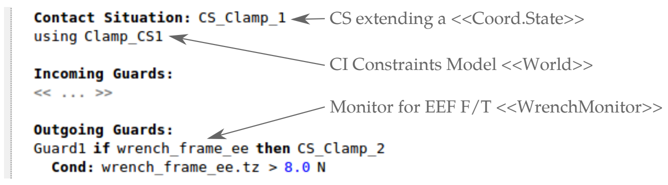


Figure 8.11: Realization of the scenario using only impedance control (top row) and using hybrid control (bottom row). In the first case, the responsibility of realizing the contact situations and i.e. upholding a certain contact force is shifted towards the trajectory. Hence, the programming of the trajectories is more challenging than in the second case, where the contact situations are enforced independent of the trajectories.




CS extending a <<Coord.State>>
CI Constraints Model <<World>>
Monitor for EEF F/T <<WrenchMonitor>>

```
Contact Situation: CS_Clamp_1
using Clamp_CS1

Incoming Guards:
<< ... >>

Outgoing Guards:
Guard1 if wrench_frame_ee then CS_Clamp_2
  Cond: wrench_frame_ee.tz > 8.0 N
```

Figure 8.12: Model excerpt showing the specification of a Guard (e.g., Guard1), which triggers a transition to another CS (e.g., CS2).

trajectories (see Figure 8.10, guards in bottom row). As a result, the CSs are responsible for achieving the desired interaction with the environment, providing a safe context for the execution of motion trajectories. This means that once in contact, the robot does not exert unwanted forces that are caused by a motion command, since the contact constraints are prioritized over the MSD constraints. Hence, increasing the tolerance for uncertainties. This approach further reduces the complexity of designing suitable trajectories, since they do not need to account for creating the desired contact forces anymore. Now, instead of precise position set-points, a suitable trajectory can consist of a combination of independent velocities and forces to achieve the task (see Figure 8.11, bottom row). Another benefit of decoupling the CSs from the task coordination, is the possibility to choose the granularity for the coordination in terms of states and trajectories to solve the assembly task. A developer for instance might benefit from a separation of the required trajectory into segments along multiple coordination states to foster exchangeability and reuse that leads to an increased comprehensibility and decrease of the design complexity. On the other end of the spectrum, a single state that executes a single trajectory can be used to achieve the task. The single trajectory might be derived from learning by demonstration [Arg+09], or generated via trajectory optimization. In both cases, the CSs enable an increased tolerance for uncertainties and provides a safe context for the execution of the chosen trajectories, as can be seen in Figure 8.13. The figure shows the snap-fitting of one clamp using the real hardware.[5]

Simple Cartesian trajectories are chosen for this case study that connect two positions linearly. In the figure it can be seen that in each CS, there is a deviation from the commanded trajectory that coincides with the directions, constrained by the modeled contact. Even though the deviations become quite large (e.g., in Z), the forces applied into the surface of the table remain at the desired value (i.e. 3.8N). This shows that the motion trajectories are not capable of influencing the behavior in a way that undesired force are exerted, avoiding damaging the clamp, the rail, or the table. By comparing the model of the CI constraints and the actual predefined forces in Figure A.7 with Figure 8.13, it can be seen that the executed behavior matches the modeled one.

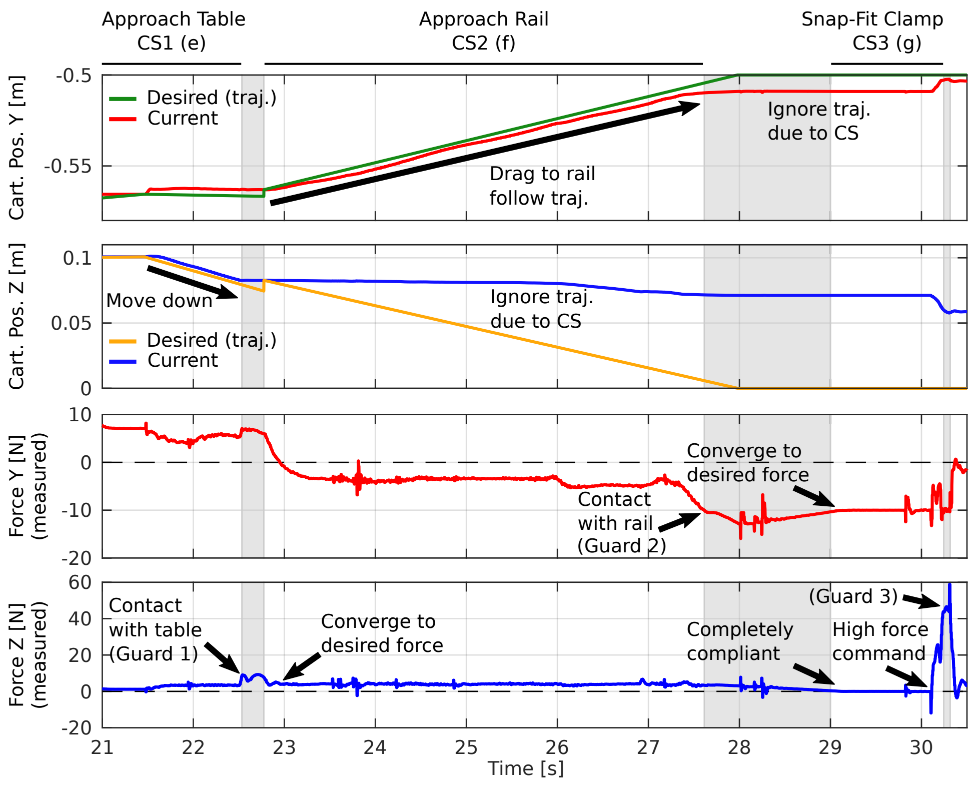


Figure 8.13: Shows the assembly of a grasped clamp. The reactivation of CS1 and the state *Retract* are omitted due to page restrictions. All quantities are shown based on the {eef} frame. It can be seen that the modeled contact forces are coupled to the interpolation of the CSs and the time allocated for the transitions (gray areas).

5 A video of the single-arm experiment can be found at `https://youtu.be/ciBBCnKsjuw`

8.4 CONCLUSION ON THE MODELING OF CI

The presented case studies show that the concepts, introduced in Chapter 6, are suitable to model and realize relevant tasks from the CI domain. Table 8.1 summarizes the constraints to model CI tasks that are currently supported in CoSiMA. The constraints are classified along their application type (i.e. behavior, scope, VM) and domain (i.e. geometrics, kinematics, dynamics). Behavior constraints define the general interaction between entities. Scope constraints set the bounds for the behavior. VM constraints allow the creation of a kinematic chain that is virtually closed on the dynamics level.

Constraint	Geometric	Kinematic	Dynamic	
Type			Motion	Force
Behavior	Geometric Relations (e.g., keep-parallel)		Mass-Spring-Damper (i.e. impedance)	Contact, Force (e.g., rigid-body, soft-body)
	Joint Types (e.g., revolute, prismatic)			
Scope	Joint Pos. Limits	Joint Vel. Limits	Joint Acc. Limits	Joint Force/Torque Limits
	Cart. Pos. Limits	Cart. Vel. Limits	Cart. Acc. Limits	Cart. Wrench Limits
VM	geometrically closed	kinematically closed	dynamically closed (incl. compensation for inertia)	

Table 8.1: Constraints for CI: supported (black) and currently unsupported (gray).

Not all the depicted constraints are prominently presented in the case studies. *Joint Types* for instance, geometrically, kinematically, and dynamically relate two robotic links to describe a desired behavior (e.g., prismatic or revolute). Joint constraints are not explicitly found in the CI task model, instead they are modeled as part of the kinematic chain or a robot platform. The robot platform for a specific robot, is then reused by the scenarios that use the same robot. Constraints, such as *Geometric Relations*, are currently not supported in CoSiMA (see Table 8.1, colored in gray). However, they are covered by related work on constraint-based programming, e.g., [BKB13; Ad14] and may be a fine addition to CoSiMA. Considering this limitation, tasks that require constraints to enforce a robot's EEF to always be parallel to a frame or another entity can currently not be modeled. As a result, a purely geometrically or kinematically closed VKC is not supported to form a controllable VM. Meaning that a VM cannot be established between entities that are not connected through physical contacts. Instead, CoSiMA supports the creation of VMs via dynamically closed chains through the use of physical contacts. With that, the VM can distribute the necessary forces to achieve a task over all involved joints. This would not be possible on the geometrics or kinematics level.

>_ *virtual kinematic chain*

The majority of the supported constraints can be represented by equality and inequality constraints. While the modeling environment already supports both kinds, the chosen PIDC implementation only supports equality constraints. While this is certainly a limiting factor of the approach, the solutions are not without downsides. A straight-forward solution would be to use a solver-based approach for the inequality constraints in combination with PIDC as suggested in [Deh+ss]. While this would enable the realization of e.g., friction constraints to optimize contact forces, joint limits would still remain a challenge. A second solution is to completely replace PIDC by a QP-based approach. However, depending on the number of constraints, it becomes more likely that the QP solver might not find a solution. In contrast to that, PIDC always returns a result, however without any guarantees regarding the fulfillment of the constraints. Further, PIDC is able to create and change prioritization structures on the fly, since it is based on projection matrices. Instead, a QP approach would need to leverage sophisticated strategies, e.g., [Kim+19; LTP15; Jar+13], to enable continuous task transitions without discontinuity. Note that this is still an open research question of its own. Generally, PIDC and QP-based SoT are suitable control frameworks for CI tasks, both having certain advantages and disadvantages. In fact, in the RobMoSyS [Rob16] ITP project CMCI [CMC20] we demonstrated a prototypical integration of the QP-based SoT framework OpenSoT [Roc+15] into CoSiMA and realized a wiping task similar to the example used in Section 6.2. Even though the preliminary results are promising, it is not in the focus of this thesis.

8.5 CONCLUSION ON THE EXECUTION OF CI

For each of the 3 case studies, an executable component-based control system is generated that uses OROCOS RTT as execution environment and PIDC as control framework. The control systems are synthesized from the CI model as described in Chapter 7 based on the developed reference architecture model. The first step of the entire generation process is the synthesis of the system's control architecture. As it can be seen in Figure 8.14, the models of the 3 scenarios only differ in the predefined extension points (green border) of the reference architecture, while the remaining part stays the same. In the next step, the synthesized model is enriched with model elements for other system concerns. This includes concerns such as the component coordination, safety, deployment, and timing. Since the execution time behavior is especially important for this thesis (see Chapter 4), the timing model for the experiments is analyzed as to what degree it is influenced by variations in the synthesized system model. As it can be seen in Figure 8.15, the sense-react chains for the scenarios only differ in the predefined extension points of the reference architecture (green border). The generated schedule shows a similar pattern (see Figure 8.16), although this might partially be a coincidence, since the solver for the schedule has no explicit knowledge about the structure of the reference architecture. However, due to the precedence and core constraints, the solution space for the schedule is drastically reduced, which leads to similar appearances for the 3 schedules. Figure A.8 depicts the data collected during one real execution cycle. It shows that the generated schedule model based on the modeled timing constraints is executable and upholds the precedence-perseverance property. Note that the implementation that executes the schedule is not optimized, since it is not the main focus of this thesis and there exist numerous works on that topic alone.

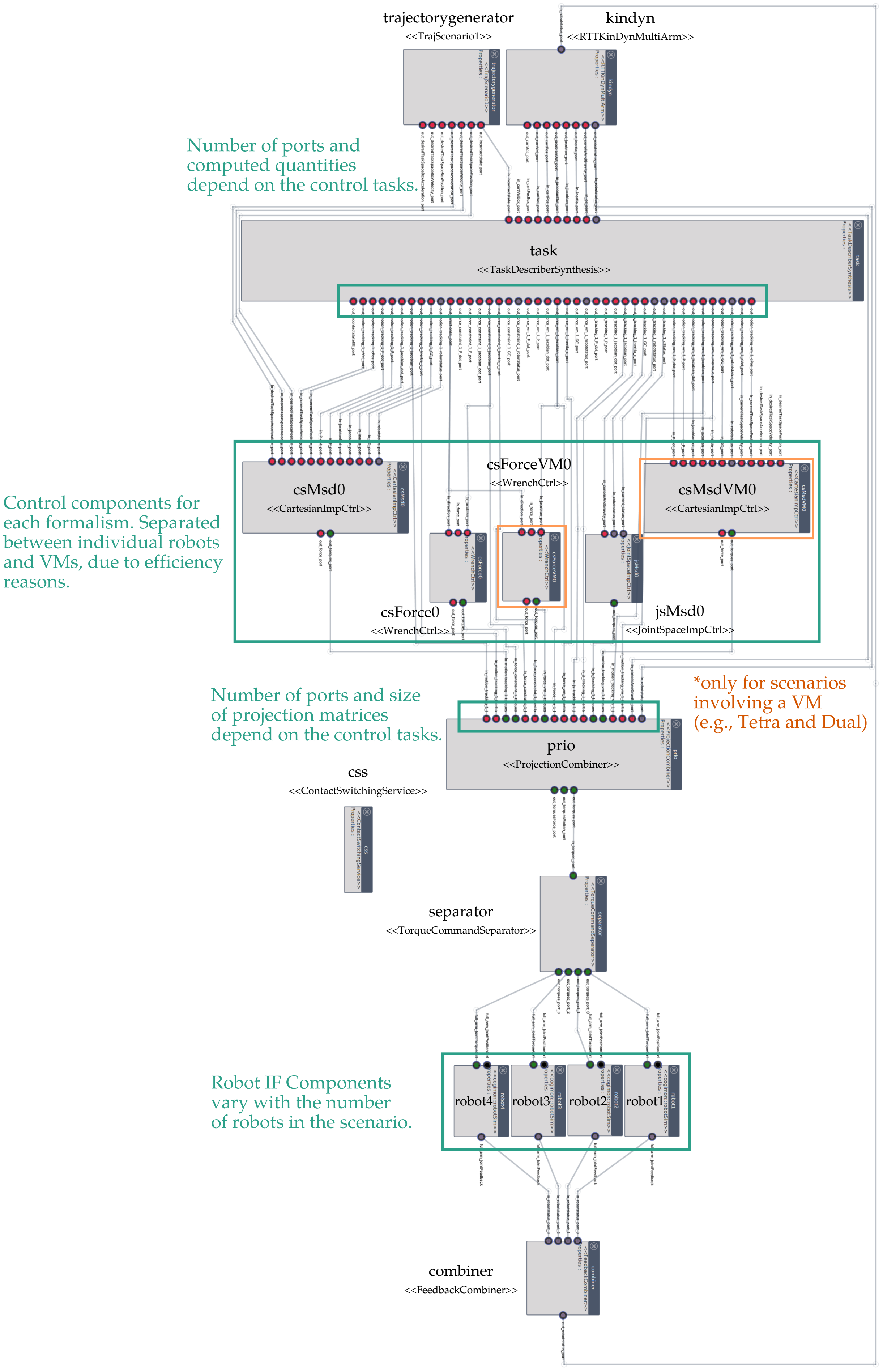
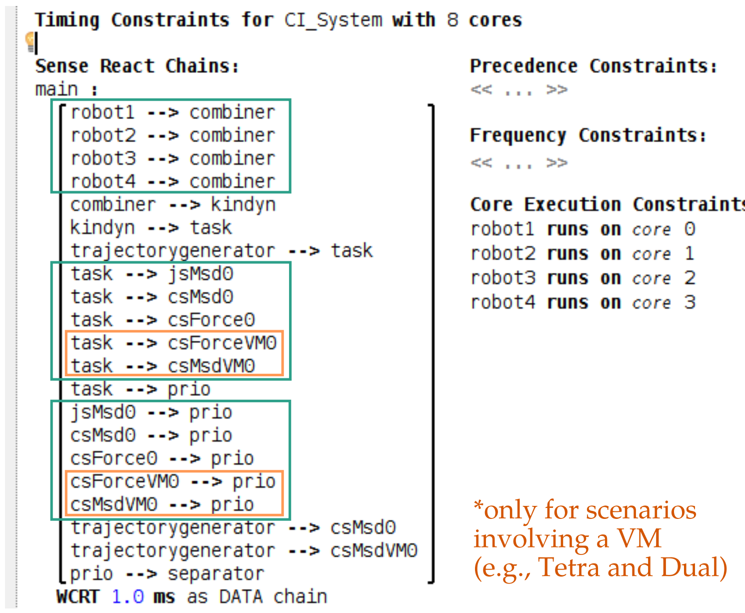

trajectorygenerator
<<TrajScenario1>>
kindyn
<<RTTKinDynMultiArm>>
Number of ports and computed quantities depend on the control tasks.
task
<<TaskDescriberSynthesis>>
Control components for each formalism. Separated between individual robots and VMs, due to efficiency reasons.
csMsd0
<<CartesianImpCtrl>>
csForceVM0
<<WrenchCtrl>>
csMsdVM0
<<CartesianImpCtrl>>
csForce0
<<WrenchCtrl>>
jsMsd0
<<JointSpaceImpCtrl>>
Number of ports and size of projection matrices depend on the control tasks.
*only for scenarios involving a VM (e.g., Tetra and Dual)
prio
<<ProjectionCombiner>>
css
<<ContactSwitchingService>>
separator
<<TorqueCommandSeparator>>
Robot IF Components vary with the number of robots in the scenario.
robot4
robot3
robot2
robot1
combiner
<<FeedbackCombiner>>

Figure 8.14: Screenshot of the complete system model, instantiated from the reference architecture. Variations points are marked by a green border.



```
Timing Constraints for CI_System with 8 cores

Sense React Chains:                          Precedence Constraints:
main :                                       << ... >>
  [robot1 --> combiner
   robot2 --> combiner                       Frequency Constraints:
   robot3 --> combiner                       << ... >>
   robot4 --> combiner
   combiner --> kindyn                       Core Execution Constraints:
   kindyn --> task                           robot1 runs on core 0
   trajectorygenerator --> task              robot2 runs on core 1
   task --> jsMsd0                           robot3 runs on core 2
   task --> csMsd0                           robot4 runs on core 3
   task --> csForce0
   task --> csForceVM0
   task --> csMsdVM0
   task --> prio
   jsMsd0 --> prio
   csMsd0 --> prio
   csForce0 --> prio
   csForceVM0 --> prio
   csMsdVM0 --> prio
   trajectorygenerator --> csMsd0
   trajectorygenerator --> csMsdVM0
   prio --> separator                ]
  WCRT 1.0 ms as DATA chain
```

*only for scenarios
involving a VM
(e.g., Tetra and Dual)

Figure 8.15: Timing constraints for the case studies with variation points (green).

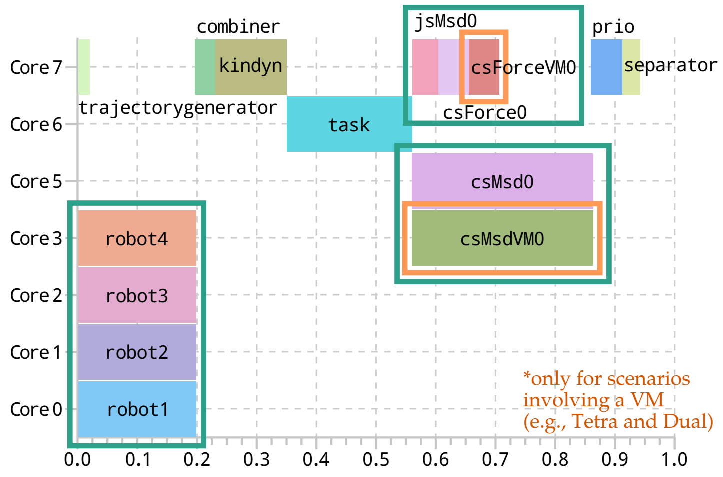


Figure 8.16: Schedule model generated from the timing constraints (see Figure A.9). Variations of the schedule based on the requirements of the case studies are marked in green.

prioritization structure

The main cause of the variations is grounded in the number of different control components that are instantiated for the particular case study. The control components are instantiated depending on the control tasks in the PS models (τ in e.g., Figure 8.3). As described in Chapter 7, control tasks are grouped by their control formalism, instantiating a single suitable control component per formalism. In these experiments, an additional separation between single robots and VMs is used for computation time benefits.

- jsMsd0 as joint-space imp. controller for the redundancy resolution.
- csMsd0 as an imp. controller to realize task-space motions.

- csForce0 as a direct force controller that constantly applies a desired force and handles the internal force in case of a VM.
- csMsdVM0 and csForceVM0 realize task-space VM motions and forces.

All control tasks using the same formalism are stacked and handled by the associated control component. Table 8.2 shows the mapping of control tasks to the control components for each case study.

Component	jsMsd0	csMsd0	csMsdVM0	csForce0	csForceVM0
Tetra	$\left[\begin{array}{c}\overset{\tau_{3,1}}{\mathtt{red1}_1}\\\hline \dots\\\hline \overset{\tau_{3,4}}{\mathtt{red1}_4}\\\hline \overset{\tau_5}{\mathtt{redundancy1}}\end{array}\right]$	$\left[\begin{array}{c}\overset{\tau_{2,1}}{\mathtt{msd1}_1}\\\hline \dots\\\hline \overset{\tau_{2,4}}{\mathtt{msd1}_4}\end{array}\right]$	$\left[\overset{\tau_4}{\mathtt{motion1}}\right]$	$\left[\begin{array}{c}\overset{\tau_{1,1}}{\mathtt{fctrl1}_1}\\\hline \dots\\\hline \overset{\tau_{1,4}}{\mathtt{fctrl1}_4}\\\hline \overset{\tau_{vm1}}{\mathtt{ifctrl1}}\end{array}\right]$	$\left[\overset{\tau_6}{\mathtt{compliance1}}\right]$
Dual	$\left[\begin{array}{c}\overset{\tau_{3,1}}{\mathtt{red1}_1}\\\hline \overset{\tau_{3,2}}{\mathtt{red1}_2}\\\hline \overset{\tau_5}{\mathtt{redundancy1}}\end{array}\right]$	$\left[\begin{array}{c}\overset{\tau_{2,1}}{\mathtt{msd1}_1}\\\hline \overset{\tau_{2,2}}{\mathtt{msd1}_2}\end{array}\right]$	$\left[\overset{\tau_4}{\mathtt{motion1}}\right]$	$\left[\begin{array}{c}\overset{\tau_{1,1}}{\mathtt{fctrl1}_1}\\\hline \overset{\tau_{1,2}}{\mathtt{fctrl1}_2}\\\hline \overset{\tau_{vm1}}{\mathtt{ifctrl1}}\end{array}\right]$	$\left[\overset{\tau_6}{\mathtt{compliance1}}\right]$
Single	$\left[\overset{\tau_3}{\mathtt{red1}}\right]$	$\left[\overset{\tau_2}{\mathtt{msd1}}\right]$	$\emptyset$	$\left[\overset{\tau_1}{\mathtt{fctrl1}}\right]$	$\emptyset$

Table 8.2: Mapping of the control tasks $\overset{\tau_{i,j}}{x_j}$ to the executable components per scenario, where $\tau_{i,j}$ refers to the control task instance i in the prioritization graph representation. x_j refers to the same control task, but represented as a controller in the respective PS model. If j is present, it indicates that the same control task is used for multiple kinematic chains (e.g., robot or VM). In that case, j denotes the index of the chain. The order of the control tasks shows the stacking as processed by the individual controller components (i.e. column headers).

For the experiments in this thesis, OROCOS RTT is used as execution environment and generation target for the control system model. It is possible to use another framework, such as ROS2.0 [OSR] or XBotCore [Mur+17]. If the framework supports the deterministic hard real-time execution and communication of multiple components, a one-to-one mapping between the modeled and the real software components is straight-forward. Thus, in that regard, the required code generator would be very similar to the one developed for OROCOS RTT. Further, the framework would need to be capable of realizing the modeled schedule. Any restrictions in that regard need to be lifted to the modeling level, preventing the design of an invalid schedule. For instance, if the real-time execution of components in parallel is not supported by the framework, this restriction needs to be reflected in the design space of the schedule. In general, the component executor in ROS2.0 is neither real-time capable nor deterministic [Cas+19]. However, an advanced executor is presented in [SLL20], overcoming this limitation by providing deterministic executions, subject to domain-specific requirements.

“ *Science is the process that takes us from confusion to understanding.* ”

— Brian Greene

9 CONCLUSION

In this thesis I have investigated how to bridge the gap between an envisioned compliant interaction task and the actual robot's behavior, by increasing the explainability and predictability through model-driven engineering. The gap is caused by the inability of explaining and predicting certain parts of the robot's behavior that are related to essential but often neglected concerns of the desired task and the robotic system that produces the behavior.

I argue that by explicitly modeling the individual concerns in a domain-specific way and by traceably composing them into a unified model that also describes their mutual interactions, the explainability and predictability can be significantly increased. Unfortunately, most of the current approaches fall short in these aspects.

The proposed approach offers a solution to this problem by using a model level to enable the domain-specific and explicit modeling of individual robotics concerns, such as the compliant interactions with the environment. Considering the current trend of robotics tasks in increasingly contact-rich situations, it is crucial to explicitly model the compliant interactions as part of the task description. Unfortunately, most of the current approaches neglect or hide the interaction, which potentially causes undesired and even dangerous situations. In contrast, the developed approach uses physically grounded abstractions that originated from the conducted domain analysis to enable the accurate modeling of the desired behavior (see RQ 3). These abstractions build the foundation to verify the exhibited robot's behavior.

In addition to task-related concerns (e.g., the compliant interaction), system-related concerns including software and hardware aspects need to be modeled and composed to create a consistent realization on the model and on the source code level. I proposed a flexible and robotics-specific composition approach to combine the heterogeneous concerns while maintaining their modularity (see RQ 1). Different model levels are introduced to allow the domain-specific representation of the concerns as well as the description of the concern-overarching interactions. In contrast to related works that often neglect or hide certain model levels, all levels are explicitly modeled to avoid the need for hidden assumptions that have a negative impact on the explainability and predictability.

The proposed synthesis links the task and the system concerns by expressing and enforcing the modeled requirements of the task description on the system level. This way, the system's implementation is generated conform to the modeled task and system concerns. Thus, increasing the predictability of the robot's behavior. Using this link, the robot's behavior and system design decisions can be explained by being traced back through the model levels to the associated requirements they originated from. This way, the user can tell e.g., due to what task-related requirement a certain control algorithm is used in the system, and is also assured that the resulting behavior matches the specification.

I presented CoSiMA as an implementation of the proposed approach for the domain-specific modeling of real-time capable (see RQ 2) robotic systems for compliant interaction tasks, which are synthesized (see RQ 4) into executable component-based control architectures that can be simulated as well as executed on the real robotic hardware. The analysis of CoSiMA shows that its high degree of DSL and generator reuse and its explicit support for language evolution yields a solid foundation for the modeling of robotic systems with the ability to incrementally cover further concerns of the heterogeneous robotics domains. Hence, offering the potential to further bridge the gap with every additional concern.

The experiments realized in CoSiMA (see RQ 5) show that the domain-specific formalization empowers behavior developers (e.g., non-robotics experts) to express compliant interaction tasks without the need to resort to hidden assumptions of e.g., the used control algorithms, as is often required by other approaches. For instance, setting the equilibrium point of an impedance controller into a surface to establish a contact is not needed and not encouraged in the presented approach, since it does not accurately represent the desired task. Further it is shown that the presented abstractions are suitable to model different compliant interaction tasks by explicitly specifying and composing the relevant robotics concerns, including the used software frameworks, the chosen robotic hardware, and the system's execution time behavior.

An investigation of the execution of the synthesized systems' control architectures shows that the produced behaviors conform to the modeled requirements. The ability to conduct this investigation also shows that the modeled concerns form a suitable foundation for verification.

Overall, this work applies the expertise of model-driven engineering to the domain of robotics to successfully decrease the gap between an envisioned task and the robot's behavior by increasing the explainability and predictability of real-time robot control systems in compliant interaction with the environment through the explicit modeling and composition of the relevant concerns.

OUTLOOK

The insights gained through this thesis build the foundation for investigating further promising research questions. Considering the increasing demand for robotics applications that involve compliant interaction, the increasing availability of affordable COBOTs, and the increasing interest of businesses in automation, two intriguing research directions arise:

One potential direction is to further empower non-experts in specifying and executing desired robotics tasks. This is motivated by the increasing interest of small to middle-sized businesses in robotic automation that rarely have in-house expertise in robotics. The developed approach in this thesis chooses to explicitly focus on the formalization and composition of robotics system concerns, while currently neglecting user experience in favor of flexibility, scalability, and explainability. Manually specifying the compliant interactions could—depending on the task and the environment—become very complex. By using simultaneous *task and motion planning* [Gar+21; MPS21] to automatically find the right sequence of interactions, in terms of contact situations, contact transitions, and the necessary motion and force trajectories, the user experience could be significantly increased.

In contrast to creating a desired behavior to achieve a goal based on the algorithmics of different planners, the question can be investigated on how to express an actual demonstration using the compliant interaction abstractions. Using the explicit and formalized description of the abstractions as a base for *learning by demonstration* [ZH18] would offer three major advantages: First, the demonstrated behavior would be made explainable by being interpreted in terms of compliant interactions. Second, a behavior could be produced that achieves a desired task using the extracted knowledge from the demonstration, allowing the integration of expert knowledge to influence how a specific task should be solved. Third, the predictability and safety of the robot's behavior is ensured due to it being produced by a valid model that is subject to additional requirements of the different concerns (e.g., execution timing constraints).

A

APPENDIX

The following is a selection of supplementary material of no particular order.

A.1 GENERATION OF COMPONENT-LEVEL BEHAVIOR

In general, a component instance that is solely based on the *Component DSL*, does only model the interface of the component, not its behavior. Hence, it relies on predefined (and possibly unmodeled) component behaviors. Considering the SoC, it makes sense to separate out the computational from the coordination aspects. The *Systems Coordination DSL* offers the possibility to model a component's behavior via an additional life cycle, which is based on the metamodel from the *Coordination DSL*. Thus, a state machine can be used to model the component behavior.

The *OROCOS Component Generator* provides a transformation to generate the component and its state machine, which manages the behavior coordination, into a C++ software component and a OSD script that defines the state machine. The script is loaded into the software component during runtime. The input for the transformation are `Component` model fragments that contain a `StateMachine` as life cycle specialization and that are associated with an annotated `ComponentInst` for the generation towards OROCOS RTT. The transformation results in an OROCOS-specific `ComponentShell` abstraction that is slightly higher than C++, to which the OROCOS annotation is passed on. This is necessary to enable the generation of the state machine fragments, by the *OROCOS Coordination Generator* and the *OROCOS System Coordination Generator*. The application of the two generators depends on the DSLs, the model fragments of the state machines are based on. Eventually, the `ComponentShell` is translated into text-based C++ OROCOS RTT source code, which contains the file path to the OSD artifact, resulting from the state machine transformation. An exemplary generator pipeline is visualized in Figure A.1 on the following page.

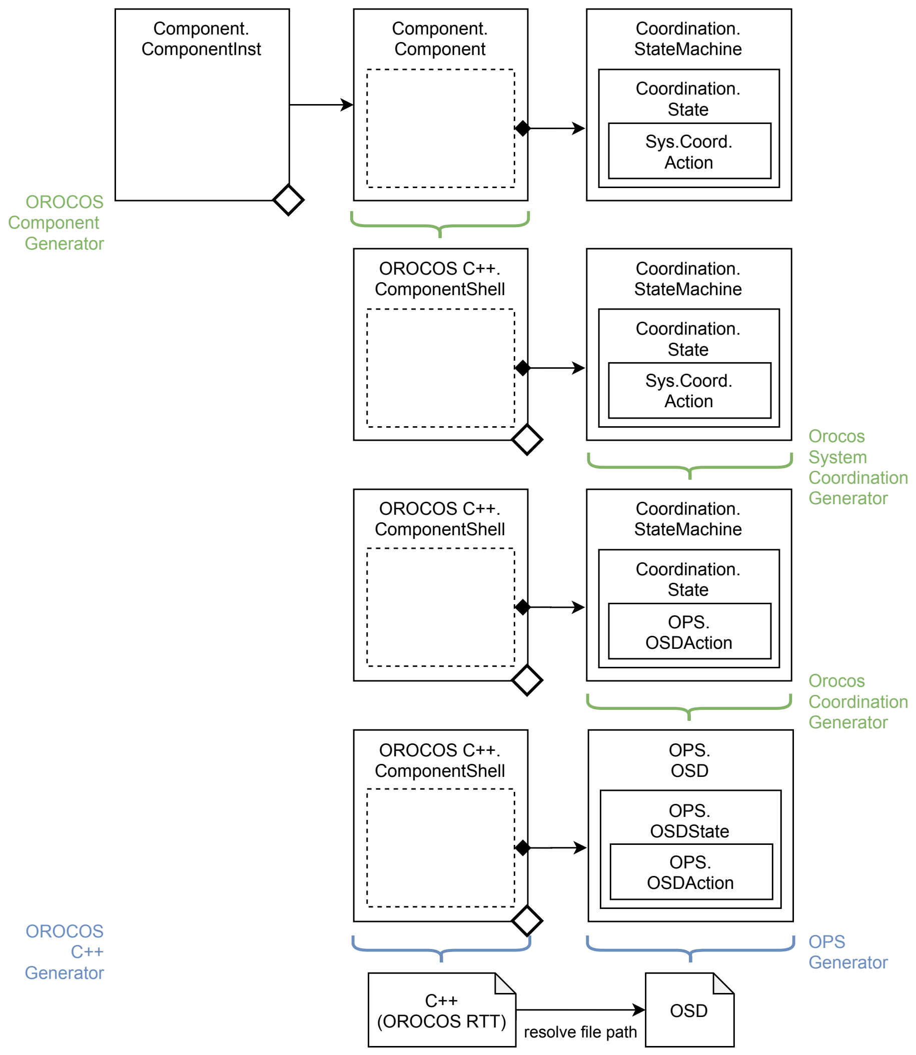


Figure A.1: A generation pipeline that transforms a component instance that has its behavior defined via a state machine, into OROCOS C++ source code that exposes the interface for communication which is required by the generated state machine script (i.e. OSD).

A.2 GENERATION OF A ROBOT INTERFACE COMPONENT CONFIGURATION

In CoSiMA a generic robot interface component is used that essentially needs the kinematic and dynamic description of the robot and the interface that the robot should be controlled through to determine which predefined OROCOS RTT component needs to be used and how it needs to be configured. Once the right component is identified, the *OROCOS Robot Platform Component Generator* creates a `ComponentInst` model for the chosen interface component. Meanwhile, the robot description is already transformed by the *URDF Generator* into an XML-based `Document` model. Since currently CoSiMA only support the generation of robot models towards URDF, all robot models are automatically transformed without the need of any kind of annotation. In spite of this transformation, the created `ComponentInst` maintains a reference to the already (or not yet) generated robot model through an `Operation`, which belongs to the default component interface of a `RobotComponentInst`. By using the IL and translating into a `ComponentInst`, the *OROCOS Component Generator* can be reused, which transforms that component instance into OPS statements. While the URDF that is generated from the robot description represents an actual file, the OPS statements can only exist as part of a larger `Document`. This means that the generator pipeline displayed in Figure A.2 shows one aspect of the generation process in Figure 3.14 (right-most column) in more detail.

>_ *intermediate layer*

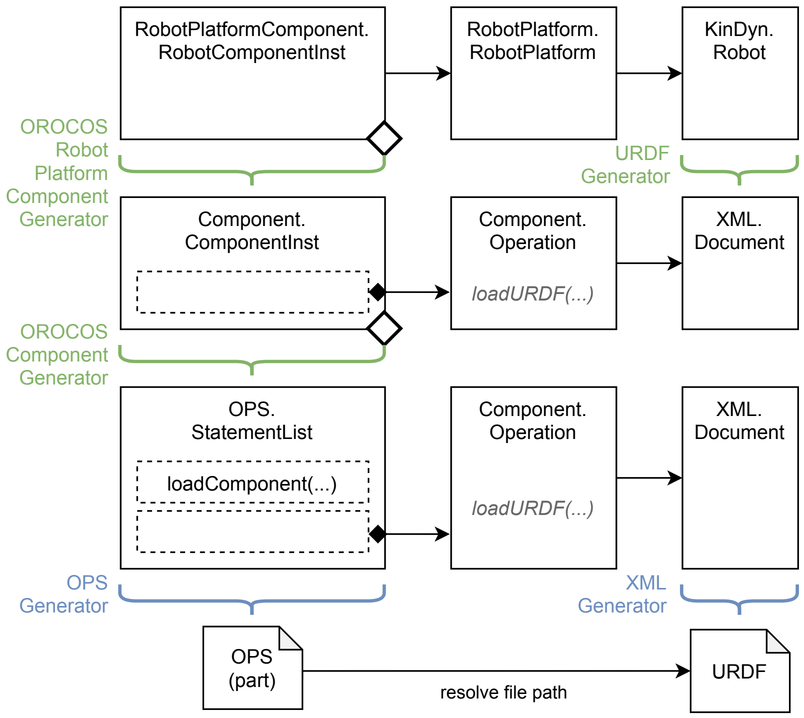


Figure A.2: A generation pipeline that transforms a robot component instance model into OPS statements to configure an OROCOS-based robot interface component that loads the necessary robot description in form of a generated URDF file.

A.3 GENERATION OF A SCHEDULING CONFIGURATION

The *Timing OROCOS Component Generator* provides two transformations. Both use the `RTTTimingActivity` from the *Timing OROCOS Component DSL* as input. The first transformation handles the generation of an OROCOS activity for each `ComponentInst`. The transformation iterates through each `ComponentInst` and checks the `IMOrocos` annotation for an `RTTTimingActivity` that is used to satisfy the `ActivityDemand` of OROCOS' software platform. Depending on the referenced `TimingConstraints`, which includes the synthesized schedule and the SRC, two types of *OPS DSL* statements are generated. First, an idle activity is generated to satisfy the requirements of OROCOS. Second, statements related to the precedence and core constraints as well as to the associated Core Scheduler are generated for the respective `ComponentInst`. Beyond this point,

>_ *sense-react chain*

the transformations of the *OPS Generator* are reused to translate the model into (plain) OPS text. The generation pipeline is shown in Figure A.3.

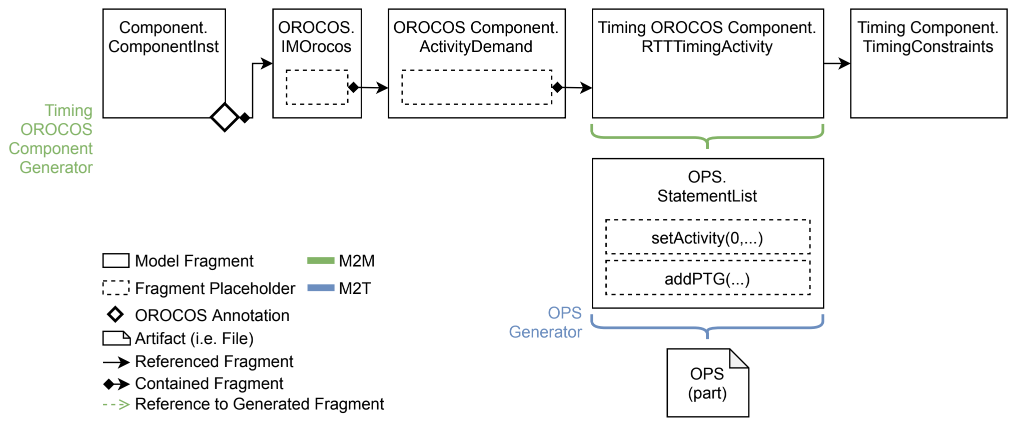


Figure A.3: A generation pipeline that transforms each `RTTTimingActivity` into OPS statements that reflect the Core Scheduler configuration for the associated `ComponentInst`.

The second transformation is only executed once. If at least one `RTTTimingActivity` is present in the model, an initial configuration of the Core Schedulers is needed. The `System` in which the Core Schedulers need to be configured can be traced back via the referenced `TimingConstraints`. Once the system is found, the OPS `Document` in which it is eventually transformed into is resolved and additional OPS statements are added (i.e. *weaved*) retrospectively. These statements are injected by the *Timing OROCOS Component Generator* to configure the necessary amount of Core Schedulers and to associate them to their respective core. The generation pipeline is shown in Figure A.4.

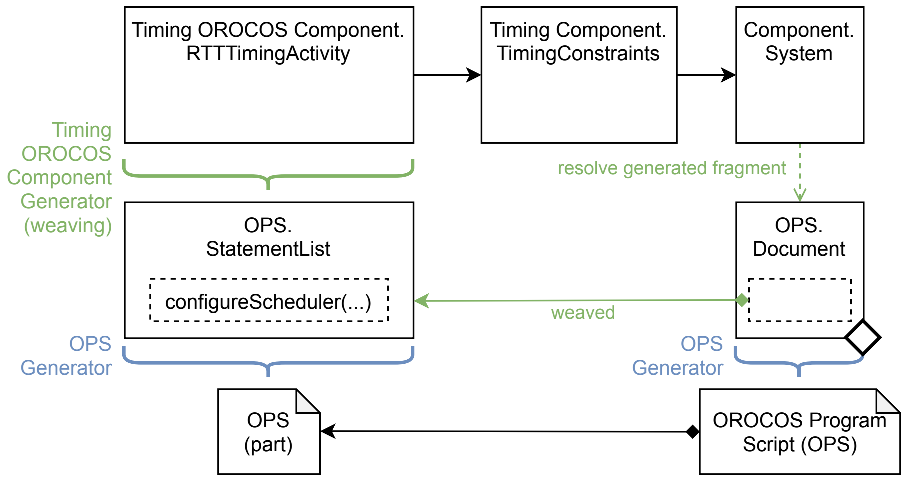
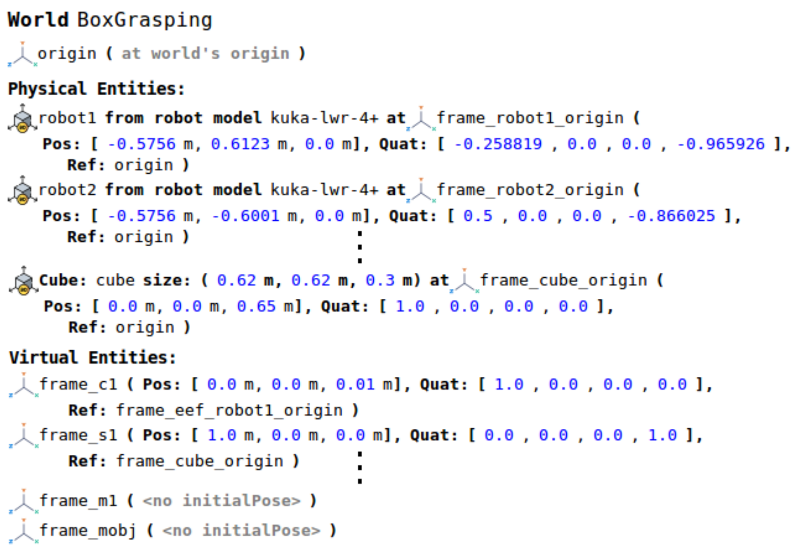


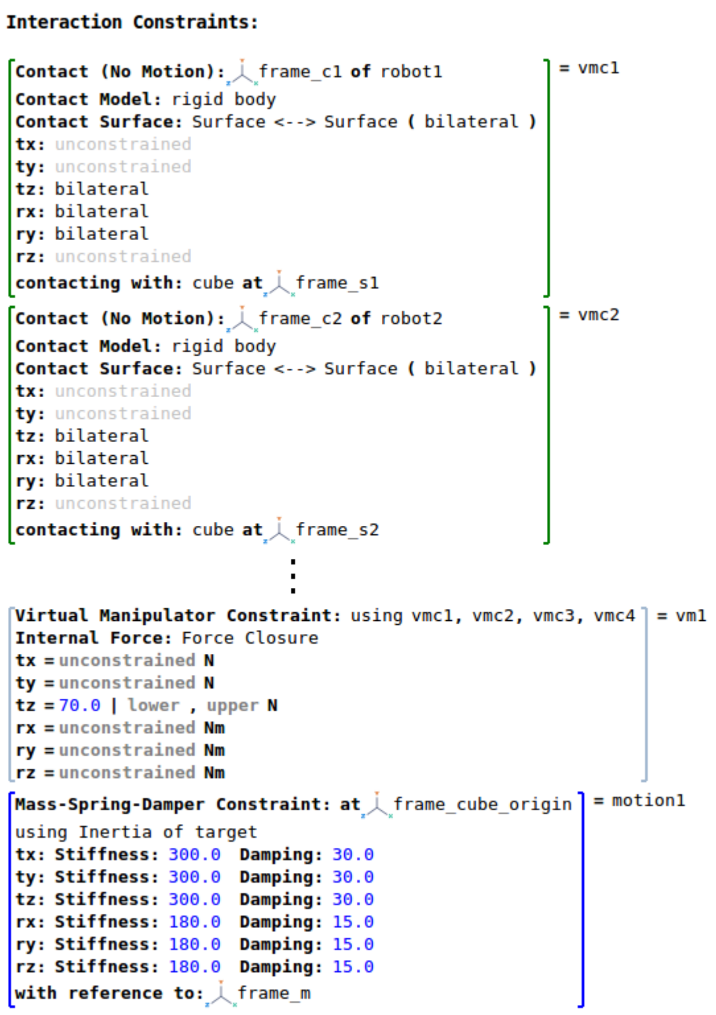
Figure A.4: A generation pipeline that adds (i.e. *weaves*) the general configuration of the Core Scheduler into an existing OPS `Document` model. This pipeline is only executed once.

A.4 CI WORLD AND CONSTRAINT MODELS



```
World BoxGrasping
origin ( at world's origin )
Physical Entities:
robot1 from robot model kuka-lwr-4+ at frame_robot1_origin (
   Pos: [ -0.5756 m, 0.6123 m, 0.0 m], Quat: [ -0.258819 , 0.0 , 0.0 , -0.965926 ],
     Ref: origin )
robot2 from robot model kuka-lwr-4+ at frame_robot2_origin (
   Pos: [ -0.5756 m, -0.6001 m, 0.0 m], Quat: [ 0.5 , 0.0 , 0.0 , -0.866025 ],
     Ref: origin )
                                      ⋮
Cube: cube size: ( 0.62 m, 0.62 m, 0.3 m) at frame_cube_origin (
   Pos: [ 0.0 m, 0.0 m, 0.65 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0 ],
     Ref: origin )
Virtual Entities:
frame_c1 ( Pos: [ 0.0 m, 0.0 m, 0.01 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0 ],
     Ref: frame_eef_robot1_origin )
frame_s1 ( Pos: [ 1.0 m, 0.0 m, 0.0 m], Quat: [ 0.0 , 0.0 , 0.0 , 1.0 ],
     Ref: frame_cube_origin )       ⋮
frame_m1 ( <no initialPose> )
frame_mobj ( <no initialPose> )
```

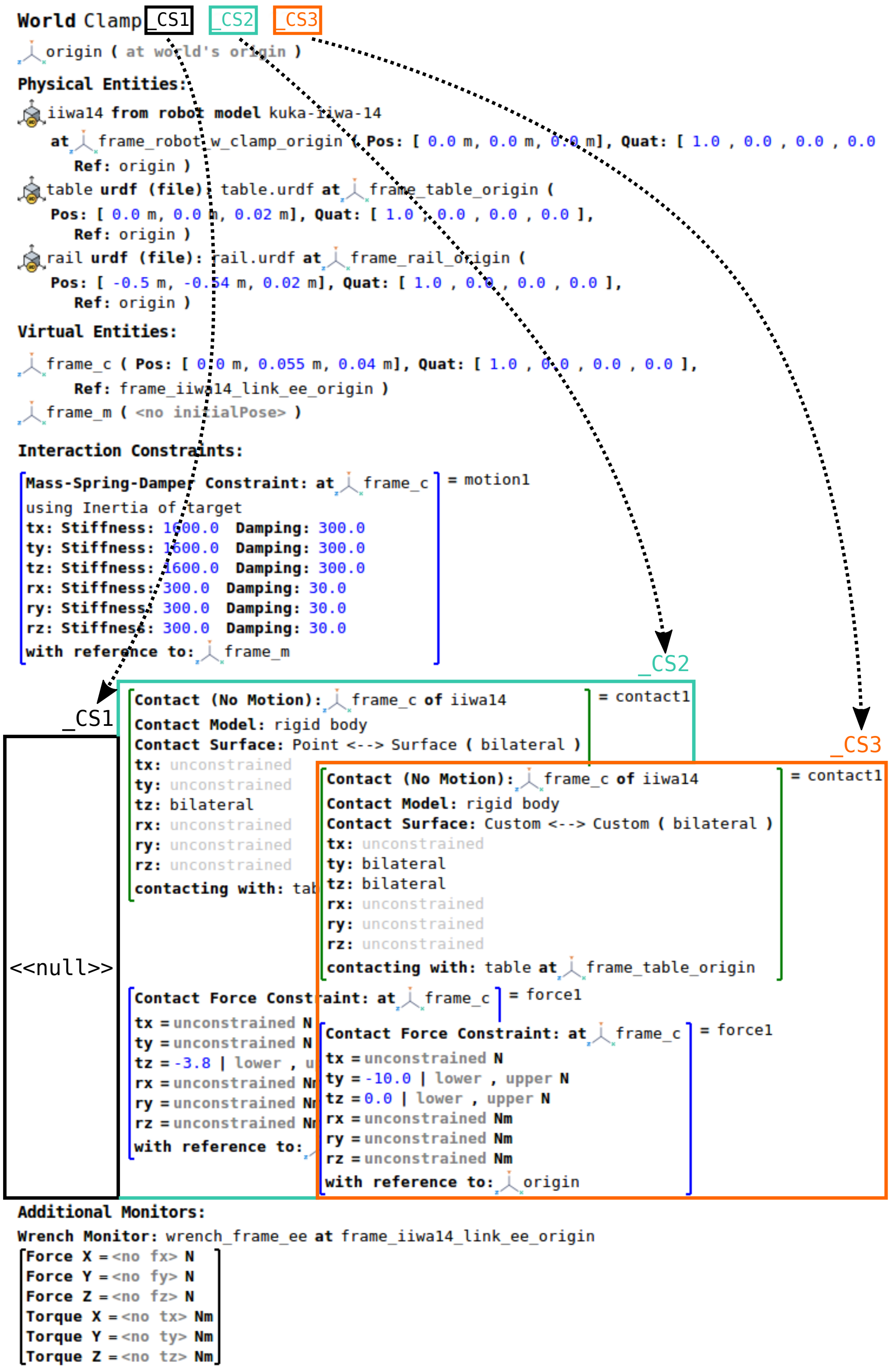
Figure A.5: Physical and virtual entities model for the Tetra-Arm case study.



```
Interaction Constraints:

[Contact (No Motion): frame_c1 of robot1                  ] = vmc1
 Contact Model: rigid body
 Contact Surface: Surface <--> Surface ( bilateral )
 tx: unconstrained
 ty: unconstrained
 tz: bilateral
 rx: bilateral
 ry: bilateral
 rz: unconstrained
 contacting with: cube at frame_s1
[Contact (No Motion): frame_c2 of robot2                  ] = vmc2
 Contact Model: rigid body
 Contact Surface: Surface <--> Surface ( bilateral )
 tx: unconstrained
 ty: unconstrained
 tz: bilateral
 rx: bilateral
 ry: bilateral
 rz: unconstrained
 contacting with: cube at frame_s2
                                 ⋮
[Virtual Manipulator Constraint: using vmc1, vmc2, vmc3, vmc4] = vm1
 Internal Force: Force Closure
 tx = unconstrained N
 ty = unconstrained N
 tz = 70.0 | lower , upper N
 rx = unconstrained Nm
 ry = unconstrained Nm
 rz = unconstrained Nm
[Mass-Spring-Damper Constraint: at frame_cube_origin] = motion1
 using Inertia of target
 tx: Stiffness: 300.0  Damping: 30.0
 ty: Stiffness: 300.0  Damping: 30.0
 tz: Stiffness: 300.0  Damping: 30.0
 rx: Stiffness: 180.0  Damping: 15.0
 ry: Stiffness: 180.0  Damping: 15.0
 rz: Stiffness: 180.0  Damping: 15.0
 with reference to: frame_m
```

Figure A.6: Excerpt of the constraints for CS2 depicted in Figure 8.2.



```
World Clamp _CS1  _CS2  _CS3
origin ( at world's origin )
Physical Entities:
iiwa14 from robot model kuka-iiwa-14
   at frame_robot_w_clamp_origin ( Pos: [ 0.0 m, 0.0 m, 0.0 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0
      Ref: origin )
table urdf (file): table.urdf at frame_table_origin (
   Pos: [ 0.0 m, 0.0 m, 0.02 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0 ],
      Ref: origin )
rail urdf (file): rail.urdf at frame_rail_origin (
   Pos: [ -0.5 m, -0.54 m, 0.02 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0 ],
      Ref: origin )
Virtual Entities:
frame_c ( Pos: [ 0.0 m, 0.055 m, 0.04 m], Quat: [ 1.0 , 0.0 , 0.0 , 0.0 ],
      Ref: frame_iiwa14_link_ee_origin )
frame_m ( <no initialPose> )
Interaction Constraints:
[Mass-Spring-Damper Constraint: at frame_c ] = motion1
 using Inertia of target
 tx: Stiffness: 1600.0  Damping: 300.0
 ty: Stiffness: 1600.0  Damping: 300.0
 tz: Stiffness: 1600.0  Damping: 300.0
 rx: Stiffness: 300.0  Damping: 30.0
 ry: Stiffness: 300.0  Damping: 30.0
 rz: Stiffness: 300.0  Damping: 30.0
 with reference to: frame_m

_CS1                                                    _CS2
<<null>>

[Contact (No Motion): frame_c of iiwa14 ] = contact1
 Contact Model: rigid body
 Contact Surface: Point <--> Surface ( bilateral )
 tx: unconstrained
 ty: unconstrained
 tz: bilateral
 rx: unconstrained
 ry: unconstrained
 rz: unconstrained
 contacting with: tab

[Contact Force Constraint: at frame_c ] = force1
 tx = unconstrained N
 ty = unconstrained N
 tz = -3.8 | lower , u
 rx = unconstrained N
 ry = unconstrained N
 rz = unconstrained N
 with reference to:

                                                        _CS3
[Contact (No Motion): frame_c of iiwa14 ] = contact1
 Contact Model: rigid body
 Contact Surface: Custom <--> Custom ( bilateral )
 tx: unconstrained
 ty: bilateral
 tz: bilateral
 rx: unconstrained
 ry: unconstrained
 rz: unconstrained
 contacting with: table at frame_table_origin

[Contact Force Constraint: at frame_c ] = force1
 tx = unconstrained N
 ty = -10.0 | lower , upper N
 tz = 0.0 | lower , upper N
 rx = unconstrained Nm
 ry = unconstrained Nm
 rz = unconstrained Nm
 with reference to: origin

Additional Monitors:
Wrench Monitor: wrench_frame_ee at frame_iiwa14_link_ee_origin
[Force X = <no fx> N
 Force Y = <no fy> N
 Force Z = <no fz> N
 Torque X = <no tx> Nm
 Torque Y = <no ty> Nm
 Torque Z = <no tz> Nm]
```

Figure A.7: Excerpt of the constraints for the CSs depicted in Figure 8.10.

A.5 TIMING CONSTRAINTS AND SCHEDULE

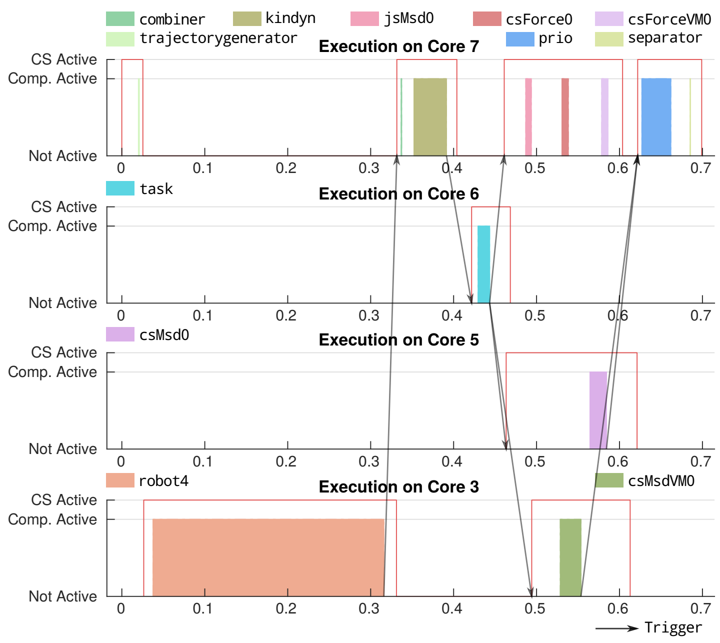


Figure A.8: One real execution cycle of the synthesized schedule, displayed in Figure 8.16. The robot interface components 1-3 are omitted in favor of readability. A single core scheduler component is associated to each core. The execution of a core scheduler is indicated by a red border. Note that the concrete execution times are not relevant, since they heavily depend on the properties of the processing hardware. Only the execution order and the data-flow is representative even for the execution on different hardware.

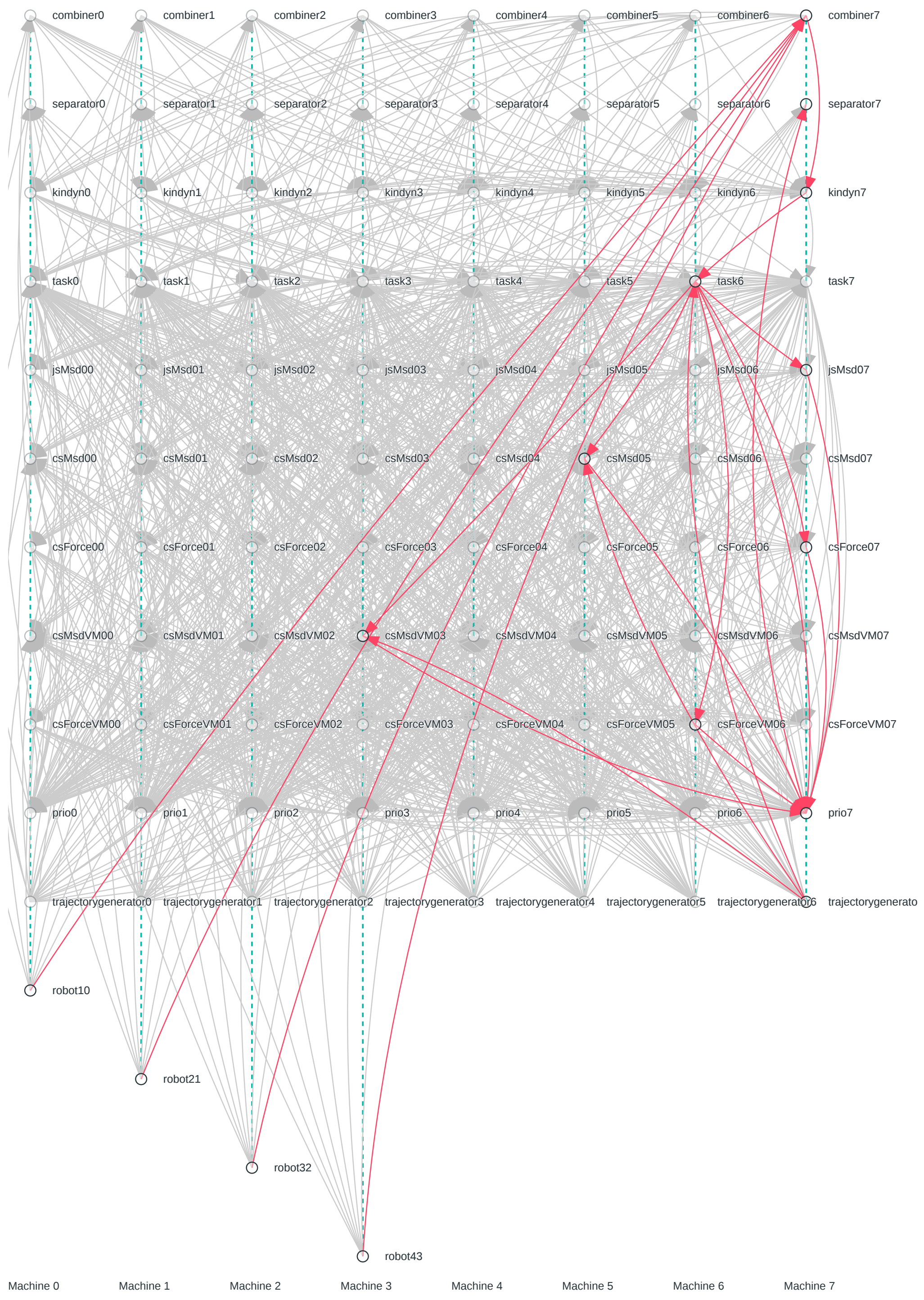


Figure A.9: Full set of core and precedence constraints for the tetra-arm scenario, which is used as input for the schedule solving process.

ACRONYMS

A

AST
: abstract syntax tree. *used on: pp. 29, 32, 43, 121*

C

CBRS
: component-based robotic systems. *used on: pp. 23–25, 31, 38–40, 48, 49, 60, 62, 65–68, 70, 71, 73, 86*

CBSE
: component-based software engineering. *used on: p. 23*

CCA
: Compliant Control Architecture. *used on: p. 112*

CF
: Compliance Frame. *used on: p. 95*

CI
: compliant interaction. *used on: pp. xii, 3–11, 35, 39, 40, 45, 48, 87, 91, 92, 100, 101, 103, 105, 106, 111, 112, 114, 117–127, 130, 133, 134, 137, 139, 141, 148, 152–154*

CIM
: computation-independent model. *used on: pp. 30, 33*

CNN
: Convolutional Neural Network. *used on: p. 149*

COBOT
: Collaborative Robot. *used on: pp. 3, 5, 103, 161*

CoSiMA
: Compliant Simulation and Modeling Architecture. *used on: pp. 11, 48, 49, 60, 65, 78, 82, 86, 121, 122, 125, 129, 137, 160*

CPC
: Component-Port-Connector. *used on: pp. 38, 49, 60*

CS
: Contact Situation. *used on: pp. 98–100, 112, 127, 128, 136, 141–143, 145, 146, 148–152, 167, 168*

CSP
: Constraint Satisfaction Problem. *used on: pp. 76, 78*

D

DoF

degree-of-freedom. *used on: pp. 3, 4, 7, 16, 18–20, 25, 43, 93, 94, 96, 97, 109–111, 115, 116, 124, 131–133, 135, 142*

DSL

domain-specific language. *used on: pp. v, xv, xvi, 7, 29, 30, 32, 33, 37, 38, 42–46, 48–51, 53, 55, 57, 59–63, 66, 71, 72, 75, 82, 83, 86, 103–105, 115, 119–122, 124, 125, 160, 163*

DynGHC

dynamically consistent Generalized Hierarchical Control. *used on: p. 100*

E

E2ERT

end-to-end response time. *used on: pp. 68, 73, 74, 76*

EEF

end-effector. *used on: pp. 16–18, 93–95, 97, 100, 108, 111, 142, 143, 146, 153*

EMF

Eclipse Modeling Framework. *used on: p. xvi*

F

FJSSP

Flexible Job Shop Scheduling Problem. *used on: p. 76*

FRI

Fast Research Interface. *used on: pp. 41, 51*

FSM

finite state machine. *used on: pp. 50, 56, 60*

G

GPL

general-purpose language. *used on: p. 29*

I

IL

intermediate layer. *used on: pp. 46, 57, 59, 165*

IPT

independent processing time. *used on: pp. 73, 76–78, 81, 85*

L

L3Dim

Layered 3 Dimensions. *used on: pp. 39–43, 45–48, 57–60, 63–65, 71, 119*

LM&C

language modularization and composition. *used on: pp. 30–33, 37–39, 42, 43, 137*

M

M2M

model-to-model. *used on: pp. 55, 56, 114, 127, 130*

M2T

model-to-text. *used on: pp. 55, 56, 81*

MDE

model-driven engineering. *used on: pp. 7, 9, 10, 28, 29, 33, 37, 63, 103, 159, 160*

MoT

Math of Task. *used on: pp. 115, 121, 122*

MPS

JetBrains Meta Programming System. *used on: p. 43*

MSD

mass-spring-damper. *used on: pp. 94, 95, 110, 142–144, 147, 149, 151*

O

OMG

Object Management Group. *used on: pp. 30, 38*

OPS

OROCOS Program Script. *used on: pp. 55, 56, 79, 81, 82, 133, 137, 165, 166*

OSD

OROCOS State Description. *used on: pp. 56, 137, 163, 164*

P

PC

principal contacts. *used on: pp. 96, 109, 110*

PIDC

Projected Inverse Dynamics Control. *used on: pp. v, 8, 10, 11, 21, 22, 35, 47, 64, 115, 122, 125–127, 132, 134, 137, 141, 153, 154*

PIM

platform-independent model. *used on: pp. 30, 31, 33, 38, 40, 44, 58*

PS

prioritization structure. *used on: pp. 112–117, 120–122, 127–131, 136, 144, 145, 156, 157*

PSI

platform-specific implementation. *used on: pp. 31, 38*

PSM

platform-specific model. *used on: pp. 31, 33, 38, 40, 58*

PTG

Precedence Task Graph. *used on: pp. 68–71, 74*

Q

QP

Quadratic Programming. *used on: pp. 21, 22, 47, 115, 122, 154*

R

RTS

real-time system. *used on: pp. 66, 67*

S

SEA
: series elastic actuator. *used on: p. 2*

SoC
: separation of concerns. *used on: pp. 24, 28, 29, 33, 37–41, 43, 57, 63, 163*

SoR
: separation of roles. *used on: pp. 26, 28, 33, 37, 43, 57, 63*

SoT
: Stack of Tasks. *used on: pp. 19, 20, 22, 47, 122, 154*

SotA
: state-of-the-art. *used on: pp. 15, 33*

SRC
: sense-react chain. *used on: pp. 68, 70, 71, 73, 74, 76–79, 81, 85, 86, 165*

T

TFF
: Task Frame Formalism. *used on: pp. 16–18, 94, 104, 105*

TRL
: Technology Readiness Level. *used on: p. 23*

U

URDF
: Unified Robot Description Format. *used on: pp. 51, 60, 143, 165*

V

VKC
: virtual kinematic chain. *used on: pp. 105, 121, 123, 135, 136, 153*

VM
: Virtual Manipulator. *used on: pp. 98, 111, 132, 135, 136, 141–147, 153, 156, 157*

VSA
: variable stiffness actuator. *used on: p. 2*

W

WCE2ERT
: worst-case end-to-end response time. *used on: pp. 67, 68, 70, 71, 74, 77–79*

WCET
: worst-case execution time. *used on: pp. 66, 67, 70–73, 78, 85*

WCRT
: worst-case response time. *used on: pp. 66, 67, 70, 72*

GLOSSARY

A

abstract syntax

A data structure for the definition of a model, without any notational details (cf. concrete syntax). *used on: p. 29*

adapter language

A language to realize dependent model fragments while ensuring the independence of the underlying languages [Gam+95; Völ+13]. *used on: pp. 45, 49*

afferent coupling

The dependencies on a module from other modules. *used on: pp. 59–61*

C

capability

An isolated functional and non-functional concern of a robotic system, such as coordination, vision-based perception, motion generation, timing, task description, or the structural concern of a system, etc. *used on: pp. 6, 15, 39, 40, 43–45, 51, 53, 57, 60, 64, 65, 70, 71, 73*

compliance frame

Refers to the frame in which the compliant control behavior is expressed. It usually relates to a contact point or the EEF of a robot. *used on: pp. 16, 108*

compliant interaction

Describes every situation in which external forces influence or even define the behavior of a robot. This includes the handling, compensation, and exploitation of physical interactions through contacts, which are natural interfaces for the exchange of forces. See Section 1.1. *used on: pp. v, 1, 3, 4, 10, 15, 17–19, 22, 37, 40, 89, 90, 92, 94, 100, 103, 106, 108, 117, 159–161*

component

A unit of composition that must be composable with other components to form a system in a predictable way. Here, a component either refers to an entity in a component-based framework (e.g., OROCOS RTT or ROS) or to a modeled abstraction thereof. *used on: p. 23*

component-based architecture
An architecture that represents the structural concerns of a system based on a component model. *used on: pp. 4, 27*

composability
Composability refers to the ability to combine and recombine parts meaningfully into a whole for different purposes [PW03; Rob19]. *used on: pp. 6, 58, 90, 103*

concern
Originated as "a principle in computer science and software engineering[,] identif[ying] and decoupl[ing] different problem areas to [be] view[ed] and solve[d] independent [of] each other [Dij82]" [Rob19]. A cross-cutting concern, however, "affects multiple properties and areas in a [robotic] system possibly at different levels of abstraction" [Rob19]. *used on: pp. 4, 7–9, 11, 15, 23–28, 30–33, 37–40, 42–45, 48, 50, 55, 57, 60–62, 65, 66, 70–72, 82, 83, 91, 137, 154, 159*

concrete syntax
The concrete syntax of a DSL is what the user interacts with to create a model. It can be a combination of textual, graphical, tabular, etc. [Völ+13] *used on: pp. 29, 32, 43*

control architecture
An architecture that combines the required structural elements to represent a control law to command the robot. It can be designed as a component-based architecture (CBA). It is part of a robotic system. *used on: pp. 7, 8, 11, 125*

coupling
Describes the interactions between contacting entities. It is characterized by the forces that can be transmitted through the contacts and the allowed relative motion of the contacting entities. Not to be confused with afferent coupling or efferent coupling. *used on: p. 92*

D

data freshness
Is one of the most important data quality attributes in information systems. The freshness refers to the following questions: How old is the data with respect to the users expectations? Is there a more recent data sample? When was the data sample produced? [Bou04; CG00]. *used on: p. 86*

dimension
Describes a horizontal SoC along semantic categories of the target domain. In CoSiMA, the robotics-specific dimensions are: Capability, Software Platform, and Hardware Platform. *used on: p. 41*

domain
"An area of knowledge or activity; especially one that somebody is responsible for" [Dic21]. Domains related to robotics are aerospace, telecommunication, and automotive. The robotics domain itself can be divided into different subdomains, such as kinematics, sensing, motion control, reasoning, etc. [Nor+16a]. *used on: pp. 7, 9–12, 18, 22–31,*

33, 37, 41, 42, 48, 49, 54, 57, 60, 61, 63–66, 71, 72, 74, 76, 77, 86, 100, 101, 103, 104, 110, 117, 118, 124, 129, 149, 153, 157, 159, 160

E

Ecore

Ecore is the meta-model included in the core Eclipse Modeling Framework (EMF) to describe models and runtime support, based on Meta Object Facility (MOF). *used on: p. xvi*

efferent coupling

The dependencies of a module on other modules. *used on: p. 61*

G

general-purpose language

A programming language that lacks domain-specific features and is usable in a wide variety of application domains (e.g., C++ or Python). *used on: p. 29*

H

heterogeneous transformation

A transformation where the input and output models are based on different meta-models [Com+16]. *used on: p. 29*

homogeneous transformation

A transformation where the input and output models are based on the same meta-model [Com+16]. *used on: pp. 29, 75, 78*

L

language evolution

Describes the fact that a DSL has to reflect changes of requirements over time. These changes can be driven by changes in the domain, target platform, and implementation-related dependencies [Völ+13]. *used on: pp. 42, 43, 47*

language module

A module that technically implements (parts of) a DSL. *used on: pp. 29, 43, 60*

language workbench

The term language workbench, coined and popularized by Martin Fowler in 2005 [Fow05], is a tool that provide high-level mechanisms for the efficient design, reuse, and composition of (domain-specific) languages. *used on: p. 43*

M

meta-model

A model that describes the abstract syntax of a language [Com+16]. *used on: pp. 29, 38, 49–52, 54, 58, 163*

middleware

A “software layer that provides a programming abstraction as well as [a] masking [of] the heterogeneity of the underlying networks, hardware, operating systems and programming languages” [CDK05]. *used on: p. 27*

model
: A model is an abstraction of reality (for a given purpose) [Rot+89]. In the context of MDE a model is produced by instantiating the meta-model concepts of a DSL. *used on: pp. 28, 29*

model fragment
: A part of a model whose AST is a sub-graph of the model's AST. In contrast, a model element refers to a single instance of a concept (i.e. a node) in the AST. *used on: pp. 32, 40–42, 45, 47, 55, 56, 82, 119, 124, 163*

P

precedence constraint
: A (temporal) constraint that enforces the execution of a real-time task to start after the completion of the preceding task. *used on: pp. 68–70, 73, 74, 76–79, 81*

program synthesis
: A synthesis in the context of generative (meta-)programming is referred to as a computation (i.e. model transformation) that when executed generates a target program from a meta-program (i.e. DSL model), used as input [TAD07]. *used on: pp. 8, 10, 11, 37, 75, 79, 85, 86, 124–127, 133, 136, 137, 141, 154*

projectional
: Projectional tools, editors, or views refer to the support of different projections for the same AST in terms of concrete syntaxes, such as textual, graphical, or tabular. *used on: p. 43*

R

robot behavior
: Refers to the overall behavior exhibited by a robotic system. On one side the behavior describes the exhibited (physical-) interaction of the robot (e.g., impedance control behavior and generated forces and motions). On the other side, it refers to the software system's behavior. This includes the coordination of different actions as well as the execution, communication, and scheduling of the involved system's components. *used on: pp. 3, 15, 37, 65, 89, 103, 138, 144, 159*

robotic system
: A system that covers robotics-related hardware and software aspects. Every robotic system uses an architectural structure [KSB16]. A control architecture represents the structural aspects related to control the robot. *used on: p. 23*

Robotics DSL Zoo
: A curated website for DSL publications in robotics [Nor+16b]. *used on: p. 33*

S

sense-react chain
: A sequence of components realizing a control cycle that starts with sensed information and ends with a respective action. *used on: pp. 25–27, 68, 74, 75, 79, 81, 154*

skill

An action that a robot is capable of performing. Depending on the literature, it might refer to a screwing motion, the picking of an object, or a peg-in-hole assembly process, etc. See Section 5.1. *used on: pp. 5–8, 15, 16, 26, 37, 89–91, 103, 104, 122–124*

structural concern

Represents all aspects that are significant to the structural viewpoint of a (component-based) system. It focuses on the architectural components, connections, constraints and styles that are required to describe and reason about the system's structure [Cre+01]. *used on: p. 38*

BIBLIOGRAPHY

GENERAL

[ABB] ABB. *YuMi.* URL: https://new.abb.com/products/robotics/de/industrieroboter/yumi (visited on 2020-12-01). *used on: p. 3*

[Ad14] Erwin Aertbeliën and Joris de Schutter. "eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs." In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2014*. Piscataway, NJ: IEEE, 2014, pp. 1540–1546. DOI: 10.1109/IROS.2014.6942760. *used on: pp. 92, 105, 106, 153*

[AGD18] Deniz Akdur, Vahid Garousi, and Onur Demirörs. "A survey on modeling and model-driven engineering practices in the embedded software industry." In: *Journal of Systems Architecture* 91 (2018), pp. 62–82. DOI: https://doi.org/10.1016/j.sysarc.2018.09.007. *used on: p. 103*

[Agh05] F. Aghili. "A unified approach for inverse and direct dynamics of constrained multibody systems based on linear projection operator: applications to control and simulation." In: *IEEE Transactions on Robotics* 21.5 (2005-10), pp. 834–849. DOI: 10.1109/TRO.2005.851380. *used on: pp. 21, 22*

[AIS18] AIST. *[HRP-5P] Capable of working with heavy materials at construction sites*. National Institute of Advanced Industrial Science and Technology (AIST) https://www.aist.go.jp/index_en.html. 2018-10-30. URL: https://www.youtube.com/watch?v=fMwiZXxo9Qg (visited on 2020-12-01). *used on: p. 2*

[Ajo+14] A. Ajoudani et al. "A manipulation framework for compliant humanoid COMAN: Application to a valve turning task." In: *2014 IEEE-RAS International Conference on Humanoid Robots*. 2014, pp. 664–670. DOI: 10.1109/HUMANOIDS.2014.7041434. *used on: p. 51*

[AK16] D. Almeida and Y. Karayiannidis. "Folding assembly by means of dual-arm robotic manipulation." In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 3987–3993. DOI: 10.1109/ICRA.2016.7487588. *used on: p. 2*

[Alo+10] Diego Alonso et al. "V3cmm: A 3-view component meta-model for model-driven robotic software development." In: *Journal of Software Engineering for Robotics* 1.1 (2010), pp. 3–17. *used on: pp. 31, 32*

[Ama+20] Vishnu Dev Amara et al. “On the efficient control of series-parallel compliant articulated robots.” In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 385–391. DOI: 10.1109/ICRA40945.2020.9196786. *used on: p. 2*

[AMS07] Charles André, Frédéric Mallet, and Robert de Simone. “Modeling Time(s).” In: *Model Driven Engineering Languages and Systems*. Ed. by Gregor Engels et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 559–573. *used on: pp. 7, 65, 66*

[Ara+21] Edson de Araújo Silva et al. “A survey of Model Driven Engineering in robotics.” In: *Journal of Computer Languages* 62 (2021-02), p. 101021. DOI: 10.1016/j.cola.2020.101021. *used on: p. 28*

[Arg+09] Brenna D. Argall et al. “A survey of robot learning from demonstration.” In: *Robotics and Autonomous Systems* 57.5 (2009), pp. 469–483. ISSN: 0921-8890. DOI: 10.1016/j.robot.2008.10.024. *used on: p. 152*

[Arn00] Ken Arnold. *Embedded Controller Hardware Design*. L L H Technology Publishing, 2000. ISBN: 978-1-878707-52-9. *used on: p. 65*

[AS16] F. Aghili and C. Su. “Control of constrained robots subject to unilateral contacts and friction cone constraints.” In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016-05, pp. 2347–2352. DOI: 10.1109/ICRA.2016.7487385. *used on: p. 22*

[Aud+14] H. Audren et al. “Model preview control in multi-contact motion-application to a humanoid robot.” In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 4030–4035. DOI: 10.1109/IROS.2014.6943129. *used on: p. 99*

[Bag+12] J. Andrew (Drew) Bagnell et al. “An Integrated System for Autonomous Robotics Manipulation.” In: *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012-10, pp. 2955–2962. *used on: p. 25*

[Bar19] William Barnett. “Architectural Data Modelling for Robotic Applications.” In: (2019). *used on: p. 64*

[Bd96] H. Bruyninckx and J. de Schutter. “Specification of force-controlled actions in the task frame formalism-a synthesis.” In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 581–589. DOI: 10.1109/70.508440. *used on: pp. 16, 17, 93*

[Ben+09] Saddek Bensalem et al. “Designing autonomous robots.” In: *IEEE Robotics & Automation Magazine* 16.1 (2009), pp. 67–77. DOI: 10.1109/MRA.2008.931631. *used on: p. 69*

[Béz05] Jean Bézivin. “On the Unification Power of Models.” In: *Software and System Modeling* 4.2 (2005), pp. 171–188. *used on: p. 29*

[BG16] Davide Brugali and Luca Gherardi. “HyperFlex: A Model Driven Toolchain for Designing and Configuring Software Control Systems for Autonomous Robots.” In: *Robot Operating System (ROS)*. Ed. by Anis Koubaa. Vol. 625. Studies in Computational Intelligence. Cham: Springer International Publishing, 2016, pp. 509–534. ISBN: 978-3-319-26052-5. DOI: 10.1007/978-3-319-26054-9_20. *used on: pp. 38, 49*

[Bis+10] Rainer Bischoff et al. “BRICS - Best practice in robotics.” In: *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)* (2010), pp. 1–8. *used on: pp. 6, 41*

[Bjö+11] Anders Björkelund et al. "Knowledge and Skill Representations for Robotized Production." In: *IFAC Proceedings Volumes* 44.1 (2011). 18th IFAC World Congress, pp. 8999–9004. DOI: 10.3182/20110828-6-IT-1002.01053. *used on: pp. 16, 90, 91*

[BK11] K. Bouyarmane and A. Kheddar. "Using a multi-objective controller to synthesize simulated humanoid robot motion with changing contact configurations." In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2011-09, pp. 4414–4419. DOI: 10.1109/IROS.2011.6094483. *used on: pp. 20, 115*

[BKB13] Georg Bartels, Ingo Kresse, and Michael Beetz. "Constraint-based movement representation grounded in geometric features." In: *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2013)*. New York: IEEE, 2013, pp. 547–554. DOI: 10.1109/HUMANOIDS.2013.7030027. *used on: pp. 105, 106, 153*

[BLW05] Paul Baker, Shiou Loh, and Frank Weil. "Model-Driven Engineering in a Large Industrial Context — Motorola Case Study." In: *Model Driven Engineering Languages and Systems*. Ed. by Lionel Briand and Clay Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 476–491. *used on: p. 4*

[Boh+09] M. Bohlin et al. "Simulation-Based Timing Analysis of Complex Real-Time Systems." In: *IEEE Int. Conf. Embedded and Real-Time Computing Systems and Applications*. 2009-08, pp. 321–328. DOI: 10.1109/RTCSA.2009.41. *used on: pp. 65, 66*

[Bor+16] Gianni Borghesan et al. "Introducing Geometric Constraint Expressions Into Robot Constrained Motion Specification and Control." In: *IEEE Robotics and Automation Letters* 1.2 (2016), pp. 1140–1147. DOI: 10.1109/LRA.2015.2506119. *used on: pp. 7, 91*

[Bou04] Mokrane Bouzeghoub. "A Framework for Analysis of Data Freshness." In: *Proceedings of the 2004 International Workshop on Information Quality in Information Systems*. IQIS '04. Paris, France: Association for Computing Machinery, 2004, pp. 59–67. DOI: 10.1145/1012453.1012464. *used on: p. 176*

[Bri08] Robert Bringhurst. *The Elements of Typographic Style*. Point Roberts, WA: Hartley & Marks, Publishers, 2008. ISBN: 978-0-88179-206-5. *used on: p. 207*

[Bro+05] A. Brooks et al. "Towards component-based robotics." In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2005, pp. 163–168. *used on: p. 23*

[Bru+13] Herman Bruyninckx et al. "The BRICS Component Model: A Model-Based Development Paradigm for Complex Robotics Software Systems." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: Association for Computing Machinery, 2013, pp. 1758–1764. DOI: 10.1145/2480362.2480693. *used on: pp. 24, 25, 31, 32, 66*

[Bru+16] Peter Brucker et al. *The Scheduling Zoo*. 2016. URL: http://schedulingzoo.lip6.fr (visited on 2021-05-21). *used on: p. 69*

[Bru15] D. Brugali. “Model-Driven Software Engineering in Robotics: Models Are Designed to Use the Relevant Things, Thereby Reducing the Complexity and Cost in the Field of Robotics.” In: *IEEE Robotics Automation Magazine* 22.3 (2015), pp. 155–166. DOI: 10.1109/MRA.2015.2452201. *used on: pp. 4, 28*

[BS09] D. Brugali and P. Scandurra. “Component-based robotic engineering (Part I) [Tutorial].” In: *IEEE Robotics Automation Magazine* 16.4 (2009), pp. 84–96. *used on: pp. 23, 89*

[BS10] D. Brugali and A. Shakhimardanov. “Component-Based Robotic Engineering (Part II).” In: *IEEE Robotics Automation Magazine* 17.1 (2010), pp. 100–112. *used on: pp. 23–26*

[BSS11] Mirko Bordignon, Kasper Stoy, and Ulrik Pagh Schultz. “Generalized programming of modular robots through kinematic configurations.” In: *Proc. Int. Conf. Intelligent Robots and Systems*. IEEE. 2011, pp. 3659–3666. *used on: p. 41*

[BTv08] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. “Worst Case Reaction Time Analysis of Concurrent Reactive Programs.” In: *Electronic Notes in Theoretical Computer Science* 203.4 (2008). Proceedings of the International Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P 2007), pp. 65–79. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.05.011. *used on: p. 67*

[Buc+14] Jacob Pørksen Buch et al. “Applying Simulation and a Domain-Specific Language for an Adaptive Action Library.” In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Davide Brugali et al. Cham: Springer International Publishing, 2014, pp. 86–97. *used on: pp. 90, 91*

[Buc+20] Antonio Bucchiarone et al. “Grand challenges in model-driven engineering: an analysis of the state of the research.” In: *Software and Systems Modeling* 19.1 (2020-01), pp. 5–13. DOI: 10.1007/s10270-019-00773-6. *used on: p. 4*

[But11a] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Sci. & Business Media, 2011. *used on: pp. 68, 74*

[But11b] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Springer US, 2011. DOI: 10.1007/978-1-4614-0676-1. *used on: p. 68*

[BWV14] A. Bubeck, F. Weisshardt, and A. Verl. “BRIDE - A toolchain for framework-independent development of industrial service robot applications.” In: *ISR/Robotik 2014; 41st International Symposium on Robotics*. 2014-06, pp. 1–6. *used on: pp. 38, 40*

[Cal+12] Daniele Calisi et al. “Design choices for modular and flexible robotic software development: the OpenRDK viewpoint.” In: *Journal of Software Engineering for Robotics (JOSER)* 3.1 (2012), pp. 13–27. DOI: 10.6092/JOSER_2012_03_01_p13. *used on: p. 22*

[Car12] Jan Carlson. “Timing analysis of component-based embedded systems.” In: *ACM SIGSOFT Symp. Component Based Software Engineering*. ACM. 2012, pp. 151–156. *used on: pp. 65, 66*

[Cas+19] Daniel Casini et al. “Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling.” In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Ed. by Sophie Quinton. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 6:1–6:23. DOI: `10.4230/LIPIcs.ECRTS.2019.6`. *used on: p. 157*

[CBG01] Matteo Corti, Roberto Brega, and Thomas Gross. “Approximation of Worst-Case Execution Time for Preemptive Multitasking Systems.” In: *Languages, Compilers, and Tools for Embedded Systems*. Ed. by Jack Davidson and Sang Lyul Min. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 178–198. *used on: p. 67*

[CDK05] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005. *used on: p. 177*

[CFD19] L. Chen, L. F. C. Figueredo, and M. Dogar. “Manipulation Planning Using Environmental Contacts to Keep Objects Stable under External Forces.” In: *2019 IEEE-RAS 19th International Conference on Humanoid Robots (Humanoids)*. 2019, pp. 417–424. DOI: `10.1109/Humanoids43949.2019.9034998`. *used on: p. 92*

[CFK16] Wan Kyun Chung, Li-Chen Fu, and Torsten Kröger. “Motion Control.” In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Cham: Springer International Publishing, 2016, pp. 163–194. ISBN: 978-3-319-32552-1. DOI: `10.1007/978-3-319-32552-1\_8`. *used on: p. 18*

[CG00] Junghoo Cho and Hector Garcia-Molina. “Synchronizing a Database to Improve Freshness.” In: *SIGMOD Rec.* 29.2 (2000-05), pp. 117–128. ISSN: 0163-5808. DOI: `10.1145/335191.335391`. *used on: p. 176*

[Cho+13] L. K. Chong et al. “Integrated Timing Analysis of Application and Operating Systems Code.” In: *IEEE Real-Time Systems Symp. (RSS)*. 2013-12, pp. 128–139. DOI: `10.1109/RTSS.2013.21`. *used on: pp. 65, 66*

[CK16] Imran Ali Chaudhry and Abid Ali Khan. “A research survey: review of flexible job shop scheduling techniques.” In: *International Transactions in Operational Research* 23.3 (2016), pp. 551–591. DOI: `10.1111/itor.12199`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/itor.12199`. *used on: p. 76*

[CL02] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. USA: Artech House, Inc., 2002. ISBN: 978-1-58053-327-0. *used on: p. 23*

[CMC20] CMCI. *RobMoSys ITP: Composable Models for Compliant Interaction Control - CMCI*. RobMoSys `https://robmosys.eu/`. 2020-11-01. URL: `https://robmosys.eu/cmci/` (visited on 2021-03-01). *used on: pp. 64, 126, 154*

[Cog19] CogIMon. *Cognitive Interaction in Motion - CogIMon*. CogIMon `https://cogimon.eu/cognitive-interaction-motion-cogimon-0`. 2019-05-07. URL: `https://www.youtube.com/watch?v=8fDuNwFkaxg` (visited on 2020-12-01). *used on: pp. 2, 5, 48, 126*

[Com+16] B. Combemale et al. *Engineering modeling languages*. Chapman & Hall/CRC innovations in software engineering and software development. Boca Raton: Taylor & Francis, CRC Press, 2016. ISBN: 978-1-315-38793-2. *used on: pp. 29, 177*

[Cor+20] Enrique Coronado et al. “Visual Programming Environments for End-User Development of intelligent and social robots, a systematic review.” In: *Journal of Computer Languages* 58 (2020), p. 100970. ISSN: 2590-1184. DOI: 10.1016/j.cola.2020.100970. *used on: p. 16*

[Cor18] Rick Cory. *IIWA Bimanual Manipulation (Sim vs Reality Side-by-Side)*. Toyota Research Institute Robotics Team. 2018-06-01. URL: https://www.youtube.com/watch?v=X9QuMrx-psk (visited on 2020-12-01). *used on: p. 2*

[Cou15] Erwin Coumans. “Bullet Physics Simulation.” In: *ACM SIGGRAPH 2015 Courses*. SIGGRAPH ’15. Los Angeles, California: Association for Computing Machinery, 2015. DOI: 10.1145/2776880.2792704. *used on: p. 48*

[COU16] Marta Cialdea Mayer, Andrea Orlandini, and Alessandro Umbrico. “Planning and execution with flexible timelines: a formal account.” In: *Acta Informatica* 53.6 (2016-10), pp. 649–680. ISSN: 1432-0525. DOI: 10.1007/s00236-015-0252-z. *used on: p. 66*

[CP00] Antoine Colin and Isabelle Puaut. “Worst Case Execution Time Analysis for a Processor WithBranch Prediction.” In: *Real-Time Syst.* 18.2 (2000-05), pp. 249–274. ISSN: 0922-6443. DOI: 10.1023/A:1008149332687. *used on: p. 67*

[CP10] Walter Cazzola and Davide Poletti. “DSL Evolution Through Composition.” In: *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*. RAM-SE ’10. Maribor, Slovenia: ACM, 2010, 6:1–6:6. DOI: 10.1145/1890683.1890689. *used on: p. 42*

[Cre+01] Valentin Crettaz et al. “Integrating the ConcernBASE Approach with SADL.” In: *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Ed. by Martin Gogolla and Cris Kobryn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 166–181. *used on: p. 179*

[Crn+02] Ivica Crnkovic et al. “Basic Concepts in CBSE.” In: *Building reliable component-based software systems*. Artech House computing, 2002, pp. 1–22. *used on: p. 24*

[CWP96] J. E. Colgate, W. Wannasuphoprasit, and M. A. Peshkin. “Cobots: Robots for Collaboration with Human Operators.” In: *International Mechanical Engineering Congress and Exposition, Atlanta*. 1996, pp. 433–439. *used on: p. 3*

[de +05] J. de Schutter et al. “Unified Constraint-Based Task Specification for Complex Sensor-Based Robot Systems.” In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 2005-04, pp. 3607–3612. DOI: 10.1109/ROBOT.2005.1570669. *used on: p. 16*

[de +07] Joris de Schutter et al. “Constraint-based Task Specification and Estimation for Sensor-Based Robot Systems in the Presence of Geometric Uncertainty.” In: *The International Journal of Robotics Research* 26.5 (2007), pp. 433–455. DOI: 10.1177/027836490707809107. *used on: pp. 5, 16, 17, 91*

[de +21] Edson de Araújo Silva et al. “A survey of Model Driven Engineering in robotics.” In: *Journal of Computer Languages* 62 (2021). ISSN: 2590-1184. DOI: 10.1016/j.cola.2020.101021. *used on: p. 64*

[Deh18] Niels Dehio. “Prioritized Multi-Objective Robot Control.” In: *Universitätsbibliothek Braunschweig* (2018). DOI: 10.24355/dbbs.084-201812051220-0. *used on: pp. 20, 98, 129, 134, 136*

[Dej16] Cyrille Dejemeppe. "Constraint programming algorithms and models for scheduling applications." Dissertation. Belgium: Louvain School of Engineering, 2016. *used on: pp. 69, 76*

[Dho+12] Saadia Dhouib et al. "RobotML, a Domain-specific Language to Design, Simulate and Deploy Robotic Applications." In: *Proc. 3rd Int. Conf. Simulation, Modeling, and Programming for Autonomous Robots*. SIMPAR'12. Tsukuba, Japan: Springer-Verlag, 2012, pp. 149–160. DOI: `10.1007/978-3-642-34327-8_16`. *used on: pp. 32, 38, 42, 49, 66*

[Dic21] Oxford Learner's Dictionaries. *Domain*. Oxford University Press. 2021. URL: `https://www.oxfordlearnersdictionaries.com/definition/english/domain?q=domain`. *used on: p. 176*

[Die+17] Christian Dietrich et al. "SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems." In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2017, pp. 37–48. DOI: `10.1109/RTAS.2017.37`. *used on: p. 67*

[Dij82] Edsger W. Dijkstra. "On the Role of Scientific Thought." In: *Selected Writings on Computing: A personal Perspective*. New York, NY: Springer New York, 1982, pp. 60–66. ISBN: 978-1-4612-5695-3. DOI: `10.1007/978-1-4612-5695-3_12`. *used on: p. 176*

[Dim] Dimensions. *Dimensions*. Digital Science & Research Solutions, Inc. URL: `https://app.dimensions.ai/discover/publication` (visited on 2021-02-13). *used on: p. 104*

[DKS18] N. Dehio, D. Kubus, and J. J. Steil. "Continuously Shaping Projections and Operational Space Tasks." In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 5995–6002. DOI: `10.1109/IROS.2018.8593400`. *used on: p. 22*

[DOA15] Alexander Dietrich, Christian Ott, and Alin Albu-Schäffer. "An overview of null space projections for redundant, torque-controlled robots." In: *The International Journal of Robotics Research* 34.11 (2015), pp. 1385–1400. DOI: `10.1177/0278364914566516`. *used on: pp. 20, 22, 99*

[Don85] B. Donald. "On motion planning with six degrees of freedom: Solving the intersection problems in configuration space." In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. 1985-03, pp. 536–541. DOI: `10.1109/ROBOT.1985.1087334`. *used on: p. 95*

[DRS15] N. Dehio, R. F. Reinhart, and J. J. Steil. "Multiple task optimization with a mixture of controllers for motion generation." In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 6416–6421. DOI: `10.1109/IROS.2015.7354294`. *used on: p. 19*

[DS19] Niels Dehio and Jochen J. Steil. "Dynamically-consistent Generalized Hierarchical Control." In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 1141–1147. DOI: `10.1109/ICRA.2019.8793553`. *used on: p. 100*

[DT19] Giacomo Da Col and Erich C. Teppan. "Industrial Size Job Shop Scheduling Tackled by Present Day CP Solvers." In: *Principles and Practice of Constraint Programming*. Ed. by Thomas Schiex and Simon de Givry. Cham: Springer International Publishing, 2019, pp. 144–160. *used on: p. 76*

[Dup07] Lyn Dupré. *BUGS in writing. A guide to debugging your prose*. Rev 10th ed. Boston: Addison-Wesley, 2007. ISBN: 978-0-201-37921-1. *used on: p. 207*

[EB15] C. Eppner and O. Brock. “Planning grasp strategies That Exploit Environmental Constraints.” In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4947–4952. DOI: 10.1109/ICRA.2015.7139886. *used on: p. 92*

[EGR12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. “Language Composition Untangled.” In: *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. LDTA ’12. Tallinn, Estonia: ACM, 2012, 7:1–7:8. DOI: 10.1145/2427048.2427055. *used on: pp. 32, 42*

[Emia] Franka Emica. *Panda*. URL: https://www.franka.de/ (visited on 2020-12-01). *used on: p. 3*

[Emm+13] C. Emmerich et al. “Assisted Gravity Compensation to cope with the complexity of kinesthetic teaching on redundant robots.” In: *2013 IEEE International Conference on Robotics and Automation*. 2013-05, pp. 4322–4328. DOI: 10.1109/ICRA.2013.6631189. *used on: p. 19*

[Erd+15] Sebastian Erdweg et al. “Evaluating and Comparing Language Workbenches.” In: *Comput. Lang. Syst. Struct.* 44.PA (2015-12), pp. 24–47. ISSN: 1477-8424. DOI: 10.1016/j.cl.2015.08.007. *used on: p. 42*

[EWS03] D. Erickson, M. Weber, and I. Sharf. “Contact Stiffness and Damping Estimation for Robotic Systems.” In: *The International Journal of Robotics Research* 22.1 (2003), pp. 41–57. DOI: 10.1177/0278364903022001004. *used on: p. 92*

[Fan] Fanuc. *CRX*. URL: https://www.fanucamerica.com/products/robots/series/collaborative-robot-crx#new-era (visited on 2020-12-01). *used on: p. 3*

[FBC12] Marco Frigerio, Jonas Buchli, and Darwin G Caldwell. “Model based code generation for kinematics and dynamics computations in robot controllers.” In: *Proc. Workshop Software Development and Integration in Robotics*. St. Paul, Minnesota, USA, 2012. *used on: pp. 7, 41*

[FBC13] Marco Frigerio, Jonas Buchli, and D.G. Caldwell. “A Domain Specific Language for Kinematic Models and Fast Implementations of Robot Dynamics Algorithms.” In: *Proc. Workshop Domain-Specific Languages and models for Robotic systems*. 2013. eprint: arXiv:1301.7190v1. *used on: p. 41*

[FBJ06] J Randall Flanagan, Miles C Bowman, and Roland S Johansson. “Control strategies in object manipulation tasks.” In: *Current Opinion in Neurobiology* 16.6 (2006-12), pp. 650–659. DOI: 10.1016/j.conb.2006.10.005. *used on: p. 99*

[Fes] Festo. *Bionic Handling Assistant*. URL: https://www.festo.com/group/en/cms/10241.htm (visited on 2020-12-01). *used on: p. 1*

[Flo+09] G. Florez-Puga et al. “Query-enabled behavior trees.” In: *IEEE Transactions on Computational Intelligence and AI in Games* 1.04 (2009-10), pp. 298–308. ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2009.2036369. *used on: p. 25*

[FO16] Roy Featherstone and David E. Orin. “Dynamics.” In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Cham: Springer International Publishing, 2016, pp. 37–66. ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_3. *used on: p. 18*

[Fow05] Martin Fowler. “Language Workbenches: The Killer-App for Domain Specific Languages?” In: (2005). *used on: pp. 43, 177*

[Fow10] Martin Fowler. *Domain-specific languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 978-0-321-71294-3. *used on: p. 29*

[FTK99] R. Featherstone, S. S. Thiebaut, and O. Khatib. "A general contact model for dynamically-decoupled force/motion control." In: *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*. 1999-05, 3281–3286 vol.4. DOI: 10.1109/ROBOT.1999.774098. *used on: pp. 93, 94*

[Gam+95] Erich Gamma et al. *Design Patterns – Elements of Reusable Object-Oriented Software*. 1st ed. 37. Reprint (2009). Amsterdam: Addison-Wesley Longman, 1995. ISBN: 978-0-201-63361-0. *used on: p. 175*

[Gar+21] Caelan Reed Garrett et al. "Integrated Task and Motion Planning." In: *Annual Review of Control, Robotics, and Autonomous Systems* 4.1 (2021), pp. 265–293. DOI: 10.1146/annurev-control-091420-084139. *used on: p. 161*

[Gar09] Willow Garage. *Unified Robot Description Format*. Organisation Open Source Robotics Foundation (OSRF). 2009. URL: http://wiki.ros.org/urdf (visited on 2020-08-04). *used on: p. 51*

[Gho10] Debasish Ghosh. *DSLs in action*. Manning Publications Co., 2010. *used on: p. 7*

[Gob+16] N. Gobillot et al. "Measurement-based real-time analysis of robotic software architectures." In: *IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. 2016-10, pp. 3306–3311. DOI: 10.1109/IROS.2016.7759509. *used on: pp. 65, 66*

[Goo] Google. *Google Trends*. Google LLC. URL: https://trends.google.com (visited on 2021-02-13). *used on: p. 104*

[Has+05] R. Haschke et al. "Task-oriented quality measures for dextrous grasping." In: *2005 International Symposium on Computational Intelligence in Robotics and Automation*. 2005, pp. 689–694. DOI: 10.1109/CIRA.2005.1554357. *used on: p. 98*

[HC01] George T. Heineman and William T. Councill, eds. *Component-Based Software Engineering : Putting the Pieces Together*. eng. 1. print. Boston: Addison-Wesley, 2001. ISBN: 978-0-201-70485-3. *used on: p. 23*

[Hen+17] B. Henze et al. "Multi-contact balancing of humanoid robots in confined spaces: Utilizing knee contacts." In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 697–704. DOI: 10.1109/IROS.2017.8202227. *used on: p. 2*

[Hoc+13] Nico Hochgeschwender et al. "A model-based approach to software deployment in robotics." In: *Proc. Int. Conf. Intelligent Robots and Systems*. 2013, pp. 3907–3914. *used on: p. 38*

[Hof+17a] E. M. Hoffman et al. "Robot control for dummies: Insights and examples using OpenSoT." In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. 2017-11, pp. 736–741. DOI: 10.1109/HUMANOIDS.2017.8246954. *used on: pp. 99, 115*

[Hof+17b] Enrico Mingo Hoffman et al. "Robot control for dummies: Insights and examples using OpenSoT." In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. Ed. by IEEE-RAS International Conference on Humanoid Robotics. [Piscataway, NJ]: IEEE, 2017, pp. 736–741. DOI: 10.1109/HUMANOIDS.2017.8246954. *used on: p. 122*

[Hog85] Neville Hogan. “Impedance Control: An Approach to Manipulation: Part I—Theory.” In: *Journal of Dynamic Systems, Measurement, and Control* 107.1 (1985-03), pp. 1–7. DOI: 10.1115/1.3140702. *used on: pp. 19, 95*

[HR13] Andreas Horst and Bernhard Rumpe. “Towards Compositional Domain Specific Languages.” In: *7th Workshop Multi-Paradigm Modeling* (2013), pp. 1–5. *used on: pp. 42, 57*

[HS85] John M. Hollerbach and Ki Suh. “Redundancy resolution of manipulators through torque optimization.” In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation* 2 (1985), pp. 1016–1021. *used on: p. 19*

[HST92] T. Hasegawa, T. Suehiro, and K. Takase. “A model-based manipulation system with skill-based execution.” In: *IEEE Trans. Robot. Autom.* 8.5 (1992-10), pp. 535–544. ISSN: 2374-958X. DOI: 10.1109/70.163779. *used on: p. 7*

[HT21] Enrico Mingo Hoffman and Nikos G. Tsagarakis. “The Math of Tasks: a Domain Specific Language for constraint-based task specification.” In: *International Journal of Humanoid Robotics* (2021-06). DOI: 10.1142/s0219843621500080. *used on: p. 122*

[Hut+16] Marco Hutter et al. “ANYmal - a highly mobile and dynamic quadrupedal robot.” In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 38–44. DOI: 10.1109/IROS.2016.7758092. *used on: p. 3*

[Ion21] Tudor B Ionescu. “Leveraging Graphical User Interface Automation for Generic Robot Programming.” In: *Robotics* 10.1 (2021), p. 3. *used on: p. 16*

[Iun+20] Aníbal Iung et al. “Systematic mapping study on domain-specific language development tools.” In: *Empirical Software Engineering* 25.5 (2020-08), pp. 4205–4249. DOI: 10.1007/s10664-020-09872-1. *used on: p. 64*

[Jar+13] G. Jarquín et al. “Real-time smooth task transitions for hierarchical inverse kinematics.” In: *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*. 2013, pp. 528–533. DOI: 10.1109/HUMANOIDS.2013.7030024. *used on: pp. 22, 154*

[JC92] Roland S. Johansson and Kelly J. Cole. “Sensory-motor coordination during grasping and manipulative actions.” In: *Elsevier BV Current Opinion in Neurobiology* 2.6 (1992-12), pp. 815–823. DOI: 10.1016/0959-4388(92)90139-c. *used on: p. 99*

[JZ17] Víctor Juan Expósito Jiménez and Herwig Zeiner. “A Domain Specific Language for Robot Programming in the Wood Industry – A Practical Example.” In: *Proceedings of the 14th International Conference on Informatics in Control, Automation and Robotics*. 2017, pp. 549–555. ISBN: 978-989-758-264-6. DOI: 10.5220/0006397205490555. *used on: p. 28*

[Kaj+03] S. Kajita et al. “Biped walking pattern generation by using preview control of zero-moment point.” In: *2003 IEEE International Conference on Robotics and Automation*. 2003, pp. 1620–1626. DOI: 10.1109/ROBOT.2003.1241826. *used on: p. 83*

[Kar+09] Gabor Karsai et al. “Design Guidelines for Domain Specific Languages.” In: *Proc. 9th OOPSLA Workshop Domain-Specific Modeling*. 2009, pp. 7–13. *used on: p. 42*

[KB12a] Markus Klotzbücher and Herman Bruyninckx. "Coordinating Robotic Tasks and Systems with rFSM Statecharts." eng. In: *JOSER: Journal of Software Engineering for Robotics* 3.1 (2012), pp. 28–56. ISSN: 2035-3928. *used on: p. 25*

[KB12b] I. Kresse and M. Beetz. "Movement-aware action control — Integrating symbolic and control-theoretic action execution." In: *2012 IEEE International Conference on Robotics and Automation*. 2012, pp. 3245–3251. DOI: 10.1109/ICRA.2012.6225119. *used on: p. 92*

[KH04] Nathan Koenig and Andrew Howard. "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator." In: *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*. Sendai, Japan, 2004-09, pp. 2149–2154. *used on: p. 48*

[Kha87] O. Khatib. "A unified approach for motion and force control of robot manipulators: The operational space formulation." In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53. DOI: 10.1109/JRA.1987.1087068. *used on: pp. 19, 20*

[Kim+19] Sanghyun Kim et al. "Continuous Task Transition Approach for Robot Controller Based on Hierarchical Quadratic Programming." In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 1603–1610. DOI: 10.1109/LRA.2019.2896769. *used on: p. 154*

[KKK16] Mohammad Khansari, Ellen Klingbeil, and Oussama Khatib. "Adaptive human-inspired compliant contact primitives to perform surface–surface contact under uncertainty." In: *The International Journal of Robotics Research* 35.13 (2016), pp. 1651–1675. DOI: 10.1177/0278364916648389. *used on: p. 95*

[KLB08] Imin Kao, Kevin Lynch, and Joel W. Burdick. "Contact Modeling and Manipulation." In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 647–669. ISBN: 978-3-540-30301-5. DOI: 10.1007/978-3-540-30301-5_28. *used on: pp. 93, 94, 96, 99*

[Klo+11] M. Klotzbücher et al. "Reusable hybrid force-velocity controlled motion specifications with executable Domain Specific Languages." In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Ed. by IEEE Staff. [Place of publication not identified]: IEEE, 2011, pp. 4684–4689. DOI: 10.1109/IROS.2011.6094782. *used on: pp. 5, 17, 90, 91, 104, 106*

[KLV09] J. Krueger, T. Lien, and A. Verl. "Cooperation of human and machines in assembly lines." In: *Cirp Annals-manufacturing Technology* 58 (2009), pp. 628–646. *used on: p. 3*

[KLW11] O. Kanoun, F. Lamiraux, and P. Wieber. "Kinematic Control of Redundant Manipulators: Generalizing the Task-Priority Framework to Inequality Task." In: *IEEE Transactions on Robotics* 27.4 (2011), pp. 785–792. DOI: 10.1109/TRO.2011.2142450. *used on: pp. 21, 22*

[Kou16] Anis Koubaa, ed. *Robot Operating System (ROS)*. Studies in Computational Intelligence. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-26052-5. DOI: 10.1007/978-3-319-26054-9. *used on: p. 23*

[Koy+08] K. Koyanagi et al. “A pattern generator of humanoid robots walking on a rough terrain using a handrail.” In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2008-09, pp. 2617–2622. DOI: 10.1109/IROS.2008.4650686. *used on: p. 4*

[Kra10] Johan Kraft. “Enabling Timing Analysis of Complex Embedded Software Systems.” Doctoral dissertation. Mälardalen University Press, 2010-08. *used on: pp. 65, 66*

[KSB16] David Kortenkamp, Reid Simmons, and Davide Brugali. “Robotic Systems Architectures and Programming.” In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Cham: Springer International Publishing, 2016, pp. 283–306. ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_12. *used on: pp. 15, 23, 25, 178*

[KSP08] Oussama Khatib, Luis Sentis, and Jae-Heung Park. “A Unified Framework for Whole-Body Humanoid Robot Control with Multiple Constraints and Contacts.” In: *European Robotics Symposium 2008*. Ed. by Herman Bruyninckx. Vol. 44. Springer Tracts in Advanced Robotics. Berlin and Heidelberg: Springer, 2008, pp. 303–312. ISBN: 978-3-540-78315-2. DOI: 10.1007/978-3-540-78317-6{\textunderscore}31. *used on: p. 92*

[Kuk] Kuka. *IIWA*. URL: https://www.kuka.com/en-gb/products/robotics-systems/industrial-robots/lbr-iiwa (visited on 2020-12-01). *used on: p. 3*

[Lab15] Trac Labs. *ATLAS turning a valve at DRC Finals*. TRACLabs https://traclabs.com/. 2015. URL: https://www.youtube.com/watch?v=MXyy3AV9Mmk (visited on 2020-12-01). *used on: p. 2*

[Lau+14] Michaël Lauer et al. “End-to-end latency and temporal consistency analysis in networked real-time systems.” In: *International Journal of Critical Computer-Based Systems* vol. 5.n° 3/4 (2014), pp. 172–196. DOI: 10.1504/IJCCBS.2014.064667. *used on: p. 72*

[LDC12] Charles Lesire, David Doose, and Hugues Cassé. “Mauve: a Component-based Modeling Framework for Real-time Analysis of Robotic Applications.” In: *Proc. 7th Workshop Software Development and Integration in Robotics*. 2012. *used on: p. 38*

[LDC18] E. Lutscher, E. C. Dean-León, and G. Cheng. “Hierarchical Force and Positioning Task Specification for Indirect Force Controlled Robots.” In: *IEEE Transactions on Robotics* 34.1 (2018), pp. 280–286. DOI: 10.1109/TRO.2017.2765674. *used on: p. 98*

[Lee+17] Chiwon Lee et al. “Soft robot review.” In: *International Journal of Control, Automation and Systems* 15.1 (2017-01), pp. 3–15. DOI: 10.1007/s12555-016-0462-3. *used on: p. 1*

[Lei19] Daniel Sebastian Leidner. *Cognitive Reasoning for Compliant Robot Manipulation*. Vol. 127. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-04857-0. DOI: 10.1007/978-3-030-04858-7. *used on: pp. 2, 3, 6*

[Len+13] Sébastien Lengagne et al. “Generation of whole-body optimal dynamic multi-contact motions.” In: *The International Journal of Robotics Research* 32.9-10 (2013), pp. 1104–1119. DOI: 10.1177/0278364913478990. *used on: p. 92*

[Lie77] Alain Liegeois. “Automatic Supervisory Control of the Configuration and Behavior of Multibody Mechanisms.” In: *IEEE Transactions on Systems, Man, and Cybernetics* 7.12 (1977-12), pp. 868–871. DOI: 10.1109/TSMC.1977.4309644. *used on: p. 19*

[Lin+18] Hsiu-Chin Lin et al. “A Projected Inverse Dynamics Approach for Multi-arm Cartesian Impedance Control.” In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. United States: Institute of Electrical and Electronics Engineers (IEEE), 2018-09, pp. 5421–5428. DOI: 10.1109/ICRA.2018.8461202. *used on: pp. 22, 98*

[LMT84] Tomás Lozano-Pérez, Matthew T. Mason, and Russell H. Taylor. “Automatic Synthesis of Fine-Motion Strategies for Robots.” In: *The International Journal of Robotics Research* 3.1 (1984), pp. 3–24. DOI: 10.1177/027836498400300101. *used on: p. 95*

[Lot+15] Alex Lotz et al. “Modeling Non-Functional Application Domain Constraints for Component-Based Robotics Software Systems.” In: *Sixth Int. Workshop Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*. 2015. *used on: pp. 68, 71*

[Lot+16] A. Lotz et al. “Combining robotics component-based model-driven development with a model-based performance analysis.” In: *IEEE Int. Conf. Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. 2016-12, pp. 170–176. DOI: 10.1109/SIMPAR.2016.7862392. *used on: pp. 65, 66, 85*

[Lot18] A. Lotz. “Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System.” Dissertation. München: Technische Universität München, 2018. *used on: pp. 4, 66*

[LRS11] Thomas Lens, Katayon Radkhah, and Oskar von Stryk. “Simulation of dynamics and realistic contact forces for manipulators and legged robots with high joint elasticity.” In: *2011 15th International Conference on Advanced Robotics (ICAR)*. IEEE, 2011-06. DOI: 10.1109/icar.2011.6088619. *used on: pp. 92, 93, 95*

[LTP15] Mingxing Liu, Yang Tan, and Vincent Padois. “Generalized hierarchical control.” In: *Autonomous Robots* 40.1 (2015-05), pp. 17–31. DOI: 10.1007/s10514-015-9436-1. *used on: pp. 20, 154*

[MAC97] Douglas C. MacKenzie, Ronald Arkin, and Jonathan M. Cameron. “Multiagent Mission Specification and Execution.” In: *Autonomous Robots* 4.1 (1997-03), pp. 29–52. DOI: 10.1023/a:1008807102993. *used on: p. 25*

[Mal08] Frédéric Mallet. “Clock constraint specification language: specifying clock constraints with UML/MARTE.” In: *Innovations in Systems and Software Engineering* 4.3 (2008-10), pp. 309–314. ISSN: 1614-5054. DOI: 10.1007/s11334-008-0055-2. *used on: p. 66*

[Man+09] N. Mansard et al. “A versatile Generalized Inverted Kinematics implementation for collaborative working humanoid robots: The Stack Of Tasks.” In: 2009. *used on: pp. 19, 22*

[Mar95] R. Martin. “OO Design Quality Metrics: an Analysis of Dependencies.” In: *ROAD* 2.3 (1995). *used on: pp. 59, 61, 62*

[Mas81] M. T. Mason. “Compliance and Force Control for Computer Controlled Manipulators.” In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.6 (1981-06), pp. 418–432. DOI: 10.1109/TSMC.1981.4308708. *used on: pp. 3, 5, 8, 16, 93*

[Mat] MathWorks. *Simulink*. URL: https://www.mathworks.com/products/simulink.html (visited on 2020-12-01). *used on: p. 7*

[MB17] Ali Ahmad Malik and Arne Bilberg. “Framework to Implement Collaborative Robots in Manual Assembly: A Lean Automation Approach.” In: *Proceedings of the 28th DAAAM International Symposium*. 2017, pp. 1151–1160. DOI: 10.2507/28th.daaam.proceedings.160. *used on: p. 3*

[MB20] T. Migimatsu and J. Bohg. “Object-Centric Task and Motion Planning in Dynamic Environments.” In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 844–851. DOI: 10.1109/LRA.2020.2965875. *used on: p. 92*

[MBT19] Jöm Malzahn, Eamon Barrett, and Nikos Tsagarakis. “A Rolling Flexure Mechanism for Progressive Stiffness Actuators.” In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8415–8421. DOI: 10.1109/ICRA.2019.8794004. *used on: p. 3*

[Mer17] Marjan Mernik. “Domain-Specific Languages: A Systematic Mapping Study.” In: *SOFSEM 2017: Theory and Practice of Computer Science*. Ed. by Bernhard Steffen et al. Cham: Springer International Publishing, 2017, pp. 464–472. *used on: p. 4*

[MFN06] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. “YARP: Yet Another Robot Platform.” In: *International Journal of Advanced Robotic Systems* 3.1 (2006), p. 8. DOI: 10.5772/5761. eprint: http://dx.doi.org/10.5772/5761. *used on: p. 41*

[Mir+18] Seyed Sina Mirrazavi Salehian et al. “Transitioning with confidence during contact/non-contact scenarios.” In: *Workshop on Towards Robots that Exhibit Manipulation Intelligence. IROS 2018*. IEEE, 2018. *used on: p. 100*

[MK97] J.D. Morrow and P.K. Khosla. “Manipulation task primitives for composing robot skills.” In: *Proceedings of International Conference on Robotics and Automation*. IEEE, 1997. DOI: 10.1109/robot.1997.606800. *used on: p. 96*

[MM06] S. Martin and P. Minet. “Worst case end-to-end response times of flows scheduled with FP/FIFO.” In: *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*. 2006, pp. 54–54. DOI: 10.1109/ICNICONSMCL.2006.231. *used on: p. 67*

[MMS12a] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. “Implementation of End-to-End Latency Analysis for Component-Based Multi-Rate Real-Time Systems in Rubus-ICE.” In: *2012 9th IEEE International Workshop on Factory Communication Systems*. 2012, pp. 165–168. DOI: 10.1109/WFCS.2012.6242562. *used on: p. 69*

[MMS12b] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. “Translating end-to-end timing requirements to timing analysis model in component-based distributed real-time systems.” In: *ACM SIGBED Review* 9.4 (2012), pp. 17–20. *used on: pp. 68, 74*

[Moh20] Pouya Mohammadi. “Context Aware Body Regulation of Redundant Robots.” en. Doctoral dissertation. 2020. DOI: 10.24355/DBBS.084-202007271032-0. *used on: p. 21*

[Mor+13a] F. L. Moro et al. "An attractor-based Whole-Body Motion Control (WBMC) system for humanoid robots." In: *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*. 2013, pp. 42–49. DOI: 10.1109/HUMANOIDS.2013.7029953. *used on: pp. 19, 115*

[Mor+13b] F. L. Moro et al. "An attractor-based Whole-Body Motion Control (WBMC) system for humanoid robots." In: *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*. 2013-10, pp. 42–49. DOI: 10.1109/HUMANOIDS.2013.7029953. *used on: p. 20*

[Mor15] Matteo Morelli. "Automated Generation of Robotics Applications from Simulink and SysML Models." In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. New York, NY, USA: ACM, 2015, pp. 1948–1954. DOI: 10.1145/2695664.2695882. *used on: pp. 7, 65, 79*

[MPS21] Masoumeh Mansouri, Federico Pecora, and Peter Schüller. "Combining Task and Motion Planning: Challenges and Guidelines." In: *Frontiers in Robotics and AI* 8 (2021), p. 133. DOI: 10.3389/frobt.2021.637888. *used on: p. 161*

[MR11] Michael Mistry and Ludovic Righetti. "Operational Space Control of Constrained and Underactuated Systems." In: *Robotics: Science and Systems VII*. Robotics: Science and Systems Foundation, 2011-06. DOI: 10.15607/rss.2011.vii.031. *used on: pp. 21, 22*

[Mur+17] Luca Muratore et al. "XBotCore: A Real-Time Cross-Robot Software Platform." In: *2017 First IEEE International Conference on Robotic Computing (IRC)*. 2017, pp. 77–80. DOI: 10.1109/IRC.2017.45. *used on: p. 157*

[MW01] H. Mosemann and F. M. Wahl. "Automatic decomposition of planned assembly sequences into skill primitives." In: *IEEE Transactions on Robotics and Automation* 17.5 (2001-10), pp. 709–718. ISSN: 2374-958X. DOI: 10.1109/70.964670. *used on: pp. 15, 16, 90, 91*

[Nag+15] Vineet Nagrath et al. "Dynamic electronic institutions in agent oriented cloud robotic systems." In: *SpringerPlus* 4.1 (2015-03). DOI: 10.1186/s40064-015-0810-4. *used on: p. 31*

[Näg+18] F. Nägele et al. "A Prototype-Based Skill Model for Specifying Robotic Assembly Tasks." In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018-05, pp. 558–565. DOI: 10.1109/ICRA.2018.8462885. *used on: pp. 4, 5, 7, 16, 17, 90, 91, 104*

[Näg+19] Frank Nägele et al. "Composition and Incremental Refinement of Skill Models for Robotic Assembly Tasks." In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE, 25.02.2019 - 27.02.2019, pp. 177–182. DOI: 10.1109/IRC.2019.00034. *used on: pp. 2, 104, 106*

[NBB16] M. Neunert, T. Boaventura, and J. Buchli. "Why Off-The-Shelf Physics Simulators Fail In Evaluating Feedback Controller Performance - A Case Study For Quadrupedal Robots." In: *Advances in Cooperative Robotics*. World Scientific, 2016, pp. 464–472. DOI: 10.1142/9789813149137_0055. eprint: http://www.worldscientific.com/doi/pdf/10.1142/9789813149137_0055. *used on: p. 66*

[Neu] Neura-Robotics. *Maira*. URL: https://neura-robotics.com (visited on 2021-09-07). *used on: p. 3*

[NH87] Yoshihiko Nakamura and Hideo Hanafusa. "Optimal Redundancy Control of Robot Manipulators." In: *The International Journal of Robotics Research* 6.1 (1987), pp. 32–42. DOI: 10.1177/027836498700600103. *used on: p. 19*

[Nor+16b] Arne Nordmann et al. *Robotics DSL Zoo: An Overview of Domain-Specific Languages in Robotics*. 2016. URL: https://corlab.github.io/dslzoo/ (visited on 2020-08-07). *used on: pp. 4, 66, 178*

[Nor16] Arne Nordmann. "Modeling of motion primitive architectures using domain-specific languages." dissertation. University of Bielefeld, 2016. *used on: pp. 7, 29, 32, 119*

[NWS15] A. Nordmann, S. Wrede, and J. Steil. "Modeling of movement control architectures based on motion primitives using domain-specific languages." In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015-05, pp. 5032–5039. DOI: 10.1109/ICRA.2015.7139899. *used on: pp. 5, 7, 91*

[Ogr12] Petter Ogren. "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees." In: *AIAA Guidance, Navigation, and Control Conference*. American Institute of Aeronautics and Astronautics, 2012-08. DOI: 10.2514/6.2012-4458. *used on: p. 25*

[Ouh+11] Yassine Ouhammou et al. "Towards a Simple Meta-model for Complex Real-Time and Embedded Systems." In: *Model and Data Engineering*. Ed. by Ladjel Bellatreche and Filipe Mota Pinto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 226–236. *used on: pp. 7, 65*

[Ouh13] Yassine Ouhammou. "Model-based Framework for Using Advanced Scheduling Theory in Real-Time Systems Design." Dissertation. France: Laboratory of Computer Science and Automatic Control for Systems, 2013. *used on: pp. 69, 70*

[PA17] L. Peternel and A. Ajoudani. "Robots learning from robots: A proof of concept study for co-manipulation tasks." In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. 2017, pp. 484–490. DOI: 10.1109/HUMANOIDS.2017.8246916. *used on: p. 2*

[Pai+15] Nicholas Paine et al. "Actuator Control for the NASA-JSC Valkyrie Humanoid Robot: A Decoupled Dynamics Approach for Torque Control of Series Elastic Robots." In: *J. Field Robot.* 32.3 (2015), pp. 378–396. ISSN: 1556-4959. DOI: 10.1002/rob.21556. *used on: p. 3*

[PB93] Simeon P. Patarinski and Roumen G. Botev. "Robot force control: A review." In: *Elsevier BV Mechatronics* 3.4 (1993-08), pp. 377–398. DOI: 10.1016/0957-4158(93)90012-q. *used on: p. 93*

[PCW18] Yuan-Chih Peng, David S. Carabis, and John T. Wen. "Collaborative manipulation with multiple dual-arm robots under human guidance." In: *International Journal of Intelligent Robotics and Applications* 2.2 (2018-03), pp. 252–266. DOI: 10.1007/s41315-018-0053-y. *used on: p. 2*

[Ped+13] Mikkel Rath Pedersen et al. "On the Integration of Hardware-Abstracted Robot Skills for use in Industrial Scenarios." English. In: Second international workshop on Cognitive Robotics Systems : Replicating Human Actions and Activities, CRS 2013 ; Conference date: 03-11-2013. 2013. *used on: p. 91*

[Ped+16] Mikkel Rath Pedersen et al. “Robot skills for manufacturing: From concept to industrial deployment.” In: *Robotics and Computer-Integrated Manufacturing* 37 (2016), pp. 282–291. DOI: 10.1016/j.rcim.2015.04.002. *used on: pp. 16, 90, 91*

[Per+12] M. Peraldi-Frati et al. “A Timing Model for Specifying Multi Clock Automotive Systems: The Timing Augmented Description Language V2.” In: *2012 IEEE 17th Int. Conf. Engineering of Complex Computer Systems*. 2012-07, pp. 230–239. *used on: pp. 7, 65*

[PK05] Jaeheung Park and O. Khatib. “Multi-Link Multi-Contact Force Control for Manipulators.” In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 2005-04, pp. 3613–3618. DOI: 10.1109/ROBOT.2005.1570670. *used on: p. 99*

[PK08] Jaeheung Park and Oussama Khatib. “Robot multiple contact control.” In: *Robotica* 26.5 (2008), pp. 667–677. DOI: 10.1017/S0263574708004281. *used on: pp. 95, 99*

[PKM02] J. Pratt, Benjamin T. Krupp, and Christopher J. Morse. “Series elastic actuators for high fidelity force control.” In: *Industrial Robot-an International Journal* 29 (2002), pp. 234–241. *used on: p. 2*

[Pra+09] Erwin Prassler et al. “The Use of Reuse for Designing and Manufacturing Robots.” In: 2009. *used on: p. 24*

[Pro19] Team iYU Pro. *The Finalists of the KUKA Innovation Award 2019: Team iYU Pro, robot for personalized massages*. Capsix Robotics https://capsix-robotics.com/. 2019-11-08. URL: https://www.youtube.com/watch?v=tyKUsq1whMI (visited on 2020-12-01). *used on: p. 2*

[PW03] Mikel D Petty and Eric W Weisel. “A composability lexicon.” In: *Spring 2003 Simulation Interoperability Workshop*. Orlando, USA, 2003. *used on: p. 176*

[Que+14] J. F. Queißer et al. “An active compliant control mode for interaction with a pneumatic soft robot.” In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 573–579. DOI: 10.1109/IROS.2014.6942617. *used on: p. 1*

[Qui+09] Morgan Quigley et al. “ROS: an open-source Robot Operating System.” In: *ICRA Workshop on Open Source Software*. 2009. *used on: pp. 24, 41*

[Rad+17] Samuel Rader et al. “Highly integrated sensor-actuator-controller units for modular robot design.” In: *2017 IEEE International Conference on Advanced Intelligent Mechatronics (AIM)*. 2017, pp. 1160–1166. DOI: 10.1109/AIM.2017.8014175. *used on: pp. 2, 3*

[Rat+12] Daniel Ratiu et al. “Implementing modular domain specific languages and analyses.” In: *Proc. Workshop Model-Driven Engineering, Verification and Validation*. ACM. 2012, pp. 35–40. *used on: pp. 32, 41*

[RC81] M. H. Raibert and J. J. Craig. “Hybrid Position/Force Control of Manipulators.” In: *Journal of Dynamic Systems, Measurement, and Control* 103.2 (1981-06), pp. 126–133. ISSN: 0022-0434. DOI: 10.1115/1.3139652. eprint: https://asmedigitalcollection.asme.org/dynamicsystems/article-pdf/103/2/126/5777717/126_1.pdf. *used on: p. 93*

[RDN10] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. "Language Boxes: Bending the Host Language with Modular Language Changes." In: *Proceedings of the Second International Conference on Software Language Engineering*. SLE'09. Denver, CO: Springer-Verlag, 2010, pp. 274–293. DOI: 10.1007/978-3-642-12107-4_20. *used on: p. 42*

[RE96] Matthias Radestock and Susan Eisenbach. "Coordination in evolving systems." In: *Trends in Distributed Systems CORBA and Beyond*. Ed. by Otto Spaniol, Claudia Linnhoff-Popien, and Bernd Meyer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 162–176. *used on: p. 24*

[RGG94] Arvind K Ramadorai, U Ganapathy, and F Guida. "A generic kinematics software package." In: *Proc. Int. Conf. Robotics and Automation*. IEEE. 1994, pp. 3331–3336. *used on: p. 41*

[Rin+16] Jan Oliver Ringert et al. "Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems." In: *Journal of Software Engineering for Robotics* 6.1 (2016), pp. 33–57. *used on: pp. 7, 38*

[RMT14] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. "Model-Driven Software Development Approaches in Robotics Research." In: *Proceedings of the 6th International Workshop on Modeling in Software Engineering*. MiSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 43–48. DOI: 10.1145/2593770.2593781. *used on: pp. 30, 31, 33, 65, 66*

[Rob] Universal Robot. *UR*. URL: https://www.universal-robots.com/ (visited on 2020-12-01). *used on: p. 3*

[Rob16] RobMoSys. *RobMoSys Grant Agreement: Excerpt of Section 1 Excellence*. 2016. *used on: pp. 64, 154*

[Rob19] RobMoSys. *RobMoSys Glossary*. H2020—ICT—732410 RobMoSys. 2019-05-20. URL: https://robmosys.eu/wiki/glossary (visited on 2020-08-04). *used on: p. 176*

[Roc+15] Alessio Rocchi et al. "OpenSoT: A whole-body control library for the compliant humanoid robot COMAN." In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 26.05.2015 - 30.05.2015, pp. 6248–6253. DOI: 10.1109/ICRA.2015.7140076. *used on: pp. 20, 21, 64, 99, 154*

[Rod15] Alberto Rodrigues da Silva. "Model-driven engineering: A survey supported by the unified conceptual model." In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155. ISSN: 1477-8424. DOI: 10.1016/j.cl.2015.06.001. *used on: p. 28*

[Rot+89] Jeff Rothenberg et al. "The Nature of Modeling." In: *in Artificial Intelligence, Simulation and Modeling*. John Wiley & Sons, 1989, pp. 75–92. *used on: pp. 28, 178*

[RS04] J.L. Rossello and J. Segura. "An analytical charge-based compact delay model for submicrometer CMOS inverters." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 51.7 (2004), pp. 1301–1311. DOI: 10.1109/TCSI.2004.830692. *used on: p. 67*

[SB18] S. S. M. Salehian and A. Billard. "A Dynamical-System-Based Approach for Controlling Robotic Manipulators During Noncontact/Contact Transitions." In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 2738–2745. DOI: 10.1109/LRA.2018.2833142. *used on: p. 92*

[Sch+09] C. Schlegel et al. “Robotic software systems: From code-driven to model-driven designs.” In: *2009 International Conference on Advanced Robotics*. 2009, pp. 1–8. *used on: pp. 22, 23, 28, 31, 32*

[Sch+10] Christian Schlegel et al. “Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering.” In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Noriaki Ando et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 324–335. *used on: pp. 4, 27, 28*

[Sch+15] Christian Schlegel et al. “Model-driven software systems engineering in robotics: Covering the complete life-cycle of a robot.” In: *it - Information Technology* 57.2 (2015), pp. 85–98. DOI: 10.1515/itit-2014-1069. *used on: p. 28*

[Sch+18] Casper Schou et al. “Skill-based instruction of collaborative robots in industrial settings.” In: *Robotics and Computer-Integrated Manufacturing* 53 (2018), pp. 72–80. ISSN: 0736-5845. DOI: 10.1016/j.rcim.2018.03.008. *used on: pp. 16, 90, 91*

[Sch+19] Marie Schumacher et al. “An introductory review of active compliant control.” In: *Robotics and Autonomous Systems* 119 (2019), pp. 185–200. DOI: 10.1016/j.robot.2019.06.009. *used on: p. 95*

[Sch06] D. C. Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering.” In: *Computer* 39.2 (2006), pp. 25–31. *used on: p. 28*

[Sci] Scilab. *Scilab*. URL: https://www.scilab.org/ (visited on 2020-12-01). *used on: p. 7*

[Sci+16] Enea Scioni et al. “Hierarchical Hypergraphs for Knowledge-centric Robot Systems: a Composable Structural Meta Model and its Domain Specific Language NPC4.” In: *Journal of Software Engineering for Robotics* 7 (2016), pp. 55–74. DOI: 10.6092/JOSER{\textunderscore}2016{\textunderscore}07{\textunderscore}01{\textunderscore}P55. *used on: p. 91*

[Sci16] Enea Scioni. “Online Coordination and Composition of Robotic Skills: Formal Models for Context-aware Task Scheduling.” Doctoral dissertation. 2016. *used on: p. 90*

[SG07] H. Sahin and L. Guvenc. “Household robotics: autonomous devices for vacuuming and lawn mowing [Applications of control].” In: *IEEE Control Systems Magazine* 27.2 (2007), pp. 20–96. DOI: 10.1109/MCS.2007.338262. *used on: p. 89*

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Pearson Education, 2002-01. ISBN: 0-201-745572-0. *used on: pp. 23, 24*

[SH20] A. Solis and J. Hurtado. “Reutilización de software en la robótica industrial: un mapeo sistemático.” In: *Revista Iberoamericana de Automática e Informática industrial* 17.4 (2020-09), p. 354. DOI: 10.4995/riai.2020.13335. *used on: p. 64*

[Sha+20] Mohit Sharma et al. “Learning to Compose Hierarchical Object-Centric Controllers for Robotic Manipulation.” In: *Proceedings of (CoRL) Conference on Robot Learning*. 2020-11. *used on: p. 97*

[Sic90] Bruno Siciliano. “Kinematic control of redundant robot manipulators: A tutorial.” In: *Journal of Intelligent and Robotic Systems* 3.3 (1990), pp. 201–212. DOI: 10.1007/bf00126069. *used on: p. 19*

[Sie14] Jon Siegel. "Object Management Group Model Driven Architecture (MDA) MDA Guide rev. 2.0." In: (2014). *used on: pp. 30, 31*

[SLL20] Jan Staschulat, Ingo Lütkebohle, and Ralph Lange. "The rclc Executor: Domain-specific deterministic scheduling mechanisms for ROS applications on microcontrollers: work-in-progress." In: *2020 International Conference on Embedded Software (EMSOFT)*. 2020, pp. 18–19. DOI: 10.1109/EMSOFT51651.2020.9244014. *used on: p. 157*

[SLS17] Christian Schlegel, Alex Lotz, and Dennis Stampfer. *D2.1 Modeling Foundation Guidelines and Meta-Meta-Model Structures*. Deliverable. RobMoSys: Composable Models and Software for Robotic Systems, 2017. URL: https://robmosys.eu/wp-content/uploads/2017/03/D2.1_Final.pdf. *used on: pp. 6, 90*

[Smi+08] Ruben Smits et al. "iTASC: a tool for multi-sensor integration in robot manipulation." In: *MFI 2008*. [Piscataway, N.J.]: [IEEE], 2008, pp. 426–433. DOI: 10.1109/MFI.2008.4648032. *used on: pp. 16, 17, 105*

[Soe06] Peter Soetens. "A Software Framework for Real-Time and Distributed Robot and Machine Control." http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf. Doctoral dissertation. Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, 2006-05. *used on: pp. 23, 24*

[SS91] B. Siciliano and J. J. E. Slotine. "A general framework for managing multiple tasks in highly redundant robotic systems." In: *Fifth International Conference on Advanced Robotics Robots in Unstructured Environments*. 1991, 1211–1216 vol.2. DOI: 10.1109/ICAR.1991.240390. *used on: pp. 19, 21*

[Sta+16] Dennis Stampfer et al. "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software." In: *Journal of Software Engineering for Robotics* 7.1 (2016), pp. 3–19. *used on: p. 9*

[Sto09] A. Stoytchev. "Some Basic Principles of Developmental Robotics." In: *IEEE Transactions on Autonomous Mental Development* 1.2 (2009), pp. 122–130. DOI: 10.1109/TAMD.2009.2029989. *used on: p. 89*

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2006. ISBN: 978-0-470-02570-3. *used on: p. 28*

[SW09] William Strunk and Elwyn Brooks White. *The Elements of Style*. New York: Longman, 2009. ISBN: 978-0-205-30902-3. *used on: p. 207*

[SWW] Jochen Steil, **Dennis Leroy Wigand**, and Sebastian Wrede. "D2.3 Domain Analysis Compliant interaction." In: (). *used on: p. 93*

[SWW18] F. Steinmetz, A. Wollschläger, and R. Weitschat. "RAZER—A HRI for Visual Task-Level Programming and Intuitive Skill Parameterization." In: *IEEE Robotics and Automation Letters* 3.3 (2018-07), pp. 1362–1369. DOI: 10.1109/LRA.2018.2798300. *used on: pp. 16, 90, 91*

[TAD07] Salvador Trujillo, Maider Azanza, and Oscar Diaz. "Generative Metaprogramming." In: *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 105–114. DOI: 10.1145/1289971.1289990. *used on: p. 178*

[Tal+04] Jean-Pierre Talpin et al. "Formal Methods and Models for System Design." In: ed. by Rajesh Gupta et al. Norwell, MA, USA: Kluwer Academic Publishers, 2004. Chap. Behavioral Type Inference: Part II - Behavioral Type Inference for System Design, pp. 245–282. ISBN: 978-1-4020-8051-7. *used on: p. 79*

[The15] The Economist. *Style Guide. The Bestselling Guide to English Usage*. 11th ed. New York: Public Affairs, 2015. ISBN: 978-1-61039-538-0. *used on: p. 207*

[Tho+13] U. Thomas et al. "A new skill based robot programming language using UML/P Statecharts." In: *2013 IEEE International Conference on Robotics and Automation*. 2013-05, pp. 461–466. DOI: 10.1109/ICRA.2013.6630615. *used on: pp. 5, 7, 16, 17, 90, 91, 104, 106*

[Van+13a] B. Vanderborght et al. "Variable impedance actuators: A review." In: *Robotics and Autonomous Systems* 61.12 (2013), pp. 1601–1614. ISSN: 0921-8890. DOI: 10.1016/j.robot.2013.06.009. *used on: p. 1*

[Van+13b] Dominick Vanthienen et al. "Rapid application development of constrained-based task modelling and execution using domain specific languages." In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Piscataway, NJ: IEEE, 2013, pp. 1860–1866. DOI: 10.1109/IROS.2013.6696602. *used on: pp. 5, 17, 91, 105, 106*

[Vd16] Luigi Villani and Joris de Schutter. "Force Control." In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Cham: Springer International Publishing, 2016, pp. 195–220. ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_9. *used on: pp. 18, 95*

[Vil04] Petr Vilím. "O(nlogn) Filtering Algorithms for Unary Resource Constraint." In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Ed. by Jean-Charles Régin and Michel Rueher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 335–347. *used on: p. 69*

[VKB14] Dominick Vanthienen, Markus Klotzbücher, and Herman Bruyninckx. "The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming." In: 2014. *used on: p. 24*

[vKV00] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-specific languages: an annotated bibliography." In: *ACM SIGPLAN Notices* 35.6 (2000), pp. 26–36. DOI: 10.1145/352029.352035. *used on: p. 28*

[Voe11] Markus Voelter. "Language and IDE Modularization and Composition with MPS." In: *Proc. 4th Int. Summer School Generative and Transformational Techniques in Software Engineering*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Braga, Portugal: Springer Berlin Heidelberg, 2011, pp. 383–430. ISBN: 978-3-642-35992-7. DOI: 10.1007/978-3-642-35992-7_11. *used on: pp. 32, 42, 43*

[Voe15] Markus Voelter. *Language-Oriented Business Applications: Helping End Users become Programmers*. SPLASH 2015 SPLASH-I. 2015-11-29. URL: http://voelter.de/data/presentations/voelter-splash-i-LOBA.pdf (visited on 2020-08-04). *used on: p. 30*

[Völ+13] Markus Völter et al. *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013, p. 558. ISBN: 978-1-4812-1858-0. *used on: pp. 9, 29, 31, 43, 45–47, 175–177*

[W3C15] The World Wide Web Consortium (W3C). *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. W3C Recommendation 1 September 2015 (https://www.w3.org/TR/scxml). 2015. *used on: pp. 50, 119*

[WDW20] **Dennis Leroy Wigand**, Niels Dehio, and Sebastian Wrede. “Model-Based Specification of Control Architectures for Compliant Interaction with the Environment.” In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 7241–7248. DOI: 10.1109/IROS45743.2020.9340718. *used on: pp. 5, 12, 89, 103, 125, 134, 141, 148*

[Whi+09] Jules White et al. “Improving domain-specific language reuse with software product line techniques.” In: *IEEE software* 26.4 (2009), pp. 47–53. *used on: p. 42*

[WHR14] J. Whittle, J. Hutchinson, and M. Rouncefield. “The State of Practice in Model-Driven Engineering.” In: *IEEE Software* 31.3 (2014), pp. 79–85. DOI: 10.1109/MS.2013.65. *used on: p. 28*

[Wie18] Johannes Wienke. “Framework-level resource awareness in robotics and intelligent systems. Improving dependability by exploiting knowledge about system resources.” In: (2018). DOI: 10.4119/UNIBI/2932136. *used on: p. 23*

[Wig+17a] **Dennis Leroy Wigand** et al. “Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case.” In: *Journal of Software Engineering for Robotics* 8.1 (2017), pp. 45–64. ISSN: 2035-3928. DOI: 10.6092/JOSER2017_08_01_p45. *used on: pp. 4, 12, 37, 48, 119, 134, 141, 145*

[Wig+17b] **Dennis Leroy Wigand** et al. “Modularization of Domain-Specific Languages for Extensible Component-Based Robotic Systems.” In: *2017 First IEEE International Conference on Robotic Computing (IRC)*. Ed. by IEEE International Conference on Robotic Computing. IEEE, 2017-04, pp. 164–171. DOI: 10.1109/IRC.2017.34. *used on: pp. 12, 37, 60, 119, 134*

[Wig+18] **Dennis Leroy Wigand** et al. “An Open-Source Architecture for Simulation, Execution and Analysis of Real-Time Robotics Systems.” In: *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. Ed. by Modeling IEEE International Conference on Simulation and Programming for Autonomous Robots. Piscataway, NJ: IEEE, 2018, pp. 93–100. DOI: 10.1109/SIMPAR.2018.8376277. *used on: pp. 12, 37, 48, 65*

[Wol+16] Sebastian Wolf et al. “Variable Stiffness Actuators: Review on Design and Components.” In: *IEEE/ASME Transactions on Mechatronics* 21.5 (2016), pp. 2418–2430. DOI: 10.1109/TMECH.2015.2501019. *used on: p. 2*

[WW11] Johannes Wienke and Sebastian Wrede. “A Middleware for Collaborative Research in Experimental Robotics.” In: *IEEE / SICE International Symposium on System Integration (SII 2011)*. IEEE, 2011, pp. 1183–1190. DOI: 10.1109/SII.2011.6147617. *used on: pp. 41, 48*

[WW19] **Dennis Leroy Wigand** and Sebastian Wrede. “Model-Driven Scheduling of Real-Time Tasks for Robotics Systems.” In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 2019, pp. 46–53. DOI: 10.1109/IRC.2019.00016. *used on: pp. 7, 12, 48, 65, 68, 73*

[XJ01] Jing Xiao and Xuerong Ji. “Automatic Generation of High-Level Contact State Space.” In: *The International Journal of Robotics Research* 20.7 (2001), pp. 584–606. DOI: 10.1177/02783640122067552. *used on: pp. 95, 96, 98*

[Yun+99] Yun-Hui Liu et al. “Model-based adaptive hybrid control for manipulators under multiple geometric constraints.” In: *IEEE Transactions on Control Systems Technology* 7.1 (1999-01), pp. 97–109. ISSN: 2374-0159. DOI: 10.1109/87.736761. *used on: p. 99*

[ZB20] Wu-Le Zhu and Anthony Beaucamp. “Compliant grinding and polishing: A review.” In: *International Journal of Machine Tools and Manufacture* 158 (2020), p. 103634. DOI: 10.1016/j.ijmachtools.2020.103634. *used on: p. 115*

[ZC06] Ji Zhang and Betty H. C. Cheng. “Model-Based Development of Dynamically Adaptive Software.” In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE ’06. Shanghai, China: Association for Computing Machinery, 2006, pp. 371–380. DOI: 10.1145/1134285.1134337. *used on: p. 28*

[ZH18] Zuyuan Zhu and Huosheng Hu. “Robot Learning from Demonstration in Robotic Assembly: A Survey.” In: *Robotics* 7.2 (2018). DOI: 10.3390/robotics7020017. *used on: p. 161*

[ZM94] R. Zurawski and MengChu Zhou. “Petri nets and industrial applications: A tutorial.” In: *IEEE Transactions on Industrial Electronics* 41.6 (1994), pp. 567–583. *used on: p. 25*

[ZP18] Xian Zhou and Quang-Cuong Pham. “Can robots assemble an IKEA chair?” In: *Sci. Robotics* 3.17 (2018). DOI: 10.1126/scirobotics.aat6385. *used on: p. 2*

SOFTWARE PACKAGES

[Dra] Drag&Bot. *Drag&Bot*. Drag&Bot. URL: https://www.dragandbot.com/product/ (visited on 2021-01-18). *used on: p. 16*

[Emib] Franka Emika. *Desk*. Franka Emika. URL: https://www.franka.de/capability (visited on 2021-01-18). *used on: pp. 5, 16*

[MPS] MPS. *Meta Programming System (MPS)*. JetBrains. URL: https://www.jetbrains.com/mps/ (visited on 2020-08-07). *used on: p. 43*

[OSR] Open Source Robotics Foundation (OSRF). *Robotic Operating System 2 (ROS2)*. Version ROS2 Foxy Fitzroy. OSRF. URL: https://index.ros.org/doc/ros2/ (visited on 2021-01-27). *used on: pp. 23, 157*

MEDIA

[Cog] CogIMon. *Cognitive Interaction in Motion – CogIMon*. Horizon 2020 ICT-23-2014 https://cogimon.eu/. URL: https://www.youtube.com/watch?v=8fDuNwFkaxg&t=59s (visited on 2021-03-16). *used on: p. 5*

Several icons used in this thesis are made by Freepik, Becris, and Smashicons from https://www.flaticon.com/.

DECLARATION OF AUTHORSHIP

According to the Bielefeld University's doctoral degree regulations §8(1)g: I hereby declare to acknowledge the current doctoral degree regulations of the Faculty of Technology at Bielefeld University. Furthermore, I certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. Third parties have neither directly nor indirectly received any monetary advantages in relation to mediation advises or activities regarding the content of this thesis. Also, no other person's work has been used without due acknowledgment. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged. This thesis or parts of it have neither been submitted for any other degree at this university nor elsewhere.

Dennis Leroy Wigand	Place, Date

COLOPHON

This thesis is typeset using the `classicthesis` typographical look-and-feel developed by André Miede and is based on the template created by Johannes Wienke. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*" [Bri08]. The writing style has been influenced by Strunk and White [SW09], Dupré [Dup07], and The Economist [The15].