Dissertation

Coordination and Composition Patterns in the "Curious Robot" Scenario

Ingo Lütkebohle

September 6th, 2011

Abdruck der genehmigten Dissertation zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.). Der Technischen Fakultät der Universität Bielefeld am 27. April 2010 vorgelegt. Verteidigt und genehmigt am 30. September 2010.

Gedruckt auf alterungsbeständigem Papier nach ISO 9706

Abstract

The advancement of robotics as a field towards new applications and domains has always been accompanied with and enabled by advances in robot software architecture. This is true for the first plan-based systems, introducing flexibility in action sequences, through behavior-based architectures, introducing speed, agility, and emergent behavior, to hybrid layered architectures with their combination of aspects of both. Most recently, a focus on the software engineering aspects and component-based architectures aims to increase engineering efficiency and maintainability, while also studying the architectural proposals in more empirical depth.

This thesis contributes to this development by advancing architectural principles needed for better Human-Robot-Interaction (HRI) systems, in particular, situated, social HRI. These systems are characterized by a tight integration of interaction and action, and by the need to rapidly iterate designs, particularly the interaction, during research and development.

Specifically, the well-known principle of describing action execution with finite-state machines has been developed into a general, abstract coordination interface, the Task-State Pattern. Firstly, the historical emergence of a commonality in otherwise different coordination systems has been identified and formalized as an architectural pattern. Secondly, a means of implementation has been proposed that enhances abstract observability, allowing diverse tasks to be observed in a general fashion. Thirdly, the benefits of implementing the pattern in a service-level toolkit have been demonstrated.

Moreover, the efficient design and engineering of components based on the data-flow principle has been studied. In particular, a decomposition method to increase the re-usability of the constituent nodes in such components has been proposed, based on ideas from event-based software integration. The suitability of the method for the robotics domain has been studied, to contribute to the design knowledge, and to suggest possible remedies for situations where data-flow alone is not adequate, such as maintaining global state.

All of these contributions have been integrated, and empirically evaluated, in a comparative fashion on multiple successive iterations of an HRI scenario, the "Curious Robot". Such comparative evaluation is, to the best of the authors knowledge, a first in this area. The identified pattern is applied here to provide tight integration between coordination components in the interaction and manipulation sub-system without large changes to each. As a result, the scenario realizes detailed mixed-initiative, multimodal interaction in an object-learning setting.

Acknowledgements

The following work would have been impossible in a very real sense without Julia Peltason and Robert Haschke, my two primary collaborators in the ERBI project. I thank them deeply for going with that odd Curious Robot idea and making it much more than I ever could have done alone.

Thanks to my advisor, Sven Wachsmuth, for sticking with me through many changes of mind and managing the unmanageable. Thanks to Sebastian Wrede, for showing me that it can be done, and in style. Thanks to Franz Kummert, for telling me what I must strive for, and for keeping the bottom line solid enough to do so. Thanks to my external reviewer, Reid Simmons, for accepting to review the work of a stranger.

All this time, I've had the pleasure of being part of many amazing groups of people. First of all, the V9 "subgroup": Christian Thurau, Dirk Stößel, Marc Hanheide, Volker Wendt and Oscar Carrascero, I owe thanks for many lunches and it ended way too soon! Thanks to everyone on the DESIRE team, particularly Lars Schillingmann, Christof Elbrechter, Risto Koiva and Florian Schmidt of the ERBI team for sharing the curiosity (and the CeBit crunch!). Thanks to Uli Reiser, Christian Faubel, Thilo Grundmann, Christopher Parlitz and all the others from the TP team for sticking with it through the end. Thanks to everyone on Q2 for having me despite deserting them, just as many thanks to the CLF crowd for accepting me as one of their own. Christian Peters, thanks for many discussions and Florian, thanks (not only!) for all the video. Not least, many thanks to the ToBI team, for having me in Graz.

Outside of the immediate circle, I would like to give a shout to Roland Philippsen for introducing me to the conference circle and approving of my bad puns, to Davide Brugali for supporting us in the TC, and to Yukie Nagai for showing us her country.

To my friends, I cannot emphasize enough how much being anchored through you means to me. Andi und Tommi, ich verlass mich drauf, dass ihr mich auch mit Dr. noch genauso wenig ernst nehmt. Simon, Holger, Agnes, Till, Mirco, Andi D. and Daniel, many thanks for climbing with me. Sarah, Antonia, Claudius and Simone, I thank for a different perspective and sharing spaces with me. Jim, Bill and Meghan and the others in Vancouver, Victoria and the isles, thanks for a great time.

Last, and most, many thanks to my family, for supporting me without question and to Julia, for many common journeys, on the road and in life.

Contents

List of Tables

List of Figures

Part I.

Motivation and Scenario

1. Robot-Software for Mixed-Initiative Human-Robot-Interaction

Robotics as a field is characterized by a progression from fixed, pre-taught automation tasks towards flexible activity in open environments. By supporting integration and complexity management, robot software architecture has been an essential enabler on this path, first integrating complex sensors in the sense-plan-act paradigm, then ideas from biology and psychology, towards reactive, situated architectures.

Along that path, robot software systems are growing not only in capability, but also in complexity, and size. The first factor is that, usually, more capabilities also means more software realizing them, plus still more for choosing and combining them. The second contributor is that many of the components have become individually more complex, e.g. to cope with unstructured, dynamic environments. The most developed sub-systems, such as navigation, now routinely rival or surpass that which was previously only seen in whole systems.

Despite these advances, robots are still quite a ways from being functional in the real-world, both due to old challenges not being entirely solved and due to new ones becoming apparent. In the search for solutions, it is not unusual that researchers have to explore solutions that change existing assumptions, both on the functional and on the architectural level.

An example of changing architectural requirements that will be a guiding concern in this thesis currently occurs in Human-Robot-Interaction (HRI). While traditionally HRI has been command-oriented, social and affective methods are now being added, to improve interaction success and efficiency. Functionally, these require many new and experimental components, e.g. for affect perception, dialog and learning. Architecturally, they require a much tighter and *bi-directional coordination interface*, to enable detailed feedback about and interaction with the actions of the robot.

Moreover, the growing scope and enhanced capabilities of robots also affect their development. In particular, capable robots are now usually created by teams, and this requires a guiding architecture that supports developers, both during development and integration. Team-work also increases co-development, where multiple interdependent components of a system are experiencing rapid change at the same time. Architecturally, such situations place a strong emphasis on *interfaces* and *composability* of functional blocks.

Therefore, this thesis will go beyond the functional issues and also examine solutions with regard to enabling efficient team-work. Towards this, the solutions are intended to realize architectural qualities, such as low coupling and encapsulation, which are known to positively influence overall complexity. Moreover, they will be evaluated in a team context, over multiple iterations of a significant system.

Besides simple pragmatism, this approach is based on a hypothesis about robotics research in general: That a good combination of approaches could often simplify the individual solutions, but such a path is rarely chosen because of overly high integration effort. As such effort detracts from what is perceived as the primary research goal, people are more likely to stay on their methodological islands. While integration is certainly not the only issue, if robotics is to advance as a systems science, it needs to solve this problem, and this thesis examines how to do so.

1.1. Context: Social Robots that Learn in Interaction

The majority of the work to be presented has been carried out in the context of a project to realize a social robot that learns from interaction with people. The scenario is called the "Curious Robot" (Lütkebohle et al., 2009a), and its goal is to explore interactive teaching in a setting where both partners are active in the process, loosely inspired by children learning. Figure 1.1 shows two exemplary interaction steps from it: One where the robot asks for an object on its own initiative (1.1(a)), the other where it puts an object away (1.1(b)).

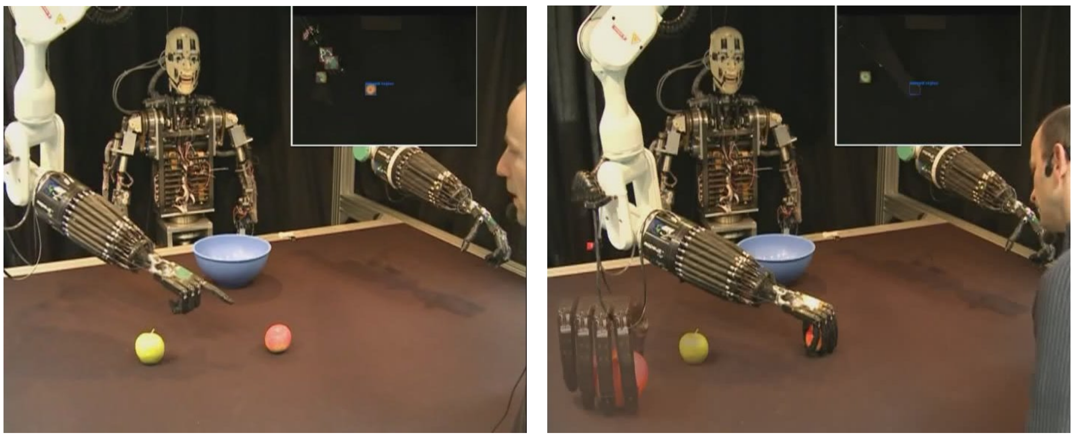

(a) Robot asks for the red apple's label. (b) Robot grasps the red apple.

Figure 1.1.: Example Interaction Steps

Architecturally, the scenario is interesting for two reasons: Firstly, it uses speech as a concurrent control modality, with a tight, yet flexible, level of integration that goes beyond the typical command interaction. In particular, interaction is possible at any stage of robot activity, and both robot and human can take the initiative for interaction. To facilitate this, the system provides detailed feedback about what it is doing, enabling the user to modify these activities at any time. Apart from controlling activities, verbal interaction is also used to query the robot's acquired knowledge. Several social structuring mechanisms, both verbal and non-verbal, are provided to assist this process.

Secondly, the system comprises a full set of perception and action components, developed by a diverse team, and working across two different (but integrated) physical platforms: An anthropomorphic robot for interaction, and a hand-arm system for powerful manipulation. Together, these attributes ensure realistic architectural challenges, and that a representative set of issues is covered. A summary of the resulting challenges will now be given, to outline the scope of the work undertaken.

1.1.1. Enabling detailed, yet general, verbal commentary

A seemingly trivial, but often neglected, ability is to provide detailed verbal commentary both on what the robot is doing ("reporting"), and what it understood from the human ("feedback"). Even with highly trained personnel, such as astronauts, the lack of verbal responses has been found to impair task success (Fong et al., 2006a). The ability of the dialog system to provide commentary based on a *generic* interface to other components has been a crucial architectural starting point, providing a foundation for what came after.

In practice, there are two fundamentally different ways to achieve such commentary: One way is to place *output generation calls* into the components performing the actions. While this is simple and straightforward, it is essentially similar to placing a debugging "printf" statement into code – useful in the short term, but not a general solution. It can only report what the author of the component thought of, in the terms that the author considered, and targeted at the specific output. Furthermore, it is an add-on that must be maintained in addition to the core functionality, and thus is at risk of falling out of sync with that functionality. While there is not much information available in the literature, anecdotal evidence suggests that this is currently the most widespread method for verbal reporting in systems which are not explicitly about speech dialog.

In contrast, *commentary from observation* generates messages by observing the state-changes an action component undergoes in the course of carrying out its action. This also requires that the component maintains such state in an observable manner, but as this form of state-keeping directly supports carrying out the action, instead of being an add-on, it is not at as much risk of falling out of sync. Furthermore, it is often already being provided in robotics systems, as part of the coordination or planning and execution system. More generally, in keeping with the "printf"-analogy, commentary from observation could be likened (architecturally) to the function of a debugger, support for which is now standard in all operating systems.

Thus, the requirement for verbal commentary are a subset of those for coordination: It needs information about which actions are carried out, and what state they are in. For coordination, the interested party is the coordination manager, for reporting, it is the interaction manager component.

In both cases a general, action-independent protocol to convey this information is desirable to prevent a static, inflexible coupling. Again similar to coordination, reporting also needs detailed state information, for example to differentiate an action being requested from one actually being undertaken (due to situational constraints, this is not generally the same).

That said, the typical structure of coordination did require some adaptation to achieve the non-functional qualities mentioned earlier. Most importantly, traditionally the mapping from action-specific to action-independent states is often performed in the coordination component. This does not provide optimal re-use (it would have to be done again in other interested "listener" components). Therefore, it is suggested to place this mapping in the task server, so that all messages sent out by the server already have both task-specific and abstract state information available for observation throughout the system.

In spite of the general goal of abstraction, specific information may be required for better feedback. For example, during grasping, both the motion of the arm to the object, and the process of grasping itself, is of interest – this corresponds to the visibility of all tasks, not just global goals. Also, while the abstract level can report starting and stopping of tasks, it is certainly more helpful to provide information on what is starting. This requires action-specific details.

Therefore, this thesis will present a means of communicating such information that i) provides an abstract level for high re-use and decoupling, ii) is associated with more detailed descriptions for inspection by specialized extensions, and iii) is observable by many parts of the system.

Furthermore, the goal is to do this in a way which is not specific to the system described here, but generalizes to most systems. This can only be achieved by taking proven approaches as a foundation, and thus, a pattern-based approach has been adopted, that generalizes and extends existing work in this area. General models of task life-cycle, leading to reusable frameworks for coordinating execution have been explored by several authors, for example Lefebvre and Saridis (1992); Simmons (1994); Simmons and Apfelbaum (1998); Kortenkamp et al. (1998); Wrede et al. (2006). The commonalities, and differences, as well as implementational caveats of these have not so far been systematically explored, however. One contribution of this thesis is to generalize them as an architectural design pattern, called "Abstract Task-State-Pattern". It is described fully in chapter 4.

1.1.2. Continuous, bi-directional interaction-action coupling

While feedback is important in many applications, for the current system, it has been only a prerequisite for a more important function: Enabling the human and the robot to interact about the actions being carried out while they occur. Knowing when something is going on, and what, is important for that, but the system is also intended to enable modification, or cancellation, of what is going on.

Such requirements are common to many robotics application, they are the traditional domain of coordination. What is different in the "Curious Robot" system is the desire to provide this through natural, situated interaction. This joins two individually complex sub-systems (motion-control and interaction) at the hip, with tight integration. The goal is to allow interaction while the action is still being executed. For example, during grasping, the pose might be slightly off, which is fairly easy to correct during the action ("more to the left"), because the action context is implicit. It could also be that the robot tries to grasp the wrong object, and the action needs to be aborted. This is obviously required to occur while the action is being performed, otherwise it would be useless. Conversely, the motion sub-system needs to be able to report failure of a task, or failure to update the goal.

On the one side, this requires that the context be addressable, and that the action tracking can capture the necessary concepts. More importantly, however, as mentioned above, two individually complex sub-systems are involved here, and integrating these while keeping overall complexity low is the primary challenge. This has initially been approached by treating one of them as the service, and the other as the coordinator, and re-using existing coordination interfaces. While such an approach resulted in a good decoupling, it suffered from limited integration. Over time, it became clear that an extension of the coordination protocol is necessary to achieve better integration, and that the resulting system actually has two executives, which coordinate amongst each other.

The resulting system structure has been studied on several representative cases, with an eye towards i) ensuring that the necessary protocol enhancements are restricted to well-defined extension points, and ii) that the overall system retains low coupling. Chapter 5 describes the initial, and chapter 6 the final experiments.

1.2. Architectures for iterative development

In the requirements discussed so far, the driving force has been better functionality. However, particularly but not only in research situations, the development process presents its own challenges. The team-based nature of robotics research has already been mentioned in the beginning, but another crucial issue is to enable fast *iterations*, which, here, will be primarily approached by making it easier to assemble and change components.

On the one hand, building up a system through an iterative development process is simply good engineering practice. On the other hand, however, it is also necessary for research into system architectures: It enables *empirical* comparison, through before-after analysis, where a suggested change is compared to a previous iteration of the same component and/or system.

As systems are growing in size, the problem of *composition*, i.e. how to assemble a functional system from components, or a functional components from parts, consumes are larger share of the work. This is the traditional area of architecture, with many diverse proposals. This thesis broadly follows the approach of hybrid/layered architectures (cf. Kortenkamp and Simmons (2008)), and in these, particularly the model of recursively layered structures of increasing abstraction, e.g. as proposed by Albus (1992). The resulting architectural schema will be presented in chapter 3. Within this schema, the primary focus of this thesis is on the *interconnections* between and within components in an architecture, an aspect that can be hard to generalize across implementations, but that for exactly this reason is also particularly important, and, in the present author's opinion, currently insufficiently studied.

The first of these interconnections, that between components, strongly relates to the communications interface, which constitutes the main contribution for *between*-component composition. As it has already been discussed in the previous section, due to its direct relation to HRI, this section will only discuss the second contribution, towards *within* component composition.

1.2.1. Within-component composition

Compositing components from existing building blocks is not a novel issue, and various approaches of dealing with it have been proposed. Architecturally, the "pipes-and-filters" (Meunier, 1995), and the more general dataflow (Lee and Messerschmitt, 1987; Lee and Parks, 1995) styles are well-known. In languages, visual languages (Johnston et al., 2004) are popular for "end-user programming" and high-level prototyping, whereas scripting languages such as Python[1] and Ruby[2] are commonly used to combine existing libraries. Not least, behavior definition languages popular with layered architectures (Firby, 1994; Gat, 1998; Bonasso, 1991) serve a similar purpose.

All that said, it must be noted that the direct approach for building components, through manual assembly, is still widely used, possibly even dominant. Therefore, the question addressed in this thesis is not whether it is possible to use one of these approaches, as it almost certainly is, but what real benefits it offers, as measured through empirical studies.

The basic approach taken is to model components as a graph, with nodes as the processing blocks, and edges transporting data. Conceptually, this is most similar to the data-flow model, particularly as realized through process networks. The explicit graph model supports the desired re-use and flexibility goals in the most direct manner, because the selection and connectivity of the nodes is externally specified. Moreover, the data-flow model is agnostic with respect to the implementation language – it represents a pure composition model that can be combined with different languages, a necessity for any research system. Last, but not least, graphs are more expressive than the simple pipeline architectures popular in some domains, and which are insufficient for many algorithms, such as those using iterative methods.

Taking this model as the basis, the question about the benefits becomes one about the suitability of data-flow for typical robotics' components. While the approach can support a variety of tasks, two issues that have been prominent in the work undertaken at Bielefeld University are i) scaffolding code and ii) platform independence. They were thus chosen for case studies of the proposed approach. The resulting work will now be introduced, with full details found in chapter 8.

[1]`http://www.python.org`, also see the Python bindings for ROS ros (2011)
[2]`http://www.ruby.org/`

Moving scaffolding to composition descriptors

Scaffolding code (also known as “glue code” or “boilerplate code”) is common on the borders of components. Particularly when components require information from various sources to perform their function, the code necessary for communication, type conversion, and so on, can make up a significant portion of the components codebase. This is despite the fact that it does not directly contribute to the function of the component.

We have examined this on an information fusion component, with many inputs, various data type conversions, and common transformations. Such components are commonly found in robotics systems, both on their own and as part of other components. Due to their many inputs and simple structure, they constitute a good baseline case. The particular items of interest have been how easy to compose, and change, the component is, and how much re-use of nodes could be realized.

Two difficulties made this nontrivial: Firstly, glue code is often configuration-rich, e.g. to specify endpoint names or the exact types for conversion. This should be handled in the model in a manner that keeps this information easy to change. Secondly, there is usually a reason such code is not simply encapsulated in a library call, be it that it is too specific, or that a minimal API is desired. Therefore, it became clear that intermediate level of code granularity is desirable – above function calls, but below the component levels.

Towards this, it is proposed to use principles known from event-based architectures to decompose functionality into easily recombinable nodes (cf. section 7.4). The level of re-use achieved is determined through case studies on conversion of components to the toolkit (see section 8.1), and by studying the adaptability, and re-usability of the resulting graph specifications (section 8.1.4). Furthermore, conclusions can be drawn on the necessary utility support to make said graphs efficiently creatable (cf. section 8.2.4).

Making platform specific code easily replaceable

The scenarios described in this thesis use several different robots, whose control interface, while similar in many respects, also contained important differences. For example, while all robots accept commands to set a desired joint position, they differ in whether later commands silently overwrite earlier ones or not, which directly influences the external interfaces. The adaptation necessary to achieve a consistent interface has in some ways been the most complex part of the low-level driver implementation.

Ironically, the platform issue is becoming more prominent as hardware platforms become more standardized. Widely available platforms usually come with software frameworks, such as NAOqi (NAO, 2011) for the Aldebaran Nao, ROS (ros, 2011) for Willow Garage’s PR2 (and many others), Yarp (Fitzpatrick et al., 2011) (for the iCub), and others. These frameworks are incompatible in many ways, such as the middleware used or, again, their control semantics. Given the different scope of the platforms, complete compatibility is unlikely to be achieved anytime soon, but there are enough similarities that some re-use of functionality appears possible. To achieve such reuse, it would be useful if i) the framework-specific parts constituted a replaceable part of the component, and ii) replacing them would be as easy as using a different configuration file.

One would expect that a graph-oriented composition approach could satisfy both of these requirements and, for the platforms used in this thesis, that is indeed the case, as explained in section 8.2. More interesting, however, is how well it does so. Two experimental studies have been conducted, to examine the complexity of the graphs created, and how easy it is to make changes to them.

1.3. Methodology: Case studies on successive iterations

The work reported in the following combines system construction with analysis of empirical case studies. Construction is a necessary first step, but empirically based analysis is needed to derive generalizable results.

Unfortunately, comparison across systems is not currently the norm in architecture research, for two primary reasons: Firstly, it is often prohibitively expensive, with current technology, to build a system of any significant complexity using completely different approaches. Secondly, systems built independently often differ in very many aspects, which makes comparison hard, if not impossible. This situation is in stark contrast to many other research areas, where comparative analysis is the norm.

This thesis addresses this problem in two complementary ways: Firstly, it will present comparative analyses on *successive iterations* of a single system (cf. chapter 9). There are both studies that vary individual components only (described in chapters 5 and 8), as well as studies that add or replace components (described in chapters 5 and 6). By looking at defined iterations of a single system, comparison at reasonable effort is made possible, and the resulting architectural qualities become visible.

Secondly, the subject matter of these studies, as outlined above, are methods that aim to reduce construction and integration effort. If proven successful, these methods are expected to lead to opportunities for comparing substantially different systems.

Moreover, these studies are intended both as a first step towards more overall comparison and to report on the construction of the present system in a way that others can compare against. The chosen method for reporting, case studies, provides insight not just into the *outcome*, but also into the *process* that lead to it, and perhaps most importantly, the *context* in which it was achieved. This is important because the process, including human dynamics, may have as much to do with the outcome as the methods used and a purely analytic approach is considered to be insufficient to capture such effects (Runeson and Höst, 2009).

This is not to say, however, that process will be the primary item examined. In contrast, the created artifacts will be described using a number of measures and models. Regarding metrics, several established software metrics have been applied, a summary of which is given in appendix A.1.1. The models used are primarily UML and state-machine models, slightly adapted for distributed systems, as outlined in the respective chapters.

In the words of Fenton, the goal is to

> ... gradually build up an empirical body of knowledge, simply by providing relevant quantitative information about real projects that we are involved with (Fenton, 2001, p. 196).

1.4. Reusable Software Toolkits

The software developed during this thesis is freely available on the author's homepage[3]. Most importantly, this includes the coordination and dataflow toolkits. The versions used during the present thesis are written in the Java(TM) (Gosling et al., 2005) language and run on any platform that supports Java. Besides the present work, they have been used in several other local projects and are considered stable and reasonably mature. Work to provide these toolkits in C++ is ongoing – experimental versions have been built, but are nowhere near complete.

The system described in the following also uses many other software packages – some of which the present author had a hand in, but also many contributions by others – which are generally not publicly available. An important exception is the XCF middleware toolkit (Wrede, 2008)[4].

In particular, both C++ (for Linux) and Java implementations of the "Active Memory" event-bus, which is required by the coordination toolkit, are available from that page. Both are implemented as a network server and can be used regardless of client language. The C++ implementation has more features, particularly regarding persistence, but the Java version has significantly fewer dependencies and is recommended for trial use.

1.5. Summary

In sum, this thesis will, firstly, present the *Abstract Task-State pattern*, a general coordination pattern that facilitates both detailed reporting of action during interaction, and a bi-directional interface to change action through verbal commentary. Secondly, it will present a decomposition strategy for component decomposition based on graphs, and a toolkit in support of the graph-based composition approach for component construction.

These contributions are thoroughly validated on several specific case studies, and proven throughout the construction of three successive iterations of the "Curious Robot" system. For the Task-State-Pattern four case studies of components in the "Curious Robot system" at various levels of integration are presented. An overall low

[3]`http://ingo.fargonauten.de/phd-soft/`
[4]`https://code.ai.techfak.uni-bielefeld.de/trac/xcf`

level of coupling, and tight integration with a generic interface will be demonstrated. For the graph-based composition approach, improved re-use and flexibility will be shown on two case studies, information fusion and generalized serial control.

2. The "Curious Robot"

Architectures are a means to an end, and as outlined in the introduction, the aim of the architecture developed in this thesis has been to support a Human-Robot-Interaction (HRI) scenario, the so-called "Curious Robot". This scenario, and the questions and developments from HRI it is intended to address, have been a central driving force for architecture design and research. The relevant questions and approaches will now be introduced, to provide a foundation for later chapters.

2.1. "Intuitive" Human-Robot-Interaction?

Human-Robot-Interaction (HRI) is a broad area that includes any means by which a human and a robot can directly interact to accomplish a task. This encompasses screens-and-buttons interfaces placed on a robot, physical HRI where humans guide a robot through physical contact (Billard et al., 2008), and social HRI (Breazeal et al., 2008) which focuses on using social cues with natural modalities such as speech and gesture. A borderline area is tele-operation, where the robot is remote controlled.

In the present work, a system will be considered where interaction occurs through "natural" modalities, particularly speech and gesture. The goal of such an approach is an intuitive interaction, that exploits humans' existing competence in using these modalities.

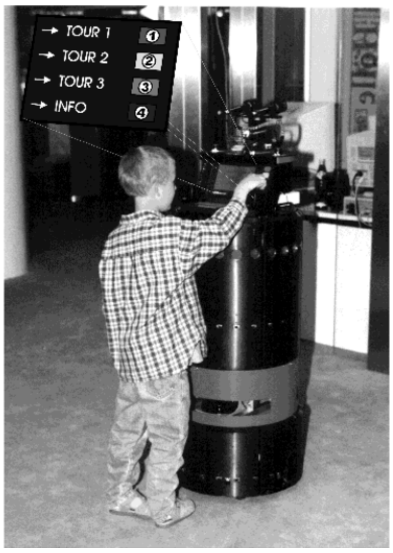


Figure 2.1.: User interface of Rhino, the mobile tour guide robot (from Burgard et al., 2000).

However, just using such modalities does not guarantee an intuitive interaction. An illustrative example is that of the two mobile robots "Shakey" (Nilsson, 1984) and "Rhino" (Burgard et al., 2000). Shakey uses a natural language interface which accepts command sentences typed in on a keyboard. Rhino, in contrast, uses a simple screen-and-button interface (shown in figure 2.1).

Of these two, Rhino is immediately usable by a naive user because the available choices are shown on the screen. For Shakey, however, control requires knowledge of the command language, which suffers from the well-known "vocabulary problem" (Furnas et al., 1987): The language is far less rich than human language and no untrained user knows which sentences are understood and which are not. Thus, Shakey cannot be used immediately.

In its defense, it must be said that Shakey is by far the older robot and, furthermore, was not intended for direct interaction. However, it demonstrates that what some might view as a more "natural" or "advanced" interface can actually be more difficult to use intuitively.

Having said this, language obviously can be much more powerful than four buttons for tour selection and has a strong appeal for HRI. The point to be made here is simply that "just using language" on its own does not yet make a good interface. Rhino's example, for instance, emphasizes that communicating a few initial activity options is necessary for intuitive use.

In the remainder of this chapter, the scenario in which this thesis has been undertaken will be motivated, related to the state of the art and then described in detail. In particular, the human-robot-interaction and the hardware capabilities will be described.

2.2. Scenario Rationale

The application area of the scenario is domestic service robotics, in which a robot is expected to perform services in human homes. In this area, it concentrates on the manipulation of objects, that is, picking them up, stowing them away, etc. Such capabilities form the basis for many other, more sophisticated tasks, in which they must be sequenced according to the particular task at hand.

It is expected that a robot will come with basic knowledge of objects and manipulation but that such knowledge is limited. Typical things that are hard to prespecify include the particular names for objects used by the robot's owner, the sheer diversity of objects, particulars of manipulation which have to be obeyed due to local constraints, the exact order of sequences to carry out, and so on.

Therefore, robots must be able to acquire the missing knowledge during operation and this has been a relatively active field of research. In it, one strain emphasizes autonomous exploration by the robot, whereas the other emphasizes human tutoring. Autonomous exploration can yield knowledge of what an object looks like and how it moves (for example Metta and Fitzpatrick (2003)). It can also, for manipulators with limited degrees of freedom, yield possible grasp poses fairly quickly (e.g. Zhang and Rössler (2003)).

In other areas, the purely autonomous approach is too limited, however. On the one hand, this includes complex search spaces, such as those posed by a manipulator close to the human hand with its many degrees of freedom. Search in such spaces can be sped up drastically by emulating demonstration actions (Billard et al., 2008; Steffen et al., 2009). On the other hand, it includes everything which is specific to a particular person or set of persons, such as object labels. This is knowledge which must necessarily be either synchronized with a human or learned from scratch and such learning is the focus of the present scenario.

2.2.1. Learning from Humans

Of the many aspects that could be acquired from human input, for learning of object manipulation, two areas are of primary interest: Learning of robot motion and learning of object attributes (such as labels).

Learning a motion in itself poses many challenges, which have been called a “Pandora’s box of important computational questions in perceptual motor control” (Schaal, 1999, p. 239). Motion learning is divided into the acquisition of so-called “motion primitives”, and learning sequences of such primitives that achieve complex actions. Motion primitives may be basic actions (such as reaching) or compound actions such as “grasping a cup” or “walking”. The compound representation is less flexible but results in a more compact state-space. The chosen abstraction level also has implications for interaction and this work will use abstract primitives.

For the present scenario, it is not so much the learning of motion primitives itself – they are assumed to be already known – which is interesting, but their configuration. Motion primitives are generic and have to be “adjusted for a specific goal” (Schaal, 1999, p. 238), e.g. to specify the target of a grasp. In a given interaction situation, this specification may be either determined automatically from the scene or specified by the human partner. Thus, the interaction and the motion sub-system must be able to communicate about these specifications.

Furthermore, the sequence of actions executed throughout the interaction could be an excellent basis for the learning of activities. While the present scenario has concentrated more on the interplay between interaction and a single action, learning about activities would be a natural extension and most likely possible using the proposed interaction paradigm.

Learning of object attributes

The label is the most obvious candidate, of the possible object attributes, for learning from humans, because it is not intrinsic to the object and cannot be acquired in any other way. However, other attributes, such as softness, may also be very worthwhile to learn from humans, to prevent unsavory accidents, or just to select the most appropriate and efficient means of manipulation.

Furthermore, human feedback can also make a hard task easier. For example, the method shown in figure 2.2 eventually manages to grasp the egg autonomously without breaking it (and generalizable, too), but it is actually very difficult to *distinguish correct from incorrect* grasps in a general way. A flexible ball may exhibit similar compression characteristics without breaking, so just detecting compressibility is not enough. Alternatively, breaking the egg might actually be the goal, e.g. when cooking.

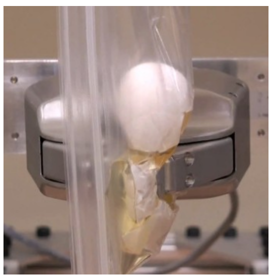


Figure 2.2.: Egg crushing (Piccoli, 2009).

In many such cases, human feedback can provide the necessary answer. It also bears noting, however, that for such interaction to be endurable by the human partner, the algorithm must not take too long to learn.

2.2.2. Differences in interaction style

An often neglected aspect of such learning scenarios is the interaction style. In most cases, either the designers of the system use it, or users are instructed prior to interaction. Not accidentally, such scenarios are also often known as "tutoring scenarios", with the implicit assumption that the human is teaching (e.g. compare Breazeal et al. (2004); Kruijff et al. (2006); Rohlfing et al. (2006) and the survey by Goodrich and Schultz (2007)).

As a result, many programming-by-demonstration scenarios ignore how to tell the robot to start learning and also ignore how the "tutor" knows what to do. HRI studies which focus on such aspects have shown, however, that the start of an interaction can be problematic (Hanheide and Sagerer, 2008), and they have further highlighted the importance of the recipient's reactions (Lohan et al., 2009; Vollmer et al., 2009). In essence, this comes back to the vocabulary problem mentioned initially, just extended: Users do not only lack knowledge about the vocabulary of a robot, they also lack knowledge about what the robot does and does not know. When humans encounter similar problems, the recipient is instrumental in communicating its understanding to ensure smooth interaction (Pitsch et al., 2009).

Put another way, the central issue is that there are two areas of knowledge: One is the target area, e.g. object labels, motions, and so on. In this, the human is the expert. The other, however, is the interaction style and here, the robot is in some sense more "knowledgeable" than the human or at least, it is more restricted and thus constrains the interaction and the goals.

Mixed-Initiative Interaction

An approach that explicitly acknowledges the flexible change of roles in interaction is represented by so-called mixed-initiative interaction (MII):

> Mixed-initiative interaction refers to a flexible interaction strategy in which each agent (human or computer) contributes what is best suited at the most appropriate time (Marti A. Hearst, introducing Allen et al. (1999)).

An example of MII in robotics is "traded control" (Kortenkamp et al., 1997), where robots are partially autonomous but may require human assistance. The principle has also been applied to a mixture of robot teams and humans, where robots can both interact amongst themselves and with humans (Murphy et al., 2000).

A crucial aspect in mixed initiative is that, firstly, participants must have some degree of autonomy and, secondly, there must be a criterion for deciding when to take or relinquish initiative. In the examples mentioned above, these are either failure situations or complementary activities (e.g., where a particularly expensive sensor is only present in few robots).

While mixed-initiative introduces flexibility into the interaction, the applications so far still assume the human user to be an expert who knows how to react. This makes sense in their target applications, but in a learning situation, many other aspects often need to be dealt with, such as the focus of attention, and additionally, the initiative can change much more often than in the aforementioned situations. Therefore, additional guidance is necessary.

Social HRI

In contrast to the above, social HRI focuses explicitly, and sometimes exclusively, on the social aspects of an interaction. The characteristics desired for social interactions are "efficient, enjoyable, natural and meaningful" (Breazeal et al., 2008). Note here the converse to the previous situation: That the interaction is *effective* is tacitly assumed.

In fact, much research on social HRI focusses not on the task but on achieving prerequisites for successful interaction, which may then lead to successful task achievement. Apart from the functional side, much of this research is also motivated by an interest in "social intelligence" as such, e.g. because of studies that suggest that social intelligence, including general emotion and empathy, is a precursor and/or base for rational reasoning in general (cf. Dautenhahn, 1995; Damasio, 1994).

For the present work, however, one particular aspect of the research in social robotics is most interesting: The wide-spread use of paralinguistic (nonverbal) cues inspired by human behavior. Such cues can be used by both the robot and the human to improve the flow of interaction. Breazeal et al. provide a taxonomy of such paralinguistic cues as either (Breazeal et al., 2008, p. 1352; my emphasis):

1. **Regulators:** expressions such as gestures, poses and vocalizations that are used to regulate/control conversational *turn-taking*.

2. **State displays:** signs of internal state including affect, cognitive, or conversational states that improve *interface transparency*.

3. **Illustrators:** gestures that *supplement information* for the utterance. These include pointing gestures and iconic gestures.

All of these are potential candidates for reducing confusion in naive users but it is surmised that state displays and illustrators will be most relevant for communicating what the the task is about, whereas regulators are more important to manage the overall interaction and possibly detect errors.

Autonomy

Bekey (2005) defines autonomous systems as "systems capable of operating in the real-world environment without any form of external control for extended periods of time.” Interestingly, while he also discusses robot learning at length (Bekey, 2005, chapter 6), it is primarily seen as a *prerequisite* for autonomy, furthering the goal for robot to be “able to operate in a world of humans” (Bekey, 2005, p. 471).

This work certainly shares the goal of enabling robots to live amongst humans and also that learning is required to achieve that goal. However, more than just a prerequisite, learning and autonomy are viewed to be *mutually supporting*. In addition to learning *for* autonomy, the following two aspects are also considered important.

Autonomy in learning. A robot needs to be autonomous in asking for information that can then be used for improved autonomy. This includes both taking the initiative, as well as deciding when such information would be advantageous. Most importantly, it relieves the user of the burden to have to know what to teach.

Learning about autonomy. Just as important, a robot must be able to learn about restrictions on its actions, so that it does not repeatedly initiate actions that are undesirable. This also includes the ability to react to input from the human when not expected (because, hopefully, the robot would not have undertaken an undesirable action when it *expected* it to be countermanded).

On an abstract level, this view is similar to the requirement that “inner directing mechanisms can be reflected upon and/or selectively modified”, specified by Boden (2008, p. 306). To that, the present work would like to add that such reflection and/or modification can also occur external to the system (i.e., by a human) and that therefore, it is beneficial to be able to treat human input in a similar way.

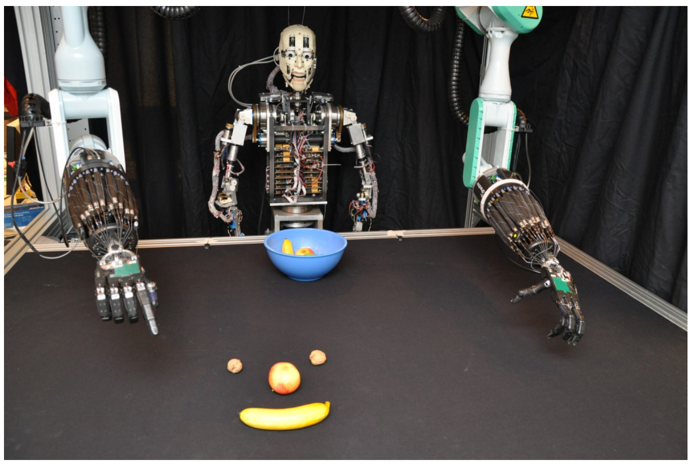
2.3. The “Curious Robot” Scenario

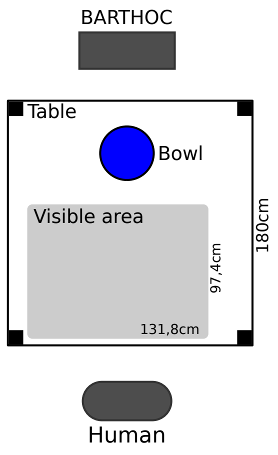
The central interaction idea explored in this thesis is that the robot structures the interaction. On the surface, this takes the form of asking questions, and commenting on actions while they are executed. Thereby, the robot provides information on what it knows and requires, and also implicitly, when interaction is possible at times that the human might not initially except. Regarding the task at hand, the robot remains in a learning position, but regarding the interaction, it assumes an initially driving role, with *interaction autonomy*. After the human has learned about the interaction style, she can assume more initiative.

The purpose of such learning about the dialog is that it integrates everything into one, complete interaction – starting from the point where a human walks up to the system, through the learning interaction, to the point where the results of learning are verified. The ultimate goal of this research are systems that can be used by naive users without prior instruction, because the scenario provides integrated guidance. Naturally, achieving this goal requires more than just the architectural contributions presented in this thesis, but they provide the required foundation on which to realize better interaction components.

2.3.1. Main Experimental Platform

The main experimental platform consists out of two fixed 7-DOF industrial manipulator arms that carry a Shadow "Dextrous Hand" each, and the anthropomorphic robot torso BARTHOC (Hackel et al., 2005). The industrial manipulators are mounted from the top, the humanoid torso is situated at the back. A view of the setup from the point of view of human partner is shown in figure 2.3(a) and figure 2.3(b) shows a birds-eye sketch of the setup. Besides the visible hardware, there is a camera (640x480, RGB) looking down at the table from the ceiling. Two stereo microphones left and right of the humanoid robot provide speech localization, and a head-set is used for speech recognition. Lastly, a CyberGlove II (LLC) provides hand posture sensing.


(a) View from the human's position



(b) Sketch of setup

Figure 2.3.: The static interaction setup.

2.3.2. Development History

Several different integrated systems have been created over the course of this thesis, as specified in table 2.1. The dates given are the completion dates for integration. Partial integrations or developments on the level of individual components which did not result in a significant change of the scenario have not been listed.

Date	Name	Comments
Feb 2008	Pre-Study	BARTHOC only, no manipulation, cf. 9.1
Jun 2008	Curious Robot	Bi-Manual Interaction, cf. 2.3.3
Mar 2009	CeBit 2009	No manipulation, first gaze feedback, cf. 9.5.1
Nov 2009	Final Demonstrator	Haptic learning, advanced dialog, cf 9.5.2.

Table 2.1.: Scenario Creation Dates

Of these scenarios, the June 2008 scenario has been the one with which the most user testing was carried out. A version of the scenario demonstrated at CeBit trade fair 2009 also received a lot of user exposure, but it was limited in functionality and the trade fair situation precluded capturing personalized data. As of the time of this writing, the final demonstrator has been demoed and tested at various occasions, but not with naive users.

2.3.3. Interaction Overview

The overall goal in the scenario is that the robot learns facts about everyday objects, specifically, the names humans use for them, and which of its grasp types it may apply. This information is determined in an interaction setting where these objects are present, through interaction with a human partner in a natural speech dialog. Once it has acquired the necessary information, the robot will move away the object and proceed to the next one.

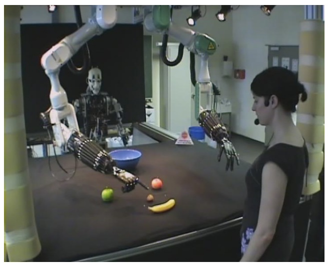
At any point during this basic scheme, the human may interject with corrections, requests to abort an action, requests to present the knowledge acquired, and so on. Doing this during the activities themselves is much more efficient than after, e.g. to abort an action as soon as the human has detected an error, or filling wait time by asking for knowledge. This should of course be possible without requiring detailed information about the implementation, e.g. by learning from the robot's explanations.

	Speaker	Dialog-Act	Verbal	Non-verbal
1	Human	-	Hello, robot.	-
2	Robot	Greet	Hello.	-
3	Robot	Learn label	What is that?	Point
4	Human	- .	That is a banana.	-
5	Robot	Confirm hypothesis.	Banana. OK.	-
6	Robot	Learn grip	How can I grasp the banana?	-
7	Human	- .	With power grasp.	-
8	Robot	Confirm hypothesis.	Power grasp. OK.	-
9	Robot	Explore grip	I am going to grasp the banana.	-
10	Robot	Confirm	OK, I start grasping now.	Grasp
11	Human	-	Stop!	Release
12	Robot	Abort action	OK, I stop.	-
13	Human	-	Grasp the banana!	-
14	Robot	Confirm start	OK, I start grasping now.	Grasp
15	Robot	Confirm end	OK.	
16	Human	-	Good bye.	-
17	Robot	Say goodbye	Good bye.	-

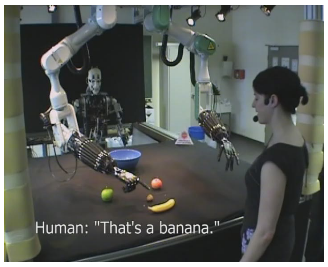
Table 2.2.: *Example* Dialog (adapted from Lütkebohle et al., 2011, table 2)

Some example photographs from the sequence in table 2.2 are shown in figure 2.4.

They are also reproduced at full size in appendix B. Please note that the images have been taken from the initial version of object learning system (cf. section 2.3.2), as it was the one that has received most user testing.

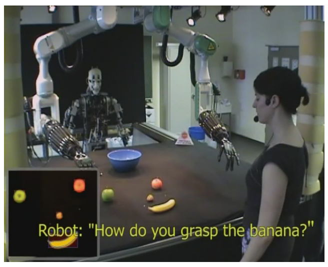


(a) “What is that?”

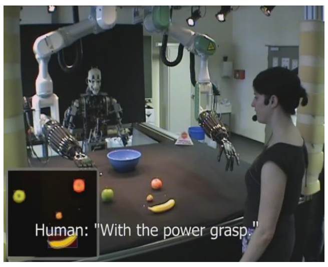


(b) “That’s a banana.”

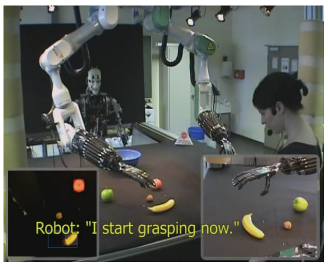
(c) “Banana, OK”



(d) “How do I grasp...”



(e) “... power grasp”



(f) “Grasping now”

Figure 2.4.: Frames from the interaction sequence with utterances overlaid.

2.3.4. System Capabilities

In the capabilities of the system, a trade-off had to be achieved between the overall complexity and realistic interaction challenges. On the one hand, it was felt that the project held enough challenges in manipulation and interaction (see next section) and that, therefore, perception had to be simplified to make the project feasible. On the other hand, the characteristics of perception are an important influence on the interaction and must not be stubbed out entirely.

Therefore, object detection and learning has been included and must run quickly, so that it can be performed during the interaction. This was achieved at comparatively little effort by using simple features and a black background. Furthermore, real speech recognition was used, because any real interaction system will have to be able to cope with the errors introduced by imperfect recognition. However, to prevent the robot’s noise (which is substantial, both during operation and otherwise) from impairing recognition too much, a head-set was used for speech input and the stereo microphones were only applied for localization, which is more robust.

Correspondingly, the challenges have been selected to further the interaction:

- *No prior object knowledge* is present in the system, both regarding an object’s visual appearance and regarding its shape or the way it could be grasped. All of these aspects must be learned. Naturally, the position of objects is also variable.

- *No unnatural interaction restrictions.* In particular, the interaction partner should not need to be known in advance, both regarding position and appearance. This also requires speaker-independent speech recognition.
- *No prior instruction* The human interaction partners should not require prior instruction to operate the system[1].

2.4. Interaction Challenges and Remedies

At first glance, the scenario might appear trivial: Ask for two pieces of information, then put the object away. The dialog example in table 2.2 already illustrates that a substantial dialog can result from these simple premises and then there are the errors. The potential errors are so numerous that the entire following section will be devoted to them. Afterwards, the chosen interaction strategy will be described, as it is specifically intended to prevent and/or reduce these issue.

2.4.1. Scenario Challenges

As in any realistic application, numerous things can go wrong perceptually and during action execution. Furthermore, the dialog must be able to recover from interaction errors, which is a non-trivial undertaking. Both of these problems are well-known and the following list is intended as a summary to convey the range of potential issues resulting therefrom.

- *Speech recognition errors.* Speech recognition is imperfect, with typical errors being that words are mistaken for other words (“substitutions”, S), not recognized at all (“deletions”, D), or that noise (e.g., coughing) is recognized as a word (“insertions”, I). The correctness of recognition is measured by the “word error rate” (WER), as $(S+D+I)/N$, with N being the total number of words spoken.

  The WER is not the only factor in play, as some words are more important than others and it may still be possible to pick out a few usable chunks from many errors (Hüwel et al., 2006), or through combination with other modalities (Wachsmuth and Sagerer, 2002). However, the WER does provide an indication of the errors the rest of the system must cope with.

  Spontaneous speech is particularly hard, due to its more irregular nature, with typical word error rates between 20-40%, as measured by DARPAs SWITCHBOARD and CALL HOME tasks (Rabiner and Juang, 2008, p. 533). Even higher rates are possible due to error cascades (Karat et al., 2000).

[1]Strictly speaking, this is impossible to achieve, because appearances are already informative, and experimental settings also require some form of cover story. However, the goal here is to give as little information as possible.

More positively, for task-constrained speech recognition error rates as low as 2.5% have been achieved as measured by DARPAs "Airline Travel Information System" (ATIS) task (Rabiner and Juang, 2008, p. 523). This underlines the importance of contextual constraints.

The causes for speech recognition errors are many and particularly relevant for speech based interaction, so they will be shortly summarized here:

- *Out-of-vocabulary words.* Words that are not in the training set of the recognizer may lead to errors. At best, they cannot be matched to any known word and are rejected. At worst, they are matched to the most similar word, which is then recognized erroneously. Both of these errors may further contribute to grammar errors.
- *Grammar violations.* Grammar is an important statistical recognition cue, because certain word orders are much more likely than others. However, human conversational speech is frequently ungrammatical and, further, word recognition errors may lead to correct sentences becoming ungrammatical. Grammar design is thus a trade-off between providing helpful constraints and overconstraining the problem.
- *Inapplicable training sets.* Speech training sets are expensive to create and many existing training sets only contain speech from particular situations (such as reading newspaper articles) or a particular pronunciation (e.g., British English). This influences the pronunciation greatly and can also cause certain errors (such as ungrammatical speech or filler words) to be underrepresented, thus skewing the language models.
- *Foreign speakers.* A special case of the previous point, foreign speakers are often particularly hard to recognize due to differences in phonetic realization.

- *The vocabulary problem.* The vocabulary available to the robot is usually significantly smaller than that of humans and is not known to the human interaction partner (Furnas et al., 1987). This may cause the human to use words or concepts the robot cannot understand, leading to the aforementioned out of vocabulary errors. More importantly, however, it may also lead to the human being confused and not saying anything at all, or asking about possibilities.

- *Unwarranted expectations.* Closely related to the previous problem is the fact that, without further information, humans often form expectations about a system's capabilities from visual appearance alone (Duffy, 2003; Lohse, 2009). Such expectations often overestimate the true capabilities of the system (Hanheide and Sagerer, 2008). This, again, may cause the humans to produce sentences which the robot does not understand. For examples, see section 9.3.

- *Uninterpretable constructions.* Even when speech recognition is correct, dialog systems cannot interpret all valid sentence constructions. In particular, relative clauses, elided words and deictic constructions are difficult and may sometimes require external information for interpretation.

- *Turn-taking confusion.* Long processing by the robot or confusion on the part of the human may lead to pauses which confuse turn-taking in the dialog, thus causing, for example, both partners to speak up at once or both wait for the other to speak next.

- *Spurious object detections.*Object detection may get confused and produce objects which are not there because of shadows or highlights, motion of the human and/or the robot's own motion. This can cause the robot to refer to objects that are not actually present, or not present anymore. While some of these spurious results may be suppressed through temporal averaging, this increases latency (compare previous point).

- *Wrong object detections.* Visually similar objects may be confused by the robot, internally assigning the wrong label. This can lead to confusion in the dialog when an object is referenced that is not actually present. Conversely, another object of the same type may not be recognized as such due to different lighting or orientation, leading to repeated questions, which again confuses the user who assumes that the object has been learned already.

- *Temporary failure.* As interaction and action execution are largely decoupled, users may request actions when they are not currently possible. Additionally, the system may fail to perform an action but not notice this, and the user needs a means to inform the system about this.

From this list of possible errors alone, it is not hard to understand why many attempts at Human-Robot-Interaction have resulted in frustration for both users and developers, or led to restricted scenarios operable only by trained personnel. The simple fact is that good solutions for many of these problems are not yet known. Fortunately, it may be possible to reduce some of these issues by combining different solutions and exploiting the constraints provided by an *active* robot system.

2.4.2. The Dialog Structuring Approach

The thesis followed in this scenario is that, of the many potential issues outlined, many can be prevented, and the remaining can be reduced in impact, through active dialog structuring by the robot. The concrete structuring strategy followed will now be described.

The most important means for structuring is the *implicit capability display*, whereby the robot reveals its capabilities through its actions. Furthermore, to aid in the crucial start phase, the robot has partial *interaction autonomy* and can start acting immediately. The various phrases reveal capabilities, and regulate, as follows:

- *"What is that?" (Label query)* This reveals that the robot has a concept of objects, but that it does *not* know their names, yet. The phrase also avoids use of descriptive attributes.

  In the first iteration, this phrase is accompanied by an illustrator gesture, whereby the robot points at the object. In subsequent iterations, it is also accompanied by a state display: The robot either looks at the object in question or at the user.

- *"How do I grasp the L?" (Grip query).* This has been the initial grip query phrase. It reveals that the robot can grasp objects, but it does not reveal how (cf. section 9.3 for why use it). Furthermore, it reveals that the robot has learned the label, thus making it available for future reference.

- *"Show me how to grip L, please."* This grip query phrase has not been used with naive users, yet, but it is the candidate phrase to indicate that gripping should be demonstrated using the CyberGlove. It reveals that the robot can make use of the demonstration (and that it can do so now).

- *"OK, power grasp"* This informational phrase provides the robot's technical terms for grasp types after demonstration, as a shortcut for future use.

- *"OK, I start grasping now"* This is, firstly, an explicit state display. Furthermore, the use of the word "start" indicates that the action will take a while. Lastly, it indicates that the action could be aborted, though this is a weak cue that should be reinforced through other means.

- *"I'm passing the bowl. Let me know when to release it."* This response to the bowl passing request contains an explicit instruction on what to do when the user is ready to take over the bowl. It also provides a cue as to which word to use for that request ("release").

On a more general level, the *use of questions* in structuring provides two cues:

- Questions are a regulatory cue that lets the user know she can act, once the question utterance is complete.

- Questions also predispose users to answer in a way that continues the question, instead of a full sentence.

As an example for the latter, consider object label presentations: When a tutor explains it directly, full sentences with deictic references are likely to occur in many different variations. For instance, possible constructions include "That is a ...", "Look at the ...", "Look, a ...", "The ... here". Furthermore, the same information may be given implicitly in many different ways, e.g. as "The X belongs on top of the Y". In contrast, when asked like "What is that?", only one answer construction is used (cf. section 9.3).

2.5. Summary

To enable robots to share their environment with humans and learn novel tasks, they must be able to learn from humans, but such learning is difficult for both the robot and the human. To improve this, social cues, natural speech dialog and *interaction autonomy* by the robot have been combined in a scenario, where the robot learns objects attributes in mixed autonomy. In particular, the system attempts to prevent many potential problems by posing questions autonomously, thus providing both information on its goal and on its current state. The resulting behavior resembles that of curious kid, and gave rise to the name “Curious Robot”.

Three different iterations of the system have been built, and evaluated in various forms. Achieving the integration, and flexible role change, while keeping the system maintainable, has caused many interesting questions on the architectural level. The respective solutions, and the corresponding evaluations, makes up the bulk of the remainder of this thesis. Last, but not least, details on the user-centered evaluations are found in chapter 9.

3. The Task-Interaction Architecture

The architecture design presented in the following is not a wholly new proposal, but builds on proven structures, particularly layered architectures (Kortenkamp and Simmons, 2008). Based on these, the present architecture intends to advance the state of the art on integration between Human-Robot-Interaction and robot activities. This already suggests a, in comparison to earlier descriptions, increased emphasis on the *interconnections* between components. Furthermore, to support rapid, incremental research prototypes, it emphasizes the maintainable construction of systems from multiple, largely independent sub-systems.

In particular, novel aspects of the resulting system are the tight interplay between two independent coordination components, one for motion control, and one for dialog management, and a more concrete description of the functional structure within sub-systems and components. The principles upon which these are built will be introduced in this chapter, and the results are frequently touched upon throughout the remainder of this thesis.

3.1. The Role of Architecture for Robotics

Before going into details of the present architecture, first some context from both robotics and software engineering.

A rough but useful definition of the role of architecture for robotics is given in Russell and Norvig (2002, p. 786) as “The architecture of a robot defines how the job of generating actions from percepts is organized”. Kortenkamp and Simmons (2008, p. 187) elaborate a bit more on the ingredients of research in robot software architecture by distinguishing, on the one hand, the subdivision of a robot’s system into subsystems, and their interaction (the *structure*) and the “computational concepts” that shape its design (the *style*), which includes the communication and control style.

The raison d’être for architectures is succinctly summarized by the same authors as: “Robot software systems tend to be complex” and the goal of architecture to:

> [. . .] facilitate development by providing beneficial constraints on the design and implementation of robotic systems, without being overly restrictive (Kortenkamp and Simmons, 2008, p. 188).

In other words, there are usually many ways of realizing the same functionality in a robot software system, but not all of them are equally good, particularly when it comes to the management of complexity. An architecture should guide the developers towards the better solutions. Complexity management and beneficial constraints for design are the main benefits an architecture should provide and by which architecture proposals can be compared.

This is generally in line with the software engineering view on software architecture, which views architecture as a sub-field of software design, with complexity management one of the main goals (Shaw and Garlan, 1996, p. 1).

However, while all systems have an architecture, some in the robotics community have questioned whether such research will lead to *generalizable* knowledge on robot software. For example, Bonasso reports that

> There has always been some skepticism in the community that there is any generality to be derived from using software architecture with robots [...] (in Kortenkamp, Bonasso, and Murphy, 1998, p. 193).

There are also those that doubt the possibility of a common architecture not for theoretical but for the purely practical reason, that it is difficult to achieve consensus in a large community. For example, Smart (2007, p. 5) has asked to "divide the fields into subfields", to facilitate agreement on a common middleware, which is one architecturally relevant component of a system.

This practical concern is echoed by Kortenkamp and Simmons, who observe that "in many existing robot systems it is difficult to pin down precisely the architecture being used" (Kortenkamp and Simmons, 2008, p. 187) and that one cause of this is that "architecture and the domain-specific implementation are often intimately tied together".

In essence, this means that the diversity of robotics is a problem in finding common ground. One big contributing factor is that most roboticists have only experience with one or a few domains and tend to make assumptions in their implementation which are not valid in other domains, or for other architectures.

Having mentioned this, it appears prudent to step back a little and re-evaluate the original goals. While a single joint architecture looks unlikely, this is not all that uncommon. Many fields have several competing architectures, which usually share a few common principles but differ in others. Also, it is probably the case for applications from many other fields that architecture and implementation details become blurred with time and re-factoring would be needed to clean things up.

Component-Based Systems

Fortunately, a few common ideas on how to architect robot software systems have emerged and while these may not always give the complete picture, they at least provide some guidance. Regarding the many differences between systems, maybe an older quote from Firby still applies:

> Many details differ between these proposals, particularly in the area of philosophical commitment, but they share the common idea that the actual behavior of the robot at any given moment is the result of a set of interacting processes acting on input from the environment (Firby, 1994).

As Firby alludes to, many differences may be not for technical but for other reasons (the "philosophical commitment" refers to the representation debate kicked off by Brooks (1991)). In other words, the proposals often differ in *how functionality is broken up* into parts, but they agree that there are fairly *independent parts which must be integrated coherently.* This suggests the possibility of an underlying technical core which is common to all, or at least many, systems.

In fact, recent tutorials by Brugali et al. (2007); Brugali and Scandurra (2009) have emphasized this point, by making a connection to software engineering and distributed systems research. In these areas, the type of system common in robotics is called a "component-based distributed system" and a rich literature for design and integration exists.

In contrast to the focus on the organization of control prevalent in earlier architecture research, software engineering focuses mainly on *re-use* and *flexibility.* Therefore, clear specifications for component interfaces are said to be "key to its successful use" (Brugali and Scandurra, 2009, p. 89). This approach is based on the fundamental tenet that "only what is hidden can be changed without risk" (Parnas, 1972b,a). Furthermore, explicit contracts are emphasized to lay down "mutual obligations", such as "preconditions, postconditions, invariants, and protocol specifications" (Brugali and Scandurra, 2009, p. 92). Thus, components are treated as black boxes, but have explicit, well-specified behavior.

Of course, earlier architectures also had component interfaces, as these are requirements of an implementation. However, they were rarely specified or reported on. Regarding contracts, it can probably be said that the control organization constitutes a contract. Here, the difference to previous research is that *explicit* specification of contracts is requested and that such contracts could be used for functional obligations, too. Besides better system specification, explicitly specified contracts also enable better use of supporting tools.

In general, the concept of component-based systems for robotics is not so much a novel insight as a bridge to encompass valuable research from other domains on how to engineer such systems in a disciplined manner.

Separation of architectural concerns

A well-known strategy to reduce system complexity is to break up the problem into parts that can be solved individually and then recombined in a principled fashion. On the architecture level, Radestock and Eisenbach (1996) have proposed the separation of communication, computation, configuration, and coordination. These are defined in Radestock and Eisenbach (1996, section 2) as:

Communication describes *how* something is communicated, for example, whether it occurs through RPC or message passing.

Computation realizes the "behavior" and thus *what* is communicated.

Configuration defines the interaction structure, i.e. who is communicating with whom.

Coordination defines patterns of interaction, or *when* something is communicated.

This particular separation is claimed to enable a "high degree of re-use and easier maintenance". It must be noted that achieving such independence cleanly can depend on the particular choices made – for example, RPC-style communication makes it more difficult to choose the target externally, because the sender already specifies one. While adaptation or interception can make it possible, they also incur additional complexity, which must be weighed against the advantages gained.

Therefore, and this is also the area the previous authors have targeted, such separation is usually only sought at the component level of a distributed systems. The approach in distributed systems particularly concerned with this separation is known as *event-based* distributed systems (DEBS) (Mühl, Fiege, and Pietzuch, 2006).

In contrast to adaptation, DEBS emphasize that components generate and consume *event notifications*, which have no target (Mühl et al., 2006, p. 3). Thus, there is no need for adaptation, though it requires components to accept inputs at some later time instead of immediate results. In notification, there is room for considerable variation. For example, a typical event-bus, of which the "Active Memory" used in this thesis is an example, broadcasts events and then filters based on the event contents or metadata. This works on the inter-component level, and is maximally flexible, but also adds some matching overhead.

A second example, the data-flow framework proposed in chapter 7, uses a fixed – but externally specified – connection between nodes that comes close to a function call in efficiency. It works on the intra-component level, which requires more efficient event notification and has less of a need for variation in communication mechanisms.

In sum, the present work shares the goal of separating these concerns and intends to further it, with an eye on the possible gains and reasonable effort. To this end, two mechanisms already mentioned – one on the inter-, the other on the intra-component level – will be introduced. The question whether the desired re-use and flexibility can indeed be achieved will be one important aspect of evaluation.

Coordination in robotics

Typical coordination concerns in robotic systems are sequencing basic actions to achieve a complex activity, managing access to scarce resources, and selecting the next activity to do. This is also often referred to as "control", but to avoid confusion with low-level control (as in a PID controller), the term coordination is preferred.

Originally, two fairly different styles have been proposed for this purpose: In the (much earlier) sense-think-act style, a strict sequence of perception, planning and acting is repeated throughout. The planning stage predetermines the sequence of actions to do, taking resource constraints into account, and thus covers coordination.

In contrast, in the behavorial style, a number of action components are running concurrently, reacting to inputs. Here, coordination is split into two phases: Firstly, behaviors only become active when certain conditions hold. This may still result in several conflicting behaviors attempting to run, and therefore, an arbitration step is added. Typical arbitration methods are a priority scheme, as in the subsumption architecture (Brooks, 1986) or combination of the output requests, as in motor schemas (Arkin, 1989).

The relative merit of these styles has been the subject of intense discussion (e.g. compare Kortenkamp and Simmons (2008, section 8.2)), much of it because these questions have also been related to models of cognition. Apart from that, however, the advantage of behaviors (fast reaction to local stimuli) and planning (optimizing longer-term activities) proved individually too attractive and led to combined architectures.

For example, in Gat's "ATLANTIS" architecture – which is typical of many architectures at the time – there are three hierarchical layers: "controller", "sequencer" and "deliberator", in this order (Gat, 1998, p. 200). The control layer is supposed to contain "primitive behaviors", so this layering could be considered to use a behavioral style at the bottom, with behavior-selection and coordination on top. The use of the term "primitive behavior", however, already signifies that these proposals may use different granularities than a pure behavior-based architecture.

This leads to an architectural issue in three-layered architectures: How to assign functionality to the various layers. For example, Mataric (1992) and Connell (1992) represent two early architectures for mobile robot navigation that use the same underlying path planning approach and the same landmark types, but drastically different distribution of functionality – Mataric encodes the map in a linked behavior hierarchy and Connell uses a symbolic model.

Similarly, Volpe et al. (2001) point out that there is still considerable debate about the best assignment of functionality between planning and execution. Furthermore, they note that the three-layer model obscures the internal hierarchies and sub-systems contained within the layers.

Decision-making Structure vs. Functional Structure

It would appear that some of the described differences in distributing functionality are due to irreconcilable differences in views about cognitive architectures, but nonetheless, many similar functional algorithms will be needed. Thus, the primary objective of this thesis is to enable system construction in a way that facilitates reuse and reconfigurability and thus leads to better comparability between alternative architectural choices.

In particular, it would appear that the mentioned styles differ primarily in how they make decisions. *That* they make decisions, that these decisions are based on both sensory input from the world and some form of a model[1], and that they result in some kind of action in the world, these things are common. It might be said that such a description is too abstract to be useful, because it glosses over the differences in decision making. However, the position taken in this thesis is not that this view is better, but rather that it is *complementary* to the decision-oriented view. Thus,

[1]Drastically different models, but models nonetheless.

while it may obscure decision making, it highlights functional relations. Both of these views are necessary for a complete architectural description.

The functional structure used in the following is that of a loop with the environment. These loops will likely exist at various levels of abstraction, so there needs to be a way to coordinate between them. The differences in architectures are then expected to be expressed primarily in the number and type of coordination structure *between* these loops.

3.1.1. Scenario-related Architectural Challenges

Apart from the previously outlined issues, which are largely shared with all complex robot systems, the following three challenges are specific to the Human-Robot-Interaction research targeted by the previously introduced scenario.

1. *Interaction independence.* The optimal interaction policy is often not known but still the subject of ongoing research. Such research needs to evaluate alternatives, i.e. compare different systems. If a different user interaction would require changes throughout the system, the effort would make the undertaking infeasible. Conversely, if a change in the system would require changes to the interaction immediately, functional progress would be stifled. The notion of dialog independence (Sotirovski and Kruchten, 1995) from traditional human-computer-interaction calls for a separation of these sub-systems to achieve the necessary flexibility. This challenge is the overarching issue addressed.

2. *Coupling of activity execution to state monitoring.* As mentioned above, during interaction it is frequently necessary or advantageous to comment on the state of actions (e.g., whether an action is ongoing, completed, etc.) If the state of actions can only be inferred from changes to the environment, or by monitoring of internal variables, any change to the executing component would require changes to the interaction components, too.

3. *Conflicting resource access.* Interaction and perception may both require the same resource, e.g. speech output or a camera head. To resolve such conflicts, both resource availability and higher-level tasks must be considered. For example, while speech could be output continuously or the camera could be moved incessantly, this would not achieve the purpose. From an architectural point of view, it is important to ensure that such conflicting accesses can be resolved without incurring functionally unnecessary couplings between the components.

3.2. System Overview

Like many current robotics systems, the system developed as part of this thesis is a component-based, distributed system (Stewart and Khosla, 1995; Brugali and Fayad, 2002; Brugali et al., 2007). It consists out of just over twenty functional (cf. figure 3.1) components, described in the following.

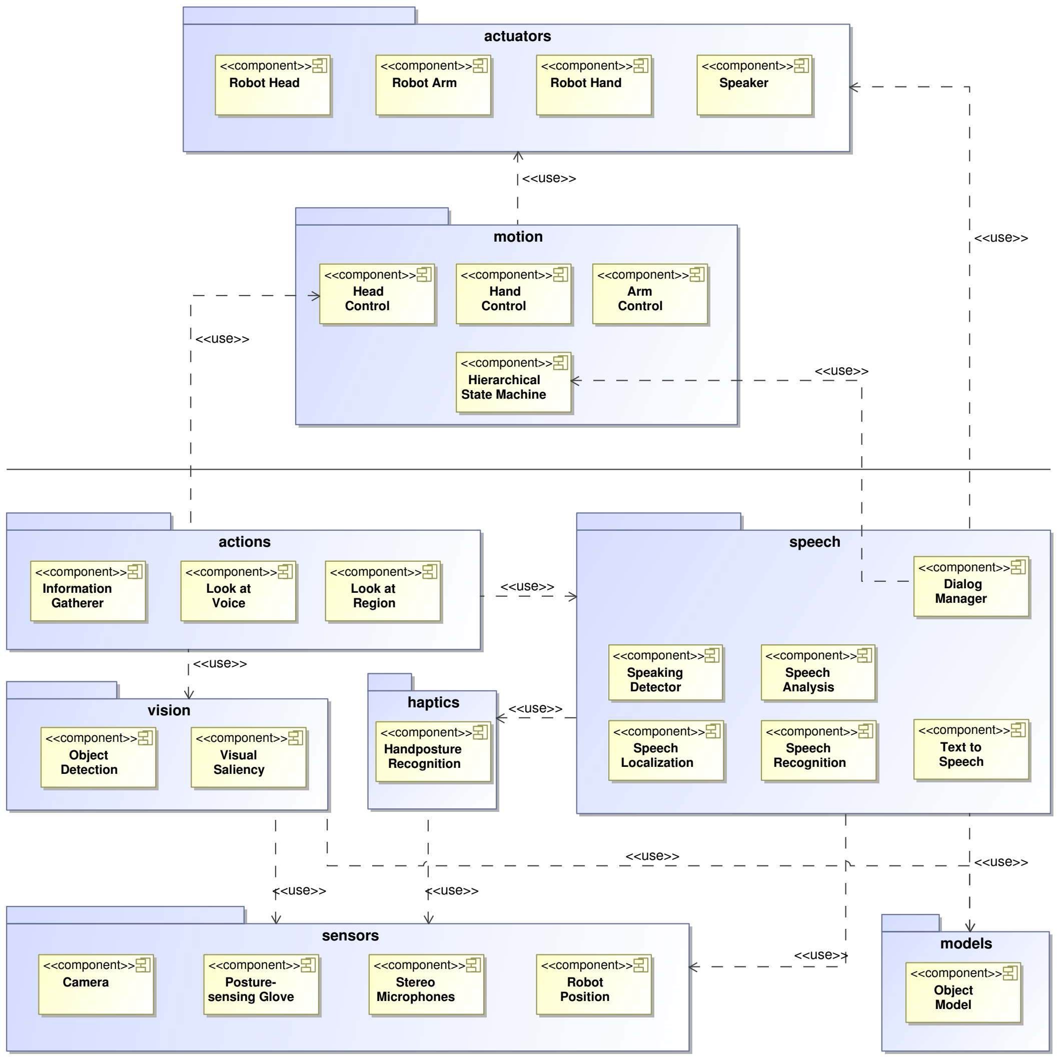

actuators
<<component>> Robot Head
<<component>> Robot Arm
<<component>> Robot Hand
<<component>> Speaker
<<use>>
motion
<<component>> Head Control
<<component>> Hand Control
<<component>> Arm Control
<<component>> Hierarchical State Machine
<<use>>
<<use>>
<<use>>
actions
<<component>> Information Gatherer
<<component>> Look at Voice
<<component>> Look at Region
<<use>>
speech
<<component>> Dialog Manager
<<component>> Speaking Detector
<<component>> Speech Analysis
<<component>> Speech Localization
<<component>> Speech Recognition
<<component>> Text to Speech
<<use>>
vision
<<component>> Object Detection
<<component>> Visual Saliency
haptics
<<component>> Handposture Recognition
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
sensors
<<component>> Camera
<<component>> Posture-sensing Glove
<<component>> Stereo Microphones
<<component>> Robot Position
models
<<component>> Object Model

Figure 3.1.: Package and component overview of the whole system. Except for the "models" package (which has been placed to reduce clutter), the diagram has been laid out to indicate the different functional levels along the y-axis, with sensors at the bottom and services at the top. Thus, it is a "flattening" of the loop in figure 3.2.

The present system uses the XCF middleware toolkit (Wrede et al., 2006). XCF offers several interaction patterns, but here, only the event-based notification mechanism is used. The whole system is event-based, to decrease coupling between components (Faison, 2006; Mühl et al., 2006). Furthermore, the event notification mechanism also provides efficient communication at small overhead. For example, the components in figure 3.1 show "use" associations pointing from the consumer to the producer. However, the consumers do not have to poll producers actively, but simply register content-based event matchers on the event-bus (which is called "Active Memory" in XCF, because it can also provide notification storage). The producers push new information onto the bus as soon as it becomes available and the bus then delivers it to the consumers.

The XCF infrastructure components that perform event notification have been omitted for brevity from most of the diagrams in this thesis, with just a few exceptions where the exact interaction is relevant for the function. In all other cases, only the resulting notification is shown, directly linking producer to consumer.

3.2.1. Components & Functionality

The present system can be roughly split in two halves functionally: Motion control (shown at the top of figure 3.1) and interaction management (below the separator), which includes perception. The fact that the "use" associations only point from bottom to top already indicates that the loop between the two sub-systems is not closed on the sensor-level (though motion control is closed-loop internally). However, through the use of the life-cycle coordination pattern (cf. chapter 4), there is a bi-directional high-level interaction, so that completion and failure information can be communicated.

Robot perception uses three external sensors (for vision, speech and haptics), plus internal joint position sensing. Based on these sensors, there are components realizing hand gesture classification, object detection and salient point detection as well as speech recognition, speaker localization and voice activity detection.

Robot action is realized by three types of components, with one each for the head, the arms and the hands. Of these, hand and arm control are tightly integrated via the Hierarchical State Machine (HSM for short) component, whereas the head is controlled independent, as it does not share a work-space with the arms.

The link between perception and action is realized, primarily, by two coordination components, the dialog manager and the hierarchical state machine. This is supported through independent but coordinated behaviors for background activities (the "looking" components).

Last, but not least, there is a dedicated system initiative component, the "information gatherer". It realizes the internal information acquisition strategy by making suggestions to the dialog manager. While many of the other components provide scenario-specific functionality, they only produce it upon request. Thus, in some sense, this component is the one which creates the drive of the "Curious Robot".

3.3. Functional sub-system Schema

While the control architecture of the present systems follows the style of a recursively layered architecture (Kortenkamp and Simmons, 2008; Albus, 1992), the emphasis of this thesis is on the functional structure, and the interfaces needed between the functional parts. The general idea of a loop with the environment is depicted in 3.2: It describes the common parts, which every externally visible function of a robot needs for realization, as a *functionally layered, closed loop* with the environment.

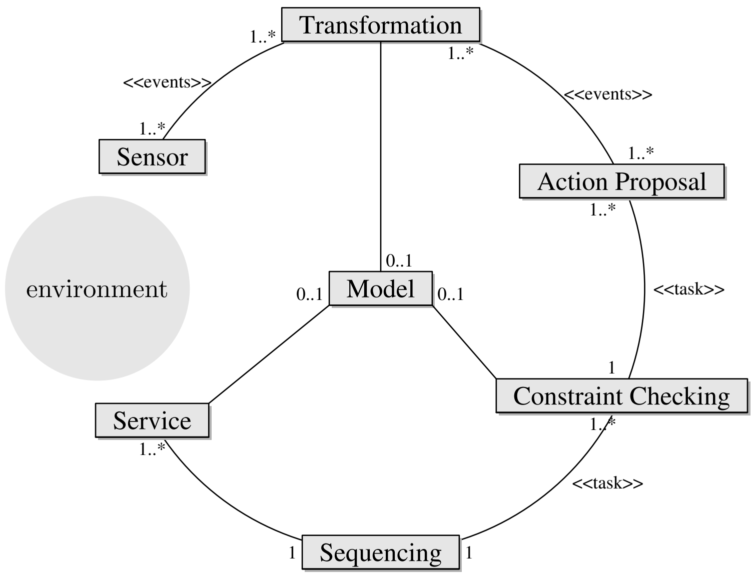


Figure 3.2.: Abstract sub-system Schema. The "environment" here can be either the real world or the world-view of an underlying sub-system, cf. figure 3.3

For a behavioral style, such loops would be tightly integrated, possibly in one component, and with a minimal model. In a pure sense-think-act style, the loops would be on different hierarchies, with a symbolic model, and action proposals exclusively passed in from higher levels. A hybrid style also uses several hierarchically structured loops, but would typically include (limited) action proposal and decision capability on lower layers.

The schema should not be construed to say that each of these functions must be in a separate component – for comparatively simple things such as a behavior or PID control, it may very well be all in one component with functions on different layers. It is only on higher and more complex levels, that the functional blocks are typically different, possibly distributed, components.

The functional parts given are not new, but have appeared in similar forms in previous architectural proposals, sometimes under different names. The will be shortly summarized now, and an overview diagram at the end of this chapter will also show the concrete assignments of the components of this system to the parts.

Sensor. Measures a conceptually separate sub-system. For the lowest layer of a robot, this measures from a hardware sensor. For other layers, it observes the data produced from a lower-level component. The output is the raw sensor value, typically one or more decimal numbers.

Transformation. Any kind of data transformation, for example, aggregation, windowing, fusion, classification, prediction, etc. The output of these transformations can be diverse and be both numeric, symbolic or combinations thereof. There are often several transformations chained together.

Action Proposal. Suggest a new action or a modification of a current action and produces a description of the desired action. Anything from changing the motor current (outputs new current value), moving a robot part (outputs target description), producing a verbal utterance, up to full-blown planning.

Constraint checking. Determines whether a proposed action is possible given the current system state. For example, this may check a motor duty-cycle limit, joint limits, the current human interaction state, etc. If the action is possible, it is output unchanged. Otherwise, a refusal is communicated back to the proposal component. This is often part of the action proposal, but separated out here, to facilitate external proposals.

Sequencing. Takes one or more checked actions and coordinates their execution. For example, this may cause successive actions to be either serialized or all but the last one being discarded. Synchronization between different services is also performed here.

Service. A generalization of the common actuator to other components that carry out longer-running actions as a service. It takes an action representation and executes it by controlling a lower level (either hardware or an underlying subsystem). It reports back whether the action was successfully started or an error occurred. Errors during execution can be either reported here or through an appropriate sensor.

Model. A model of the environment. Depending upon the type of system, this may be a very minimal representation of sensor readings, or a more elaborate, abstract representation. Examples include the speed of a motor, the configuration of a kinematic model, an estimated world-state, etc.

3.3.1. Communication

Most of the connections in the loop are specified as m:n, i.e. each component may have several inputs and each input may go into more than one component. The exception to this is the sequencing step, which is explicitly intended to coordinate the various inputs and produce a coherent action – thus, it needs to be a single component and each service has to be associated with exactly one sequencing component.

The interaction pattern for communication is pure event-based notification from sensor through transformations to the action proposal stage. After that, it is based on the life-cycle coordination pattern (cf. chapter 4). This pattern enables communication about the state of an action without introducing direct couplings between components.

That said, one note on generality: The schema has been modeled as it is to allow future extension and it is certainly conceivable that there are multiple components for most types. However, in most current systems, including the present one, true m:n communication is common only in the sensing and transformation stages and even there only at higher levels. After that, there is often only one or a few action proposal components per loop.

The reason is simply that there are often many sensors and usually even more transformation components, regardless of the type of system. In contrast, action-proposal are usually fewer, particularly in hybrid architectures with only a few background behaviors, such as the current one.

However, one other uncommon aspect is not just for future extension: While sensors and transformations are often viewed as output-only, modeling their connections bidirectionally is intended and reflects the view that these components usually also have inbound communication. At the simplest end this could be just an "active/inactive" state but it could also expose internal variables or training inputs, and thus facilitate external configuration of these transformations or adaptation through learning.

3.3.2. Model of Computation

The sub-system schema presupposes a component-based, distributed architecture, where each component has its own process. For such a model, the schema shown is complete.

It would be interesting to explore use of the schema for the internal structure of components, and in fact, its general structure has been designed to encompass low-level motor control, too. Here, however, the models of computation are more diverse (cf. Bhattacharyya et al. (2005, chapter 1) for an overview). For the schema to encompass this, additions may be required, depending upon the exact model chosen. While such extensions are beyond the scope of this thesis, it is notable that the strict layering may be impaired for some models, but not for others. If the component is fairly simple, such as PID control, this is not usually a concern. For more complex components, use of models which keep the layering (such as discrete event or process networks) is suggested.

3.4. System Schema

The second part of the schema describes how to combine multiple sub-system loops into a coherent system. This schema represents the primary architectural guidance principle proposed, by limiting the points of contact (which represent the extension points) between the loops, and using the task-state pattern for coordination between them. The proposed approach is visualized in figure 3.3.

The "transformation" stage of the left sub-system becomes the input for the higher-level "sensor" stage and the "action proposal stage" receives input from the higher-level "service" stage. These are the *only* points of contact. Note that there may be more than one sensor and more than one service in the right sub-system.

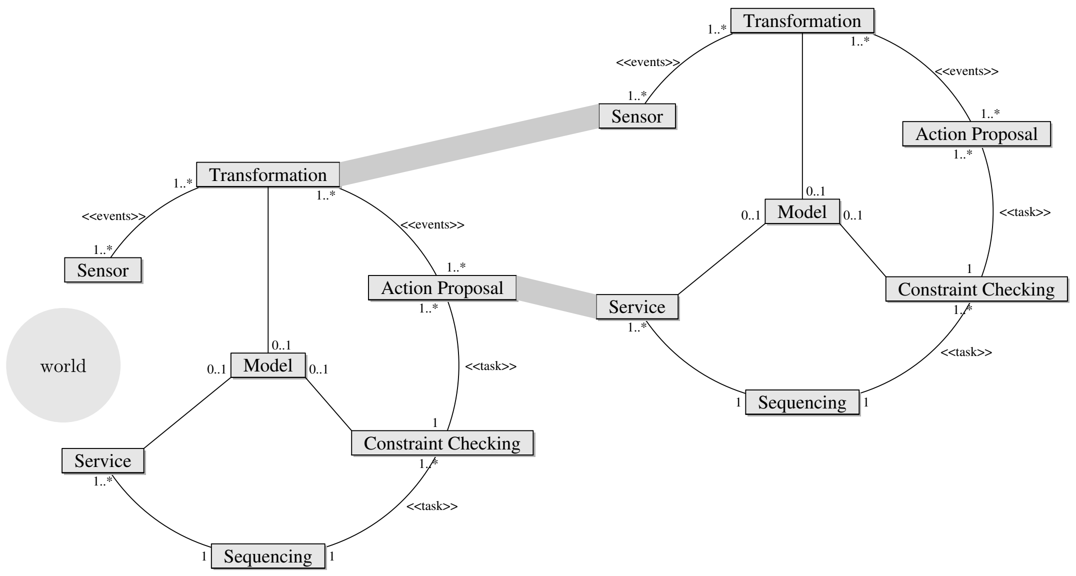


Figure 3.3.: Abstract System Schema. A full system will usually have more than two loops and more than two levels, but the general structure remains the same.

That a system is built up hierarchically is a common approach, because of the belief that "hierarchical structures reduce complexity", known as Simon's law after Herb Simon (Endres and Rombach, 2003, p. 40)[2]. The reason for this is that sub-systems may be studied individually and the effect any outside system may have on them is restricted to the connecting points, thus making it predictable and comprehensible, even if not all the external systems are known.

The exact choice of the points of contact is a design choice, however. The choice made in the present system emphasizes re-use and sub-system independence. Re-use is realized by taking the sensory data for the higher-level sub-system from the transformation stage, after post-processing and analysis.

It should be noted that the same transformation component may belong to more than one processing loop and that different loops may post-process outputs that other loops use directly as input to action proposal. Therefore, access to transformed data

[2]Simon's original account has likely idealized this principle beyond the empirical evidence, particularly regarding the way how hierarchy is created. Nowadays, the view is more that hierarchy is not inevitable but that it can be imposed to serve a useful function. For example, compare Agre (2003).

at various stages is not prohibited by the schema. However, the schema strongly suggests that higher-level layers which interact with an action proposal component use the same input as that component as the basis of their sensory processing.

Sub-system independence is achieved by injecting requests at the action proposal stage. Often, these take the forms of parameters, such as the set-point of a PID controller, which are simply used within the action proposal stage and not passed on. Therefore, control of the underlying sub-system can be said to be indirect, because the concrete action can only be influenced in ways allowed by the action proposal component. Direct access to the service layer is prohibited (and, in fact, prevented by the sub-system encapsulation), to ensure safe operation[3].

Furthermore, the design also emphasizes that requests by higher levels are never guaranteed to succeed. There are numerous reasons for failure to perform as requested, primarily changes in the environment which cannot be compensated, but also because something is requested which the underlying hardware cannot perform. Despite best efforts, such cases can occur due to bugs or inconsistencies in any of the participating components. The present schema provides a clean way of checking and reporting.

3.4.1. Feedback Coupling

The service layer of the higher-level component requires feedback about the outcome of its requests. However, in the present design, the only point of contact is the action proposal stage, whereas the outcome of actions is only known at the service stage. Thus, a question about the schema might be how the feedback is achieved. The answer to this question is the life-cycle coordination pattern, introduced in chapter 4, which provides such feedback.

Given the use of feedback from later stages, it might appear that there is hidden coupling to these stages of the loop. This may or may not be the case, depending upon the data exchanged. In principle, and this is why there is no coupling shown in the schema, coordination can rely solely on the life-cycle states, which are independent of functionality. In this case, the only data-coupling would be to the data received and sent by the action proposal stage.

However, if later stages also add information to the action representation, and such information is used by the other levels, an implicit coupling would indeed be present. The present system does not do this, but it is certainly conceivable that other systems might.

[3]The necessary checks are intended to be lightweight and quick. For example, for collision prevention during motion, one would expect checking of joint limit constraints and direct collision sensors, not full world-model based collision detection (which would rather be part of path planning, i.e. action proposal). In other words, fast checks that prevent mistakes and don't add significant latency.

3.4.2. Concrete Architecture of the Final Demonstrator

This section outlines the sub-systems of the final demonstrator, to provide the context for the remainder of this thesis. Full details will be given in chapter 9, which discusses the evolution of the scenarios and evaluates their respective architectures.

The final demonstrator consists out of three coupled sub-systems (cf. figure 3.4):

1. *Interaction Management (middle).* Realizes the "curious robot", which engages the human in a dialog about objects it perceives.
2. *Manipulator Control (left).* Coordinates control of the main robot arms and hands, for pointing and grasping, based upon action requests by the interaction management sub-system.
3. *Social Feedback (right).* Observes the dialog and produces social cues, such as gaze feedback and quasi-immediate reaction to user utterances.

To give a rough idea of the components and their interaction, figure 3.4 shows a birds-eye view of all sub-systems. The main point here is that the coupling between sub-systems is low, which is a fairly rare sight for a complex robot system. Furthermore, it can be seen that all sub-systems have components in all stages of the sub-system schema, albeit in different forms and this will be one of the discussion items during the architecture evaluation.

One further fact worth mentioning regarding figure 3.4 is that some of the components are depicted on multiple levels, including gaps (e.g. "Hierarchical State Machine (HSM)") or even almost all levels (e.g. "arm control"). Such spread violates the suggested schema. One reason for this situation is that these are monolithic components which could not be exhaustively analyzed (so they might contain internal communication that the present author is not aware of). Thus, while it appears that they follow the suggested layering internally (compare the notes on the various functionalities provided by "arm control"), this is not known for certain.

However, the more important reason, here and in the rest of the thesis, is that these figures represent the reality as it is, not what the present author would have considered best[4]. While for some of the components development has been carried out directly, or could be influenced, for other's no influence has been possible. Furthermore, for almost all systems, a look in hindsight reveals room for improvement, so such cases are likely less surprising than a perfect picture.

[4]Fortunately there is a large overlap, just not a complete one.

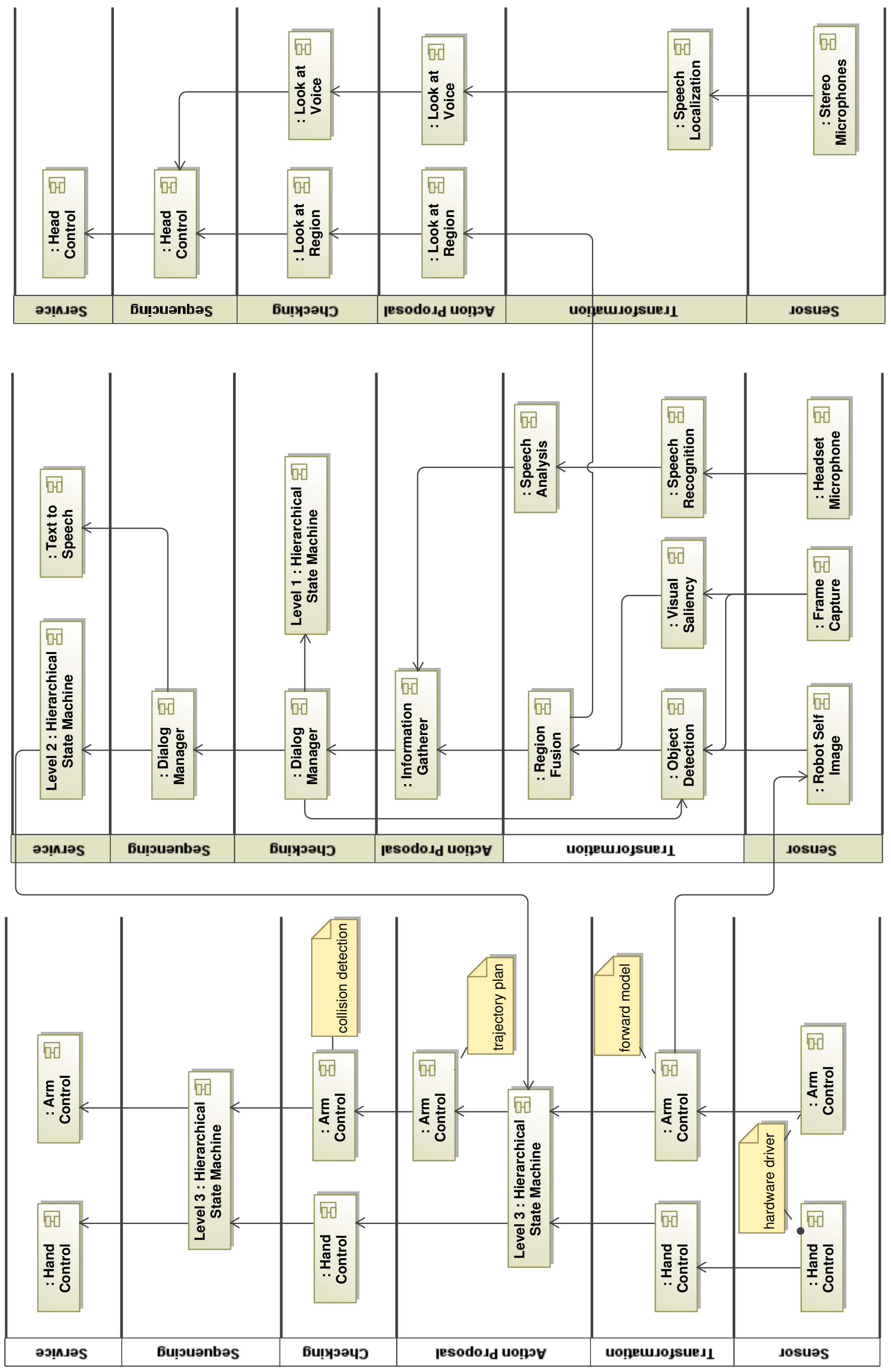


Figure 3.4.: Birds-Eye System Overview (for details, cf. sections 9.2 and 9.5.2)

Part II.

Coordination

4. The Abstract Task-State Pattern

The following chapter will introduce one of the two core contributions in the form of a software design pattern. A design pattern, according to the seminal work of Gamma, Helm, Johnson, and Vlissides (1994):

> A *design pattern* [...] describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context (Gamma et al., 1994).

The present pattern primarily addresses the architecture of systems:

> An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them (Buschmann et al., 1996, p. 12).

The use of patterns to guide construction is common in many areas. It requires that existing solutions are analyzed to identify patterns, then report those appear worth repeating for future reference. In software engineering, this practice has been popularized beginning in the late 1980s (Beck and Cunningham, 1987), achieving widespread popularity in the mid 1990s.

An early, and often cited, influence on software design patterns has been work on patterns for building architecture by Alexander, Ishikawa, and Silverstein (1967, 1977). According to these authors, patterns describe

> ...an issue that frequently recurs in our environment and explains the core of the solution for that problem, so that the solution can be applied repeatedly, without ever replicating it exactly.

At the time of writing, about 15 years later, many books and papers on patterns have been published, with considerable breadth and depth. Apart from the original focus on object-oriented software of Gamma and colleagues, software architecture has also been an important area of pattern work, with the "Pattern-Oriented Software Architecture" series now at five volumes (Buschmann et al., 1996; Schmidt et al., 2000; Kirchner and Jain, 2004; Buschmann et al., 2007a,b).

This type of work contrasts with software architecture in robotics, where the focus often seems to be on the the *differences* between approaches, despite (or possibly because of) the prevalence of two common themes, namely behavioral and layered/hybrid architectures. While this is perfectly understandable, as authors concentrate on the things the general approach could not solve, it is the present author's opinion that the pattern reporting approach is complementary and may provide a unifying view. Therefore, in the following, a pattern-based approach for a common architectural principle will be adopted.

4.1. Overview

The pattern has been distilled based on experience from a number of case studies, carried out as part of this thesis and described in chapters 5 and 6. Additionally, it incorporates a review of the relevant literature, both to establish known uses and to encompass alternative implementations.

There is not a single format for describing patterns, but all of them are are usually variations and refinements of Alexanders mantra "a solution for a problem in context". For example, Buschmann et al. (1996, section 1.2) proposes a refined context-problem-solution structure for the "Pattern-Oriented Software Architecture" series. The context for the pattern has already been described in previous chapters, so here, the structure is slightly modified to start with the problem, but otherwise follows their suggestions.

4.1.1. An Example Problem for Motivation

As a short example, consider an autonomous robot that assists a human by handing over a cup from the table. This requires a fair bit of manipulation (perceiving the goal, moving a manipulator, grasping) and Human-Robot-Interaction (specifying the goal, maybe aborting the action, reporting on robot activity). Much has been reported in the literature on how to come up with a plan of action for performing such a task, and then executing it (cf. section 3.1), but the latter usually involves putting together a number of more basic activities.

Of interest here is the complexity of this "putting together", particularly the communication necessary. Using basic activities requires deciding when (and if) to start something, whether progress is being made, and determining if it is successful or failed. This, in turn, requires knowledge about what is being done, and often also how. For example, whether the cup has been grasped depends on stable contact of the effector (what), which could be determined by haptics (how). For arm motion, success may be tracked by measuring joint angles (different how).

This diversity is a fact of (robotics) life, but it becomes a problem when such information is needed throughout the system, e.g. to provide feedback on the robot's actions, handle a variety of errors in a general manner, or, in general, for all high-level control. In particular, system integration and team-work are both less efficient in the face of growing diversity.

Therefore, the *primary goal* is to provide detailed information about ongoing activities without requiring detailed knowledge of component implementation throughout the system. In other words, we are interested in a common abstraction that achieves polymorphism: A shared interface that handles functionally different components in a unified manner. This interface is the task-state pattern.

A unified interface improves decoupling, because it allows applications to be developed against the common parts only. However, a potential problem is that it can reduce flexibility, because components must not deviate from the proscribed procedure. Therefore, *secondary goals* are, firstly, separation of concerns, to enable combining task-specific and task-independent aspects cleanly, and secondly, evolvability, in particular to allow changing the common task-state machine in a defined and maintainable manner.

4.1.2. The Solution in a Nutshell

Terminology

A **task** is an identifiable piece of work that fulfills a **goal**. In robotics, a typical task would be moving an actuator to a certain position. In distributed systems, the task goal is typically defined by one component, called **task client**, and executed by another, called **task server**. To correctly function, the client must send the goal to the server, and the server must report its progress using **task notifications**, which contain information about the **execution state** of the task. For the aforementioned actuator movement, the state could be expressed by a joint position measurement. The means for exchanging goal and state notifications is subsumed under the term **event bus**.

Definition

The "Abstract Task-State pattern"[1] structures communication between task client and server, adding and using an abstract state representation common to all tasks in the system. It has two parts: An extended task representation, and a communication interface with both synchronous and asynchronous capabilities.

The task representation consists of a pair of **abstract state** (new) and **task-specific representation** (as usual). The abstract state refers to a state-machine common to all tasks that the system can carry out, irrespective of their functional differences. Typical states include "initiated", "started", and "completed" or equivalents thereof. The resulting state-machine is called the **task life-cycle**. Compare figure 4.1 for an overview of the concepts introduced so far.

The communications interface is based on asynchronous notifications[2], so that it can deliver multiple notifications about state changes. To associate the various notifications to the task, it includes a (de-)multiplexer. A central advantage of the pattern is that, through its knowledge of the task's life-cycle, it can generically resolve some of the difficulties commonly associated with asynchronous updates. Furthermore, the client interface described by the pattern includes methods to wait for state transitions, thus offering a synchronous interface, where that is desired.

[1]Hereafter also often called Task-State Pattern or ATS pattern.

[2]Use of multiple synchronous calls is possible, and there is at least one implementation of the pattern doing so. However, for distributed components, asynchronous notification matches the decoupled nature of execution better, and thus is considered the typical implementation strategy.

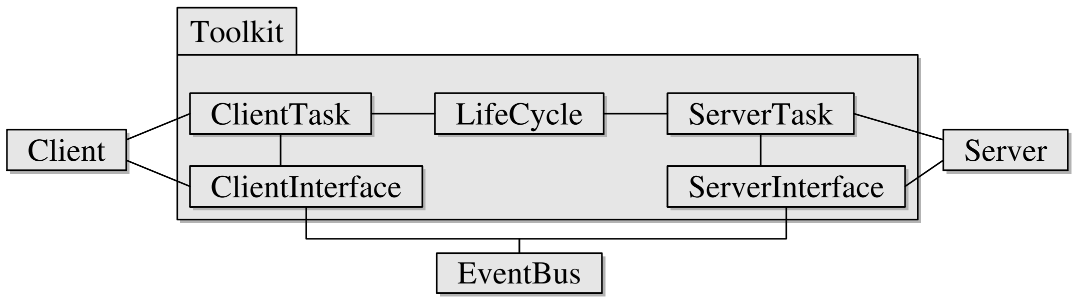


Figure 4.1.: Components involved in a task.

4.2. User's View

Before explaining how toolkits realizing the pattern are implemented, this section will shortly introduce its use by component developers. Besides guidance for these developers, we hope that this will also make them aware of the advantages of using the pattern, and ask for it in frameworks that don't yet offer it[3].

4.2.1. An example life-cycle

Component developers should be aware of the task life-cycle used in their system, so that they know which states and transitions are possible. A simple life-cycle is shown in figure 4.2: It has only two real states: An "initiated" state, which is held when the client requested it, but the server has not yet acknowledged, and a "running" state, when the server is executing the task. The end-states are also relevant, one for successful ("done"), and one for unsuccessful execution ("canceled"). Note that unsuccessful execution can also be caused by the server rejecting task execution, e.g. when it is busy with something else.

There are essentially two ways for interacting with the life-cycle: Inspecting the task-object and registering for event notifications. The latter is most appropriate if the delay between a state-change and the reaction to it should be short. We expect that this will be usually true for servers.

4.2.2. Communication

So far, in component-based robotics, client and servers exchange task-specific messages that encode the goal to be performed, and the current state of actions. For example, such data could be a target motor position, respectively the currently achieved position. To this, using the task-state pattern adds task-independent state-information, such as whether the target motion is acceptable, currently executing, or

[3]Currently available open-source toolkits are *i)* ROS Actionlib, `http://www.ros.org/wiki/actionlib`, and *ii)* the XCF Task Toolkit, `http://opensource.cit-ec.de/projects/xtt`

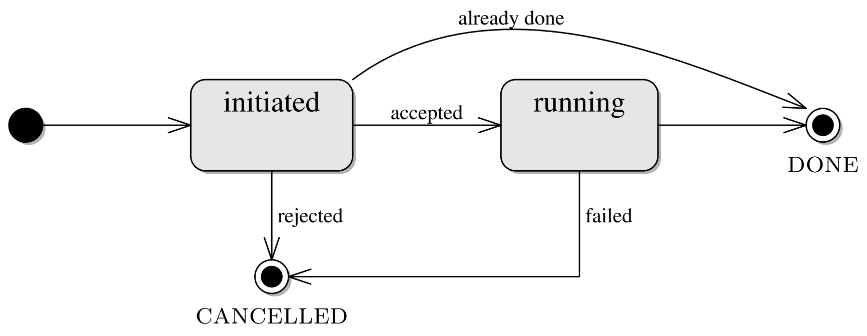


Figure 4.2.: A basic life-cycle example. The two terminal states could be fused, but then users would be required to keep track of the event which lead to it.

finished. Managing this information is usually the responsibility of a dedicated task toolkit. This toolkit takes the place of the communication layer (cf. section 4.3.1).

Interaction between client and server then entails in the client sending goals together with a task-independent transition event (e.g. [“initiate”, neck_pan=50°]). The server responds with a decision on execution or not (e.g. “accepted” or “rejected”). Both client and server may update the task-specific representation. Examples of such updates would be when the server reports intermediate positions, or when the client decides based on some external input that the motion so far is already sufficient. The client can also decide sufficiency for tasks that are not intrinsically bounded (e.g. moving a mobile base forward at a defined speed).

Because client and server are running independently of each other, situations may occur where both of them attempt to change the task at the same time. It is the toolkit's responsibility to resolve such cases (cf. section 4.3.5).

4.2.3. Using the Pattern as a Client

When the result of the task should be used immediately, the procedure is roughly as follows: i) Submit the task to the client interface, which returns a task object. ii) wait either for finish, or for the next transition. iii) check whether the task has been successful, and react accordingly.

Compared to message-passing, this procedure will provide explicit feedback on whether the task request has been executed or not. Compared to remote-procedure calls, it can provide explicit feedback both at the beginning and at the end of task execution, by waiting for the next transition in step 2, instead of just for completion.

Furthermore, more detailed life-cycles, such as the one in figure 4.10 (page 59), can provide information about, and interaction with tasks in more versatile ways, while keeping the same coherent interface. Updating goals and canceling tasks are two often-used functions a more general life-cycle can provide in a consistent manner.

Last, but not least, instead of waiting, clients may also ask to be notified of state-changes whenever they occur. This facilitates managing several tasks at the same time, and can also be convenient when only some transitions necessitate a reaction (e.g. failures).

4.2.4. Using the Pattern as a Server

Creating a server for using task-state based reporting is fairly straightforward: Primarily, a server must map from its internal view of the task's progress to the abstract state-machine, and report the transitions. Depending on the life-cycle, this may either be together with its usual messages, or in addition to them. For example, when it starts acting upon a task, the server should publish an appropriate event (in the example life-cycle, this would be "accepted"). The same applies when the task is complete.

When advanced life-cycles are used, the server implementors face more work, e.g. to support updating and cancellation. Fortunately, because of the general state-machine, the toolkits can support default implementations (rejecting the requests, or using cancel-then-restart for updates). In this way, server implementors can start using a basic implementation, then transparently add more capabilities.

Readers familiar with other coordination approaches might have noted that placing some of the mapping into the server is a distinctive feature of the proposed pattern. In fact, earlier realizations of the pattern often didn't do this, yet (cf. section 4.4). Only the one presented in this thesis (XTT) and the ROS actionlib Marder-Eppstein and Pradeep (2010), work this way. As these have been independently developed, this refinment is still considered part of the pattern. The advantage here is that, i) the life-cycle provides a standardized semantics for communication between client and server, and ii) generic observation (which only looks at the abstract states) becomes possible easily, because the mapping is performed in a single place, before messages are sent out.

4.3. Implementing the Pattern in a Toolkit

Having discussed what the pattern is good for, and how it can be used, we will now provide guidelines for implementation. This will be of most interest for toolkit developers, but also provides a general overview about the boundary between the toolkit and other components.

Patterns are rarely implemented in exactly the same way, so we will first describe the most typical implementation of the pattern, and then discuss alternatives. Furthermore, common pitfalls and appropriate remedies will be discussed.

4.3.1. Premises and Design Influences

We consider implementations for distributed systems, assume that task execution is fundamentally asynchronous, that multiple tasks may occur in parallel, and that each of them can have multiple state changes and/or results. Therefore, we assume that communication is using *asynchronous event notifications*.

The suggested implementation of the pattern is as a *toolkit layer*, i.e. a library for re-use in all components of a system. Apart from re-use, we advocate this to ensure safety and robustness: Our experience is that communication between asynchronously running components has many edge cases which are often not covered until testing reveals them. A toolkit can combine experience from many systems and cases, to provide a safe and robust behavior. Using it in a layered structure, as shown in figure 4.3, prevents the user's from bypassing it.

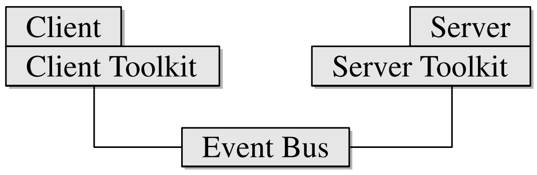


Figure 4.3.: General structure: Distributed with toolkit layers

The second most important design goal is a gradual adoption curve for developers. Developers generally prefer the simplest solution and while the pattern's life-cycle provides evolvability of the overall system, it may, at first, appear more complex than needed. Therefore, the toolkit should afford developers the choice to have their components support only a minimal subset, with opportunities for gradual growth.

4.3.2. Summary of implementation steps

This section is intended as a (short) check-list of implementation steps and a guide to the remainder of this chapter.

1 Specify the life-cycle model

The life-cycle is a finite-state machine that contains all states and their possible transitions for tracking the task's progress. More details on choosing a good state-machine will be given in subsection 4.3.6. For implementing it, we recommend using a configurable model, to aid evolving the life-cycle.

In addition to the finite-state-machine, it should be specified which transitions may be invoked by the client, which by the server, and which by both. This facilitates checking allowable transitions by the toolkit (see next step).

2 Specify client and server task objects

The task objects contain the task representation, and the current instance of the life-cycle. See figure 4.1 and section 4.3.4 for minimal attributes needed.

The task objects should be distinct for client and server, to check that only allowable transitions are invoked.

3 Implement a demultiplexer for notifications

The (de-)multiplexer maps from incoming notifications to the corresponding task object, and transmits outgoing notifications. See section 4.3.5 for its responsibilities. A typical implementation strategy is the asynchronous completion token pattern (ACT) (Schmidt et al., 2000, p.261–284).

4 Implement notification for client and server

Information about the state of ongoing changes should be available as asynchronous notifications. This saves users from having to periodically poll state objects. The Observer pattern is a typical implementation choice Gamma et al. (1994). For servers, observers are also used to initiate new tasks. See section 4.3.4 for the interfaces.

5 Implement a client interface

The client interface is the point where new tasks are submitted by clients (cf. section 4.3.4). It cooperates with the demultiplexer to send notifications out, and also manages what the client observes.

6 Implement an abstract base class for servers

The server processing has a fairly regular structure and is amenable to support through an abstract base class. Specifically, it can map incoming events to dedicated functions, and provide default implementations for optional parts of the life-cycle. See section 4.3.4 for details.

4.3.3. Advantages and Responsibilities

The toolkit's function is to communicate, and keep track of, changes in the state of a task, including start and end, between two or more (distributed) components. This is non-trivial because components are running independently, all communication incurs a transmission delay, and thus two component's view of a task may diverge. It is the toolkit's responsibility to handle these cases in a robust manner.

Conventional middleware is not sufficient here, because it has no concept of the relation between messages. In contrast, the task-state-pattern knows the life-cycle and can use it to recover from errors.

A secondary issue is that a single client/server pair may be engaged in more than just one task at the same time. Therefore, incoming notifications must be associated with a task, and the application side notified of changes as they occur.

Finally, a task toolkit should prevent components from attempting state changes that are not valid according to the life-cycle. If such illegal transitions are still received, either because of bugs or due to network errors, the toolkit must recover from them.

4.3.4. Structure

For the toolkit's structure, we distinguish between objects visible to the user of the toolkit, and internal objects. The user-visible objects are: i) an interface for clients to, firstly, submit new tasks, and, secondly, receive *change events*, which we call the *client service*, ii) a *server interface* to be implemented, and iii) an object to represent, and modify, ongoing *client tasks*, and iv) the same for the server side, a *server task* object. The toolkit-internal objects are a) the life-cycle model, and b) a demultiplexer object that associates incoming messages with their tasks. See figure 4.4 for an overview.

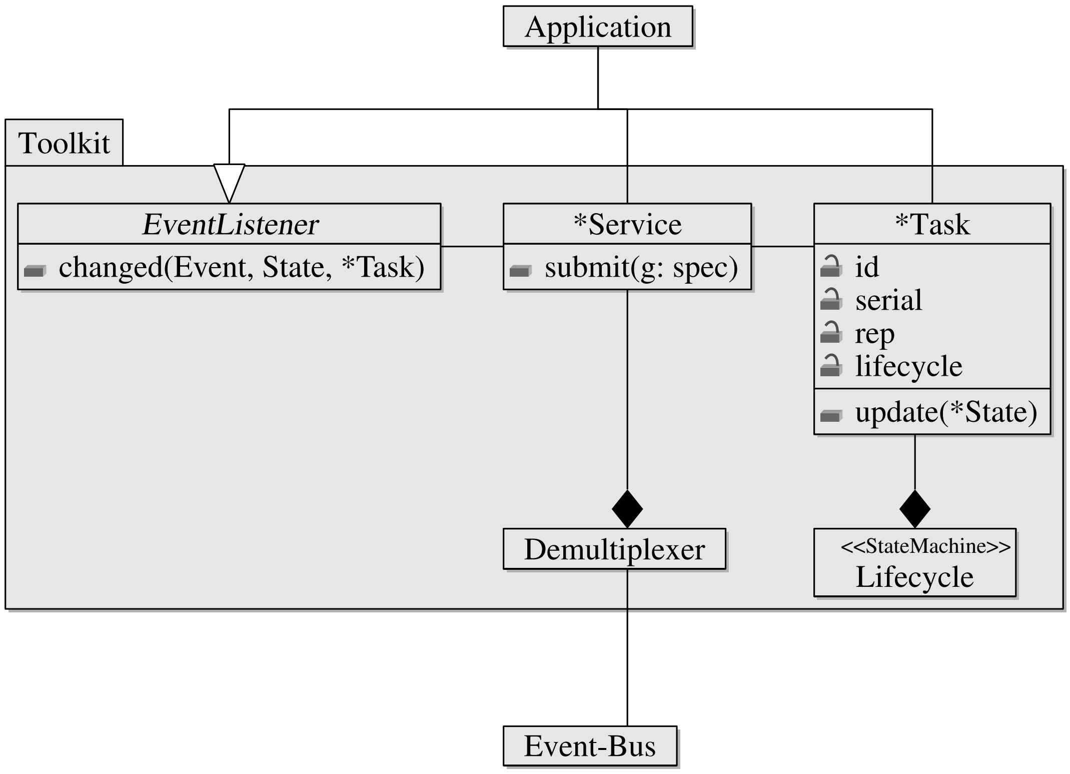


Figure 4.4.: Minimal toolkit structure for clients. The server side is similar, but combines Service and Listener into an abstract base class.

The life-cycle describes all states a task can have, but client and server differ in how they are allowed to modify it. To prevent errors, the task objects restrict possible changes: The client task object accepts only those modifications that a task client may perform, and rejects the others, and vice versa for the server side (cf. figure 4.5).

Task Object and Notification Structure

The attributes of the task object (cf. figure 4.4) are essential for the added functionality that the toolkit provides, and have to be included in all task-related notifications.

id An identifier for the task. Has to be unique at least per server, and ideally globally, to preclude potential confusion.

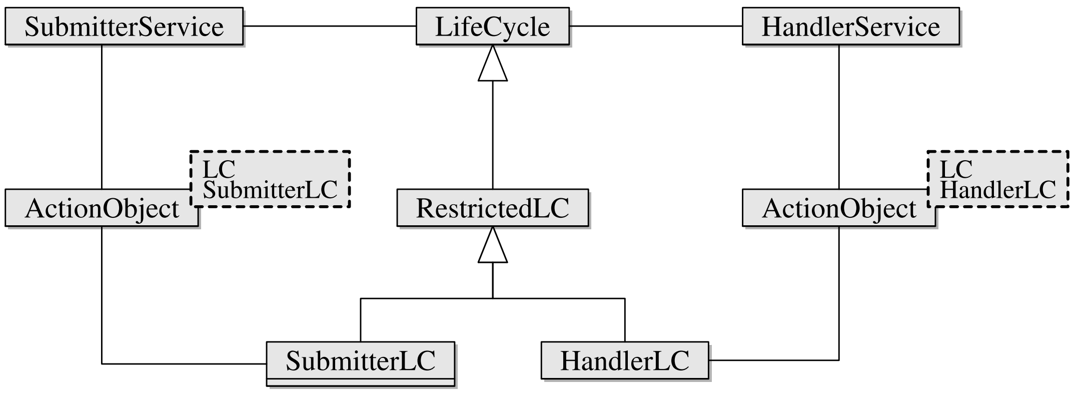


Figure 4.5.: Partitioned life-cycle access

serial A number that is incremented on each update and used to detected missed updates. Also note the *originator* field of task messages, described below.

rep A representation of the task to be performed in detail, including goal, parameters, and the current state (in task-specific terms). This field is specific to the type of the task and is not directly modified by the task toolkit, just updated from notifications.

In addition to these fields, it must be possible to determine which component sent a notification, to check for overlaps (compare section 4.3.5). Usually, this is available from the middleware, but if not, an additional *originator* field must be included in notifications. It specifies whether an update has been sent by the client or by the server.

Designing for Gradual Adoption

When creating a new server implementation, or when moving a simpler server to a more advanced life-cycle, the server implementor has to support additional features. It may be desirable to postpone this effort to a later stage, e.g. for testing, or to rapidly provide basic support before realizing more features. It may also be that a few servers cannot support all parts of the life-cycle, for example, a safety critical task may not allow cancellation.

For such cases, we have found it useful to provide an abstract base class for the server, which provides default functionality for the unsupported cases. See figure 4.6 for an example abstract server interface.

4.3.5. Dynamics

The dynamics of the task toolkit are fairly simple, primarily a pipeline, with the toolkit interjecting between client or server and the middleware. On the sending side, the toolkit checks that changes are valid and rejects them if not. The receiving

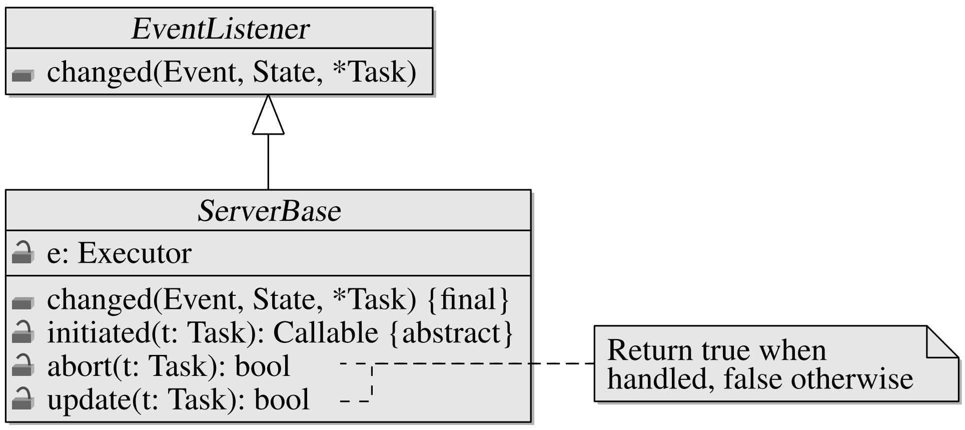

EventListener
changed(Event, State, *Task)
ServerBase
e: Executor
changed(Event, State, *Task) {final}
initiated(t: Task): Callable {abstract}
abort(t: Task): bool
update(t: Task): bool
Return true when handled, false otherwise

Figure 4.6.: Methods for example abstract server base class. The method “initiated” has to be implemented, the others provide defaults and can be overriden as necessary.

side is slightly more complex, because, firstly, it has to associate notifications with tasks, and secondly, it must detect and handle message overlap.

Figure 4.7 shows an example sequence for the sending side. It should be noted that this sequence is less fixed than the structure. While the pattern requires a number of objects, their interaction has more flexibility. For example, we show the client service organizing the various other objects, but a different distribution of responsibilities is certainly possible, e.g. with the client task creating the life-cycle. We found that, in our cases, the sequence shown minimizes associations between the objects, but the importance of that design goal is up to the implementors judgment.

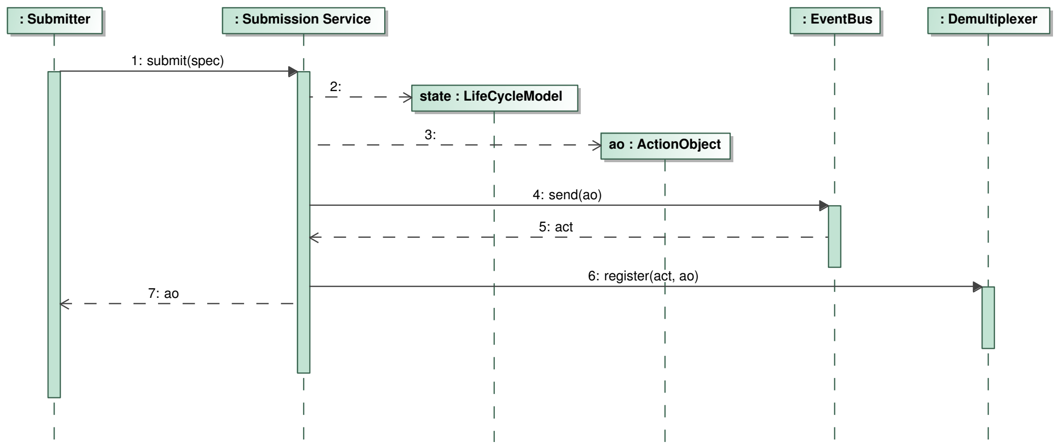


Figure 4.7.: Invocation sequence for submitting a new task.

When receiving a notifications from the event-bus, the process is essentially reversed, as shown in figure 4.8. Notifications that are not found by the demultiplexer – which means they have been created by other clients – are ignored.

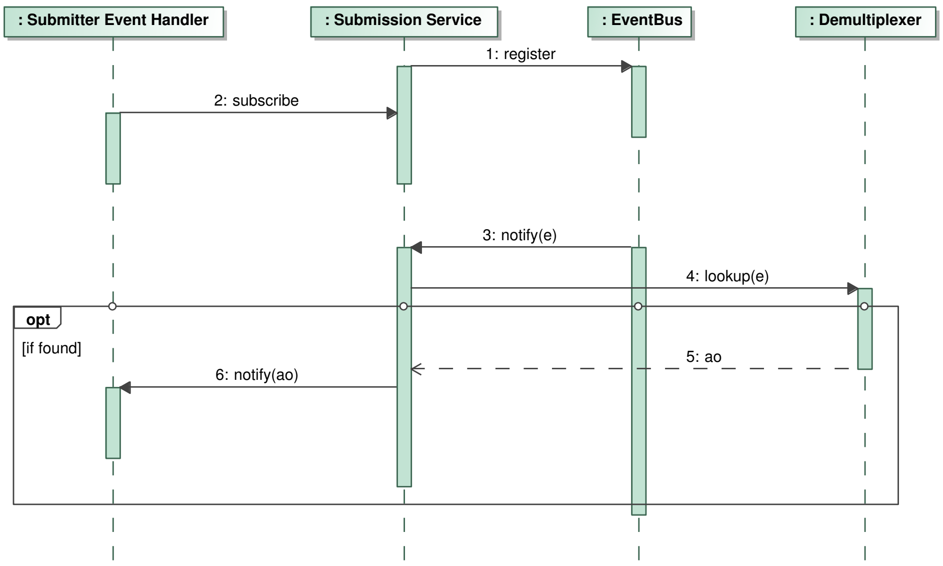


Figure 4.8.: Invocation sequence for notifying listeners.

Detecting and recovering concurrent updates

Not shown so far is the handling of message overlap. Such overlap can occur for several reasons. Firstly, client and server may both attempt to change the state of a task at approximately the same time, but transmission delays cause their changes to overlap. This case is shown in figure 4.9, where, globally, “update 1” is first and “update 2” is second, but locally (at the end points), the notification for “update 1” may arrive after “update 2” has been sent.

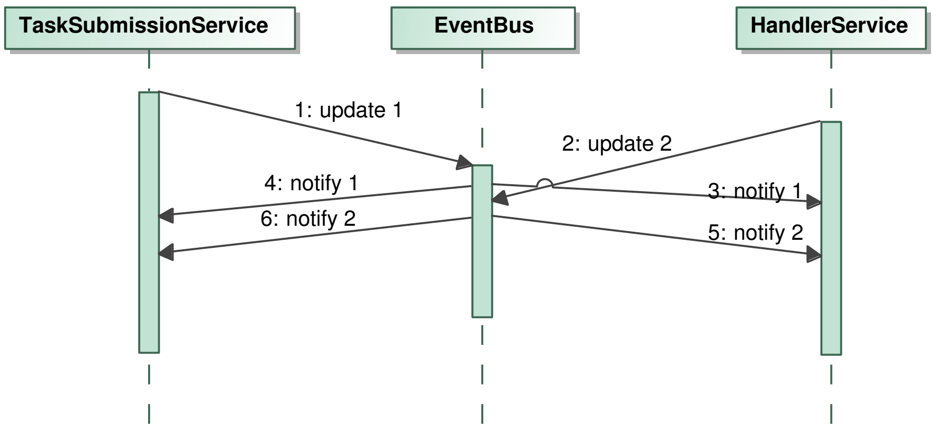


Figure 4.9.: Example of message overlap due to network delay

An easy method to detect such cases is the use of a serial number, as specified for the task objects before. Prior to sending a notification, the serial is always incremented by one. Upon receiving a new notification, its serial is compared to the local serial, with the following three possibilities:

1. If the serial is higher than the local one, the notification is a normal update and processing continues.
2. If the serial is equal to the local one, it means that both parties have sent at the same time. This calls for a recovery procedure (see below).
3. If the serial is lower than the local one, it means that the remote component has missed several notifications by the local component. This is usually a serious error and should only occur in case of network problems. Recovery may be attempted, but unless the communication problems are fixed, may not be successful.

Fortunately, case two is recoverable through checking the life-cycle for allowed sequences. A simple means to do so is to use what we call the "server dominant" approach. In this, firstly, the server always determines the current state, because it is the component executing it. Secondly, if the life-cycle allows it, the server applies the client's update after its own and sends an additional notification to that effect. If the life-cycle does not permit the client's update after the server's, it is ignored.The client always accepts the notifications from the server as the correct state, even when the local state would not ordinarily allow the transition.

A drawback of the "server dominant" approach is that clients must be prepared to receive a notification for unexpected target states. This is a common requirement, but if it is undesirable, alternative approaches may be preferred. See section 4.3.7 for a short discussion.

For case three, we recommend reporting an error. While additional checks may determine that the original problem has ceased and operation could continue, this is not well-placed within the communication infrastructure (which the toolkit is part of). Higher-level mechanisms should be employed to determine the viability of continued operation, and re-start tasks, if possible.

4.3.6. An Example Life-Cycle

The life-cycle directly determines the distinctions that can be made when tracking tasks. Because having a start and an end is central to the task concept, and execution may always fail, we consider the basic life-cycle introduced earlier to be the core of what all life-cycles must support (cf. figure 4.2). However, we expect that most users will want more features.

The necessary distinctions are dependent on the system the toolkit is designed for, but here we will present a life-cycle that has served us well, and that we consider to be a good candidate for most systems. See figure 4.10 for a visualization. In addition to the basics, it also supports updating the goal during execution, aborting tasks, and delivering intermediate results. Its states and transitions have the following meaning:

- INITIATED Initial state for newly published tasks.
  - ACCEPTED Execution is commencing.
  - REJECTED No action will be taken.
- RUNNING Execution is ongoing.
  - UPDATE Goal has changed
  - RESULT_AVAILABLE Intermediate result added
  - COMPLETED Goal reached
  - CANCEL Execution stop requested
  - FAILED Goal could not be reached
- UPDATE REQUESTED Updated goal available
  - ACCEPT Target is new goal.
  - REJECT Previous goal will remain target.
- CANCEL REQUESTED Aborting execution is requested
  - CANCEL FAILED Aborting not possible.
  - ABORTED Execution aborted.
- CANCELED Execution stopped without reaching goal.
- DONE Goal reached, execution stopped.

One design choice has been made in the general life-cycle model: it returns to the running state when an update fails. That means it will continue to act on its previous target. It would also be conceivable to attempt an abort in such a case and the model might need to be enhanced to encompass this. The argument for the current choice is that the client is generally best positioned to decide whether the failure is severe enough to warrant aborting the action.

4.3.7. Alternatives

The implementation guidelines described so far attempt to strike a balance between simplicity and efficiency. However, other implementational choices are certainly possible and this section reports some different choices we have seen so far.

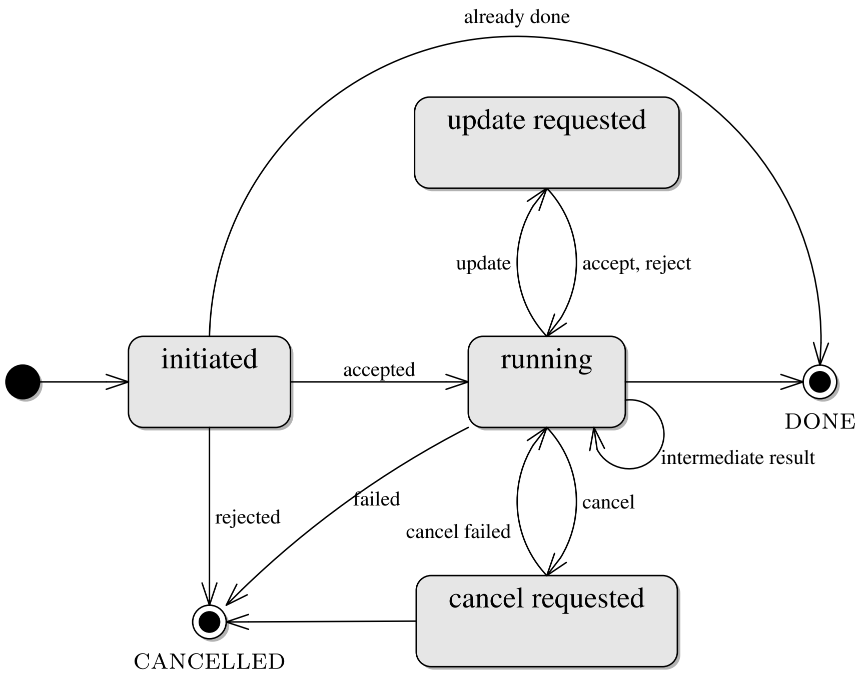


Figure 4.10.: A fairly general life-cycle.

Representing the task

Figure 4.4 has specified a single "rep" attribute that contains the entire current state of the task, both goal specification and results. The tacit assumption behind that is that the entire representation is communicated every time the state changes. Depending on how large the representation becomes, and how often it changes, this may cause communication overhead. Furthermore, it requires that the representation is modified during operation. Again depending on the complexity of doing so, this may be inefficient.

An alternative approach is to *communicate and store only updates.* This is appropriate when the representation is updated much more often than it is read. The drawback of this choice is that it makes reconstructing the current state more difficult, and it also makes observation harder, because an observer has to follow all messages to reconstruct the state.

Conflict detection and resolution

As mentioned in section 4.3.5, the "server dominant" model for overlap resolution requires that client must be able to accept whatever state the server reports, even if that state would be unreachable from the last state they have requested. This is not normally an issue, but may be considered confusing.

Furthermore, the "server dominant" model treats all overlaps in the same manner. Users may want to have more say in treating such cases, to make case-by-case decisions. While this somewhat contradicts the value of the pattern to generalize handling, we would not want to rule out the necessity entirely.

Therefore, we'll shortly present two alternatives to conflict resolution that we have observed in implementations.

- *Central broker.* Instead of communicating directly between client and server, all communication may be enforced to pass through a central broker that resolves conflicts. From a communication perspective this introduces a single point of failure. Furthermore, a central conflict resolution strategy may be hard to specify. Nevertheless, such an approach can provide predictability.
- *Delayed Transition Evaluation.* All transitions are executed with a fixed delay that is larger than the maximum possible communication delay. This strategy may be used when real-time transports are available that guarantee a known maximum transmission time. This is not recommended for best-effort communication protocols, since they could always exceed the expected maximum.

4.4. History & Known Uses

Historically, the use of life-cycle based coordination appears to have been developed as an abstraction on earlier, fine-grained but action-specific coordination mechanisms. Early approaches introduced use of life-cycle based coordination primarily to detect and handle errors in a generalized way, with the actions themselves coordinated on a finer level (Lefebvre and Saridis, 1992; Simmons, 1994; Wrede et al., 2006). In at least two published cases, an earlier mechanism for fine-grained coordination was succeeded by abstract life-cycle based coordination (Simmons and Apfelbaum, 1998; Hanheide and Sagerer, 2008).

4.4.1. Intelligent Machine Architecture (IMA) (Lefebvre and Saridis, 1992)

The so-called "intelligent machine architecture" coordinates low-level tasks, such as blob detection and robot limb motion, using Petri-Nets. The states in these petri-nets are action-specific, but have obvious life-cycle relations. For example, the blob-detection action has states "start_find_spot" and "end_find_spot". Similarly, the visual controller has states "VC_ready", "VC_start", "VC_done" and so on (Lefebvre and Saridis, 1992, figure 5).

The petri-net structure of this approach is also an example of a hierarchical model. The higher levels (such as the visual controller) contain conflict-resolution states ("ready", "available", etc.), whereas the lower levels do not have such states and just use start and end states. Error handling is similarly organized in a 3-level hierarchy:

> Errors are handled first by the Coordinator, and are passed up to the Dispatcher only when a local strategy is not adequate to resolve the con-

> dition. In some instances the Dispatcher must turn over error resolution to the Organization Level where operations may be replanned (Lefebvre and Saridis, 1992, section 5, paragraph 4).

This architecture, however, does not exploit the similarities in these models and thus its coordination mechanism is not action-independent. Additionally, the coordination hierarchy, while explicitly designed, is not modeled but implicitly contained in the petri-net structure.

Information on the implementation of the IMA is scarce, but it is mentioned that message-passing is used for communication, which has very similar semantics to an event-bus, in particular, it also uses asynchronous communication. The main difference is that messages are addressed to a known recipient, instead of being selected from the event-cloud by the handler, as described for the task-state pattern.

4.4.2. Task Control Architecture (TCA) (Simmons, 1994)

Similarly to the IMA, the Task Control Architecture uses fine-grained models encoded as so-called “task-trees”. This architecture does not yet exhibit abstract states, but already utilizes an external execution monitor, described as follows:

> In particular, monitors and exception handlers can be added without modifying existing components. [...] a module can associate an exception handler with a class of messages handled by another module, so that whenever a message is issued and added to the task tree, the exception handler is automatically added to that node (Simmons, 1994, section IV, paragraph 2).

Notably, in the above quote, “classes of messages” are mentioned, which suggests a relatively close dependence of exception handlers on the monitored tasks. Not long after, Simmons introduces a more abstract mechanism, in the form of the Task-Description-Language approach (see below).

TCA uses the IPC messaging middleware (Simmons and James, 2001), which supports both publish/subscribe as well as client-server models. It is not specified which of the two is used in the TCA.

4.4.3. Task Description Language (TDL) (Simmons and Apfelbaum, 1998)

The successor to the previously described TCA is one of the earliest examples of abstract, life-cycle model based coordination. In this architecture, all tasks are coordinated using the abstract states “disabled”, “enabled”, “active” and “completed” (Simmons and Apfelbaum, 1998, p. 1933). It also introduces the concept of an “expansion”, which describes the state of an entire task sub-tree using the same set of states [ibid].

The TDL keeps the approach of an external exception handler, but also introduces abstract states called “succeeded” and “failed” to attach it to tasks, thus again moving towards an abstract life-cycle model.

TDL, as its name implies, is a language compiled to C++ code. The resulting code uses the the Task Control Management (TCM) library as the service-level API. Like TCA before it, TCM can use the IPC message-passing middleware. Additionally, it supports Real-Time Inc's DDS[4] publish/subscribe middleware, which suggests that the TCM itself is also based on a publish/subscribe model.

4.4.4. DESIRE Architecture (Plöger et al., 2008)

A non-event based implementation of life-cycle based coordination is exemplified by the DESIRE service robotics architecture. Its life-cycle model consist out of “Running”, “Finished”, “CommandAbort” and “CommandFatalError” states, with the last state signifying an unrecoverable error. Errors are further distinguished into “Abort”, “ContractViolation” (indicating an erronous activity specification) and “FatalError” (indicating an internal error).

While the DESIRE life-cycle model does contain a state indicating an unacceptable action specification, it is still primarily a model that can only support a centrally coordinated system, because no request states are modeled. Correspondingly, the DESIRE system includes a planning component, which is integrated at the service-level.

Instead of event-based communication, the DESIRE architecture uses the RPC interaction style of a CORBA-based object-oriented middleware, but communication between requesting and handling components is mediated using the so-called “Ablaufsteuerung” (execution control) component, which also realizes the service-level. The execution control thus realizes asynchronous communication between submitter and handler.

4.4.5. Active Memory Architecture (AMA) (Wrede et al., 2006; Hanheide and Sagerer, 2008)

A coordination approach first using Petri-Nets and later on a finite-state-model has been built on the Active Memory Architecture. The architecture was originally developed for a cognitive vision system (Wrede et al., 2006; Wachsmuth et al., 2007) and later transferred to human-robot-interaction (Hanheide and Sagerer, 2008). The “Active Memory”, which gives the approach its name, is (technically) an active database coupled with an event-bus.

The earlier, Petri-Net based coordination realized a fine-grained execution monitor based on temporal monitoring of the actions and enabled, respectively disabled, recovery actions if errors were detected. It did not explicitly model the action life-cycle, because actions were continuously repeated as part of the vision-processing

[4]http://www.rti.com/products/dds/

cycle anyway. In this approach, the petri-net controller was realized as a dedicate coordination component.

The later, finite-state approach added an explicit request-acknowledgement life-cycle for action monitoring. Actions in this context were fairly coarse-grained, e.g. "acquire a map of the room". Depending on the nature of the sensor used, these larger activities could either map to a single action (e.g., analyzing a single view from a 360° degree camera) or multiple sub-actions (e.g., activation of a 180° laser scanner plus rotation of the platform to acquire a 360° view). To accommodate both types of handlers without changes to the clients, a model with several optional states was used, similarly to the one in figure 4.10. In this approach, coordination was decentralized and co-located with the participating components.

4.4.6. XCF Task Toolkit (XTT) (Lütkebohle et al., 2009a)

The XCF Task Toolkit (XTT) is a service-level coordination implementation, initially based on the life-cycle-model of the active memory architecture approach. It has been developed as part of this thesis, in order to experiment with the effect of various model-level, interface- and implementation-level changes. Detailed technical and case-study information from these experiments is found in chapter 5.

XTT uses a layered architecture which embeds the service-level into participating components, but hides the middleware communication from the application-level and also enforces adherence to the life-cycle model. Conflict detection uses serials and recovery employs the dominant handler approach. On the API level, XTT is somewhat unique in that it offers both the common Observer interfaces, as well as synchronous adapters, with a submitter-side Future and a handler-side Callable interface (Sun, 2009a).

As the name suggests, XTT is built on the event-based middleware XCF, in particular, the Active Memory-based event-bus (Wrede et al., 2006). It hides the communication mechanism, but exposes the data-transport classes of that toolkit, in conjunction with an application-level conversion registry.

4.4.7. ROS ActionLib (Marder-Eppstein and Pradeep, 2010)

The "actionlib" is a recently developed (2009) C++ toolkit for action execution based on the "Robot Operating System"[5], an Open Source initiative that aims to build a complete infrastructure for mobile robot systems. Similar to the XTT, it comprises a service-level toolkit for task state coordination.

A notably different aspect of the actionlib is that it maintains two separate state machines, one for the server and one for the client. The server machine is considered to be the reference, with the client tracking it. In term of the task-state pattern, this represents use of the "dominant handler" recovery strategy directly within the client state machine. If this recovery strategy is chosen, it is likely the most elegant way

[5]`http://www.ros.org/`

of implementing it. The resulting client state machine is not exactly simple, though (cf. figure 4.11), but still reasonable given its functionality.

Similarly to XTT, actionlib is layered on top of a middleware that supports status notification – therefore, changes of ongoing tasks may be observed by other components in the system at any time. Also similarly to XTT, the action server supports both callback-based and queue-based invocation of actions.

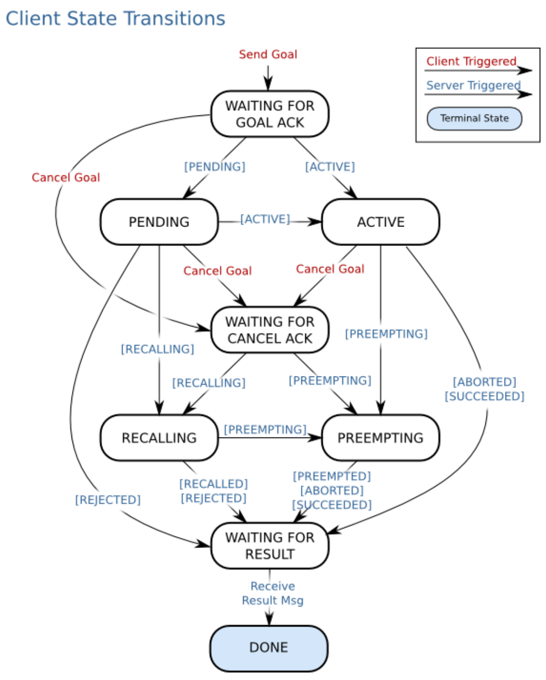


Figure 4.11.: actionlib client state machine

The actionlib is generally very full-featured, with every option of the generic life-cycle (cf. figure 4.10). The terminology is slightly different, with the most notable difference in naming for cancellation: Cancellation before action execution is called "recall", whereas cancellation after is called "preemption".

A further interesting aspect is the actionlib's cancellation handling, which supports both temporal and id-based references. Thus, there is built-in support for cancellation of all ongoing actions started before some time, as well as cancellation based on goal ids.

4.4.8. Plan Execution Interchange Language (PLEXIL) (Baskaran et al., 2007)

PLEXIL is somewhat unusual, in that it is not a coordination system as such (though a reference implementation, called "Universal Executive" is available), but a proposal for a standard interchange format, based on the concept of a tree containing processing nodes. It relates to coordination in that coordination systems are needed to execute the specified plan, and in that it contain a common, abstract life-cycle for all nodes in the plan tree.

The life-cycle itself has the states `Inactive`, `Waiting`, `Executing`, `Finishing`, `Iteration_Ended`, `Failing`, and `Finished`. As can be seen from these states already, an emphasis is placed in preparation of a task, to ensure synchronized starts. It also contains an intermediate feedback facility, the `Iteration_Ended` state.

PEXIL's life-cycle is slightly different from the approaches presented so far, in that it distinguishes possible transitions by node type. The node types themselves are categorized according to the kind of action performed, and between leaf and parent nodes, mostly to be able to reason about what kind of changes may occur when executing a PLEXIL plan. In essence, though, these distinctions are not particularly different, they just make the life-cycle's cleaner to present.

A bigger difference is that several of the transitions reference changes to *other* nodes in the tree, e.g. `Ancestor_end_condition` or `All_children_waiting_or-_finished`. These are essentially events on the coordination level, not on the task level, but PLEXIL's life-cycle has provisions for automatically propagating these coordination-level events to changes in a node's state.

In general, PLEXIL embodies a development of the centralized coordination model: It contains a rich language for coordination, based on a variety of conditions. It does not, however, explicitly specify the communication interface between the coordination engine and the services. Moreover, its conditions (at least the examples given) rely heavily on being able to inspect various component-specific attributes. Thus, while it would not be incompatible with using the pattern as described here, it suggests a different model, one where mapping from task-specific states to abstract states occurs in a central component, not in the servers.

4.5. Consequences

Coordinating systems using the task-state pattern is expected to offer the following **benefits**:

- *Separation of concerns.* The abstract life-cycle model separates the state of action execution from the details of the action to be carried out. The pattern further hides communication details from the application level, but offers customizability of model and error handling. Thus, applications need not deal with complex middleware interfaces or optimizing communication, but can concentrate on action execution.

- *Transparency and Extensibility.* The current state of action execution is exposed in the system in a well-defined, re-usable way. This opens avenues for a variety of service components, e.g. for error handling, restarting of failed activities, recovery from component failure, etc.

- *Correctness.* By dividing the distributed state-keeping mechanics, which are hard to implement correctly in the face of unreliable communication, from the application-level concerns, the overall likelihood of correctness is increased.

  Additionally, conflict detection and recovery have clear interfaces, which makes them standard instead of optional. Model constraints are further checked locally, which exposes application errors early on.

- *Encapsulation, modularity and re-usability* are increased by offering an action-independent, high-level coordination mechanism that is applicable to all actions in a system. Instead of having to implement dedicated coordination models for every possible action, one common model can be applied.

- *Incremental development.* Life-cycles with optional transitions, coupled with event-based coordination facilitate incremental system development. For example, if an optional preparation state is included in the model, but components do not initially support it, they can simply register for the following start event. If later on support for preparation is added, the submitter component can stay the same and functionality is transparently increased.

- *Bridging of different abstraction levels.* Integrating different abstraction levels, e.g. between motor control and planning, to give just one example, are a consistent problem in system building. The life-cycle model helps here by offering a means for high-level coordination of low-level components. In particular, models such as the generalized life-cycle (cf. figure 4.10) support updating targets without restarting an action, which can provide an abstract interface for highly-coupled components such as visual servoing.

Using the pattern may also incur some **liabilities**:

- *Mismatch to action life-time.* For very short-lived activities, the communication overhead of multiple state change notification messages may be unwarranted. In particular, if event-notifications are sent over standard TCP channels, latencies can add up quickly due to ack-waiting. This is potentially the case, for example, for CORBA Notification Service implementations, due to CORBA's RPC underpinnings (COR, 2004).

- *Overcomplicated life-cycle* The attempt to provide a single life-cycle that encompasses all possibilities may result in an automaton that is much too complicated for most components. In this case, if a more suitable life-cycle cannot be found through a redesign, other coordination methods should be explored.

- *Tendency to discretize* The life-cycle model, with its explicit states, may cause a discretization of activities which are more naturally described using continuous values. While obvious cases of this, such as dynamical systems, are not likely to use the task-state pattern, other cases may not be so clear cut and could be biased on this direction because of the use of the pattern.

4.6. Summary

This chapter has presented the Abstract Task-State Pattern, as a generalizable, extensible communication interface for robot tasks. Based on an analysis of the relevant literature, the pattern has been identified and generalized. Furthermore, an extension to place mapping from task-specific states to the abstract state in the server has been proposed and, in contrast to earlier implementations, it is suggested to realize the pattern in a client-server toolkit, similar to communication middleware. Detailed designs for the recurring parts of such a toolkit have been presented.

In the following chapters, the general utility of the pattern, including the extensions, will be empirically studied based on a number of case studies.

5. Separation of Concerns in Life-Cycle Coordination

This chapter describes the origins of the task-state pattern and investigates its use, and the design and implementation of the supporting toolkit, in a real, integrated system. First, an example system from the COGNIRON robotics project will be discussed, where the pattern was already identifiable, but implemented in an ad-hoc manner (cf. 5.1.1). An analysis of implementation effort indicates that a dedicated, re-usable toolkit implementing the pattern may improve design quality and reduce errors.

In the second part (section 5.2), the toolkit design is presented, with a focus on two different APIs offered to service implementors: One that completely hides the life-cycle, thus preventing protocol violations at the cost of a more restricted execution, and another that exposes all events directly, for more flexibility.

Lastly, the toolkit's architectural benefits are evaluated on a comparative study of a text-to-speech component, demonstrating reduced coupling and better cohesion.

5.1. Pattern Identification and Analysis of Historical Use

Interfaces similar to the task-state pattern have probably been developed independently at many groups all over the world, to varying degrees of formalization. Here at Bielefeld, the first incarnation came about from the interaction mechanism of the dialog manager and the navigation components, developed for the "Home-Tour" scenario of the EU-Project "COGNIRON" (Booij et al., 2008; Hanheide and Sagerer, 2008). The pattern was later encountered in the DESIRE architecture, which is based on Simmon's TCA approach (Plöger et al., 2008; Simmons, 1994). The second system triggered identification as a pattern, but the present work is based on the first.

5.1.1. Example: The COGNIRON Home-Tour

The COGNIRON "Home-Tour" is a Human-Robot-Interaction scenario where the human tutor shows a mobile robot around his or her apartment. It is characterized by an alternation between the robot following the human around and the robot acquiring a unique signature of a room that allows it to recognize the room afterwards (mapping). The change between these actions is initiated by the human, and in one instance the robot signals completion (for mapping), whereas in the other, the human signals it (for following). Additionally, the robot may encounter a failure during execution (for example, getting stuck in a door), or reject a command when it is momentarily unable to comply (for example, a follow request when mapping is ongoing).

The goal of the pattern in this context is to harmonize interaction between the various action-performing components and the dialog manager, which relayed the humans requests. Ideally, the dialog manager should have the same high-level protocol to all other components, with only the specifics of the action request changing.

However, good HRI performance requires a fairly fine-grained interaction between dialog and component, for example, to notify the human when a longer-running action is ongoing, but omit such notification, when the action is short. Amongst more technical reasons, this was decisive in precluding the use of a synchronous remote procedure call. The intertwined interaction scenario is shown in figure 5.1.

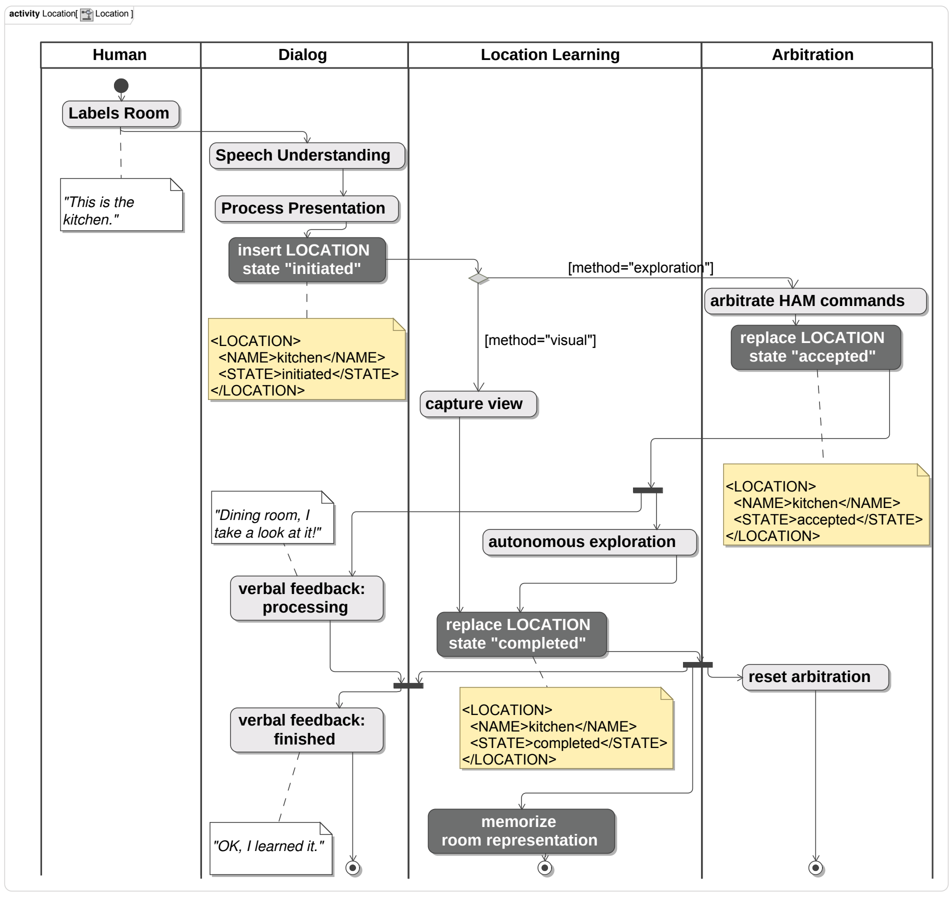

activity Location[Location]
Human
Dialog
Location Learning
Arbitration
Labels Room
"This is the kitchen."
Speech Understanding
Process Presentation
insert LOCATION state "initiated"
<LOCATION> <NAME>kitchen</NAME> <STATE>initiated</STATE> </LOCATION>
[method="exploration"]
[method="visual"]
arbitrate HAM commands
replace LOCATION state "accepted"
<LOCATION> <NAME>kitchen</NAME> <STATE>accepted</STATE> </LOCATION>
capture view
"Dining room, I take a look at it!"
autonomous exploration
verbal feedback: processing
replace LOCATION state "completed"
reset arbitration
<LOCATION> <NAME>kitchen</NAME> <STATE>completed</STATE> </LOCATION>
verbal feedback: finished
"OK, I learned it."
memorize room representation

Figure 5.1.: COGNIRON Interaction Diagram (from Hanheide and Sagerer, 2008, figure 5).

Communications in the scenario is based on event-notification using the XCF middleware (Wrede et al., 2006), where messages are encoded as XML. Because of the use of XML, it was easy to insert a “STATUS” tag just prior to sending regardless of the format of the action specification. While no explicit life-cycle is presented, from figure 5.1, the automaton in figure 5.2 can be derived.

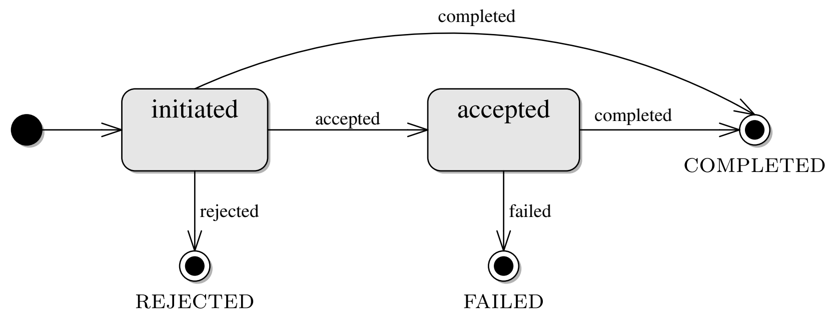


Figure 5.2.: State Machine for the Life-Cycle used in the COGNIRON system

As used in the system, the model represents a mixture of request-acknowledgement and central control: On initiation, a request can be rejected. However, for aborting a task, the state was set to "completed" and this could not be refused (Peltason, 2008).

Another aspect of that life-cycle is that it had two different failure states: "rejected" and "failed". This is because the state-name is always equal to the transition and the condition is based just on that. Therefore, it cannot distinguish logically equivalent states which have different causes. Instead, both causes have to be handled individually. With just one instance of this, it probably was not considered a major nuisance, but for extended life-cycles, it can become more of a hassle and generally reduces abstraction.

5.1.2. Verdict: Successful, but onerous to get right

The use of the state-based coordination in the home-tour scenario is generally considered a success by all involved, firstly because of the abstraction it afforded the dialog manager and secondly, because it facilitated monitoring of the current state of an interaction (Peltason, 2008). This was also evidenced by the fact that, while originally designed only for the "following" activity, it was re-used basically unchanged for mapping, too.

However, at this time, all action-handling components had to implement their side of the protocol over and over again. Additionally, on the dialog manager side, handling of the action specification (which is different between tasks) has been intertwined with protocol management, thus preventing re-use, even though the potential is obvious.

5.1.3. Analysis of a typical component: "Following"

To get an idea of the consequences of the situation, let us take a slightly closer look at one of the components involved. While this is only a cursory analysis, it reflects the considerations that have led to the implementation of a dedicated toolkit.

One component in the above system is the "following" behavior, which, in its original form, consists out of 1,593 source lines of C++ code (SLOC). Of that, the layer for state-management amounts to 241 SLOC, or 15.1% [1]. While this percentage is not insignificant, it could hardly be considered prohibitive.

However, firstly, this size does not reflect the significant human effort which went into defining the model, and communicating its exact mechanics and semantics to all developers. This is likely the far costlier effort.

Secondly, it is telling how often the code in question is changed. In this particular component, the state-management code was factored out into its own class a year earlier. Since that time, it has been involved in 29% (23 of 79) of all changes, despite the fact that a) the model has not changed and b) it is only called from three places of the main application code!

From a manual inspection of the class in question[2], it has been determined that this high rate of change is caused by coupling with related, but different functionality. It is this other functionality that changes often, because it deals with manipulating the action specification, which is enhanced more often. However, the risk that regressions occur in the state-management code, due to the unrelated changes, is high, and at the same time completely unnecessary.

In addition, many potential communication errors have to be handled, as outlined in the pattern description (cf. section 4.3) and it is probably not an accident that the current (year 2009) implementation of the generic task-management library runs to about 1500 SLOC, or almost six times the size. While the general implementation also has substantial extra functionality, the difference in size is partially due to its handling of error-prone edge cases that are missed by the ad-hoc implementations used previously.

Having said this, existing toolkits for task management were also considered. The known toolkits, however, required a particular form of modeling action execution on the lower-levels. For example, the Task-Description Language implementation models activities as task-trees (Simmons and Apfelbaum, 1998). While use of such a model may have been interesting, and probably even beneficial, a more incremental approach was preferred, which could do without explicit models and instead re-use action handlers as they were.

Thus, the decision to implement a re-usable task-coordination library was made, as the effort appeared negligible. As is often with such judgments, it proved to be not quite as simple, but probably still easier than switching the middleware and, in particular, only required switching effort for those researchers that were going to benefit from the new toolkit.

[1]Calculated based on data generated using David A. Wheeler's 'SLOCCount' on revision 180 (svn revision r19364) of component "Following_AM".

[2]In file "AMDataShortTerm.cc"

5.2. The XCF Task Toolkit (XTT)

The XCF Task Toolkit (XTT) has been named, in one part, after the XCF middleware on which it was implemented (Wrede et al., 2006), and in the other part, after the "task" concept in Simmons's task control architecture (Simmons, 1994). However, it was never intended as a re-implementation of TCA, but is based on the view that the TCA, the DESIRE architecture and the previously described coordination model all represented different implementations of the same underlying pattern.

While XTT has been designed and solely implemented by the present author, as part of this thesis, it has been used by several other researchers in the course of the "ERBI" sub-project of the German service robotics initiative[3]. Correspondingly, XTT's design owes many ideas, and requirements, to the discussions with and experimental experiences of these colleagues, most notably Julia Peltason and Robert Haschke.

The toolkit has had three releases, which were mostly driven by enhancements to the protocol level, to improve fault-tolerance. In addition, some minor extensions to the life-cycle model were made in the third release.

5.2.1. XTT Design Guidelines

Based on the implementational difficulties observed in the original system, the following requirements are specified for a service-level implementation.

1 *Separate State-Management from Activity Specifications.* The main advantage to be gained from a toolkit is that it should be re-usable for differing activities. In particular, this requires that the activity specification, which is different for each activity, be independent from the state storage (even though for transport, the state must be attached to the activity specification).

2 *Scheduler Independent.* The implementation should not require a particular way of scheduling low-level actions, but instead support whatever means the handlers were using already.

3 *Protocol Compatibility.* Both for reasons of compatibility, and to facilitate incremental development, the protocol level has to be completely compatible with previous implementations. Future extensions can then be explored in an incremental way, while keeping backwards compatibility.

4 *Life-Cycle Correctness.* The toolkit should only offer operations that cannot violate the life-cycle. This is likely to require dedicated submitter- and handler-side interfaces. Where an operation is legal only some of the time, runtime checking must occur.

[3]`http://www.service-robotik-initiative.de/`

5 *Reduction of Implementation Effort.* Recurring functionality should be provided by the toolkit as much as possible, to reduce the development effort for handler developers.

6 *Prevent Direct Middleware Access.* Both to reduce coupling and to prevent application-developers from bypassing the checking enforced by the toolkit, the underlying middleware should not be directly accessible.

7 *Use Middleware Transport Datatype.* The XCF middleware uses a very flexible transport type based on the XML data model, which is essentially an attribute-value tree, for the main data plus optimizations for large binary data. This data-type should be kept and exposed to the application level, to allow manipulation of the action representation directly. Correctness can be ensured by checking the representation prior to middleware communication

5.2.2. XTT Structure

As shown in figure 5.3, the XCF Task Toolkit has basically two branches (with the "TaskSubmissionService" for the submitter-side, the others for the handler side), both of which sit on the middleware, in package "net.sf.xcf". The application-level interface consists of the services, and implementations of the standard "Future" and "Callable" interfaces.

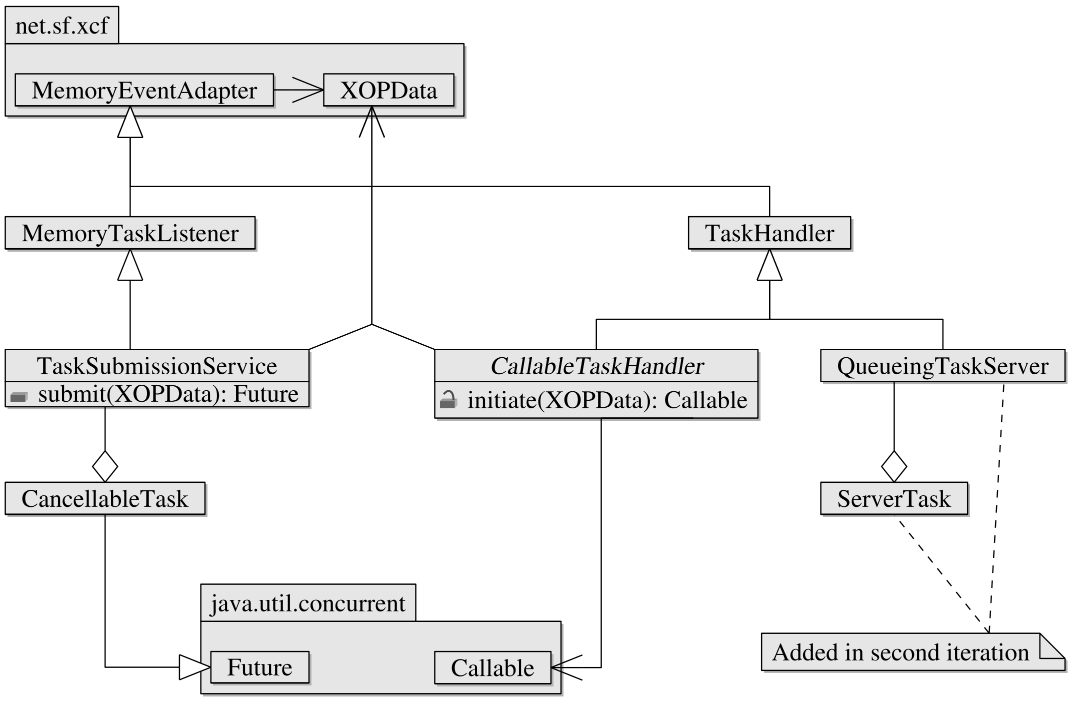


Figure 5.3.: Structure of components in the XCF Task Toolkit.

In the initial implementation, life-cycle state changes were realized by implementing patterns for mapping between synchronous and asynchronous execution, taken from the standard “java.util.concurrent” package. These patterns, “Future” and “Callable” represented the main interface for life-cycle interaction. For reference, they are reproduced in figure 5.4.

Using these interfaces, the life-cycle could be mapped as follows, first for the submitter side:

- `TaskSubmissionService::submit` set state to “initiated”
- `Future::get` wait until a terminal state is achieved. If terminal is “completed”, return the result. If terminal is “failed” or “rejected” throw corresponding Exception.
- `Future::cancel` set state to “abort”.
- `Future::isDone` true if a terminal state has been achieved
- `Future::isCancelled` true if a terminal state is achieved and it is not “completed”

Correspondingly, on the handler side:

- `CallableTaskServer::initiate` If a Callable is returned, state is set to “accepted” and the callable is called. Otherwise, if an exception is thrown, the state is set to “rejected”.
- `Callable::call` If an exception is thrown, state is set to “failed”. Otherwise, state is set to “completed” and the result is returned.
- State “abort” received from client. Interrupt the thread executing the `Callable::call`. If interruption succeeds, set state to “aborted”, otherwise to “abort_failed”.

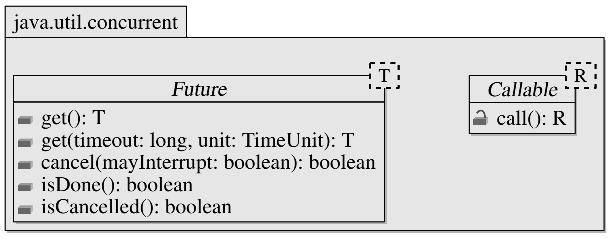


Figure 5.4.: The Future and the Callable interfaces (Sun, 2009c).

As a result of these mappings, the life-cycle is never explicitly manipulated by the application-level. The idea behind that is that the life-cycle is managed in the service-level only and that correctness can be guaranteed (assuming the service-level can be made to be correct).

5.2.3. System Interaction

The network-level interaction of the service-level with the “Active Memory” event bus is shown in figure 5.5. The unconnected messages represent the interface to the application level, which has been omitted for brevity. Please note the use of both synchronous (filled triangle, with reply) and asynchronous calls (open arrow). Also, please note that *reception* of asynchronous notification is *not guaranteed* to occur in the sequence shown, due to network latency (cf. figure 4.9).

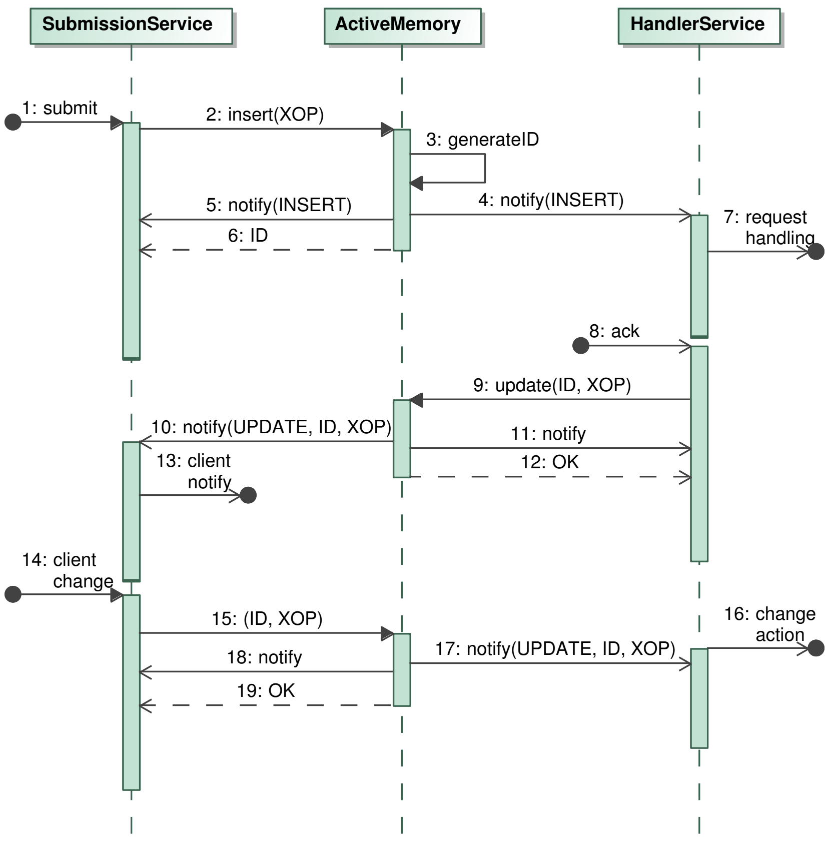


Figure 5.5.: Interaction of service level with the event bus.

The task identifier is generated by the active memory and used in subsequent “update” notifications. The service implementations use this to demultiplex messages to tasks. The active memory also generates self-notifications, which are filtered out in the service-level.

5.2.4. Task Server Implementations

There are two different task-handler base classes, for different use cases. The first of these, the "QueueingTaskServer" is straightforward: It only delivers notifications into a queue, for handlers which prefer to handle state changes explicitly. The second, however, completely manages life-cycle state changes for anything that can be encapsulated as a simple function call, as described in section 5.2.2.

Mapping to function calls involves several edge cases, to be able to support accept/reject, updates and cancellation. See figure 5.6 for an overview of reject support. When the request to create a handler callable fails, the task is rejected, otherwise it is executed.

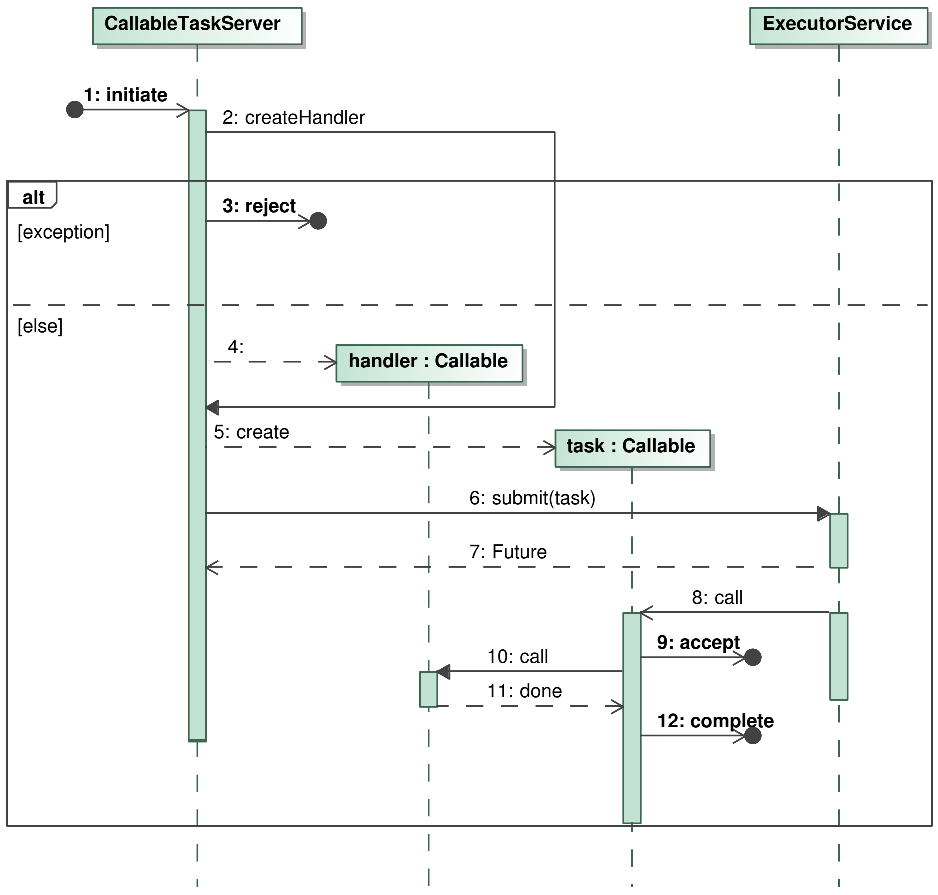


Figure 5.6.: Callable task server handler accept/reject interaction.

In a similar way, the created Future (cf. step 7) supports cancellation, which can be triggered when an "abort" signal is received. This will cause the "ExecutionService" to interrupt the thread executing the task callable. If interruption succeeds, "aborted" is set, otherwise "abort_failed". This interaction has been left out of the diagram for brevity.

The fact that showing the full callable/task interaction would have made the diagram twice as large is an indication of how onerous it is to implement the various cases. This is certainly one reason why developers prefer to implement just a simple RPC handler, with all the drawbacks that has. However, the CallableTaskServer completely encapsulates this and derived classes only have to provide one handler function. Thus, it is hoped that such an adapter will ease use of life-cycle based coordination considerably.

5.2.5. Summary

The XCF Task Toolkit provides a service-level implementation, encapsulating the middleware details and, if desired, life-cycle state management for service implementations. Middleware encapsulation solves the various network race issues identified previously and reduces overall complexity considerably. An optional task server that manages the life-cycle of a simple function call is provided to ease the transition from RPC-style services to the life-cycle protocol. The overall goals of toolkit have been correctness and support for separation of concerns in applications.

5.3. Proof of Concept: Text-to-Speech Service

The proof of concept has been undertaken before the first wider release of the task toolkit, to validate that it is functional and easily usable. The results reported in this section are largely based on quantitative and qualitative analysis of the components' source code, with a focus on dependency and complexity analysis, using the metrics described in section A.1.

5.3.1. Preventing Insider Bias and 2nd System Effect

In this study, which directly measures a particular implementation, it is important that this implementation not be influenced by a knowledge of the goal of the study. However, due to the early stage of the toolkit and the needs of the overall project at the time of the experiment, only the present author was available to carry out the modification for use of the toolkit.

To remove the potential bias this could have caused, the implementational parts of the study have been split. In the first part, the present author has added support for the task toolkit to the component by only adding new code calling the old, in the form of a completely new class. In a second step, a student assistant has been tasked with removing any references to the previous, RPC-based, invcation method from the code. The student was forbidden to perform any other, unrelated changes or clean-ups. Thus, it is believed that the resulting code base is a valid comparison target.

A second potential risk is the second system effect, whereby an improved understanding of the target domain causes a better design on the second try. This risk has been headed off through the exclusive use of additions to existing code in the first step, which furthermore only related to the external interface, not the functionality of the component as such.

5.3.2. History and Use of the Text-to-Speech Component

Text-to-speech synthesis is a common service in systems that use spoken language to communicate. It takes a text representation as input and produces an auditory signal that resembles human speech, which is then played back to a human listener.

The test component realizes a front-end to the "Mary" Text-to-Speech System (TTS) (Schröder and Trouvain, 2003). The Mary TTS is a client-server system with a custom Remote Procedure Call (RPC) interface, developed in Java. Mary's server only produces the waveform, playback must occur on the client. The original implementation of the client accepts the string to be spoken, uses the Mary API for synthesis and then plays back the resulting waveform.

The component originally used an RPC-style interface, where the entire synthesis and playback operation is performed during one XCF remote procedure call. This call could either be performed fully blocking, where the call returns when playback is complete, or partially-blocking, where it returns as soon as synthesis is complete, but before playback. If the latter option is chosen, an estimate of playback time is returned.

If another output request is made before the previous output is complete, synthesis occurs in parallel, but playback waits until the previous output is complete. If this happens, the estimate of playback time is accurate in terms of the duration of the audio sequence, but it does not take the waiting time into account.

5.3.3. System Context

Even though text-to-speech is primarily a service component, its state and interaction is still relevant for several parts of the system. In the original component, as shown in figure 5.7, text-to-speech sits between the dialog manager and the synthesization server.

In addition, the text-to-speech component notifies the speech recognition component about imminent output, because otherwise, depending upon speaker and microphone conditions, the recognizer could try to interpret the synthesized audio output. Note that this creates a dependency from the TTS server to the speech recognizer.

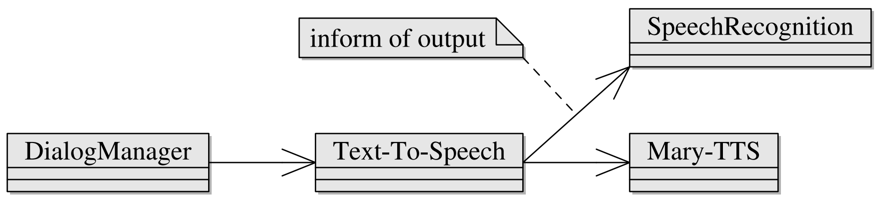


Figure 5.7.: Original Text-to-Speech context.

In contrast, figure 5.8 shows the context after use of the task toolkit: Mainly, the direct association between DM and TTS has been replaced by mediated interaction through the event-bus ("ActiveMemory"). Of course, a dependency is still present. Other interested listening components could be attached in the same way.

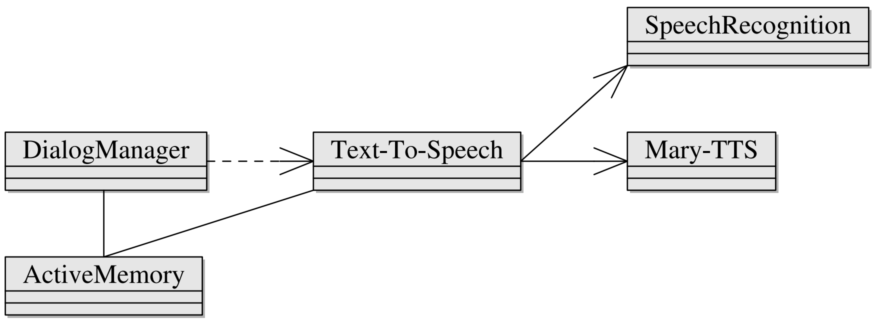


Figure 5.8.: Text-to-Speech context with Task Toolkit. This shows the actual dependency structure after the experiment – in particular, the direct notification of the speech recognition component has not been changed, even though the toolkit would in principle be capable of replacing it with observation instead. The reason has been that changing the recognizer component at the time was not an option, but this is planned for the future.

5.3.4. Stakeholder Analysis

The stakeholders for anything that affects the TTS front-end are the researchers working on the dialog manager (because it is the primary user of the TTS service), the developer of the front-end itself, researchers who are interested in fine-grained coordination of speech and gesture and system integrators.

A relevant social aspect in this context is that the first two of the stakeholders do not gain directly from application of the toolkit: The dialog manager only needs to know when playback is complete and the tts front-end has all information needed locally. While one could assume that they have nothing against a functional extension, they do have important interests: Namely, that current functionality should stay correct and their components should not become more complicated unnecessarily.

Additionally, system integrators have the interest that other components won't be adversely affected.

5.3.5. Study Design

From the previous analysis, the objectives of the prototype are:

1 Show that the "Callable" handler interface can protect handler applications from the additional complexity of the task-state pattern and ensure correctness of the life-cycle.

2 Provide the additional information needed for fine-grained coordination.

The text-to-speech frontend will be changed to use the task toolkit for incoming speech output, instead of the RPC interface. The dialog manager will be changed to use the task toolkit for submission of action request. Verification of the objectivities will be performed quantitatively, through dependency metrics, and qualitatively, by judgment of the original authors.

5.3.6. Data Analysis

The data to be used for this experiment are the source code of the text-to-speech front-end prior to and after integration of the task toolkit.

The first set of metrics pertains to the middleware dependencies in the component. In table 5.1, the total number of classes and methods is given, with the number of those that reference the middleware or task toolkits in brackets. These are compared with total the number of classes and methods referenced from the middleware and task toolkit.

Version	#Classes (ref)	#Methods (ref)	#Referenced C	#RM	avg refs
before	14 (5)	71 (12)	9	11	3.75
after	10 (5)	76 (14)	8	8	2.86

Table 5.1.: Dependencies in the TTS front-end component.

Overall, the trend is towards lower number of dependencies, but in general the numbers are fairly close together and the changes are more for technical reasons. For example, the RPC interface of the middleware requires listener classes, thus there are slightly more classes before than after.

The second set of metrics concerns the object-oriented complexity of the adapter class. For this, only the modified classes were compared in detail.

Classname	WMC	CBO	RFC	LCOM	Ca	NPM
MaryXCF	19	21	80	**101**	4	6
MaryConnector	15	13	60	**5**	3	9
MaryTS	7	12	26	**17**	1	3

Table 5.2.: Chidamber and Kemerer OO Metrics (cf. appendix A.1) before/after. "MaryXCF" is the original server, renamed to "MaryConnector" afterwards. "MaryTS" is the task-server implementation.

The Chidamber & Kemerer Metrics given in table 5.2 demonstrate that addition of the task-server has distributed complexity. While not all metrics can be simply summed, it is apparent the coupling between objects (CBO) has roughly split between the two new classes and similarly for the response for a class (RFC) and afferent couplings (Ca). The one *huge* change, however, is on the cohesion metric (LCOM), highlighted above: The methods in the two new classes are much more cohesive than before. Taken together, this strongly suggests that the desired separation of concerns has been achieved.

5.4. Summary

The coordination interface used in these experiments can provide a clear, flexible and tightly integrated model for coordination. However, they cause increased communication effort, due to the larger number of updates per task, and the common mixing of such updates with manipulation of the action representation. The effects of this could be substantiated through examination of two components which previously implemented the coordination interface directly. Components should not be concerned with such communication issues and be able to concentrate on the action executed only.

The use of a service-level toolkit successfully achieved separation of concerns for a typical service handler component. Furthermore, despite the fact that the component now supports a much more capable interaction method, the sum of couplings to middleware and task toolkit were lower than the previous couplings to the middleware alone, which suggests that the design of the component has improved.

So far, data from two isolated components has been studied as proof of concept. A service-level toolkit has further potential benefits when used in a system with several initiative-taking components. Therefore, data from a fully integrated system will be presented in chapter 6.

Acknowledgements

Many thanks go to Lars Schillingmann, a colleague in the ERBI project, who allowed adaptation of his handler component, and to Julia Peltason who specified the original requirements for the QueueingTaskServer.

6. Life-Cycle Coordination for Mixed-Initiative Human-Robot-Interaction

The prototype described in the previous chapter was deliberately restricted with respect to both its API and the life-cycle employed, to reduce the potential for errors. This worked well in the case of a simple service component, but it became clear quickly that for components that participate in, or even coordinate, multiple actions, this restricted approach was not sufficient. While coordination components are not as numerous, their higher level of interaction with task execution also means that they would benefit more from a toolkit, making the effort of specialized support worthwhile.

Therefore, the case study presented in this chapter investigates life-cycle based coordination, as used in the task-state-pattern, in the context of several use cases from a fully integrated system, specifically the "Curious Robot" scenario (cf. chapter 2). Particular emphasis has been placed on a) the suitability of the model for coordination components, b) any changes to the life-cycle and/or the API required as a result of issues identified and c) the general suitability related experience gained through application of the model by other developers and their feedback.

The rest of the chapter will first introduce some context and then introduce the study design. In the main part, the use cases are presented and analyzed. The chapter concludes with a discussion of the main results.

6.1. Context: Initiative Generation for the Curious Robot

In the "Curious Robot" system, the robot can take dialog initiative at any time, primarily based on perceptual input, but also on internal goals. To ensure that this occurs consistent with the current state of interaction with the human partner, the dialog manager coordinates human and system initiative, in the so-called "mixed-initiative" approach (Allen, 1999).

Interaction between the DM and components that would like to initiate a system action occurs based on the task-state pattern: Components propose an action and if it is currently possible, the dialog accepts and tracks it, providing commentary throughout execution. When the action is not currently possible, it is rejected. All components performing overt activity have to coordinate in this way with the dialog, either directly or indirectly, and at any time during execution, the human partner may interrupt or change the task, or propose a different one.

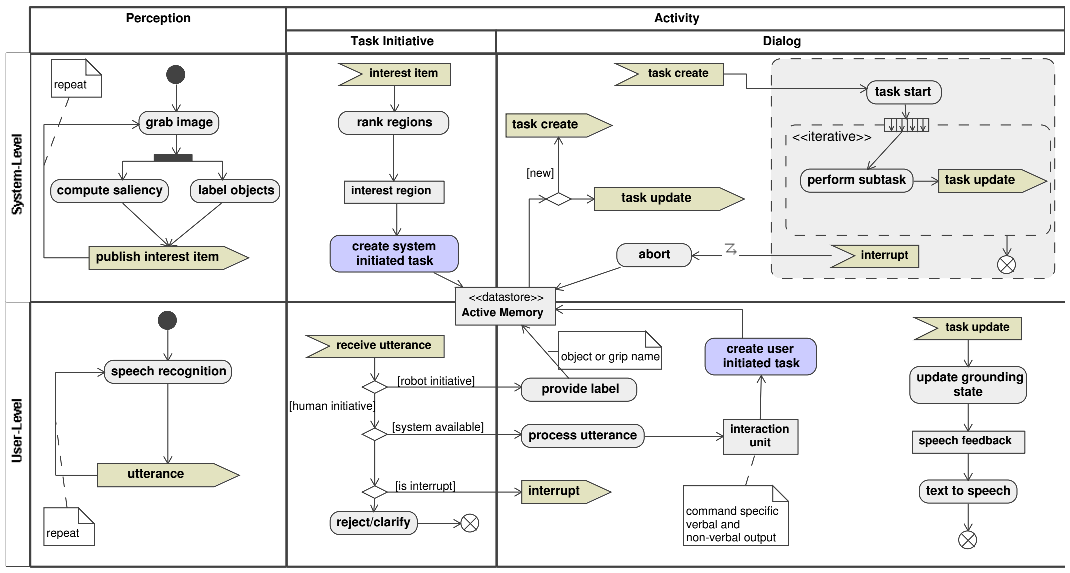


Figure 6.1.: Task Interaction in the Curious Robot Scenario.

As a result, the order of system vs. human activity is not at all fixed. The full system interaction is shown in figure 6.1. Activities related to motion planning and execution are not shown here, but are coupled through listeners on the task updates. Note that the connection of the dialog to the rest of the system is through events only, and that there are quite a number of concurrent activities in the system, even without motion control.

6.2. Study Design

The present study has been carried out as an active design study (Runeson and Höst, 2009): It has been performed during use of the toolkit and its results guided the toolkit's further development. The overarching goals have been a) to investigate consequences of and validate the design choices made in the toolkit's implementation in a realistic setting and b) to extract generalizable insights that can guide future applications of the task-state pattern in different scenarios.

The case study will examine the two coordination components in the "Curious Robot" scenario (cf. chapter 2): the dialog manager, which manages human-robot-interaction, and the hierarchical state machine (HSM), which manages arm and hand motion. For these, firstly, their direct interaction with other components will be investigated and, secondly, the resulting sub-system separation will analyzed, to determine whether the intended separation of concerns was achieved.

6.2.1. Generalization and Validity

While one case study cannot be claimed to provide generally valid results, every effort has been made to assure as much generalization as possible. Firstly, the two coordination systems examined differ in many aspects and had previously been developed independently. Secondly, the system stretches all the way from abstract human-robot-interaction to low-level motor control and thus encompasses a large part of the functionality relevant in state-of-the-art robot systems, particularly those targeted at service robotics.

Third, and not least, it should also be pointed out that the system under study underwent three development iterations during the study period (cf. chapter 9). This differentiates it from typical "one-shot" case studies and allows insight into maintenance concerns, which are known to make up the larger part of a system's life-time.

The iterative approach has also provided opportunity to verify the claimed benefits by the other involved developers. It could be said that these developers are not fully independent, because they are part of the same project and university (though in two different groups) and that is certainly true. They did use the approach during an actual project over several years, however, and when issues presented themselves, these developers were affected negatively. Thus, they were not inclined to simply please the present author, but had every interest in problems being identified and addressed.

Taken together, these characteristics of the study are expected to allow conclusions which, while not necessarily applicable to all systems, are likely to be valid and of sufficient generality to apply to a reasonably interesting subset.

6.2.2. Goal: Validate the suitability of life-cycle based coordination.

The first aspect addressed in the study is whether the life-cycle model is appropriate for the coordination requirements. An overly complicated life-cycle model would mean more effort in component construction and a more difficult validation procedure during integration. Therefore, it is to be avoided. On the other hand, an overly simple or restrictive life-cycle model would mean that aspects not covered would have to be treated outside of the model, either by convention, or by inventing additional coordination mechanisms on top. The concrete question here is under what circumstances, and in which combinations, transitions are used and if that use is appropriate.

From the viewpoint of a toolkit developer, it would also be interesting to identify widely applicable subsets of the life-cycle model. If such subsets could be identified, default implementations can be provided which handle some aspects (those which are not supported by the component) in a default manner and thus reduce effort and improve correctness for application development. The question here is whether and if yes, which, subsets of the life-cycle model are in use.

Data Used

During design of the life-cycle model, the intended use of the transitions was specified and there was also some idea of the combination in which these transitions were to be used. However, only practical experience can tell if these expectations hold.

To supply the data to answer this question, a model-based approach has been chosen, as a suitably abstracted level. The reason is primarily that individual source-code level differences and the clutter introduced by other functionality being mixed up with state-management (particularly in components not using the task toolkit already) are not of interest for this study. A second reason is that the diagrams, while not small, are sufficiently compact to be included with the analysis here and thus facilitate outside inspection.

In the following, a number of exemplary use cases are modeled using UML sequence diagrams and the models are then analyzed qualitatively. The analysis has been verified by the authors of the respective components.

A risk of the model-based approach is that the model may not reflect the reality. This is true even for automatically reverse-engineered models, as the extraction process of current tools is not error-free and has to be manually checked. In fact, it is unfortunately still the case that automatically generated models are, at the present time, more error-prone than manually created ones. Therefore, the models used in the following have been created manually and were validated for correctness by the component developers.

Parameters Used

For each component in the models, the following aspects were analyzed: a) the transitions occurring in interactions with that component, b) the action that occurred as a result of a transition and c) the ratio of *accepted* inbound to *accepted* outbound actions and d) the maximum number of concurrently active actions initiated by that component.

Aspect (a) is intended to allow identification of subsets of transitions used, whereas aspect (b) allows analysis of the fit of the transition to the action. Aspects (c) and (d) are intended to provide some insight into differences in coordination styles.

6.3. History and Evolution of the Scenario

The dialog manager used for this study originated within the COGNIRON Human-Robot-Interaction project, and has later been extended during the ERBI project. In contrast to many other dialog components, its focus is not so much on detailed, scripted dialogues or information gathering/giving but on managing the interaction state between human and robot. Its guiding principle is that of “grounding”, proposed by Clark and Brennan (1991) and realized in this case by Li (2007). It is particularly notable for flexible, nested dialog exchanges.

Later extensions focused on providing more feedback on and interaction during ongoing activities by the robot. In particular, the DM tracks action execution, provides verbal feedback about it at appropriate points and allows the human to interrupt or comment on activities. While some of these capabilities were present previously, they were developed specifically for particular actions, at considerable effort per action. To provide a generalizable and extensible component architecture, the task-state pattern was introduced. This extension, and subsequent substantial refactoring, were performed by Julia Peltason.

6.4. Coordination Use Cases

In the following, four use cases will be introduced to demonstrate the range of different applications for this case study. The first is a general overview, the second and third a look at the two different subsystems involved, and the fourth, an example of independent synchronization enabled through use of the generalized task model.

A Note on Model Interpretation

The models in the following are visualized as UML sequence diagrams, as the message sequence is the most relevant information for the purposes of this study. However, in contrast to regular sequence diagrams, there are three modifications: a) The objects are independent components, b) messages are communicated through the middleware, not in-process and c) the event-bus used for communication is not shown.

The first two of those are straightforward mappings to semantics for event-based systems. The third, however, was made for brevity: If the event-bus were included in the diagrams, the number of messages would have at least doubled. As all the shown interactions are just examples anyway, this would not have contributed additional information, but would have significantly increased clutter. However, the decoupled communication typical for event-based systems remains:

> All messages are shown as originating from the sending and terminating at receiving components. This should *not* be construed as a dependency from sender to receiver but is solely for brevity of presentation. Components are still decoupled through the event-bus.

All models will be accompanied by tables that summarize the task state transitions occuring in them by component, to indicate which subset of the life-cycle is being used.

6.4.1. Analysis of Coordination Integration

The sequence diagram for an example activity is shown in figure 6.2, and will be described in detail in the following.

The figure represents a fairly straightforward sequence: First, the system initiative creates a "label query" task (1), the dialog is notified (2) and accepts the task (4),

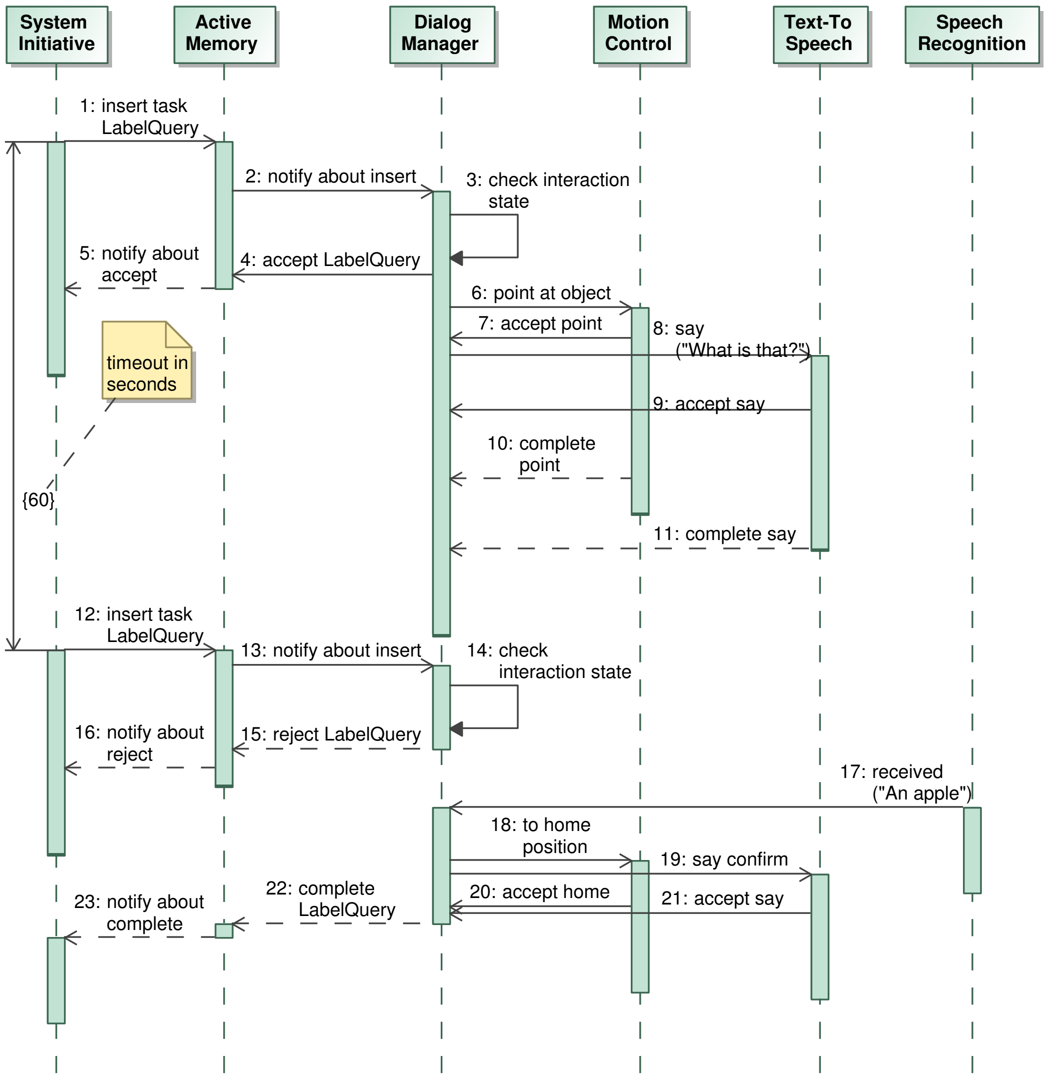


Figure 6.2.: Example 1: Label Query Interaction Sequence. All life-lines shown represent independent components which communicate through the middleware. The "Active Memory" is the event-bus component.

after checking that nothing else is pending (3). Executing the task results in two new tasks (pointing, 6, and verbalization, 8), which are accepted, too (7, 9), and eventually complete (10, 11).

Some time after that, the human provides an answer (17), which causes the robot to confirm the label received (18) and move back to home positions (19). Acceptance of these by the components (20, 21) completes the overall interaction (22, 23) (but compare figure 6.4 for an alternative ending).

In between, a timeout expires and the system creates an initiative again (12, 13), which is rejected (15, 16), after the state check (14) reveals that another task is ongoing. The timeout is a simple (and not the only) means to prevent the interaction from completely stalling if the human fails to answer.

Historical Information

This use case was the first to be realized in the “Curious Robot” scenario and there were two versions which differed in that in the first, all components had custom implementations of the task-state pattern whereas in the second, the task toolkit was used for some of them. Specifically, it has been used for the “system initiative”, “text-to-speech” and “dialog manager” components. The motion control subsystem kept its independent implementation. Because of this, it provided an ideal test-case to ensure that the new implementation was compatible (and it was).

It is also notable that in the first implementations of this use-case, all state changes were communicated, but the components ignored some of them. For example, the first system initiative component never checked whether its actions were actually executed, but simply produced new actions at about 30Hz (the vision processing rate), which, together with rejection, caused 60 messages per second largely going to waste. This mode of operation was soon found to be a sub-optimal use of resources, because even discarding events can take up noticeable time, when there is many of them. While obviously the rate could have been throttled down to more manageable levels, this would have caused a loss in reactivity to changes in the environment.

A better change has been realized based monitoring life-cycle information: The action proposal component suppresses proposals when an earlier one is currently being processed (i.e. has been accepted), up until a given timeout (60 seconds by default). This vastly reduces message traffic at no loss in functionality. Thus, while the proposal component’s functionality could be realized without life-cycle information, it can be realized more efficiently with them.

Analysis

In this typical situation, the dialog manager is both a server (of system initiative tasks) and a client (of tasks for text-to-speech and motion control). Furthermore, with regard to system initiative tasks, it constitutes the also typical case of a server which can only perform one task at a time and its conflict policy is to reject.

Table 6.1 summarizes the transitions, ratios and actions performed of the interaction shown in figure 6.2. In this simple overview, it can be seen that there is an accept/reject sequence between system initiative and the dialog, which reflects that the user can usually only tolerate one interaction at a time. This matches the intended use of that part of the model. The other transitions use accept and complete in the intended manner, too. Please note that the motion control interaction has been simplified here and will be explained in more detail later.

One interesting aspect is that the text-to-speech service does not use an accept/reject sequence, even though only one utterance can be spoken at the same time. This is because the TTS server queues requests and produces them in the order they were received, one after the other. Therefore, there can be a time gap between "initiate" and "accept", when a new request is received before the old one is complete. As all utterances are caused by the dialog, such a situation does not occur in our scenario, but it would be something to keep in mind for scenarios where it can happen.

Component	Transitions	RIO	CO	Results
System Initiative	I, A, R, C	0:2	1	A: wait 60s until next
Dialog Manager	I, A, R, C	1:4	2	I: check state, initiate sub-actions
Motion Control	I, A, C	n/a	n/a	I: initiate sub-actions
Text to Speech	I, A, C	1:0	0	I: start action

Table 6.1.: Transition codes: I – initiate, A – accept, R – reject, C – complete. RIO: rate of accepted in to accepted out. CO: max concurrent outbound. A value of "n/a" means not shown here.

6.4.2. From Actions to Activities

So far, the interaction looks very similar to a hierarchical decomposition of a larger task into multiple sub-tasks – and it is. However, let us look a bit closer at the situation when the full motion control sub-system is included into consideration.

In figure 6.3, the left-hand side shows the user-facing side of the system, up to the "Dialog". The right hand side, starting with the hierarchical state machine ("HSM"), shows what goes on within the motor control subsystem: At first, a grasp task is started and the HSM interjects with trajectory planning and the various control components to move the arm to its position, then grasp. This is already a fairly typical, decomposed interaction which the rest of the system is not particularly interested in the details of.

After the arm is in position, a user correction is received via speech input, telling the robot to grasp slightly more to the left (just as an example, of course). This causes a position correction (in this case not shown here, but it represents a fixed offset relative to the position determined by vision), and a corresponding update to the grasp target specification. The HSM then aborts the grasping process, plans a new trajectory for the arm (likely very short), repositions the arm and starts grasping

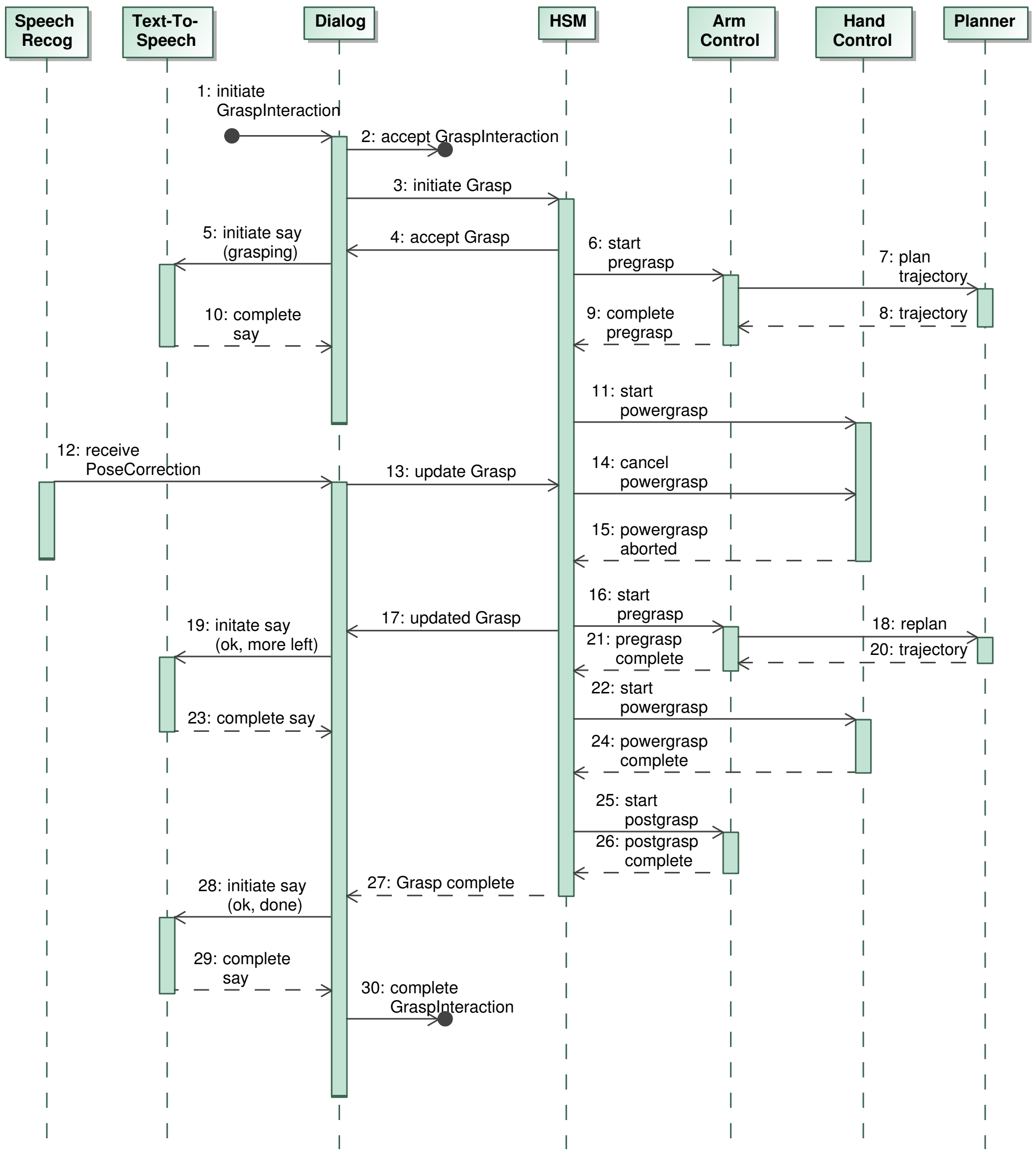


Figure 6.3.: Example 2: Motion replanning after User Correction. All life-lines represent independent components, except for "Trajectory Planner" which is contained with "Arm Control".

again. After this is complete, the grasp task is marked as complete, and the dialog produces user confirmation.

Historical Information

In user studies, we found that interaction with the robot about its physical activities is challenging for users (cf. 9.3) and that users would benefit from every help they could get, including the ability to change the action while it is still ongoing (which was not possible at the time). Therefore, the "update" transition shown in this use case was introduced, to enable such tighter coupling. It was one of several changes in the later stages of the scenario and one which we do not yet have user study information on, to determine whether it actually improves interaction performance.

From the viewpoint of the architect, it is interesting to note that adding the transition to the life-cycle of the toolkit was trivial, and changing the components to use it at all was not particularly complicated either, but that designing a human-robot-interaction to make good use of it is a real challenge. It is expected that it will take several more user studies to make progress on this issue. However, these studies can now take place on the basis of much tighter integration possible between the coordination components, without sacrificing independent development.

Analysis

This use case demonstrates the coordination of larger activities: The action carried out by one subsystem is realized as several sub-actions internally and the "update" transition of the extended life-cycle model enables the other sub-system to modify the ongoing activity, without having to be aware of the sub-activities. This could not have been realized by simply canceling the old and starting a new action, as the motion subsystem cannot infer the relation between the two on its own.

Such interactions are not exclusively found during human-robot-interaction, but may also occur, for example, in visual servoing. Visual servoing is typically characterized by a very tight, real-time coupling of vision and motor control. However, it could also be realized based on the extended life-cycle shown here, given suitable underlying communications mechanisms (and running in a single process, most likely, but still as independent threads, thus maintaining a degree of independence).

The characteristics of the components involved are summarized in 6.2. It should be noted that the interaction between the HSM and the arm/hand control components does not use the explicit life-cycle but has roughly similar operations to the basic life-cycle. The interaction between dialog and HSM uses the extended life-cycle coordination and the HSM performs the mapping.

One can see that the dialog manager and the HSM are slightly different coordination components. The dialog has overlapping activities ongoing (one to motion control and several to speech output), whereas the HSM performs strictly sequential[1]. There are two reasons for this: In this example, it is simply that grasping does

[1]In the sense that there always is only one logical activity going on, even though it can involve

Component	Transitions	RIO	CO	Results
Text to Speech	I, A, C	1:0	0	I: start action
Dialog Manager	all except Ca/Cd	1:4	2	I: check state, initiate subactions
Motion Control	all except R	1:4	1	I: initiate subactions
				U: cancel current, initiate replanning
				C: start next subaction
Arm/Hand Control	I, A, C, Ca, Cd	1:0	0	I: start, Ca: abort

Table 6.2.: Transition codes: I – initiate, A – accept, R – reject, U – updated, Ud – updated, Ca – cancel, Cd – canceled C – complete. RIO: rate of accepted in to accepted out. CO: max concurrent outbound. A value of ”n/a” means not shown here.

not make sense before the arm is in position. More importantly, however, while the HSM can move two arms at the same time, they share parts of their physical space and always have to be controlled together, to prevent collisions.

6.4.3. Overlapping Activities or “Implicit Completion”

Complementing the previous use case, let us now look closer at the interaction subsystem. A particularly interesting issues arising in Human-Robot-Interaction is that of “implicit completion”: The completion (or acknowledgement) of the prior action is not signaled explicitly, but implicitly – by successfully beginning the next action.

For example, in our object label query task, the robot confirms the label provided by the human and then the human has two options: Either simply yield the turn and allow the robot to proceed, or correct the confirmation. Correction can become necessary due to speech recognition errors and could be prevented by always asking for confirmation. This is, however, not optimal in the majority of cases where recognition is correct, so an optimization is useful. It was also felt that this approach is more “human-like”, though this remains to be established.

The resulting sequence is illustrated in figure 6.4: A “LabelQuery” system initiative task (1, asking for an object label) starts the interaction and causes both a pointing gesture to the referent (3, 5, 6, 7, 9, 10) and a verbal query (4, 8) to be performed. Then, there is a correction iteration, where the human first provides a label (11), the system uses that to train an object recognizer and confirms the name (12-14).

For this example, a case is shown where the first recognition is incorrect, so the human partner provides a correction and the system starts over (15-19). This is followed by implicit completion, where the system attempts the next query (20-22) and implicitly completes the previous task. As the human replies (25), the next task proceeds. The system could also have refused the new task and taken initiative in going back to the earlier task, which, in this mapping, would cause a new LabelQuery, initiated by the human.

both arms simultaneously.

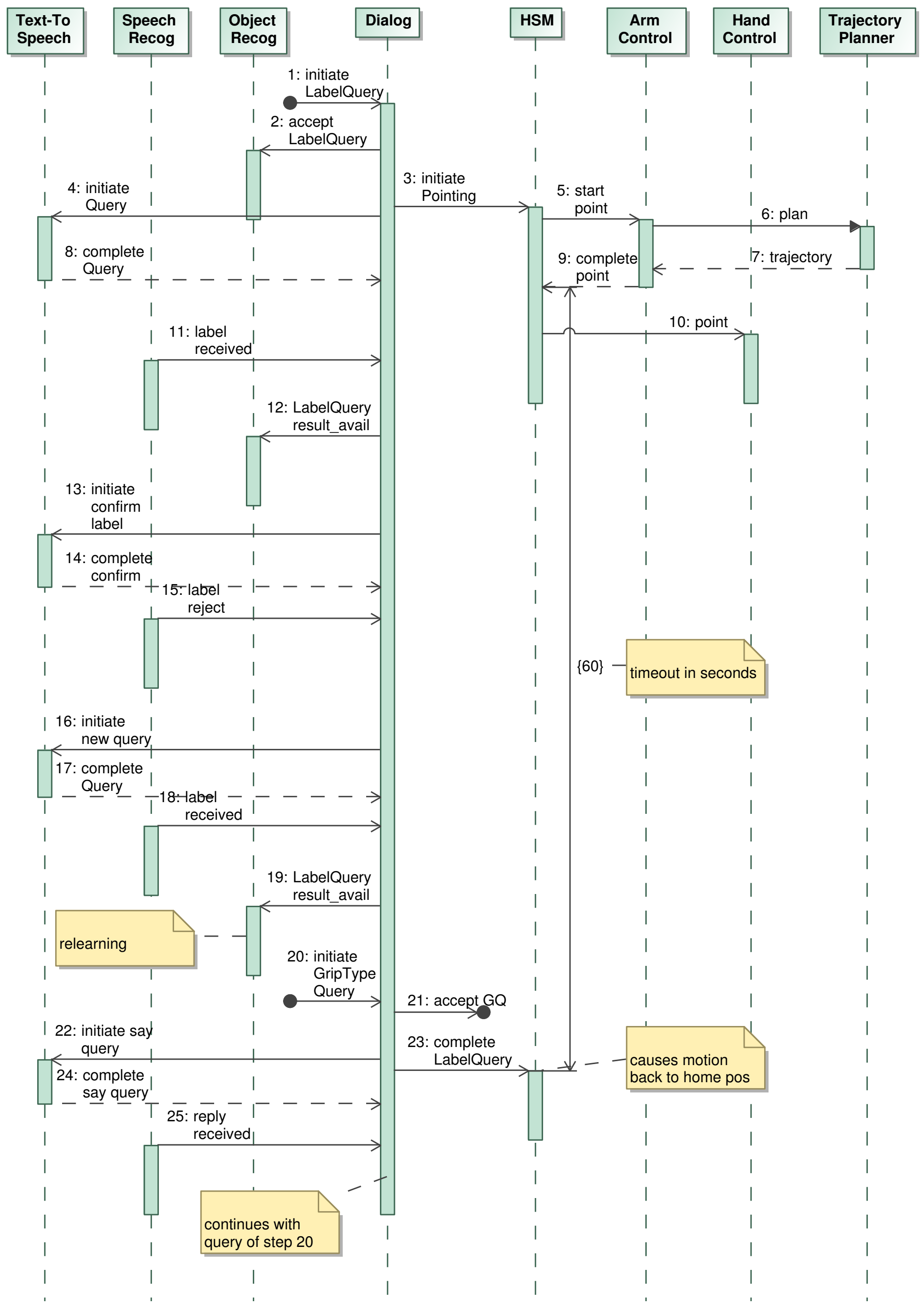


Figure 6.4.: Label correction with Implicit Completion. As before, all life-lines represent independent components (except for "Trajectory Planner" which is contained within "Arm Control") and communication is through the middleware.

Historical Information

This is the second use case that required the introduction of a transition which was not present before: “result_available”. This transition is a self-transition on the “accepted” running state and used to provide intermediate results. It was introduced because the object detection component could improve its operation when it receives the information that the training data was not to be considered complete.

At the time, there was some discussion on whether the same functionality could have been realized by a sub-action specifically targeted at the object recognizer, but it was felt that the dialog should not be required to interact with object recognition directly, as this would have increased coupling unnecessarily.

Furthermore, the use case provided a secondary validation of an API design choice made earlier: Because there are overlapping actions, the pure “Callable” task server implementation is not suitable – interaction between successive actions’ states are required. The QueueingTaskServer can provide these, simply by interleaving event notifications in the queue, and thus the necessity of such an API for coordination components was again underlined.

Analysis

This scenario demonstrates why separating interaction coordination from motion coordination is a requirement to enable experimental research: The mapping of human activities to task states shown here is the *result of one particular theory of communication*. One might just as well require an explicit confirmation by the human to complete the task, or, as another alternative, consider the task completed only after the human has replied to the new query – to name just two of the many choices that are currently tested out in research systems.

There is not a clear choice what the “best” way of interaction would be and, most likely, different trade-offs would lead to different strategies. For example, the present strategy has been chosen because, in the large majority of cases, recognition of the label works well and explicit confirmation was only adding another step that came across as tedious, time-consuming and machine-like (because of repetition for each object). If recognition performance were worse, e.g. due to outside noise or a non-native speaker, another strategy might be more appropriate. One could even use multiple strategies in the same system, and select amongst them based on current performance.

While most likely not the only one, the task-state pattern provides a way of combining different interaction management components with motion control in a way that is close to the level of abstraction required in typical HRI situations. Thereby, it facilitates independent experimentation with different strategies, as the rest of the system can stay unchanged.

Regarding the life-cycle model, the only notable addition is the interaction with the object detector: During the interaction, the validity of the label received from the user is not entirely certain, as it could have been detected in error. However, the system has to be able to act on its intermediate knowledge, because the user expects it. Therefore, the intermediate “result_available” state provides a result, even though the task is not completely finished, yet. The object recognizer, which does not otherwise participate in the interaction, listens for such transitions (as well as for “completed” transitions) and uses this intermediate information to immediately update sensor information, yet keep open the option of re-learning.

While the reporting of intermediate results may appear useful for many other cases, care must be taken because it requires a level-of-detail decision: Which level of detail will be interesting? This can be hard to decide. For example, during grasping, some components may be interested only about completion, whereas others might be interested in the time when contact between hand and object is established, or other intermediate steps. Whenever such information would be available by other means, e.g. by registering directly for tactile events, that should be preferred, in order to simplify the information conveyed in the task protocol.

In this case, the information received (a speech utterance) was produced by a speech recognizer but could only be associated to the object by the dialog, because it knows the current interaction state. Therefore, the alternatives were to either provide the new information as part of the ongoing action or to create a new action, specifically targeted at the object recognition component. In the latter case, this could have been implemented either as part of the dialog or as a dedicated component. In both of the latter, more implementation effort would have been necessary, so it appeared most straightforward to include it in the existing task.

Component	Transitions	RIO	CO	Results
Text to Speech	I, A, C	1:0	0	as before
Dialog Manager	I, A, C, Ra	2:5	2	I: check state, initiate sub-actions
				Ra: provide intermediate result
Motion Control	I, A, C	1:1	1	as before
Arm/Hand Control	I, A, C, Ca, Cd	1:0	0	as before
Object Recognition	Ra	0:0	0	Ra: train classifier

Table 6.3.: Transition codes: I – initiate, A – accept, R – reject, Ra – result available, C – complete. RIO: rate of accepted in to accepted out. CO: max concurrent outbound.

6.4.4. External Synchronization for Additional Feedback

The last use-case details external synchronization. Here, a component does not itself participate in the life-cycle, but it observes the life-cycle of another action and acts in synchrony with that other one.

As an example, gaze feedback is used. In interaction between humans, the gaze is a powerful feedback mechanism used to indicate, for example, the current focus of interest. Similarly, human-robot-interaction can benefit from the viewing direction

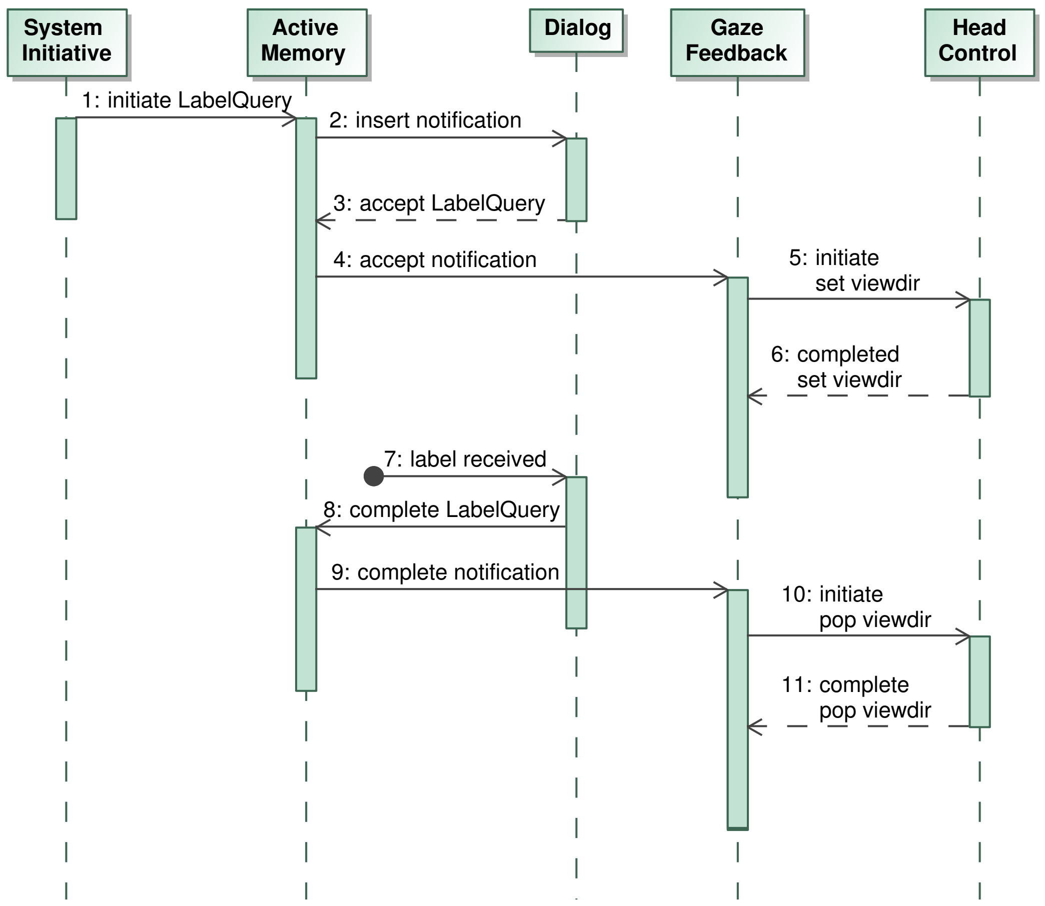


Figure 6.5.: Example 4: Gaze Feedback Sequence. All life-lines shown represent independent components. Middleware communication *is* shown (the "ActiveMemory").

of the robot being oriented at objects in its focus and to do so, the viewing direction needs to be coordinated with the current dialog state. This results, for example, in the robot looking alternatively at the currently attended object, or at the interaction partner.

However, a robot's viewing direction is more than just feedback: It is also, to give just one example, an important means of extending the visible area, by looking around. Therefore, its control scheme is not logically a part of the dialog manager and the dialog manager should not be complicated by having to address these other issues.

External synchronization offers a way to reconcile these issues: It allows the viewing direction to be separate, while still enabling synchronization to the current dialog activities. Figure 6.5 shows in detail how events on the active memory cause notifications for components subscribed to this type of event.

Historical Information

While the use case has applied the existing life-cycle model, it was relevant for toolkit implementation for another reason: It introduced the concept of a dedicated "listener" component, which receives transitions events but does not modify the state. The API had not previously supported that directly but was refactored to separate out the necessary functionality.

Analysis

Extensibility as used in the present use case has been recognized as one of the advantages of the task-state pattern: The level of abstraction allows coordination without detailed knowledge of the proceedings and the event-based method of communication allows integrating extensions without disturbing the existing system.

That being said, it must be recognized that the coupling is not as loose as it appears at first: Often, during development of a system, the abstract state-change notifications are taken to be synonymous with particular activities by the components – as observed by the developer during testing – even though this association is nowhere guaranteed. Later changes, then, may affect the external components in unforeseen ways.

Therefore, while the approach is attractive for incremental changes, the association must be recorded and should be re-examined after changes. In the "Curious Robot" system event notifications are subscribed to based on content-conditions. These conditions, and a runtime record of matching events, have proven very useful in tracing such indirect couplings and a means of recovering and recording indirect couplings from the system interactions should be considered a necessary precondition for safe system evolution.

6.5. Discussion

In the use cases presented, a diverse set of components has been integrated successfully using the task-state pattern. For evaluation of the approach with regard to the specified goals, both the historical information and the analysis presented so far will now be discussed, to provide a summary of the results and a comparison to alternative approaches.

6.5.1. Life-Cycle Design

Beyond general advice for designing finite-state automatons, there was design guidance for designing life-cycles. In the studies presented, the first state-machine has essentially been just found in existing systems, and was then later extended. Is there something we can learn from these extensions?

Functionally, the original life-cycle (cf. figure 4.2) realized a request-response pattern with two regular returns (one upon start, one upon termination) and exception notification (the distinction between "completed" and "failed"). Furthermore, it allowed the client to signal completion, which requires multiple client messages. In essence, then, this is an advanced form of asynchronous procedure call, a well-known building block for distributed systems.

To this, two capabilities were added over time: Updates and intermediate results. The second of these is a straightforward extension from two returns to multiple returns, it simply corrected an oversight of the original specification. Updates, however, are more interesting, because they could, essentially, have been realized also by aborting a current task and starting a new one. In fact, some other realizations of the pattern (ROS's actionlib) realize updates implicitly, by sending new goals.

So, what is the advantage of this new transition? The advantage is that it keeps the *context* of the task. It communicates to the executing component that this is not something new, but merely a change. This is useful only for such components that associate significant state with tasks. In fact, looking at the case studies, there are only two components which makes use of this transition: The hierarchical state machine for motion control, a coordination component, receives it and the dialog manager, another coordination component, sends it. The former benefits because it can skip initialization steps (return to home and similar things) and the latter benefits from keeping context, because aborting and starting tasks is usually associated with verbal feedback, which would need to be suppressed in the case of internally caused updates.

In essence, then, this transition has mostly been introduced to accommodate interaction between coordination components. Now that it exists, other components might use it as a shortcut instead of abort-restart, but that was not its original purpose.

Moreover, both of these components could have done without the new transition, but it made their implementations easier. This came at the cost of, potentially, making other component's implementation harder (because they have to support the new states), but this is not a big issue, because the life-cycle includes an option to refuse an update, which can be used by the toolkit to realize a default implementation. Thus, existing components did not actually need to change.

From this, we can conclude that the minimal life-cycle is essentially a finite-state machine realizing a restricted from of multi-message exchange, bound to the essential characteristics of activities of medium duration. It is fairly static, and found as such in multiple implementations.

The extended life-cycle, in contrast, relates to and is mostly useful for coordination components. Again, coordination components do not interact with tasks in an arbitrary manner. Essentially the only common coordination functionality missing from the current life-cycle is a feature to coordinate the commencement of related tasks – the current system uses sequential start-up, which was sufficient for the purpose at hand, but the improvement possibility is certainly there. Apart from this, the generic life-cycle as proposed is considered essentially complete, because it models the essential characteristics of tasks, and fulfills the common coordination requirements.

Of course, only time will tell whether this view is correct. If it is not, however, we can deduce one more piece of advice from the current case studies: Use toolkits, and always include a possibility to refuse a new transition. In this way, default implementations can safely pave the way for incremental adoption of new life-cycles, without requiring changes to components.

6.5.2. Component Types

By examining the ratio of incoming to outgoing actions as well as the number of concurrently active actions, a separation into four distinct sets of components can be made:

1. Components that do not participate in the task-state based coordination. In the use cases presented, these are all perceptual components, such as speech recognition. More generally, these can be called **pure analysis** components.
2. Components which only perform actions for others. This category includes actuators and other types of output components, but the form of interaction is also used, e.g. in information retrieval processes. Thus, the common "actuator" is eschewed for the more general term **service** components.
3. Components which only produce actions. There is only one of these at the moment, the "system initiative". In other systems, planners have this role. One might be tempted to call them "decision" components, but due to the experiences from real-world systems, a more cautious term is preferred, so they are dubbed **action proposal** components.
4. Components which both produce and handle actions (dialog and HSM). These have been called **coordination** components throughout this chapter and that is the suggested term. Other terms found in the literature are "sequencing", "executive" or "scheduling" components, though all of these appear to originate from architectures that assume central and/or preplanned control, which is why they are not used here.

6.5.3. Life-Cycle Subsets

As can be seen from the transitions in use, the original, basic life-cycle remains widely used for interaction with service components. That said, a number of possible functional enhancements have been discussed which would make some of them useful for these components, too. For example, the "abort" transition is not widely used, but would be useful, for example, with text-to-speech synthesis to abort a question when the human interject (Peltason, 2010). Similarly, the "updated" transition could be used with the same service, allowing it to insert a placeholder when the utterance is changed in mid-sentence (as humans do when they say "ehm", for example).

It is probably safe to assume, however, that not all service components will want or be able to support these extended transition – at least initially. Therefore, toolkit support should be provided (and has been provided in the XTT), to provide default handling of such transition for components which cannot support them. This default behavior should be easy to switch off, when components are enhanced.

6.5.4. Overall Suitability

Overall, the integration was a success: Three successive demonstrators were built, with steadily increasing functionality and stability. The task-state pattern contributed to this success mainly through two factors. Firstly, it reduced development effort for component developers and secondly, it provided a ready-made template for the addition of novel components. In particular the latter aspect was noticeable, as novel components were integrated late into the system with very little effort (cf. chapter 9).

Regarding the API suitability, some extensions were necessary, firstly, to support coordination and, secondly, to extend the life-cycle. They were largely due to the system becoming more capable, as outlined throughout this chapter.

Developer feedback has generally been positive, in particular because effort for implementation of the task-state pattern and generally undesired book-keeping is reduced (Peltason, 2010). This has already led to requests for toolkit implementations for other languages.

6.5.5. Distribution of Work

One interesting, and totally unplanned, effect of separating concerns through use of the task toolkit was that in the initial stages of development, when the toolkit was still being developed, it added another developer to an existing task: When there was a bug in the pattern implementation, it could be solved concurrently with the component developers working on unrelated functionality. While adding people does not always speed up development, this is a case where it did – most likely, because the functionality was split along a clearly demarcated interface.

Obviously, this is only the case when a) the toolkit developer is available and willing to work on the system at the same time as the other developers, because otherwise it would have a detrimental effect and b) there is enough unrelated functionality to work on so that component developers do not have to idle around while waiting for the toolkit to be fixed. In our experience, requirement b) is not a problem, but requirement a) can be and requires management support.

While the effect of the additional development workpower can level off after the toolkit reaches maturity, some positive effects remain because there are more people available to explain integration-related aspects – a notoriously time-consuming and tricky part of system development. Additionally, integration has already been partially solved towards a higher-level of problem definition.

Acknowledgements

Grateful acknowledgement is given to Julia Peltason and Robert Haschke, colleagues in the ERBI project. Julia Peltason undertook the work on the dialog manager and

continued to use the toolkit for several other extensions, thus providing invaluable feedback for its development. Robert Haschke is the lead developer of the HSM component and integrated it with the task-state pattern.

Part III.

Composition

7. Modeling with Directed Typed Graphs and Event-Oriented Decomposition

While the previous chapters have focused on the architecture and coordination of distributed components, the following chapter will deal with flexible construction of novel components[1]. As in systems, an overarching goal is to construct components from existing modules. In this thesis, a secondary goal is to construct components in a way that facilitates functional comparison and experimentation with alternatives.

To achieve module interconnection in a generalized way, the approach taken is that of data-flow, which treats modules as directed typed graphs (DTGs), where the nodes of the graph are functions and the arcs transport (and, if necessary, buffer) data. Furthermore, similar to visual programming, but different from lower-level data-flow approaches, connectivity is completely externally specified and thus easily changed.

An open issue in DTGs is the optimal node granularity, and how to decompose existing functionality into nodes. In this work, the potential for re-use is emphasized and it is suggested that decomposing components by the principles suggested for event-based system integration (Barrett et al., 1996) could aid such re-use. In particular, they separate application-specific assumptions from algorithmic blocks.

A toolkit to support this approach has been developed as part of this thesis and its foundations and design rationale will be given in the remainder of this chapter. It has been evaluated experimentally on the present system, the results of which are given in chapter 8.

7.1. Component Modeling for Distributed Systems

Two primary issues usually motivate construction from modules: Firstly, in integrating a core algorithm into a larger system, often a significant amount of so-called "glue" code is necessary. This includes, for example, conversion between representations, setting up and using communication channels, etc. Such code is dependent, at the very least, on a particular middleware and the particular system decomposition into components. Therefore, it is in general desirable to keep it separate from the algorithm implementation.

Secondly, most algorithm implementations are based on utility libraries, such as data structures, mathematical tools, etc. These libraries have to be embedded again

[1]In the following, "component" will be used to denote a *distributed* component and "module" will be used to denote a functional unit *within* such a component.

and again in different places, often just in different, sometimes even fixed parameterizations.

The first issue is usually approached through layering, and the question for a component model is how to identify connecting points and satisfy them upon construction. The second issue is dependent on the modeling method and in the present approach, inspiration has been drawn from the data-flow model and event-driven programming. The foundations of these will be now be introduced, to be followed by the description of the combined toolkit.

7.1.1. Graph-Oriented Program Models

Treating modules as graphs dates back at least to the dataflow model of the 1970s, which, according to a survey by Johnston, Hanna and Millar, was motivated by the desire to exploit hardware-level parallelism (Johnston et al., 2004). In this model, every instruction is a node in a graph, with the arcs transporting data between nodes. Therefore, with suitable hardware, every instruction may be executed in parallel, when its inputs are available.

When conventional imperative languages proved difficult to compile to these novel architectures, matching languages were developed. Over time, the previously radically different models were realized on standard systems through multi-processors and software threads.

Unfortunately, these early dataflow approaches did not achieve the anticipated improvements because their theoretical assumption (of hardware-level parallelity) is not realizable. They operated at a level that was too fine:

> While von Neumann architectures operate at process-level granularity (i.e., instructions are grouped into threads or processes and then executed sequentially), dataflow operates at instruction-level granularity (Johnston et al., 2004, section 3.2).

Such instruction-level granularity added huge overhead to each instruction and Silc et al. succinctly summarized the situation as follows: “pure dataflow computers [...] usually perform quite poorly with sequential code.” (cited after Johnston et al., 2004, section 3.2).

Fortunately, by this time, dataflow languages had achieved an appeal in their own right and

> The reason for the decline in dataflow research in the late 1980s and early 1990s was almost entirely due to problems with the hardware aspects of the field. There was little criticism of dataflow languages[...](ibid)

In fact, there is a large number of visual, data-flow oriented languages today, some of which are the de-facto standards in their field. Examples include NI LabView[2] for laboratory data analysis, Mathworks Simulink[3] for simulation and mod-

[2] `http://www.ni.com/labview/`
[3] `http://www.mathworks.com/products/simulink/`

eling, Max/MSP[4] for music processing and synthesis, StreamBase Studio[5] for event processing (particularly financial data), and many others.

These languages create new components from existing functions – where these functions typically contain many instructions, thus avoiding the performance problems. This suggests that the graph-oriented view may be a suitable abstraction level for module interconnection. Furthermore, the graph is a suitably abstract representation of the functionality of a component, which lends itself well to inspection and automated analysis or transformation.

Analysis

For the design of a novel toolkit, several facts from the historical overview seem particularly relevant. The first is the *granularity problem*: The level of individual instructions is much too fine for both performance and development. While it is still an open question what a suitable coarser level could be, many toolkits sidestep the issue by distinguishing "host" and "coordination" languages: The nodes themselves are implemented in a conventional, imperative language, the "host" language. In contrast, the connectivity of the graph is specified through a simpler, often visual, "coordination" language.

Furthermore, it is probably not accidental that all the successful dataflow oriented toolkits are domain-specific. They come with extensive domain-support libraries, which can be immediately plugged together to demonstrate the framework's value. The domain-specific constraints provided may also simplify execution (compare section 7.2, particularly 7.2.4).

7.1.2. Reactive Systems

Common to the dataflow approaches, with their processing nodes and data flowing through the graph is a view that Harel and Pnueli (1985) have called "transformational".

> A transformational system accepts inputs, performs transformations on them and produces outputs (Harel and Pnueli, 1985, p. 479).

In contrast, they pose the concept of a "reactive" system:

> Reactive systems, on the other hand, are repeatedly prompted by the outside world and their role is to continously respond to external inputs[...] A reactive system, in general, does not compute or perform a function[6], but is supposed to maintain a certain ongoing relationship, so to speak, with its environment (Harel and Pnueli, 1985, p. 479).

[4]`http://cycling74.com/products/`

[5]`http://www.streambase.com/products-StreamBaseStudio.htm`

[6]In the mathematical, not the general sense.

The terminology used here may be slightly misleading for robotics, because robotic systems always include both of these aspects. It might be clearer to describe transformational and reactive as complementary views on a system. Essentially, the distinction Harel and Pnueli are making is that the transformational view emphasizes the computation that a system performs, irrespective of where its input comes from and what its output achieves. In contrast, the activities of a reactive system are all about maintaining the relationship to the environment, irrespective of how that is achieved. In robotics, the *interaction with the world* embodies the reactive aspects.

An example of how this is realized in robotics is Brooks' early Subsumption architecture. While Brooks focuses mostly on the data-flow between the components of his architecture, each of these components internally contains a finite-state-machine maintaining, in the reactive fashion, a relationship with its environment (including the rest of the system) (Brooks, 1986, see p. 22).

To describe reactive systems, they propose what is, in essence, an inverse view to dataflow: The statechart (Harel, 1987). Nodes in statecharts represent the possible states and functions are carried out during transitions. What state-charts add is the concept of an "external input", to which the system reacts. This input specifies which of the paths out of a state is taken.

Analysis

The dichotomy between transformational and reactive systems is an important one and, in fact, the entire previous chapter has been devoted to a pattern that manages state between components. That said, the management of state depends on the level at which a system is viewed. A serial device driver, for example, needs to keep internal state to interpret incoming data, but can hide much of this state from the rest of the system. This rest, in turn, may have to keep higher-level state, only some of which is communicated externally.

Thus, it is the present author's opinion that the state-centric view is useful for describing the *boundaries* of a system, whereas the flow-centric view is useful for describing the *sequence of actions* that occur during state-transitions. A consequence of this view is that the toolkit, which is primarily flow-based, must incorporate functionality to interoperate with reactive processing.

7.1.3. Computer-Aided and Model-Driven-Engineering (MDE)

The previous sections introduced dataflow oriented graphs and state-machines as basics for modeling software. However, this is not to say that the goal of this chapter is a general software modeling approach. Such tools have been tried in the 1980s under the name "computer-aided software engineering" (CASE) and while they generated a lot of publicity, they were ultimately unsuited for serious software development (Schmidt, 2006).

While Schmidt attributes some of the problems of the CASE approach to the "paucity of the underlying platforms", which necessitated overly complex code generation, he also mentions the problems of a “one size fits all” approach, which is too generic and non-customizable (Schmidt, 2006). It is notable that this is almost the exact opposite of the domain-specific dataflow frameworks mentioned previously.

Model-Driven-Engineering, in contrast, attempts to close the semantic gap through modeling for domain support, or *meta-modeling.* Domain-specific modeling languages are created to aid developers in these domains (Schmidt, 2006; Balasubramanian et al., 2006). Models are built using these meta-modeling languages and translation engines then transform them to code in a host language, which can be executed normally. While at the moment these host languages are predominantly imperative, the models usually contain information that could be used to generate for other targets.

While such domain support is an important step in the forward direction (creating code from models), it does not improve the so-called “roundtrip engineering”, which updates models from code. Furthermore, the integration with host languages for node implementation in MDE tools is generally not as advanced as in dataflow-oriented environments, owing to the higher abstraction level of MDE. It remains to be seen whether this gap can be closed once MDE tools become more mature and can benefit from their generality or whether domain-specific tools will remain superior, despite their smaller user base.

7.2. Execution models

In the previous overview of modeling approaches, two means for execution have been presented: Translational ones, which transform the model to an executable form and direct ones, where the model is executed by a suitable interpreter, which may also be in hardware. Naturally, these can be combined to form hybrid execution engines and in some cases, such hybrids are created directly by developers using host and coordination languages in conjunction. In the following, the various methods will be discussed in more detail.

7.2.1. Classical data-flow model

Classically, data-flow is *data-driven*: Whenever the necessary data is available at the input(s) of a node, it will be processed and the result, if any, is placed on the output arc(s). Execution thus follows the data, as it flows through the graph – hence the name.

The necessary data is determined by a node’s “firing set”, which specifies the inputs that have to be available. Furthermore, flow graphs where every input causes one output are called “well behaved”. In the flow analogy, nodes that take input from other nodes are “downstream” with respect to these nodes. Parallelism can be achieved when upstream nodes do not have to wait for downstream nodes to be finished.

In practice, this pure model has the problem that it requires intermediate storage for data which is not yet processed. When node execution is completely decoupled, the storage required can become unbounded, unless additional constraints are met. Furthermore, it may perform unnecessary processing, e.g. when the data produced by upstream nodes is not actually used.

The pure data-flow execution models assume the engine to be in the hardware. In the absence of such hardware, or when mapped to coarser granularity, a process network engine is usually used (see below).

7.2.2. Kahn process networks

One approach to keep storage requirements from growing and prevent unnecessary computation is "demand-driven" execution, used, for example, in Kahn process networks (KPN) (Kahn and MacQueen, 1977). Here, nodes are called "processes" and they are connected by "channels". Kahn defines channels to "*behave* like unbounded FIFO queues" (Kahn and MacQueen, 1977, my emphasis), but they do not have to unbounded in practice.

The difference in the KPN model is that processes only become active when their output is requested, i.e. on demand. They then produce some data, and will not be activated again until this output has been consumed.

The disadvantage of pure demand-driven execution is that it only exploits parallelism between different input paths, not within one path. To allow parallelism within a path, Kahn suggests the use of an "anticipation coefficient A(C)", for a channel C (Kahn and MacQueen, 1977, section 3.1). Then, a node that has been activated will produce not just one but A(C) items of data, in anticipation of future use. This enables parallel processing whilst bounding storage requirements.

Furthermore, due to the demand-driven activation, the dataflow concept of "firing sets" does not make sense in KPNs. However, KPNs explicitly query their input channels, which creates an implicit internal condition to the same effect.

7.2.3. Communicating Sequential Processes

A similar, but slightly different approach, the "Communicating Sequential Processes" (CSP) model, is due to Hoare (1978) and Brookes et al. (1984). In this model, processes execute independently and are always active. However, when communicating, they block until the remote process is also available for communication. This reduces storage requirements and provides well defined synchronization semantics.

CSPs are also used slightly differently: Whereas in the dataflow approach, the connections between the nodes are created externally, CSPs specify the nodes they would like to communicate with internally. Depending on how the remote processes are named, this may require more knowledge about the surrounding system.

Furthermore, while the original data-flow approach was based on nodes executing primitive operations, CSPs are clearly high-level programs. However, since the original formulation, the data-flow concept has also developed to encompass nodes with coarser granularity, so this distinction is vanishing.

7.2.4. Synchronous Data Flow

A different solution to the practical problems is to restrict the dataflow graph in ways that make execution statically schedulable at compile time. This is done in the Synchronous Data-Flow approach due to Lee and Messerschmitt (1987). Its essential aspect is that the number of input and output items consumed respectively produced are pre-specified. Based on this specification, the execution order can be pre-determined. This is called "static" scheduling, with the traditional approach being called "dynamic" scheduling in contrast.

There are fairly elementary nodes with asynchronous behavior, such as conditionals. The SDF approach handles these by creating synchronous sub-graphs, which are statically scheduled, and selecting the correct schedule at runtime (Lee and Messerschmitt, 1987, section V.C).

7.3. The Filter-Transform-Select (FTS) Toolkit

Having presented the main influences, the following section will introduce a novel toolkit for component construction. Its design is primarily based on the dataflow idea, hence it is transformational in nature. In addition, a taxonomy will be introduced that classifies nodes as filtering, transforming or selecting and the filter and selection nodes, in particular, are intended to provide a bridge to the reactive aspects of a component. This taxonomy has also led to the name "FTS", which stands for "filter-transform-select".

First, the requirements and the design of the toolkit addresses will be introduced. Both technical and social aspects will be considered, the latter of which we consider particularly important for adoption. After that, the structure is introduced and two different execution models are compared.

7.3.1. Historical Overview

Two precursors had been created prior to designing the current toolkit. As a first experiment, the present author developed a so-called "conditional dispatch" library, which allowed incoming events to be assigned to handler functions based on a set of conditions on the content of the input. This is similar to pattern matching declarations in functional languages such as Haskell, but has been realized for XML documents using XPath for pattern matching.

The conditional dispatch library could select either the first handler from a list, or all matching handlers. Importantly, it was considered an error if no match was found. Therefore, it realized a *selection amongst options*. Furthermore, the conditions included state variables and support was present to easily *modify state variables across a number of conditions*. While this was not designed with reactive programming in mind, the association – and similar usage patterns – quickly appeared.

At the same time, Sebastian Wrede implemented so-called "message-transforming-function trees" in an experimental version of the XCF middleware framework (Wrede, 2008). It was based on a *chain of transformations* to be associated with an event-selection expression acting as a *filter*. Several such chains could be combined in a tree and evaluated in parallel.

The foundational experiments have been merged conceptually in the Filter-Transform-Select toolkit, created jointly by Sebastian Wrede and the present author (Lütkebohle et al., 2009b). It combines the ideas of selection amongst alternatives and chained transformations. Furthermore, the recognition of a common need caused the creation of a solid representation in the form of graph which could support loops. This laid the ground for a flexible component modeling approach which can be applied beyond the limited applications of the previous implementations.

7.3.2. FTS Requirements & Design

From the historical development, it is already possible to discern two major functional requirements: Firstly, in event-based components the idea of a condition, which decides what further processing to perform, appears again and again. In many cases, such conditions are embedded within other code, which makes them hard to change. Therefore, a means to separate these conditions from the handler code was desired.

Secondly, a lot of recurring glue code, such as (un)marshaling, data extraction and fusion of data and so on, is naturally described as a chain of transformations. This code occurs so often that just calling a sequence of external functions becomes tedious. Furthermore, the use of different middleware toolkits, or none at all, means that different transformational chains are used with the same core algorithm. Therefore, it was desired to separate this code from the core and make it easily changeable at the same time.

Parallel and sequential execution models

One of the most influential requirements has been flexibility with regard to execution models. The primary reason for this flexibility is to support components with different requirements regarding timing, concurrency and performance. While process networks are very flexible and sometimes required, they are neither the most efficient nor the most predictable execution model.

Furthermore, many parallel execution models leave synchronization aspects up to the developer. When nodes are not side-effect free, this can be a difficult problem and it may be desirable to ensure serialized execution.

Thus, it is required to have at least one parallel and one sequential execution mechanism. For the parallel mechanism, a *data-driven process network* approach with FIFO queues is to be used. For the sequential mechanism, a breadth-first graph traversal algorithm is employed. The reason for this choice is to be found in the way fusion is realized: When a node requires multiple inputs, these are collected through fusion nodes, and then passed as one to the target node. To allow rate adaptation, these fusion nodes also support caching data. If a depth-first traversal were used, this caching would lead to the target node being called directly once the first of the inputs has been updated. This is only desired when no updates for the other inputs are available in this time-slice. Breadth-first processing, however, gives the other data inputs a chance to update the fusion node's cache before further processing occurs.

A functional node interface

The node interface is one of the most crucial design aspects. On the one hand, it must be powerful enough to encapsulate a variety of functions. On the other hand, it must be simple enough to be easily, and correctly, usable. The latter requirement is particularly important to facilitate adoption of the toolkit, as discussed above.

The following choices have been made to realize this:

- *Informational annotations.* Some execution models (such as synchronous dataflow) require additional information. However, such information should be optional, to keep the minimal interface simple and clean. Thus, the use of *external annotations* has been chosen, whose addition is optional.

- *Functional interface.* Different execution models may require different data exchange mechanisms, such as FIFO queues, shared buffer or direct parameter passing. Thus, the base mechanism must be versatile enough to support all of these without being too complicated. For this toolkit a *functional interface with single input/output* has been chosen.

While restricting the interface to single inputs and outputs appears restrictive at first, the inputs and outputs can also be structures containing multiple elements, so it is not really restriction, just an interface simplification for the execution engine. Sometimes, the application domain may provide such a structure naturally, for other cases, the framework supports lists and maps as generic data structures. Furthermore, when several output elements are requested, it may be a useful optimization to compute them incrementally.

Marking layers and hierarchies

Layering, with information hiding between layers, is a well accepted design principle and should be supported in the toolkit. When connectivity is externally specified, information hiding between nodes is already realized, because the nodes are not aware of where their input comes from, and have no direct access to the source object. However, the graph structure still needs to be known across layers, because, for example, to connect two sub-graphs from different layers, their output and input nodes are needed.

Thus, to hide the graph structure, a means to *mark input and output nodes* in a subgraph is to be part of the toolkit. These markings must contain enough information to establish connections between layers automatically, which will include at least the *type of the data exchanged.*

Furthermore, layering usually prohibits connections between non-adjacent layers to reduce coupling and improve abstraction. As connectivity is already externally specified in the current approach, such coupling would be on the data-structure level only. This is considered acceptable for the present design and might even enable beneficial optimizations.

However, it may certainly be the case that a lower layer produces data of the same type as a higher layer, e.g. when the higher layer performs filtering or smoothing. If connections are purely made on the basis of type, this could create accidental connections. Therefore, a means to *mark the layer of a node* is necessary. If desired, this specification could then be used to prevent non-adjacent layers from communicating, though this is not done in the present implementation.

Performance estimates

In the chosen process model, passing data between nodes occurs through blocking queues, specifically the `java.util.concurrent.ArrayBlockingQueue` introduced with JDK version 5. While queue overhead is low, it is still orders of magnitude higher than function call overhead, so an early concern was that this overhead might become too costly. Fortunately, this concern could largely be put away, due to the following considerations.

Early estimates have been that a typical data fusion component working on vision data could come to about 3,000 node executions per second, and a serial control component up to about 10,000 executions per second. As it turned out, these numbers overestimated the number of nodes required considerably, and in practice numbers about one order of magnitude lower are actually observed. Still, it is better to err on the high side and also leave some room for growth, so they were used for an estimate to ensure future scalability.

Benchmarking data-passing with multiple threads and contention is a complex subject, but from measurements available in the literature, it is known that the locking primitives used for such queues take between 30 and 400 nanoseconds (ns), depending on contention, on Intel Pentium4® class processors (Lea, 2005, tables 2 and 3). Further, data-transfer through this kind of queue takes about 5,000ns or 5µs an AMD Opteron® class processors (Scherer et al., 2006, figure 3).

This means that the transfer of 10,000 messages through such a queue is estimated to take about 50 milliseconds, for an overhead of 5% when transfering this number of messages per second. This is significant, but considered tolerable. Measurements on the overall framework have since confirmed that it remains within this range. Thus, we can conclude that while the overhead is noticeable, it it is certainly within a reasonable range.

Furthermore, Java's built-in APIs are by far not the most efficient for these applications. Due to the popularity of the message-passing model for concurrent processing, vast performance increases in the underlying primitives have been achieved. For queues in Java, implementations are now available that reduce the overhead by a factor of at least three (Scherer et al., 2006, figure 7). Furthermore, new event-processing-oriented frameworks have appeared that achieve up to 25 million messages per second on generally available machines (Thompson et al., 2011). This means that, should message-passing overhead become a problem in the future, huge room for growth is available by moving to one of these frameworks.

7.4. (De-)Composition Principles

Having given a number of technical considerations for the toolkit design, the following section will focus on aspects which have been designed to provide developer support for easy composition of components from modules. As the history of CASE tools has shown, the social and economical aspects are crucial for adoption, yet most research has focused solely on technical concerns. Besides a number of design aspects, this fact has also been the motivation to study the concrete uses of the toolkit, the results of which are presented in chapter 8.

A general principle has been to keep the toolkit small and easy to understand. One consequence is that the toolkit has not been designed as a development environment, but as a framework-level library. Furthermore, a crucial goal was to keep the size of the API limited, with the original goal specified loosely as “maximum of 10 classes or interfaces”. This could not be realized by far, but its spirit has been followed by keeping the size of the *user-visible* core API in that range and through minimal interfaces.

7.4.1. Decomposition for re-use

While the concept of a graph with connected functions can encompass many types of programs, this generality is not without pitfalls. As mentioned previously, the granularity has consequences for performance. However, more importantly, from the viewpoint of a developer, it is not clear how to decompose module into nodes.

This may only be a minor issue, depending on the application area. For example, in more structured domains, such as signal processing, typical decomposition strategies may be well established. Furthermore, structured programming suggests particular control strategies, such as conditionals and loops, which also have relevance for the execution flow. Thus, it might appear natural to decompose along these lines.

However, for novel domains decomposition are still unclear and, in general, a decomposition on the granularity of individual control structures is considered too be to fine.

Therefore, a decomposition into filters, transformations and selection nodes is advocated (Lütkebohle et al., 2009b), for the following reasons:

- *Conditions vs. algorithms* One of the primary aspects in integrating legacy components is to change the conditions (hence, filters) under which the algorithms (the transformations) are applied. For example, a behavior that is executed in just a single condition on one scenario might be used for several conditions in another scenario. Such changes should be possible on the level of the processing graph and that is only possible if conditions and transformations are separated[7].

- *Selection vs. Filtering.* The "select" distinction is to be used whenever there are multiple options, at least one of which must be taken. This is distinct from a filter step, which is used to determine whether the conditions for further processing are met. For a select node, it is an error if no condition is met, whereas for a filter node, this is a normal case. This restriction, and its use in the framework, facilitates a *structured dataflow* design that prevents errors, because processing is guaranteed to continue after the select.

Besides these distinctions, the decomposition also depends on the level of interest. For example, an algorithm often contains many conditions. As outlined above, describing a graph structure on that level is considered too fine. Therefore, the decomposition into filters, transformations and selectors is suggested to be performed at the *highest possible level*, just below the component boundary. In effect, the inputs for filters and selectors should be externally provided data.

That said, it is certainly envisioned that the toolkit may be employed in a hierarchical manner to model the sub-module level – possibly using a different execution model. In such use, the decomposition level would have to be different, but that is beyond the scope of the current work.

Learning curve and testing support

The inverted control-flow of a framework may hamper testability, if care is not taken. Instead of the developer writing code that calls library functions, the developer implements interfaces, which are invoked by the framework. This is a well-established approach (Johnson and Foote, 1991), but may slightly decrease understandability and increase the learning curve. Furthermore, in a true process network, many concurrent processing threads may be active, calling nodes in no predetermined order. This can make it much more difficult to determine when and where errors occur.

To reduce the effect of this, the toolkit should include support for isolated execution of a node and stepwise execution of a graph. Furthermore, it should be possible to observe the execution of a graph while it is running, ideally in such a way that the data and execution trace can be captured and replayed[8].

7.4.2. Multi-Level Composition Support

Like many dataflow toolkits, the present toolkit should support dedicated host and coordination languages. A specialized coordination language can reduce the tedium

[7]This does not mean that all conditions can be separated out that way, some of them are essential to the way the transform works. However, many conditions that purely decide whether to perform processing or not are good candidates for extraction. Further, even some of those appearing essential may be extracted by splitting the transform at the same time.

[8]While capturing is indeed supported in the toolkit, replay has not yet been implemented.

of constructing graphs and enable high-level constructs to be expressed succinctly. However, firstly, it is not clear what the optimal abstraction level of a coordination language is and whether it should be primarily visual or textual. Furthermore, a coordination language requires a translation to an executable representation, which is another source of errors.

Therefore, a stepwise approach with three layers has been taken.

1. *Directly accessible base API.* The basic API for creating graphs is exposed to developers, so that they can use it procedurally.
2. *Low-level graph language.* Built on top of the base API, this language should have a one-to-one mapping to nodes and edges, but may already translate from abstract node types to concrete implementations.
3. *Domain-Specific Modeling Languages.* To raise the abstraction for particular domains, a translation from a domain-specific model to the low-level graph language should be supported.

The third layer has been implemented and tested experimentally, but has not been used in the current system. The remainder of this chapter will only address the first two layers.

It can already be said that creating a model programmatically also opened up possibilities to introduce hidden coupling through shared state. While this decreased the advantages to be gained from an explicit model, it also enabled the toolkit to be used in situations which would not (yet) have been supported otherwise.

7.5. Toolkit Implementation

The current implementation consists out of a core engine and a number of support libraries, all realized in Java™ (Gosling et al., 2005). Two of these libraries realize monitoring and the declarative graph specification, they will be described in the following. The remaining libraries are more specific and not relevant here.

7.5.1. User-Visible API

The elementary types in the FTS toolkit are specified through interfaces, except for the model itself ("Graph") and a generic execution runner ("EngineThread"). Abstract base classes implementing the interfaces are provided as the default extension points for users of the library. The associations between these and the interfaces are shown in figure 7.1.

The main node interface is, obviously enough, called "Node", with a number of sub-types according to the roles of the nodes in a graph. Two of these ("Source" and "Sink") are essentially tagging interfaces, adding no functionality. The other two, "AbstractFilter" and "AbstractTransform", are abstract base classes for the implementation of the corresponding types. Initially, select nodes were constructed from (Filter, Node*) pairs, but as this did not ensure the requirement that at least one path must always be taken, dedicated "SelectFirst" and "SelectAll" types have been added later on. As these classes require no extensibility, they are framework-internal and constructed through a convience factory method in the "Graph" class.

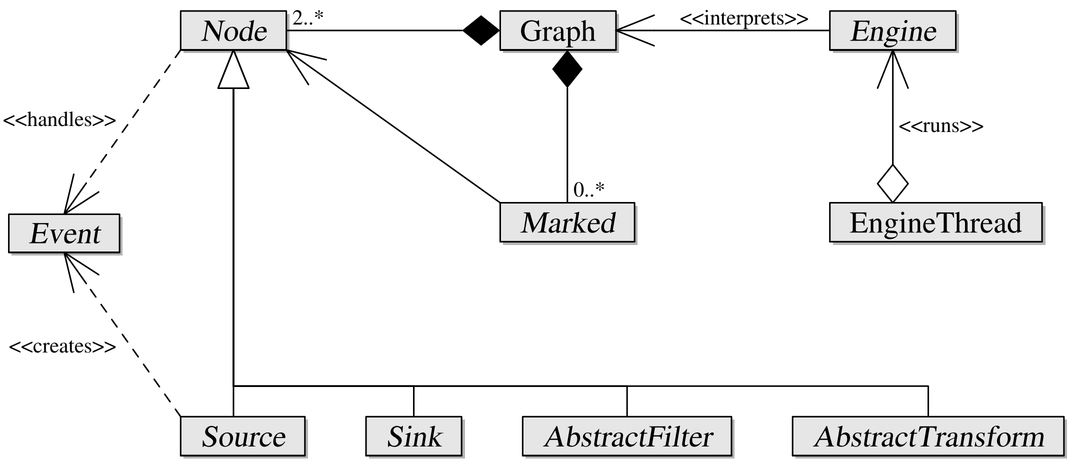


Figure 7.1.: Overview of base types.

Abstract bases and data access

One notable aspect about the base libraries is the treatment of the data communicated. Some internal functionality requires meta-data about data flowing through the graph, which should be hidden as much as possible from the users of the library – both for reasons of simplicity and to prevent changes.

The resulting base classes are shown in figure 7.2. Both these classes implement "handleEvent" and finalize it, to prevent modification in derived classes (Gosling et al., 2005, section 8.4.33). Thus, classes derived from "AbstractFilter" cannot modify their output and classes derived from "AbstractTransform" do not see the Event type at all, which keeps them from accessing and/or modifying meta-data.

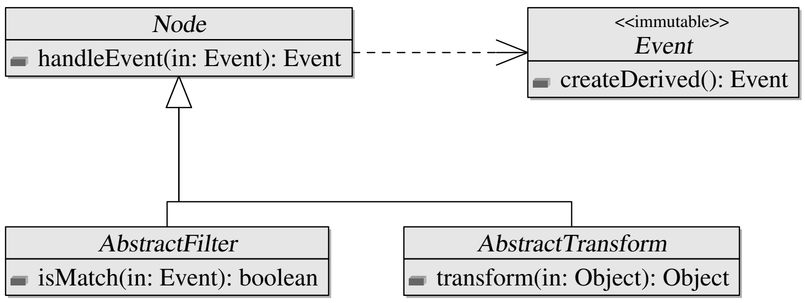

Node
handleEvent(in: Event): Event
<<immutable>>
Event
createDerived(): Event
AbstractFilter
isMatch(in: Event): boolean
AbstractTransform
transform(in: Object): Object

Figure 7.2.: Abstract bases for node implementation.

These restrictions are not strictly enforced – anybody may implement the Node interface directly and modify data. Instead, they are simply provided to prevent users from inadvertently making errors.

Model Functionality

The model stores the graph connectivity information in an adjacency-list representation. While this choice is not exposed through the interface, the available operations depend on this implementation to be efficient. The relevant methods of the "Graph" class are shown in figure 7.3.

Apart from the nodes and edges making up the graph model, the graph also stores marker information which can be used to establish connections between layers without knowledge about the structure of the participating graphs (cf. section 7.3.2). When such marks are present, they may cause new graph edges to be created when merging graphs. At the moment, merging is implemented based on types and, optionally, a scope. Marks are optional and Graphs can also be merged without marks being present.

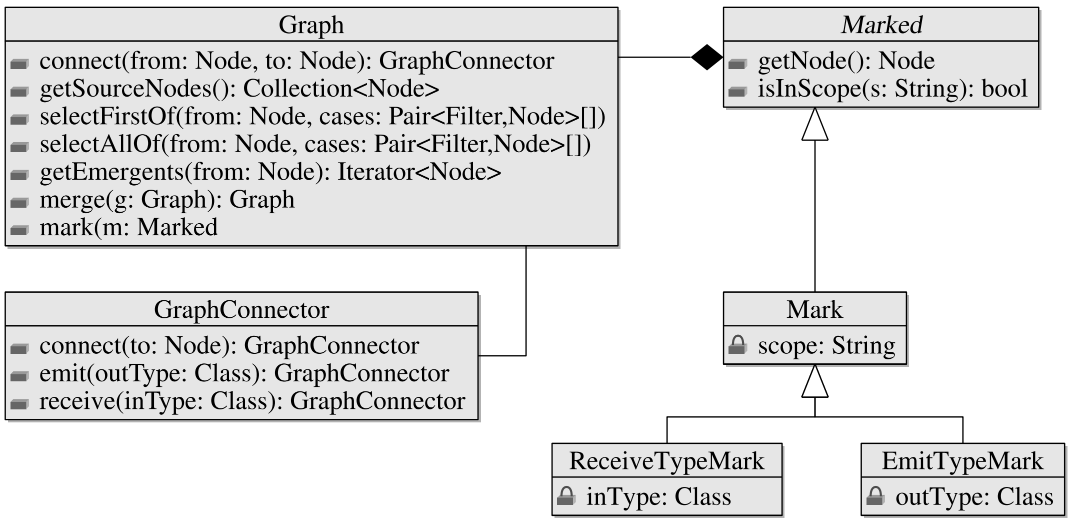

Graph
connect(from: Node, to: Node): GraphConnector
getSourceNodes(): Collection<Node>
selectFirstOf(from: Node, cases: Pair<Filter,Node>[])
selectAllOf(from: Node, cases: Pair<Filter,Node>[])
getEmergents(from: Node): Iterator<Node>
merge(g: Graph): Graph
mark(m: Marked
Marked
getNode(): Node
isInScope(s: String): bool
GraphConnector
connect(to: Node): GraphConnector
emit(outType: Class): GraphConnector
receive(inType: Class): GraphConnector
Mark
scope: String
ReceiveTypeMark
inType: Class
EmitTypeMark
outType: Class

Figure 7.3.: Graph and associated interfaces.

To simplify manual creation of the graph, a "GraphConnector" class is provided for creating node chains. It allows continuing a connection made using the "Graph" connect method and/or add marker information to the last node added. Besides reducing the amount of information that has to be passed, this connector class also facilitates creating objects only within the function call themselves, which can again reduce clutter, because declarations are elided.

State Support

Use of some nodes may require activating hardware or other background processes, some of which may start producing data immediately. Therefore, such activation must not occur prior to running the graph, which is performed by the engine. The Activatable interface, shown in context in figure 7.4, facilitates this.

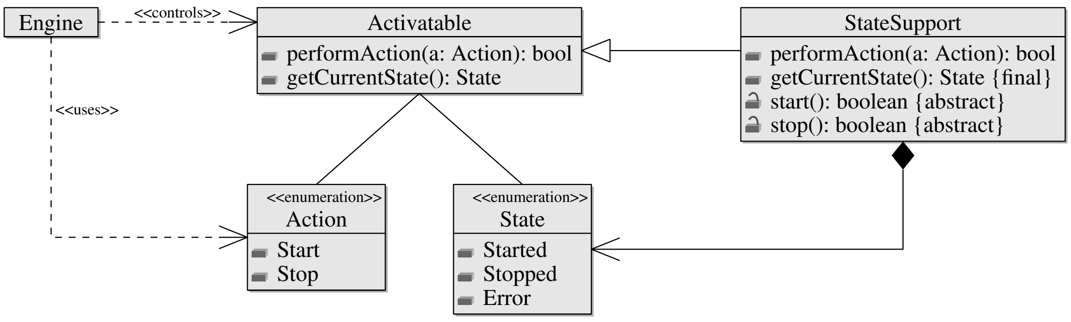

Engine
<<controls>>
<<uses>>
Activatable
performAction(a: Action): bool
getCurrentState(): State
StateSupport
performAction(a: Action): bool
getCurrentState(): State {final}
start(): boolean {abstract}
stop(): boolean {abstract}
<<enumeration>>
Action
Start
Stop
<<enumeration>>
State
Started
Stopped
Error

Figure 7.4.: Engine and state-support.

The activatable node is initially in the "stopped" state and the obvious transitions are defined as shown in figure 7.5. All actions will are triggered on exit, to indicate that the next state is not yet reached. If an action fails, the error state is entered. An alternative model for this state machine would have been to include "Starting" and "Stopping" states. However, this could be interpreted to suggest asynchronous execution, which is not the intention.

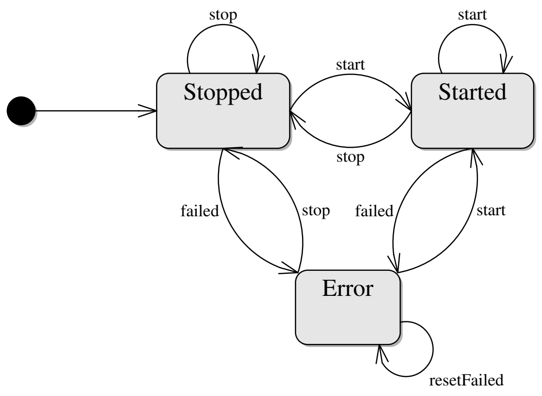


Figure 7.5.: State machine for Activatable.

While the state-machine itself is straightforward, the fact that errors may occur on any change results in substantial code-repetition. Furthermore, it is a well-accepted principle to enable extensibility through the use of abstract base-classes. Therefore, a support-implementation for the state support has been provided, which may be used to delegate state-changes to.

7.5.2. Declarative Graph Specification

Having introduced the host language API, this section describes a means of graph construction using declarative statements. This work consists of two parts: A fundamental graph construction language, including toolkit support to construct graphs from this language, and support for transformations from domain-specific languages to this language. While some experiments with such domain-specific languages have been undertaken, the focus here will be on the fundamental graph construction language only.

Apart from the graph's connectivity, the graph declaration should also contain the necessary information to create the nodes of the graph. This includes the type of the nodes and their construction arguments. While the current implementation is in Java, the description should allow other languages. Hence, the use of a type-mapping is employed, which maps generic names to the implementation-specific types.

A number of general graph description languages exist, including, for example, GraphXML (Herman and Marshall, 2001), the Graph eXchange Language Holt et al. (2002), the GML format (Himsolt, 1999) and the dot format used in the Graphviz system (Gansner et al., 2009). It was desired to use a well-known parser infrastructure, which suggested of XML-based formats, e.g. GXL or GraphXML. In their direct form, these general formats tend to generate fairly large files, which hinders manual construction. Therefore, a custom XML-based format with a mapping to the GXL has been chosen.

The primary objective for the language specification has been brevity for typical use-cases. In particular, the connectivity is inferred from the sequence of the document, unless specified otherwise.

Shortly, the language has the following elements:

model	– Root element.
node	– Declares a node – used for all types of nodes.
select	– Encloses a selection block.
target	– Encloses a single selection target.
filter	– Declares a filter node for selection.
arg	– Declares a node construction argument.
fuse	– Introduces a fusion node.

The full document type definition (DTD) is provided in appendix C.1.1.

Graph Specification Examples

A very simple graph with just two nodes, a source and a sink, is specified as follows:

```
<?xml version="1.0" encoding="utf−8"?>
<model>
  <node type="FibonacciSource"/>
  <node type="QueueSink"/>
</model>
```

Listing 7.1: Queueing Fibanocci numbers

In listing 7.1, the first node is one that produces numbers from the Fibonacci series as output and the second node is one that consumes its input and places it in a queue. They are implicitly connected in the order shown.

A more complete example is given in listing 7.5.2. It demonstrates the language support for selection expressions and explicit linkage through named nodes and the “source” attribute. Functionally, the listing is somewhat artificial: It declares two sources, both of which produce XML documents, but with differing content. The select statement then decides amongst two possibilities of selecting content from these documents and the remaining nodes print out the selection and place it into a queue again.

As can be seen from the listing, this way of creating flow-graphs is very low-level – too low-level for most purposes. Therefore, it is more likely to be produced by tools. However, it can be used manually, if required, and this is how the examples in this thesis have been produced.

```
<?xml version="1.0" encoding="utf−8"?>
<model>
  <node name="fib" type="XMLFibonacciSource"/>
  <node name="time" source="null" type="XMLTimestampSource"/>
  <select name="choice1" source="fib,time" selectMax="1">
    <target>
      <filter type="XPathFilter">
        <arg type="StringChild">/fibonacci</arg>
      </filter>
      <node type="XMLPrint"/>
      <node type="XPathTransformSingle">
        <arg type="StringChild">number(/fibonacci/value)</arg>
      </node>
      <node type="Print"/>
    </target>
    <target>
      <filter type="XPathFilter">
        <arg type="StringChild">/*</arg>
      </filter>
      <node type="Print"/>
```

```
        <node type="XPathTransformSingle">
          <arg type="StringChild">string(.)</arg>
        </node>
        <node type="Print"/>
      </target>
    </select>
    <node name="static" source="null" type="StaticSource">
      <arg type="StringChild">something else</arg>
    </node>
    <fuse sources="choice1,static" required="choice1,static"/>
    <node type="QueueSink"/>
</model>
```

Node construction

The largest implementational issue when processing a graph specification as given above is to construct the nodes within. Firstly, a mapping from the type-name in the specification and the real type name must be specified. This is to be done in a way that allows extension by developers without modifying the core toolkit. In the present work, the implementation is based on the Java(tm) Service Provider Interface (Sun, 2009b). While this API is not available for all languages, it does not make any language-specific assumptions and could be provided for other languages easily.

More troublesome is the matching of arguments to class constructors, which is based on reflection. Many languages, including C and C++, do not offer run-time reflection in an easily usable way. For such languages, translating to source code and compiling statically may provide a simpler avenue to support declarative graph specifications. This is done, for example, by Ptolemy II (Bhattacharyya et al., 2005).

7.5.3. Execution

In the current toolkit implementation, the graph-based program model is executed through interpretation: An engine class queries all sources, determines the recipients of the input data, invokes them and cycles until all data has been consumed by sinks. Then the overall cycle begins anew. If, during execution of a node, any error occurs, an error handler is invoked which may decide whether execution is terminated or continues.

Sources are queried on a background thread using the JDK-standard "CompletionService". This service queries all sources in parallel and delivers their data into a queue once it becomes available. In particular, this ensures that multiple sources are queried at approximately the same time (though this is not a hard real-time guarantee).

The basic execution loop, including the initialization of activatables, the submission of sources and the loop that distributes input data is shown in figure 7.6. This loop is shared by all current engines.

At the time of this writing, two different types of engines exist. They differ in the level of parallel execution, with one realizing a sequential and the other realizing highly parallel execution.

Figure 7.7 shows an overview of the visible engine classes. Both engine implementations are derived from an abstract base class “AbstractEngine” which realizes the common functionality, such as initializing “Activatable” nodes, querying sources and handling errors. Furthermore, a convenience factory exists to create these classes, which is also used by the default engine runner, “EngineThread”.

Breadth-first engine

This engine utilizes a single thread and traverses the graph in breadth-first order, executing nodes as they are encountered.

Execution. Nodes are executed when an input item for them is at the front of the shared input queue.

Concurrency. None.

Output handling. Output of node execution is duplicated once per child and added at the end of a shared FIFO queue. Between children, the order is not fixed.

Predictability. Due to the breadth-first strategy, all first-level children of a node are guaranteed to be executed before any of the deeper levels.

Input strategy. Sources are queried when the queue is empty, i.e. once the previous input data has been completely processed.

Due to these characteristics, the breadth-first engine realizes a fairly space-efficient, predictable execution but without any concurrency. The processing speed is determined by the slowest node, with the advantage that overruns are avoided.

In the data-flow model, this engines realizes an execution that is fairly close to that realized by naive synchronous data-flow, though scheduling is still dynamic and not as efficient.

Queuing Engine

In contrast, the queuing engine is closest to the pure data-flow execution model: It starts one thread per node.

Execution. Nodes are executed when an input item is available at the front of their respective input queue.

Concurrency. Multi-threaded, one thread per node.

Predictability. No guarantees.

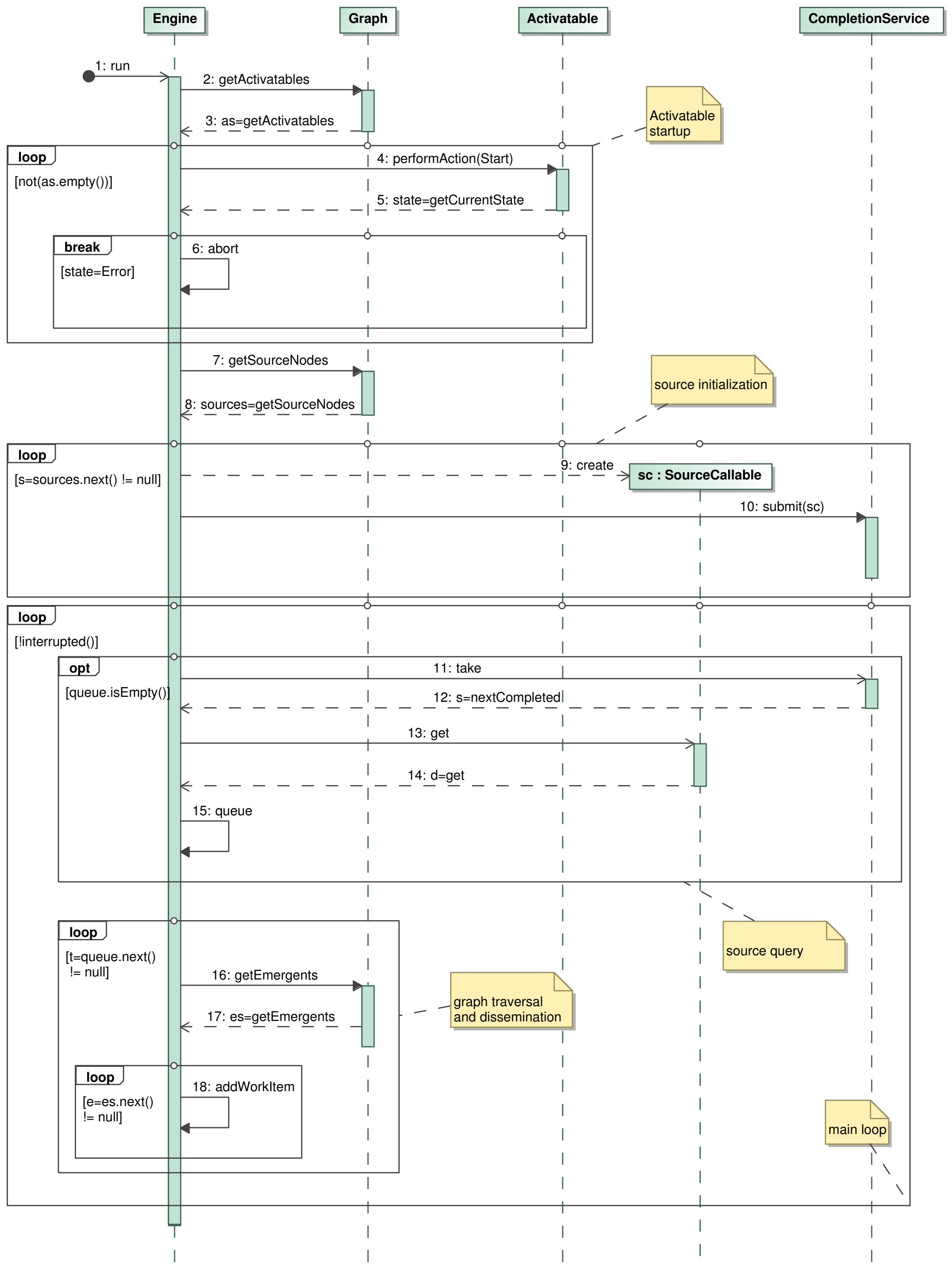


Figure 7.6.: Basic engine execution loop. "addWorkItem" is to be provided by the sub-class.

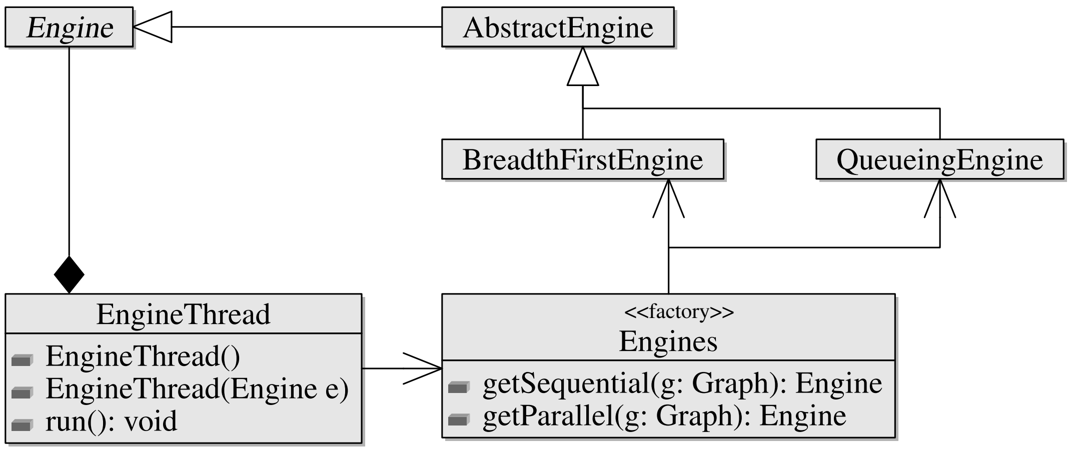


Figure 7.7.: Overview of FTS Engine classes.

Output handling. Output of node execution is first added to a share FIFO queue. A dedicated thread takes it from there, duplicates it once per recipient and adds it to the FIFO queue of the recipient node. There is an option for discarding all but the last item.

Input strategy. Sources are queried once the previous data has been distributed to the first-level children.

This engine realizes a fairly simple model for higher concurrency. The use of one thread per node is not optimal, but guarantees that each node is executed sequentially, which simplifies implementation and guarantees FIFO ordering. The processing speed is determined by the sources, i.e. it is data-driven. This may cause overruns if sources do not have a natural periodicity or if later nodes are slower than earlier ones.

It should also be noted that the scheduling overhead of the QueueingEngine can be substantial, due to the relatively large number of threads. This is particularly relevant for chain-style graphs, or chain-style subsets and reflects the general issue of using multiple threads for essentially sequential code (cf. section 7.1.1). This issue can, for example, be reduced through aggregation of node-chains into composite nodes.

Multiple engines per graph

An experimental strategy for execution is to use multiple “BreadthFirst” engines per graph. The advantage of this is that it achieves concurrency while avoiding overruns, because sources are only queried once the previous input has been processed. The disadvantage is that it guarantees neither FIFO ordering nor sequential node execution: Due to unpredictable thread scheduling, it may very well happen that an input item received later is processed earlier. Furthermore, two threads may call the same node at the same time.

Despite these problems, this method may be used to speed up processing in cases where input order is not important. As it does not reduce latency, it is only useful to increase throughput in cases where the source data-rate exceeds the throughput with one thread.

7.6. Visualization

One of the advantages of a graph model is that it affords direct visualization. By showing the graph's nodes and edges, labeled with their names and configuration information, a good overview of a component's functionality and structure can be given immediately.

Furthermore, the proposed FTS principle results in a classification of the nodes, and this can be used to pack more, easily viewed information into a more compact display. Based on node-type information, a visualization using the standard UML activity diagram notation (UML2.0, chapter 12) can be produced in a fully automated fashion, as outlined in the following.

7.6.1. Activity Diagram Semantics for FTS Graphs

The graphical elements in an activity diagram are shown in figure 7.8. The elements of an FTS graph are mapped as follows:

1. *Filters* Filters are represented as conditionals with a *single* outgoing edge. Implicitly, the other edge leads to final flow.

2. *Transformations.* Boxes with rounded-corners, the notation for an activity, are used to represent pure transformation nodes a graph.

3. *Fusion.* The AND merge type, shown with a vertical bar as in figure 7.8(a) is used to represent fusion nodes: Data produced is a combination of the inputs.

4. *Select.* Selects are represented as conditionals with multiple outgoing edges, as in figure 7.8(b). In principle, UML activity diagrams do not permit more than one outgoing branch to be taken (i.e., the conditions must be mutually exclusive). This has been relaxed for the present work, however.

5. *Data flow.* Edges represent both control- and data-flow, as the control always follows the available data. When multiple inputs are present without a fusion this is shown as an OR junction, as in figure 7.8(c). When data is duplicated amongst multiple outgoing edges, control flow also splits, indicated by a vertical bar as in figure 7.8(d).

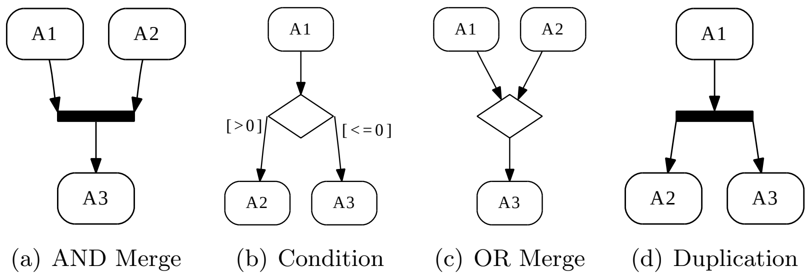


Figure 7.8.: Junction types in UML 2.0 activity diagrams (UML2.0).

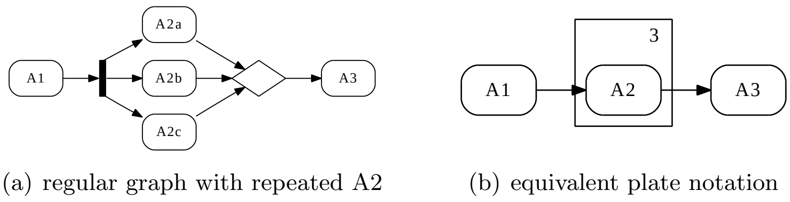

A1
A2a
A2b
A2c
A3
3
A2
(a) regular graph with repeated A2
(b) equivalent plate notation

Figure 7.9.: Plate notation example.

Plate notation

In some of the graphs presented, many nodes of the same type exist in the same position. For example, in the motor control examples shown later, there is one source node per actuator. The visual clutter caused by repetition can make interpretation difficult, which is why an extension of the UML notation has been introduced: Plate notation. This notation aggregates nodes that a) are of the same type and b) have exactly the same predecessors and successors. The visual notation has been adopted from usage in the display of probabilistic graphical models. An example is shown in figure 7.9.

7.7. Summary

This chapter has presented the design considerations and resulting implementation for a toolkit to construct components out of smaller building blocks according to the data-flow principle. Particular emphasis has been placed on application of the toolkit to existing (legacy) components, by suggesting i) a principle for decomposition that is intended to enhance reusability, ii) a simple API for users of the toolkit, and iii) a choice of sequential and parallel execution styles.

While embodying some novel principles, the primary purpose of the toolkit is to support experiments that study the data-flow approach in robot software systems. These experiments will be described in the chapter 8.

8. Data-Flow Case Studies

The suitability of using the FTS approach for component construction will be evaluated in this chapter based on a number of case studies. These encompass regular and data-flow based applications, the latter of which are realized based on the toolkit introduced in chapter 7.

The first objective is to study the differences when realizing the same functionality based on the data-flow approach or using "traditional" component construction. The intent of this comparison is *not* to make absolute value judgments – far too many variables would be a factor. Instead, the intent is to gain insight into what kind of issues to expect when *changing* between approaches.

The second objective is to study the evolution of data-flow based components, to determine whether the claimed benefits hold. In particular, the evolvability and level of re-use will be investigated. This should also serve to identify possibilities for future improvement.

In addition to the metrics introduced in section A.1, the case studies will be based on visual analysis of the graph structure of the components.

8.1. Proof of concept: Data fusion

As a proof-of-concept study, the action selection component has been chosen. This component is responsible for robot initiative, i.e. it creates action proposals at the highest level. It has been developed specifically for the "Curious Robot" scenario, thus it was a natural choice as a case study object.

Moreover, action selection is an important part of any autonomous system. While there are many different methods to make the selection, they all need to base their decisions on input from external sources. Hence, communication and interaction with other components, as well as data fusion is an important prerequisite to achieve such functionality. From an algorithmic point of view, it is also often the least interesting, so a means to reduce the effort needed for integration is likely to be interesting for many such components.

Last, but not least, data fusion typically exhibits a straightforward tree structure. This represents the simplest processing case, i.e. it is the baseline for suitability.

8.1.1. Functional component overview

The functionality of the action selection component can be summarized as follows:

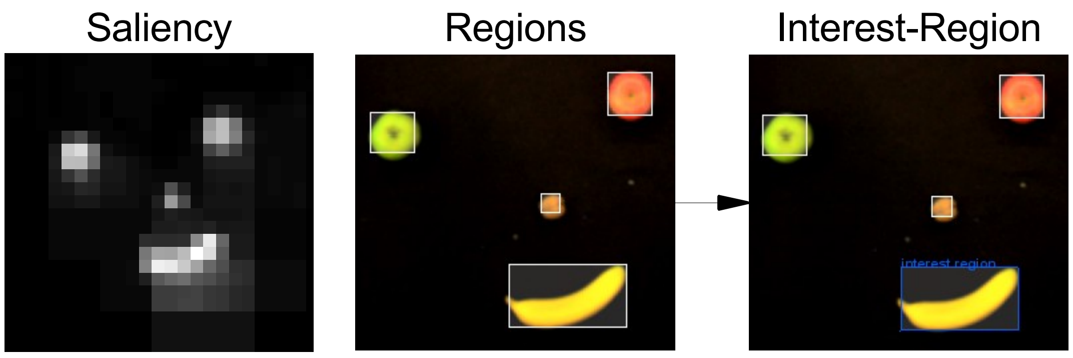


Figure 8.1.: Typical input data (from Lütkebohle et al. (2009a)).

1. *React to new visual events.* Combine visual data of various types based on spatial coherence (i.e., sufficient overlap in the image). This creates a list of merged visual regions.

2. *Rank visual regions according to saliency.* From the list of regions, the visually most interesting one is selected. This outputs a region description. In the present component, saliency information is externally computed using Nagai's method (Nagai et al., 2003b).

3. *Integrate background knowledge.* If this region has been the object of interaction before and some knowledge was acquired, merge it into the region description.

4. *Determine next most interesting piece of information.* Based on what is already known, this determines what piece of information is needed to enable robot activity.

5. *Propose an action that acquires the desired information.* This is where an action proposal for external execution is created and sent.

Example input

A visualization of typical inputs is shown in figure 8.1. The actual input consists of symbolic descriptions, with the saliency given as coordinates with an associated saliency value (cf. appendix C.4.1) and the regions given as a bounding box and, if known, an object detection label (cf. appendix C.4.2).

Discussion

At each of the processing steps outlined, a number of choices is encoded in the implementation. For example, to determine spatial coherence, a common coordinate system, and a common region representation is required. This can be simple, e.g. relative pixel coordinates (to account for different resolutions) and rectangular regions. However, it may easily be more complicated, with coordinate transformations required and/or more exact region descriptions. Fusion could also be based on more than just spatial coherence and include, for example, visual similarity. Last, but not least, sensors or analyzers might run at different rates, which would then require a history and interpolation. None of this should affect the the subsequent processing steps, however.

Additionally, it is easily conceivable that steps might be re-ordered or new steps introduced. For example, the concept of "most interesting region" could certainly include existence of background knowledge. This would require steps two and three to be re-ordered and a new saliency computation to be inserted (instead of using an external, purely visual measure, as is the case currently).

Apart from these functional considerations, communication is a further considerable contributor to complexity. In the present component, a number of data sources must be specified and their data formats must be parsed into an internal representation. To reduce coupling and enable component evolution, it should be easy to change these.

8.1.2. Analysis of original action selection

An initial, independent version of the action selection component has been implemented by Lars Schillingmann for the first iteration of the "Curious Robot" demonstrator (Lütkebohle et al., 2009a, section 2.D). It was implemented in Java and used the XCF middleware (Wrede, 2008), and XML utility code from the BonSAI toolkit[1].

To provide a reference for comparison, some metrics on the code will be given first. Table 8.1 has the Chidamber and Kemerer metrics. As can be seen, the component is small-to-mid-size, with 778 source lines of code distributed amongst 22 classes, 5 of which are small inner classes. None of the classes exhibit noticeable anomalies in the CK metrics.

From the code size, the basic COCOMO (Boehm, 1984) model would put such a component at 1.87 person-months, without overhead. While this may be an overestimate, given that it assumes a typical commercial development process, the component interfaces with many others, which adds developer communication effort. Thus, it is probably not too far off.

The functional logic of the component is contained in merely three classes: "DefaultEntityComparator" ranks regions, "GripManager" manages background knowledge and "InteractionRegionThread" sends out proposals. In the latter two classes, only one of the class's methods is actually functionally relevant, the rest just contains marshaling code. This also demonstrates a typical issue, which is that i/o and function are often intermixed.

It has to be said that the original implementation already re-uses a considerable amount of code, both regarding middleware and regarding data transformation utilities. Even then, the combination of data from various sources in a relatively low-level

[1]`http://opensource.cit-ec.de/projects/bonsai`

Classname	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM
AgingThread	4	2	0	3	24	2	1	2
DefaultEntityComparator	7	1	0	3	16	21	1	3
GripManager	10	1	0	9	21	19	5	2
GripManager$1	2	0	0	10	13	0	1	1
GripManager$2	2	0	0	19	29	0	1	1
Identifier	2	1	0	3	2	1	2	2
InteractionRegionThread	8	2	0	16	48	0	3	2
InteractionRegionThread$1	2	0	0	12	18	0	1	1
InterestRater	2	1	0	10	14	1	0	2
PointXYIdentifier	3	1	0	4	14	3	1	3
RaterConfiguration	6	1	0	2	10	11	1	5
Region	9	4	0	5	24	22	8	7
RegionListReceiver	4	1	0	15	19	2	2	1
RegionListReceiver$1	2	0	0	7	6	0	1	1
SalientEntity	11	1	0	2	14	9	9	11
SalientEntityList	14	1	0	8	37	27	9	12
SalientPoint	13	0	0	6	31	64	8	11
SalientPointListReceiver	4	1	0	14	16	2	2	1
SalientPointListReceiver$1	2	0	0	7	6	0	1	1
StateLogThread	3	2	0	5	18	0	1	2
Timestamp	2	1	0	3	9	0	1	2
Total	112	21	0	163	389	184	59	73
Average	5.33	1.00	0.00	7.76	18.52	8.76	2.81	3.48
σ	3.88	0.93	0.00	4.91	10.68	15.00	2.94	3.51

Table 8.1.: CK metrics for the original selection component (cf. section A.1).

language such as Java leads to a comparatively high amount of code for a simple purpose.

Furthermore, the mix of functional and marshaling code is not surprising, given that the functional core is the smallest part of the program. That is, the situation is as it is not because the component was badly designed. On the contrary, it is well designed regarding its primary concern: Communication. It is just that communication should *not* be the primary concern of such a component: Fusion, ranking and proposal should be!

From this situation, two conclusions have been drawn: Firstly, configuring middleware communication endpoints and (un-)marshaling code should be easier and provided in a less component specific way. Secondly, data manipulation should occur at a higher-level, to reduce the amount of code necessary.

8.1.3. Graph-based re-implementation

From the two issues outlined previously, only one is directly related to the FTS approach: Creating middleware endpoints and (un-)marshaling code can certainly be made easier in a graph-based framework. In contrast, the issue of high-level data transformation is a domain-specific one, but equally important, if not more so. Thus, it constitutes a good test to determine the suitability of the toolkit to provide domain-specific support.

The previous functionality has thus been replicated, based on the data-flow toolkit, by the present author, and tested in the second and third iterations of the demonstrator (cf. section 2.3.2). For the final iteration, visualization has been added, as described in the next section.

The initial graph-based implementation of the selection component is visualized in figure 8.2, with the full (textual) specification given in appendix C.2.1. The graph contains about 40 nodes, and the specification has 131 lines.

As can be seen, there are four different input channels reading from the event-bus (which is called "Active Memory" for historical reasons, hence the node name). The associated filters are already contained in the source nodes and thus not shown (cf. appendix C.2.1, line 3 to 40, for details).

Two of the inputs contribute to tracking of background knowledge (on the left side, leading into "ListCollector"), the two others (on the right) deliver visual perceptual data. The "Unpack" nodes create internal data-structures from the XML inputs which are then fused and ranked by "RegionInfoRank". They are then converted ("packed") to XML again and the rest of the processing occurs based on the XML representation.

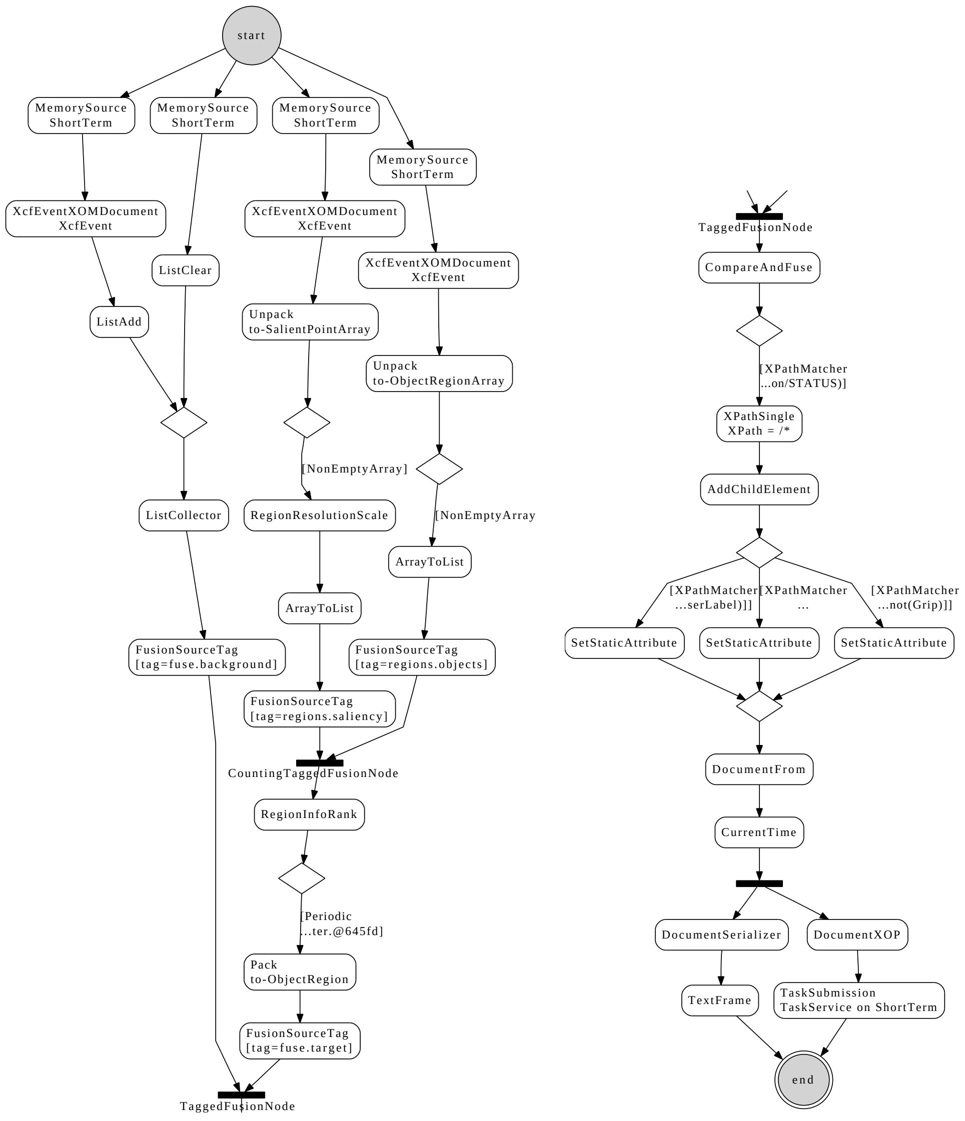
In the lower part, the jointly-delivered background information and current inputs are fused. This occurs based on three XPath selectors: One for determining which background document matches, and two for specifying what to copy from the background to the target and where, respectively. This fusion has been directly implemented (as opposed to using XQuery). While it is only used in this component, it is intended to realize a common fusion pattern.

After the fusion, three different paths can be chosen, based on the information already available in the fused data. After a choice has been made, the document is modified to reflect the next goal, timestamped and sent out. The "TextFrame" node simply displays the last document sent for the human operator.

It should be noted that while many nodes work on XML documents, the use of XML is by no means required to achieve reuse. The re-usability stems from the use of an attribute-value-tree (AVT) with associated path navigation (XPath) and modification (XQuery) languages. XML is merely a convenient choice, because of the wealth of tools available and any other AVT that has such tools could be used for similar effect.

Analysis

The first, readily apparent result of the conversion is demonstrated by the generic nature of the nodes: The graph consists of 39 nodes (including filters) that belong to 25 different types. Only one of these is completely specific to the component at hand ("RegionInfoRank") and one other is slightly specific in that it has only been used for this particular component, so far ("CompareAndFuse"). The other nodes are generic ones configured according to need and, of course, the graph structure itself is specific. Where in the previous implementation the re-use potential was only theoretical, in the graph-based implementation it is fully realized, as shown by the fact that 23 of the 25 different node types (or 37 of 39 nodes total) are completely generic. This is further supported by the fact that slightly more than half of these



(a) upper part (b) lower part

Figure 8.2.: Action selection graph. The graph as visualized has been split after step 3 of the initial functional description. That is, figure 8.2(a) shows information reception, ranking and integration of background knowledge whereas figure 8.2(b) shows the selection and proposal steps.

(13 of 23) are already being used widely in other components.

In particular, please note how all the input chains contain exactly the same node types at level one and the leftmost and the two rightmost ones up to level two. Moreover, the rightmost chain has exactly the same types as the second from the right up to level 7, where they join up, with the the second from the right having just one more transformation ("RegionResolutionScale") inserted.

For the "unpack" type nodes this is possible because of an underlying, service provider-based registration infrastructure for XML-converters. Therefore, it must be added that the actual parsing implementation differs. However, please note something which is not visible in the graph visualization: The output types of the converters are the same. This allows the rest of the graph to treat them alike.

In terms of source code, the node implementations used amount to 696 source lines of code, 103 of which are contained in the two specific nodes mentioned above. In essence, then, the graph specification and the specific nodes taken together comprise 234 lines of novel code for 0.56 person months of development time according to COCOMO basic, compared to 1.87 for the original component. While one would expect that the XML specification is faster to write than conventional source code, it is safer to err on the high side. This means that, assuming all the general nodes had been implemented already, the component could have been realized with (at most) one third of the effort.

	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM	SLOC
Total	111	3	1	113	329	169	1	59	669
Average	4.11	0.11	0.04	4.19	12.19	6.26	0.04	2.19	28.5

Table 8.2.: Aggregated CK metrics for node implementations.

Furthermore, the rewrite has resulted in an explicit representation of the component's structure, where all external configuration variables are directly editable in a simple XML file. Thus, for example, a change of sources or the addition of new goals can be realized without recompiling.

Last, but not least, it is notable that the decomposition of functionality into nodes was not problematic at all. This is likely in part due to the fact that the functionality was well understood beforehand, but it is notable that the separation was not as clean in the original, non-graph-based implementation. This suggests that the graph-oriented component design can be a mental aid during construction.

8.1.4. Adding visualization

An extension of this case study has been to add visualization support to the component discussed so far. This is work that has been carried out for the third iteration of the "Curious Robot" system and adds novel functionality which was not previously present in the action selection component.

Visualization of results is a frequently useful function during development of a system, but can create the need for a GUI or other logging functionality in components that would otherwise not need one. Alternatively, visualization can be added based on the results produced by the component to be observed, and its inputs. This requires combining all the same input processing functionality slightly differently, to enable visualization. Thus, it is a prime candidate for testing whether the claims of improved reuse can actually be achieved.

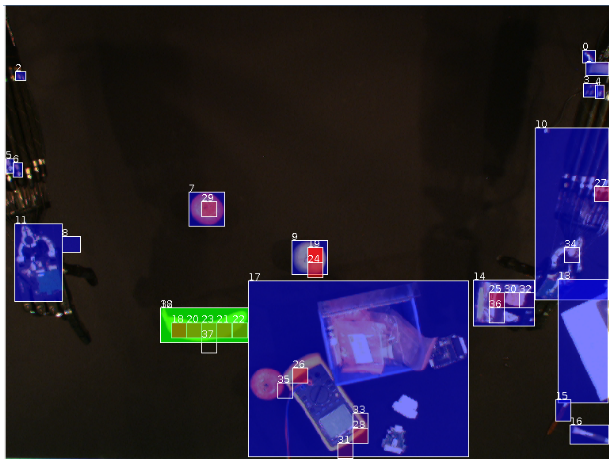
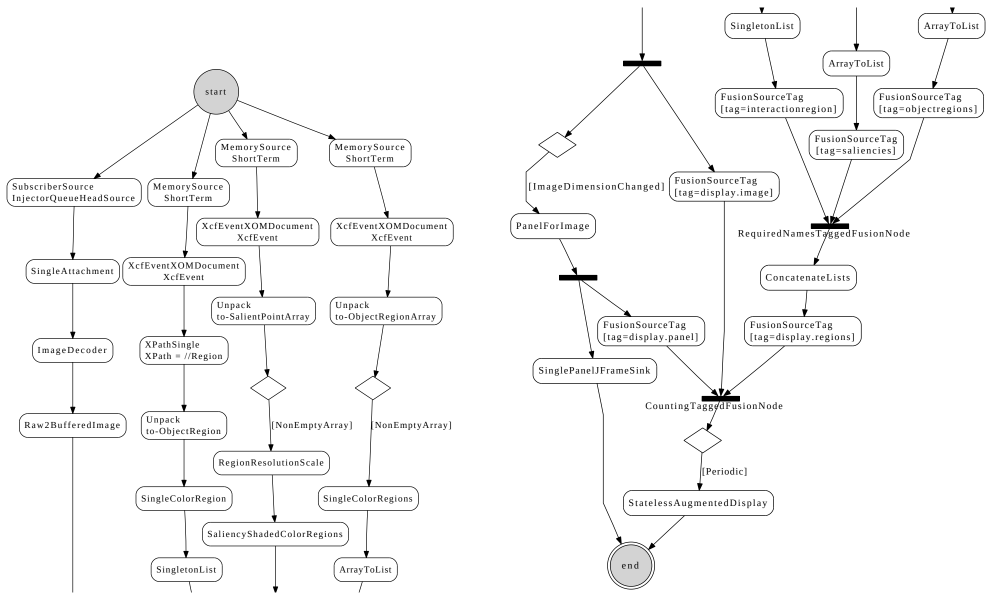


Figure 8.3.: Output of the region selection visualization – best viewed in color. The rectangles shaded in blue are candidates for object detection, the red rectangles are salient points and the green rectangle is the region selected for interaction. Please note that this screen-shot was taken during maintenance of the robot.

The output of the visualization is shown in figure 8.3 and the graph realizing it in figure 8.4. As expected, the graph shares many nodes with the action selection graph. In particular, the two right-most chains are almost exactly the same input processing chains as in the previous graph, up to the point where ranking would have occurred. Left of that is the chain processing the output of action selection and the leftmost input receives the camera image over which the information will be overlaid to provide context. Missing from the graph are the inputs for background knowledge, as these were not interesting for visualization.

The lower-left part of the graph realizes the actual visualization. First, a GUI component is created that matches the size of the incoming image and it will be updated when the size changes – this is an example of how state can be kept internal to the graph, by using a fusion node for storage. Up to this point, the *entire* graph consists of generic nodes only.

The actual visualization code is contained in the “StatelessAugmentedDisplay”, which receives a drawing surfaces and several lists of regions that will be drawn. The coloring is again generic and has been assigned by attribute-assignment nodes in the input processing chains. The visualization output itself is custom, with the


start
SubscriberSource InjectorQueueHeadSource
MemorySource ShortTerm
MemorySource ShortTerm
MemorySource ShortTerm
SingleAttachment
XcfEventXOMDocument XcfEvent
XcfEventXOMDocument XcfEvent
XcfEventXOMDocument XcfEvent
ImageDecoder
XPathSingle XPath = //Region
Unpack to-SalientPointArray
Unpack to-ObjectRegionArray
Raw2BufferedImage
Unpack to-ObjectRegion
[NonEmptyArray]
[NonEmptyArray]
RegionResolutionScale
SingleColorRegion
SingleColorRegions
SaliencyShadedColorRegions
SingletonList
ArrayToList
SingletonList
ArrayToList
ArrayToList
FusionSourceTag [tag=interactionregion]
FusionSourceTag [tag=objectregions]
FusionSourceTag [tag=saliencies]
[ImageDimensionChanged]
FusionSourceTag [tag=display.image]
PanelForImage
RequiredNamesTaggedFusionNode
ConcatenateLists
FusionSourceTag [tag=display.panel]
FusionSourceTag [tag=display.regions]
SinglePanelJFrameSink
CountingTaggedFusionNode
[Periodic]
StatelessAugmentedDisplay
end

(a) upper part (b) lower part

Figure 8.4.: Graph for the visualization component.

node accounting for 37 source lines of code (SLOC) and the display panel containing another 71 SLOC.

Analysis

Firstly, please note the added input chain (second from the left) which is fused with the two rightmost chains. The latter subgraph is a direct copy of the fusion graph from the selection component, with just two transformations added that specify a color for visualization. Moreover, the new input chain again uses many of the same node types. Thus, the predicted reuse has been achieved.

That said, a look at the detailed component specification (cf. appendix C.2.2) reveals that the graph specification itself has been copied and modified. This means that, while reuse is present at the node-level, it is not present at the level of the graph specifications. While certainly some form of inclusion specification could have provided reuse of subgraphs verbatim, the chains used here are slightly modified, with novel nodes inserted. Therefore, a means of graph transformation would have been needed, too.

A similar issue can occur in direct, library-based reuse – if the granularity of a library function is too high, inserting functionality into the middle of processing becomes impossible. In contrast, if the granularity is lower, similar to the level of the nodes used in the present case study, more effort is required during library use, due to the effort for combining them. Most languages, with the notable exception of functional ones, contain no explicit support for simple/declarative interconnection of functions.

At the moment, the lack of reuse in specifications is not considered a pressing problem, due to their relatively small size. However, a remedy would certainly be required to scale up this approach to the level of more complex components. A possible solution might combine ideas from meta-modeling (cf. section 7.1.3) and refactoring (Fowler et al., 1999)), but this is left for future work.

8.1.5. Summary

The case-study has demonstrated that an information fusion component can be realized in the FTS approach for substantially less effort. In this case, at most one third, likely less, of the effort of standard development would have been required. This is due to increased potential for reuse, which the explicit graph-structure has lifted from almost non-existent to 92% of nodes (23 of 25). A second sub-case, adding visualization support, has further demonstrated that graph-based construction aids in adding functionality to the overall system in a non-intrusive way.

While these two cases do not permit generalization just yet, they suggest that the identification of common, high-level data-structures is a crucial step to realize the reuse potential. While this is a well-established accepted fact in object-oriented development, it has not been a focus of attention for data-flow languages, so far (Johnston et al., 2004).

Last, but not least, it appears that the graph-oriented decomposition of components into nodes has resulted in a clean separation of concerns. It may also be the case that the graph model functions as a mental aid during construction, allowing the developers to maintain a high-level mental model of the component. Whether this is due to the comparatively small size of the graph specifications, which can be comprehended in their entirety, or due to other factors or a combination, would be an interesting question for future work.

8.2. Case study 2: Hardware independent serial robot control

The previous case study has investigated a data-fusion application which can be found in many systems, robotic and otherwise. Therefore, the next case study looks at an application area which is closer to hardware and control aspects: Robot control through a serial link.

Using serial links for robot control is a popular approach. The actual control algorithm is implemented on a micro-controller, often as the only function of that controller, which makes achieving real-time scheduling fairly simple. Furthermore, such micro-controllers often offer convenient sensory inputs. For example, the XMega microcontroller used in the “Flobi” head (Lütkebohle et al., 2010) has – amongst other inputs – three quadrature decoders which can be directly coupled with motor encoders, to achieve high-accuracy, high-speed velocity and position sensing for very little CPU time. Achieving the same functionality on a PC requires a real-time operating system and measurement boards that are several order of magnitude more expensive.

To link this micro-controller to the PC, a control protocol must be implemented. Usually, each robot manufacturer has a custom protocol, sometimes more than one. Some of them provide libraries that implement these protocols, some don't. In any case, to realize a hardware independence layer, these protocols must be adapted to a common abstraction layer – either based on direct protocol implementation, or by adapting the vendor-supplied libraries.

The requirements for implementing such protocols are the following:

- Transduce abstract control commands to the vendor protocol and inversely for sensor data. This may sometimes be dependent on current device state.
- Manage access to the serial link – many protocols prohibit sending a command before a reply for the previous one has been received.
- Realize a blend mode – protocols differ in whether a new command overwrites a previous one, is queued or is refused. Some protocols allow a choice, in others this must be realized through explicit queuing or cancellation.
- Deliver feedback information on start of command execution.

For the present work, a hardware independence layer for different robots has been implemented by the present author. The robots are:

1. The BARTHOC humanoid torso (Hackel et al., 2005), manufactured by MABOTIC GmbH. This robot is controlled through Atmel-based micro-controller boards delivered as part of the robot by MABOTIC, including firmware. The protocol is relatively simple.
2. The Sony EVI D31 Pan-Tilt-Zoom camera, using the VISCA™ protocol (Son, 1999). This camera is common in many robotic applications and the protocol is also used for several other cameras by Sony. This protocol is fairly elaborate and includes substantial device state.
3. The “Flobi” anthropomorphic robot head (Lütkebohle et al., 2010), developed jointly by Bielefeld University and MABOTIC GmbH. The controller boards for this robot, and their firmware, have been developed by Simon Schulz of Bielefeld University. Close cooperation was possible during their design and the command protocol was jointly specified by Simon Schulz and the present author.

8.2.1. MABOTIC robot control protocol

In principle, the MABOTIC control protocol is about as simple as it gets. It is a binary, serial protocol with only five fixed packet types, running at 38,400bps over a standard RS-232 serial port, full-duplex. The packets have a standard 4 byte header and an optional 2-3 byte payload. The header has sender, recipient, payload size and type information, each encoded as a single byte.

The main command types are a) version inquiry, b) joint position set immediate and c) joint position store, d) joint execute and e) joint position inquiry. A few more commands are available, but are only used for maintenance. The full specification, as it is, is contained in appendix D.1. Not mentioned there, but implicit is that novel set points for an actuator silently overwrite previous ones, i.e. the blend mode is overwrite.

As is already apparent from the command types, the robot is purely position controlled. Joint positions are specified in the hardware range, hence the conversion from joint angles to hardware values must by performed by the implementor.

The only complication during implementation of this protocol arises from link management: New commands must only be sent after a reply for the previous one has been received. This is because the serial controller chip (an Atmel AT90S8535), has only one byte serial buffer, and the firmware does not read from this buffer while processing a command (Schwope, 2008).

The graph implementing the protocol is shown in figure 8.5. It consists of two sub-graphs, which are executed by separate engines: One for transcoding commands to the protocol and one for reading replies. The sending graph has one input per actuator, of which there are 27, and one input for inquiry commands. The per-actuator nodes perform the conversion from angular space to device space. The “Command-Generator” node implements the transformation from the internal command format to the MABOTIC protocol. On the parsing side, the “ReplyReader” transforms the input bytes to the internal format again.

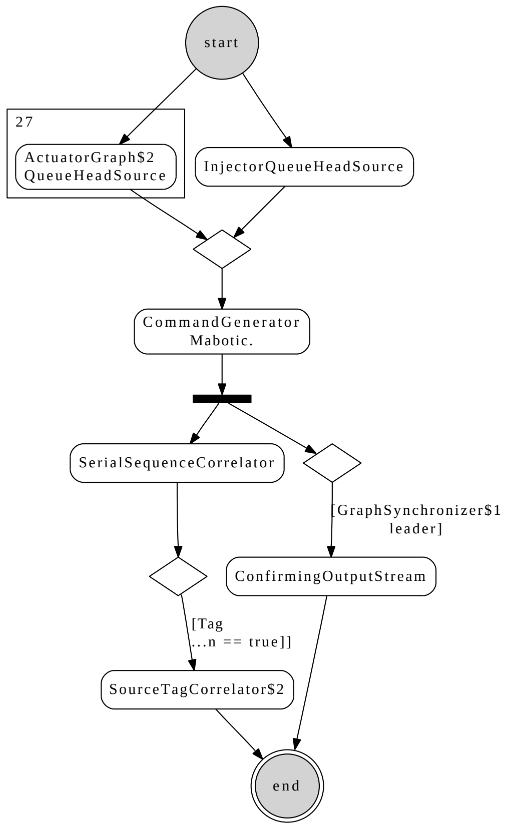
To achieve the send-read coupling mentioned in the previous paragraph, the graphs are linked through the “GraphSynchronizer” nodes. When no command it outstanding, the first of these (“leader”) passes through and sets a lock. The second (“follower”) then releases the lock. If a command arrives inbetween, the first will block until the lock is released.

Last, but not least, confirmation about start of motion cannot be derived solely from sensor information in the BARTHOC robot, because some joints are actuated by servos with no externally visible position sensors. Thus, confirmation is provided after the command has been submitted and the robot is starting to move. This functionality is realized through the “SourceTagCorrelator” node.

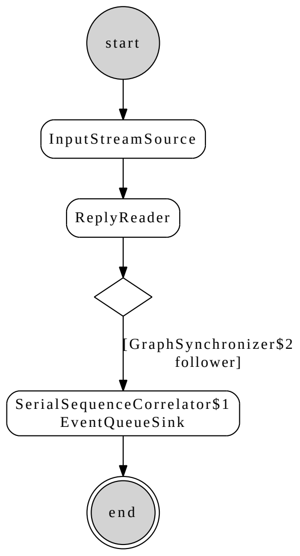
Analysis

Similar to the protocol, the graph structure is fairly straightforward for both sub-graphs, with the notable exception of the hidden link between the two graphs (from “leader” to “follower”, as explained above). This reveals a fairly interesting difference in explicitness when using a direct API and the visualization.

The presented implementation of the protocol has been split so that it can be executed in two different engines in parallel, with a link as above. In the Java code,



(a) Command generation & confirmation



(b) Input parsing

Figure 8.5.: The two subgraphs making up the implementation of the MABOTIC robot control protocol.

this link is obvious: A common “GraphSynchronizer” object is created, which has two linked nodes as attributes that are then placed into the different graphs. However, the link is not reflected in the graph structure and thus the visualization – which is only based on the explicit edges – does not show it. Likewise, all future analysis based solely on the graph will miss this important link. The goal of achieving better comparability through comparing the graphs only would not be achievable in such a situation.

Fortunately, a possible way out is to use a fusion node with history. This fusion would take both the “CommandGenerator” and the “ReplyReader” as inputs and could then pass on commands only when a reply for the last commands is also present as input. This fusion node would need a history so that it can queue up multiple commands while waiting for the replies.

Changing the synchronization to a fusion node would be beneficial in at least three ways: Firstly, it makes things more explicit. Secondly, and perhaps more importantly, it removes a lock-and-wait idiom, replacing it with a data-driven trigger. Lastly, it removes one thread from the execution, because the new graph, where no node ever blocks, can be executed with one engine instead of two[2].

Given that the MABOTIC protocol graph was one of the earlier experiments with the FTS toolkit, the lesson to be learned here is probably an old one: It takes some mental realignment to realize the benefits of a new approach. Furthermore, the solution is obvious in hindsight, probably at least in part through the help of the visualization.

8.2.2. VISCA control protocol

In contrast with the previous case, the VISCA protocol (Son, 1999) is much more powerful, and correspondingly more complex. Challenges include:

- *Transducing.* Some commands require knowledge of device state, for example cancellations need to know the command number.

- *Link access.* Motor control commands follow a three-step procedure with send, ack and complete. After sending a command, everything must wait until it is acknowledged. Later on, when the command is completed, the camera sends a reply on its own.

- *Blend mode.* The D31 camera has no consistent blend mode. There are two command sockets and when one of them is available, commands overwrite each other. However, when no socket is available, commands are refused.

- *Deliver feedback.* Because the VISCA protocol can deliver completion information at any time, it can only be associated with a command through the

[2]This is possible even though InputStream is blocking, because synchronous sources are queried on independent threads to achieve fairness (cf. fig. 7.6).

socket number, which is reported in the ACK message. This requires the device state to be tracked and communicated throughout much of the protocol implementation.

Some of these changes are beneficial – for example, receiving an explicit completion notification removes the need for sensor polling in many cases. However, the inconsistent blend mode is a clear hindrance for a consistent, device-independent external interface. Furthermore, it is greatly aggravated by the fact that the cancel command – which is needed to create a virtual blend mode of overwrite – is device-state dependent.

The result of this can be seen clearly in the initial protocol implementation, shown in figure 8.6. The graph is fairly large but please note the edges marked in blue and drawn dashed. All of these lead into the "AutoCancel" node and they are feedback from command execution. In contrast, only *one* (the remaining) edge into that node is actual control input.

Moreover, the entire left side of the graph (beginning right after the "InputScanner" compound at the top, ending at the "SocketReplyMapper" at the bottom) is needed to match replies to the commands that caused them. Whereas in the previous case study, each reply was associated with the command just sent, the possibility for intermediate completion replies from the camera precludes this. As a consequence, where previously one, general sequence-matching node was sufficient to perform this function, now ten protocol-specific nodes (including filters) are necessary.

Analysis

The cause for the circular graph structure shown is the desire to provide "overwrite" command blend mode: Newer commands should be able to abort earlier commands, if so specified, without the user having to know any details about the way the earlier command is executed on the device.

As mentioned already, the cancel command requires knowledge of which socket the previous command is in. The only way to handle this automatically is to maintain knowledge of the current socket state, which can only be determined from the replies of the camera. Hence the need for "backwards" transitions from reply receiption to a node which also transforms incoming commands.

Such cycles are certainly needed in many applications and while they may clutter the visualization slightly, they are not a bad thing. However, the existence of so many back-links is symptomatic of an underlying problem: hidden state. In contrast to the previous case study, where state could be entirely maintained in generic fusion nodes whose behavior is well-known, this time the state is contained within an opaque and protocol-specific "AutoCancel" node.

Further, it is notable that some of the select conditions used are specific and not configurable, e.g. "FirstReply" contains explicit code to select either measurement replies or acknowledgements. This is in contrast to the data fusion example where

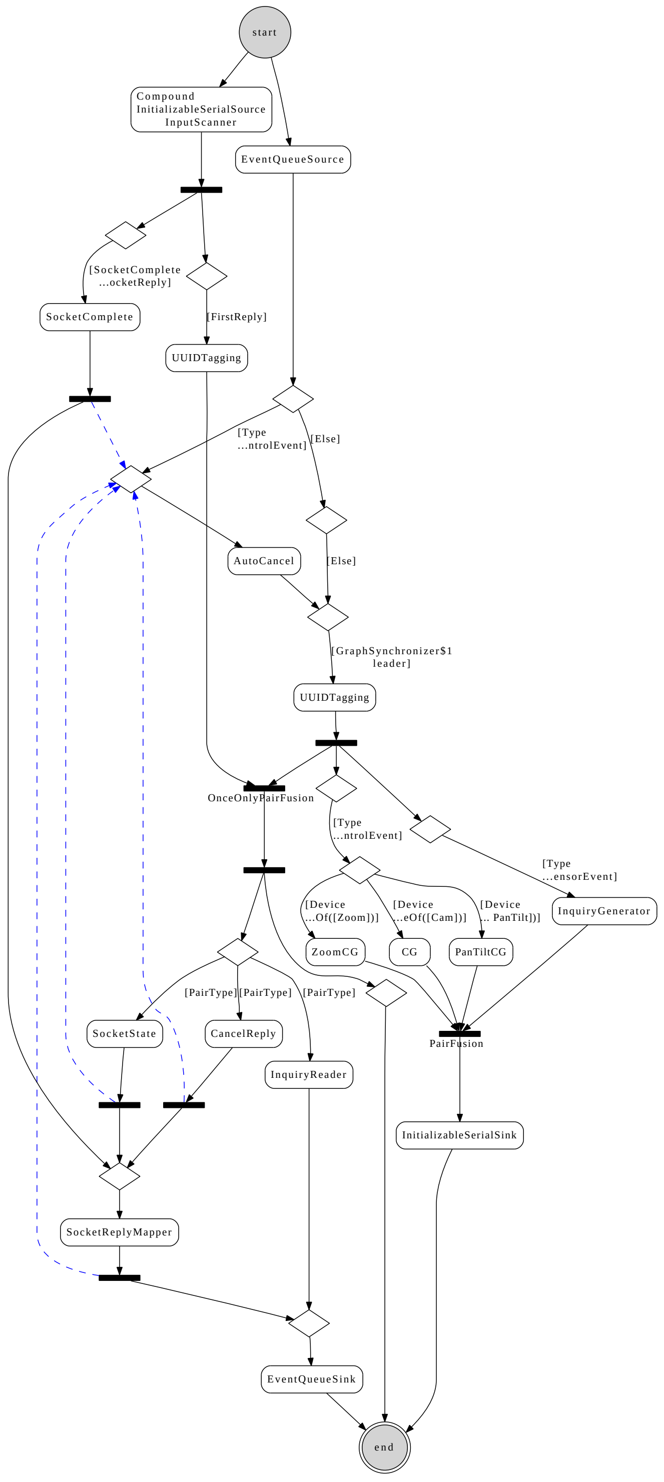


Figure 8.6.: Line driver graph for the Sony D31.

the XPath language provides an explicit, transparent means to select based on the content of data. The lack of such a language for general objects impairs reuse.

In a similar vein, the elaborate graph construction needed to match replies to commands is not something one routinely wants to construct manually. It also does not enhance reuse, as it is completely protocol specific, up to the node implementations. While there are certainly many ways of going about this, it would appear that the use of an event expression language, including temporal constraints, would be one solution that ties in particularly well with the declarative way of building graphs.

8.2.3. “scontrol” protocol driver

The “scontrol” protocol is a custom development for the “Flobi” robot head (Lütkebohle et al., 2010). The content of protocol packets is modeled on CAN bus messages (CAN), with extended payload lengths and the semantics of commands are similar to the MABOTIC protocol (i.e., a novel command for the same actuator overwrites a previous one).

Due to the similarities, the control implementation graph contains no surprises. However, one interesting aspect of the implementation is that the protocol underwent a minor revision: The first revision did not use replies for simple commands, in an attempt to reduce the potential for collisions on the bus. The lack of replies meant that multiple write commands could be sent without waiting.

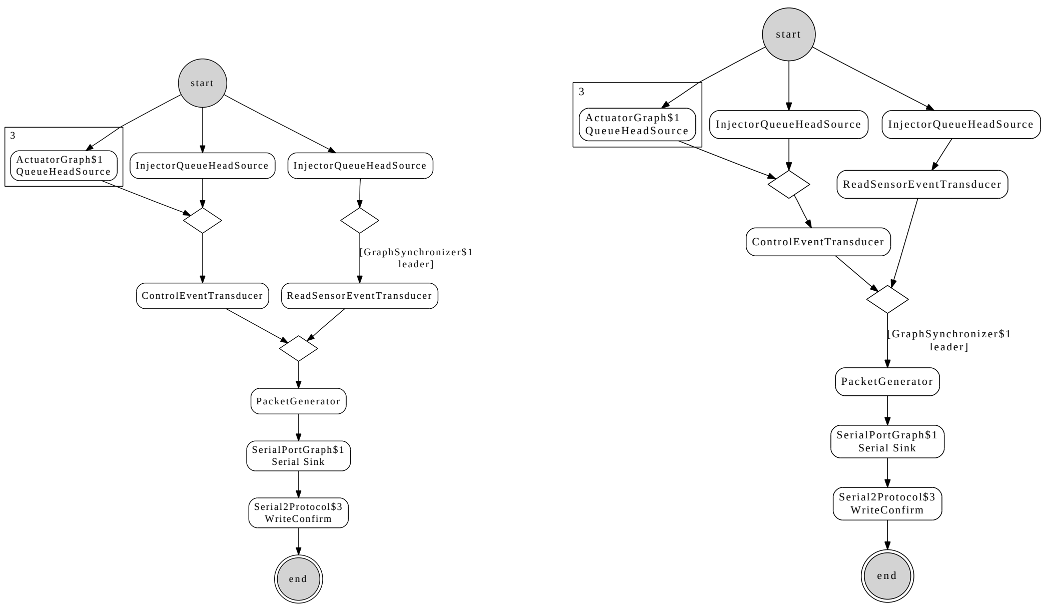
The obvious drawback is that it is unknown whether the controller has actually received the command. The link was not over CAN but RS-422 instead and a combination of an electrically noisy environment and relatively high data-rates (1Mbps) caused frequent errors. Because of this, the second revision did introduce acknowledgements for write packets, too. These then required writes to be included in link access management.

Fortunately, the necessary synchronization changes could be accommodated in the write graph purely by rewiring it, as shown in figure 8.7.

Analysis

In the scontrol protocol, the data-flow approach has delivered on both its promises: Reuse of general building blocks (serial link i/o, link management, reply association) and flexibility through external specification of node connectivity. This demonstrates that the approach is capable of exploiting protocol similarity, despite differences in packet format.

Development of the “scontrol” implementation has been continued by Patrick Holthaus, a colleague working with the “Flobi” head. This work is focused on feature additions and still ongoing, also based on the FTS toolkit. While his work is beyond the scope of this thesis, it can be said that the data-flow approach could easily accommodate all required changes to the implementation so far.


start
3
ActuatorGraph$1 QueueHeadSource
InjectorQueueHeadSource
InjectorQueueHeadSource
[GraphSynchronizer$1 leader]
ControlEventTransducer
ReadSensorEventTransducer
PacketGenerator
SerialPortGraph$1 Serial Sink
Serial2Protocol$3 WriteConfirm
end

(a) v1: read sync only (b) v2: all commands synced

Figure 8.7.: Command output graphs for scontrol revisions.

8.2.4. Discussion

The toolkit has been experimented on three different robot control protocols, across three different robot platforms. Both the reuse and the flexibility goals could be demonstrated in these case studies.

As expected, the benefit of flexible graph construction was particularly visible during evolutionary development of a protocol (the “scontrol” case study). This indicates that the approach is suitable for rapid application development.

In comparison to the previous case-study on data-fusion, it could be seen that a declarative, content-based condition language is beneficial and reuse is reduced when one is not available. For simple filtering conditions, this may be outweighed by efficiency and/or simplicity considerations. However, for more complex, temporal constraints, an expression language appears necessary to constrain complexity of the implementation graphs. This is related to work in event-processing languages (Luckham, 2002; Arasu et al., 2006; Demers et al., 2007) and future work may explore this relation.

Not surprisingly, it was found that optimal use of an data-flow approach may take some learning. In particular, synchronization and state-management are two areas which appear to be substantially different. Of these, state-management is somewhat mixed, with local state easily managed but shared state still presenting somewhat of a problem.

Part IV.

System and Conclusion

9. System Evolution

In this, final, evaluation chapter, the evolution of the architecture over the iterations of the present system will be discussed. The issues encountered while realizing more capable Human-Robot-Interaction will serve as a backdrop to evaluate the proposed architectural concepts. In this, both social and technical requirements will be considered, in keeping with the two main objectives of an architecture: a) provide beneficial constraints for development (thus guiding the developers, a social aspect) to b) realize a usable, performant system (the technical aspects).

The main scenario idea came about as a direct reversal of an older scenario: In the “Home Tour” (Hanheide and Sagerer, 2008), people are followed by a robot and direct the robot’s attention by pointing at things and labeling them. As outlined previously, this was criticized on the grounds that people cannot know a-priori what the robot does and does not know, both regarding which objects it knows and what sentences it understands. Therefore, an alternative approach is that the *robot structures interaction*. In the present system, this structuring is done by asking on its own – so-called robot-initiative.

In keeping with a reversal of the prior situation, the robot should also point at objects while asking for their names, and possibly other information, then start interaction with the objects as its facilities permit. This situation models that no knowledge about object labels and/or descriptive attributes is prespecified. Of course, detecting objects is already by no means trivial, but it was hypothesized that visual saliency, known from computational models of human attention, might provide a means to identify object-candidates from which the interaction could be started.

9.1. Scenario pre-test: “What is that?”

For the described scenario to be feasible, a crucial question is *whether people would let the robot set interaction goals*. While mixed-initiative has been successfully used before, it is usually employed in situations where humans give the initial command and the robot just takes initiative when it gets into trouble following that command. In contrast, in the present scenario the robot sets the goals of the interaction.

If humans would not accept the robot’s goals, the undertaking would have been over before it even began. If they did, there are also some more technical questions. For example, it may turn out that pointing is not a good means of referencing objects or that visual saliency does not deliver sensible object candidates.

Therefore, a small test was conducted to establish that the scenario was worth exploring. The test also provided an opportunity for early integration of basic components. While by no means representative (the test group contained just four subjects), it is a good example of the core scenario idea.

Test procedure

The test has been conducted using the “Wizard of Oz” paradigm (Kelley, 1983; Dahlbäck et al., 1993), with some live components. Action selection was provided by a human operator, whereas saliency computation and pointing used the actual algorithms to be employed in the future system. The latter aspect is intended to verify that the chosen saliency algorithm is usable. The robot’s activities followed four steps:

1. The vision system computes visual saliency of the camera image, using Nagai’s saliency algorithm (Nagai et al., 2003a).
2. The system selects the most salient point and has the right robot hand point at it.
3. Concurrently with the previous action, the question “Was ist das?” (English: “What is that?) is uttered.
4. Upon receiving an answer, the robot confirms with “Aha, ein(e) *label*” (English: “Ah, a *label*”) and repeats the procedure with the next object (the saliency map is recomputed on every iteration, but known objects are blotted out).

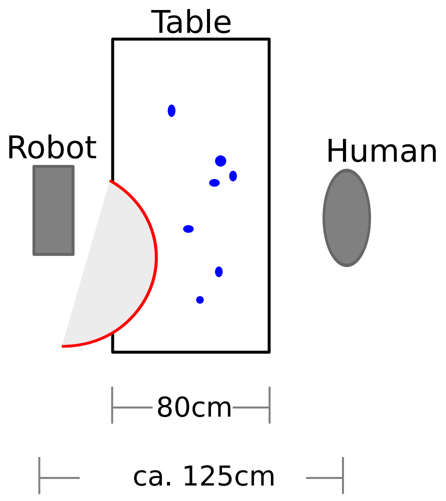
This sequence is the main sequence, with deviations due to user activities always possible. Robot activity is controlled by a human operator using a graphical user interface, shown in figure 9.1. The interface allowed both speech output and restricted robot motion. Each step of the procedure outlined is individually controllable and can be executed and/or repeated as desired. This has been used, for example, to test how subjects reacted to repeated questions for the same object.

The robot used in this test was BARTHOC Junior, a smaller version of the humanoid torso robot of the main setup. Both models have the same motion capabilities.

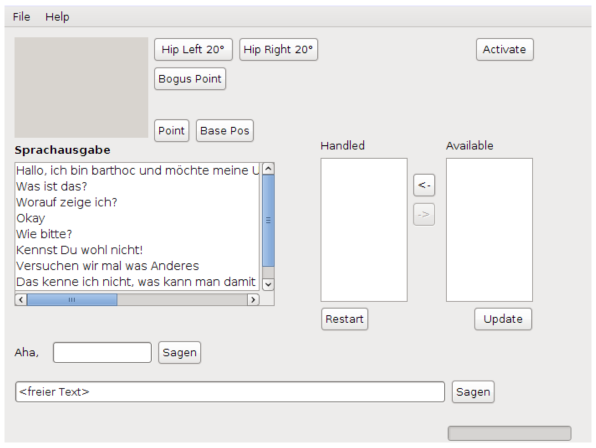
Four naive individuals (non computer-science students) made up the test sample. This sample is obviously not representative by far, and was not intended to be, but it is considered sufficient to establish the worth of proceeding further. Furthermore, it turned out to be a diverse sample, useful in establishing a number of potentially problematic issues.

Test outcome

The main outcome of the test has been that, yes, users did allow the robot to specify goals, and – not unexpectedly – also took initiative. They also asked a lot of questions at all stages during the proceedings. In particular, they queried the robot about the knowledge it acquired and they also picked up objects to demonstrate them explicitly, after being asked once.



(a) interaction setup. arm range: red/shaded; objects: blue/small.



Hallo, ich bin barthoc und möchte meine U
Was ist das?
Worauf zeige ich?
Okay
Wie bitte?
Kennst Du wohl nicht!
Versuchen wir mal was Anderes
Das kenne ich nicht, was kann man damit

(b) remote control user interface

Figure 9.1.: Interaction setup and the operator control application.

Furthermore, pointing has been established as interpretable, but the distance between the finger of the robot and the object to be pointed at is crucial. While this could not be established directly, it appeared as if the direction of the pointing finger was less important than the absolute distance to the object. If multiple objects were in the general direction, users were confused and similarly for objects which were relatively far away. Figure 9.1(a), shows an example object arrangement and it can be seen that several objects were out of range and/or close together in approximately the same direction.

Because of the resulting confusion over the target of pointing, subjects frequently used clarification questions. In these questions, some of the subjects assumed the robot to understand color attributes. Some of the subjects also pointed at objects, whereas others picked them up.

Last, but not least, visual saliency using Nagai's algorithm did turn out to be useful in suggesting image regions to ask for. Distractors, such as edges, did occur but while these did cause problems, the were much smaller than the problems caused by the pointing procedure.

9.2. Initial Object Learning System

Having established that the scenario idea is acceptable, the next step has been to transfer it to a fully autonomous setup. In particular, this required automating the choice of a next action and the integration of a dialog management component to handle the various sub-dialogs that occurred during the main sequence. Regarding perception, object and speech recognition had to be integrated

Furthermore, the previous robot did not do anything besides asking. This is considered too limiting, both for studying the interaction itself and with regard to the larger context this thesis has been undertaken in, which is about manipulation for service robotics.

In this context, "object learning" includes basic manipulation capabilities and the remainder of this section will be about the initial system created for this purpose. The initial system consists of 19 distinct components and has been developed by five persons. In the following, the scenario will first be detailed and then examined regarding the suitability of the architectural principles to facilitate technical integration and developer coordination.

9.2.1. Use Cases

The robot has acquired several more use cases, which will be specified in the following. Additionally, the possible user activities are also specified as use cases. Each of the use cases specified contains multiple activities on the system level, but on the interaction level, they constitute the steps from which the overall interaction is created. An overview of the use cases is given in figure 9.2.

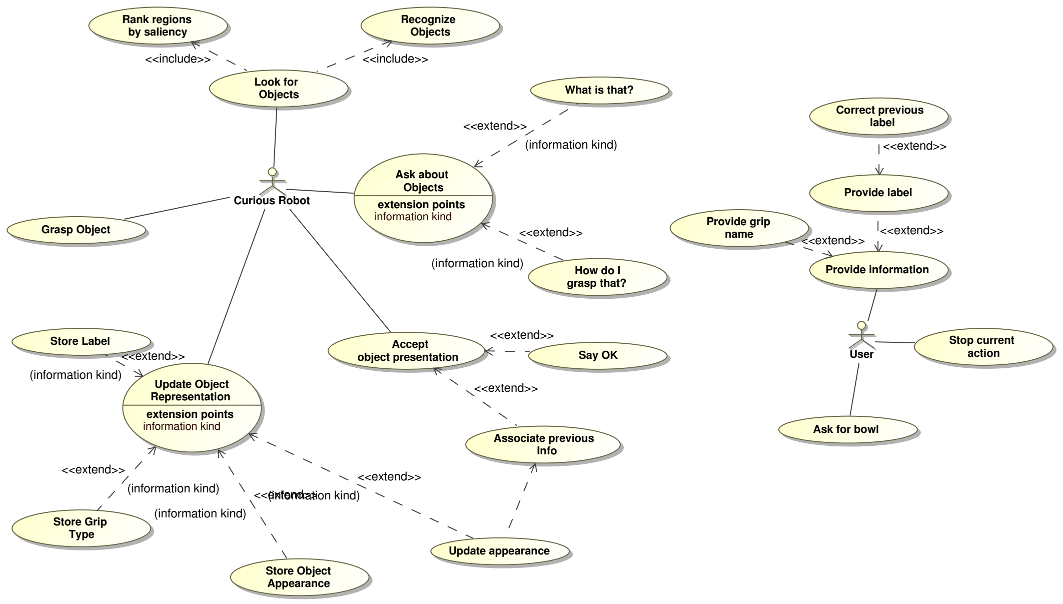


Figure 9.2.: Use cases of the "Curious Robot"

The information requests are extended by one: Asking for a grip types, in case this cannot be inferred automatically. Requesting other types of information follows basically the same model, so it has not been considered necessary to include more information requests.

Otherwise, the robot can perform two manipulation actions. Firstly, placing objects into a bowl by picking them up (using the specified grip) and dropping them into the bowl and secondly, picking up the bowl with both hands and passing it to the user.

Last, but not least, an object recognition component has been added, which is responsible for storing information about the objects. The interaction of the dialog and object recognition constitutes a major part of the scenario.

User actions

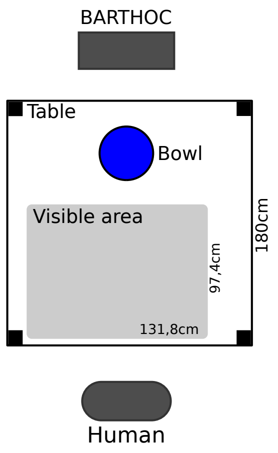
In the initial system, the user is mainly providing information, as requested by the robot. Furthermore, she can stop the robot's actions at any time. Lastly, she may request that the robot passes the bowl.

It must be said that the bowl passing action is somewhat orthogonal to the main scenario idea, as the user has to know how to do that already, instead of being told by the robot. It has been included primarily to be able to demonstrate bi-manual manipulation in conjunction with object learning.

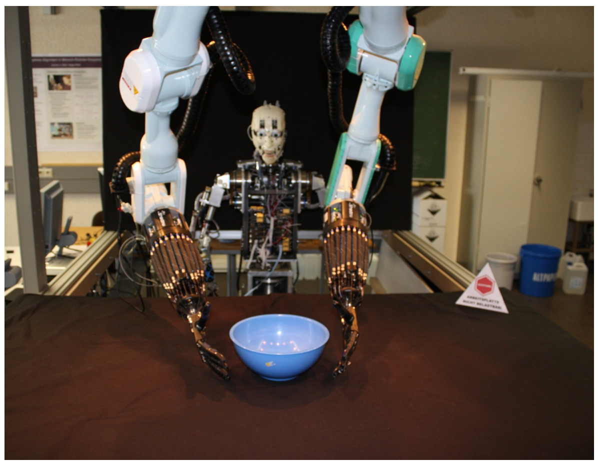
9.2.2. Hard & software components

The setup physically contains two robots: The humanoid torso "BARTHOC" and two software-coupled Mitsubishi PA-10 industrial robots, with Shadow robot hands attached. Of these, only the PA-10s actually move during the initial scenario. The humanoid is in a fixed position, looking at the table.

Part of the design has been to simplify perception. For vision, this is done by using a static camera viewing the scene from above, and a black background with known lighting. The resulting visible area is shown in figure 9.3(a). The camera used is a Sony DFW-500 at a resolution of 640x480.



(a) Sketch of setup



(b) Photo of setup

Figure 9.3.: Initial object learning setup.

The software structure of this version is shown in figure 9.4, with components arranged according to the functional areas they belong to. There is one component each for each of the active parts of the robot (arm, hand, “voice”/speaker) and the sensors (camera, joint sensors, microphone) and one derived sensor which computes a self-occlusion image.

Many of the analysis and behavioral components are standard, as follows:

Visual Saliency uses Nagai’s variant of the Itti-Koch model (Nagai et al., 2003a), embedded in the iceWing framework (Lömker et al., 2006).

Speech Recognition applies the speaker independent ESMERALDA (Fink, 1999) toolkit with HMM-based recognition, a bi-gram language model, a robust, scenario-specific grammar and German conversational speech training data from the VERBMOBIL project.

Speech Analysis is performed through a robust, chunk-based analysis component (Hüwel et al., 2006)

Text-to-Speech applies the Mary TTS toolkit (Schröder and Trouvain, 2003)

The Motion Subsystem and its components has been developed for the present setup by Robert Haschke and colleagues (Ritter, Haschke, Röthling, and Steil, 2007)

Custom components

Scenario specific additions have been made as follows:

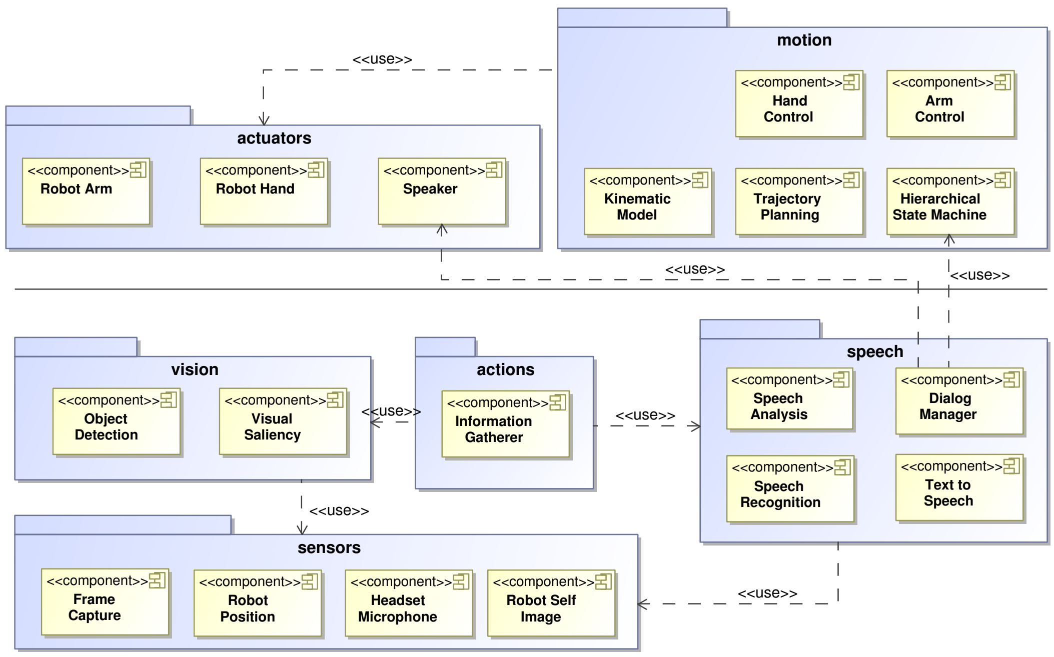
The Information Gatherer is the action selection component developed for this scenario and discussed in detail in section 8.1

Object Recognition uses a color-histogram and shape based recognizer with one-shot learning developed by Christof Elbrechter (Lütkebohle et al., 2009a)

The Dialog Manager is based in part on an earlier system, but has been substantially extended for the present scenario by Julia Peltason (Peltason et al., 2009b; Lütkebohle et al., 2009a).

Communication between these components is performed using the XCF middleware toolkit (Wrede, 2008). For the high-level interaction, the life-cycle based interaction is applied – which, in this iteration was *not* using the XCF Task Toolkit.

It is obvious, and not surprising, that the autonomous system requires much more functionality, and actual integration, compared to the Wizard-of-Oz test. The larger set of components also came with a larger research team and corresponding coordination challenges. While the test has been carried out by just two persons (Julia Peltason and the present author), the software for the autonomous system has been



motion
<<component>> Hand Control
<<component>> Arm Control
<<component>> Kinematic Model
<<component>> Trajectory Planning
<<component>> Hierarchical State Machine
actuators
<<component>> Robot Arm
<<component>> Robot Hand
<<component>> Speaker
<<use>>
vision
<<component>> Object Detection
<<component>> Visual Saliency
actions
<<component>> Information Gatherer
speech
<<component>> Speech Analysis
<<component>> Dialog Manager
<<component>> Speech Recognition
<<component>> Text to Speech
sensors
<<component>> Frame Capture
<<component>> Robot Position
<<component>> Headset Microphone
<<component>> Robot Self Image

Figure 9.4.: Components in the autonomous object learning system

built by five (Robert Haschke, Christof Elbrechter, Lars Schillingmann and the previous two).

Fortunately, a substantial functionality base already existed, which made the scenario feasible and allowed it to be built comparatively fast. The existing functionality had also already been separated into independent components for distribution on separate machines. However, the fact that these components were originally developed for different scenarios did cause some additional effort during integration

9.2.3. Notes on integration

Integration occurred at various levels in the present system (and this also applies up to the last iteration built so far). While this part of system building is known to uncover problems and generally take more time than expected, the participating researchers had all built systems before and the system was designed to avoid many of the classic, problematic issues.

Calibration. As the camera is static, it could be easily calibrated to the robot coordinate space. Hand-Arm calibration was contained in two, tightly coordinated components.

Sensor level. The only integrated sensor with several components in this iteration is vision. Because only one camera was used, the main problem to achieve integration was to agree on a common coordinate system – different algorithms ran at different resolutions, so absolute pixel coordinates were out. The resulting solution is to use relative pixel coordinates, which could be compared regardless of resolution.

Model Level. The main model that is built up during operation of the system is one of objects being interacted with. For this purpose, a common "interaction region" representation has been created, which is jointly built up by the various components and provides a common reference point. For an example, see appendix C.4.3.

Interaction-Action Level. The task-state pattern, which had been successfully used in the COGNIRON system, has been applied in the present system to integrate dialog, object recognition and the motion control coordinator (HSM).

The integration of this system is remarkable primarily because it has been so unremarkable. It appears that this is due in large part to the XCF middleware toolkit, which has been designed specifically to allow iterative integration and emphasizes a focus on representations, instead of commands (Wrede, 2008). For example, the addition of relative coordinates to the output of the vision components could take place one component at a time, because additions to data structures are backwards compatible in the XCF approach.

The present system has added task-based coordination to these principles, which focuses on a *single* interaction model. So far, it appears that the combination works well on the technical level, and the resulting system will be evaluated regarding its interaction in the next section.

This is not to say that integration has been entirely without problems. However, the problems mainly had to do with exercise in the full system uncovering bugs in individual components. This is quite different from the, not uncommon, case where integration uncovers fundamental design problems. Fortunately, such severe flaws did not occur and the system could be delivered to testing.

9.3. User evaluation of the Object Learning System

Every Human-Robot-Interaction (HRI) system must be tested with users other than its designers, to determine whether it can actually deliver what it has been designed to do. Such a test determines how well users can actually perform the intended task with the system. In the present system, the evaluation target has been the efficacy of the dialog structuring approach. The evaluation questions are:

- Does the guidance by the system give users a correct idea of what they can do?
- Does the guidance improve intelligibility of the humans actions by the system?

Apart from these, user's impressions and suggestions for the system are also of interest.

9.3.1. Experimental design

The experiment has been based on the video-study paradigm and credit is due to Manja Lohse[1] for suggesting that this form of study might be easier on the subjects than a real study with a system in its first iteration. This section is largely based on Lütkebohle et al. (2009a), with some additions regarding value judgments. For that paper, the video has been filmed with the full team, while the study itself has been designed and carried out by Julia Peltason and the present author.

In a video-study a recording of an experienced person interacting with the system is shown to the test subjects[2]. The video is stopped at predefined positions and questions are posed to the subjects, as shown in table 9.1. The questions are asked after the robot has acted, but prior to the moment where the recorded person answers, to guarantee an unbiased answer. We can then compare user's reactions in the varying interaction situations. The difference in the responses for the various situations can give us insight on the general effectiveness of the guidance (our first item), and the variability in the responses indicates whether the constraints increase predictability (our second item).

Time (mm:ss)	Situation	Question
00:07	Scenario shown	What do you think could this robot do?
		How would you instruct this robot?
00:29	"What is that?"	What would you do now?
00:47	"How can I grasp that?"	What would you do now?
00:51	"Beg your pardon?"	How would you correct?
03:40	Failed to grasp apple.	What would you do now?
06:33	Points at empty position.	What is happening?

Table 9.1.: Study Plan

The advantage of a video study like this one is that diverse interactions may be explored without frustrating the subjects, because they can show their intuitive behavior first, which may or may not be supported by the system, yet, and then continue further interactions based on the behavior the experienced test subject demonstrates. The obvious disadvantage is that results may not generalize to direct interaction. However, video studies have been shown to generalize well when correctly analyzed (Woods et al., 2006). In general, the benefits of early feedback outweigh the potential drawbacks.

Outline of the demonstration

In the video shown, the human and the robot collaboratively identify objects lying on the table, coordinate how to grasp an object and then the robot places it in a bowl (see figure 9.5). Ten test subjects were recruited from visitors to a university wide

[1]Applied Informatics Group, Bielefeld University. mlohse@techfak.uni-bielefeld.de
[2]See `http://aiweb.techfak.uni-bielefeld.de/content/curious-robot` for the video used.

outreach event and thus had varying backgrounds and ages. They did not receive prior information about the goal of the scenario but have been told that we intend to broaden the interaction capabilities of the robot and that any action they would like to take was acceptable and of interest. The video was shown to them using the procedure described above, and each of them has also been video-taped to record their motion and facial expression for qualitative analysis.

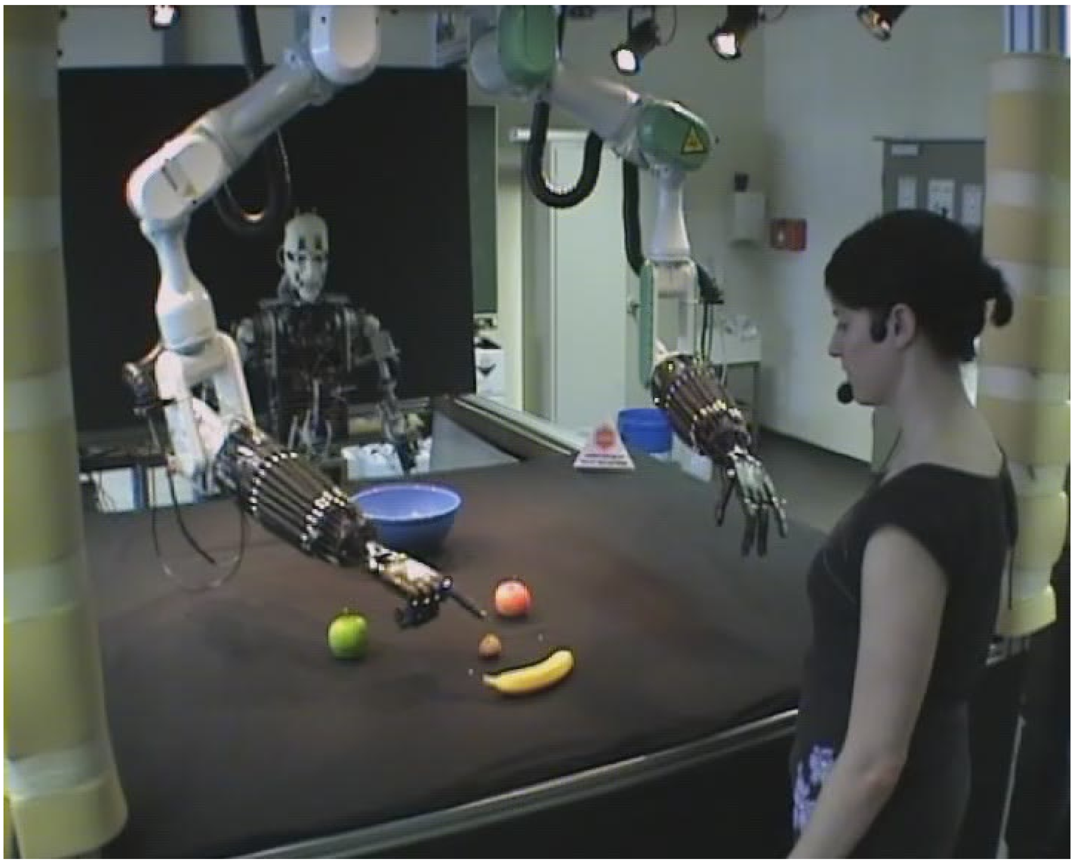


Figure 9.5.: Situation for "What is that?", as shown in the experiment video.

9.3.2. Results of the experiment

The following results are based, firstly, on a log of subject answers kept by the experimenter and, secondly, a questionnaire filled out by the subjects. The questionnaire did capture some information about the subject and let the subject judge the system. It is reproduced in appendix E.1.

Observations during initial system description

The first situation is a static image of the scenario (figure 9.5), where subjects are asked to speculate on the systems interaction capabilities by appearance alone. All subjects could deduce the task to be "placing objects into the bowl". They also agreed that the system was capable of vision, grasping and speech recognition, even though no direct indication of that was given.

After that, however, the descriptions of how they might attempt to interact with the system varied widely and no clear pattern emerged. For example, one person said "Take the green apple and put it in the blue bowl" while another provided "I would explain that it should combine the four things" and a third said "Make a fruit-salad!" (which make sense, because all the objects visible were fruit). A summary of the variations is shown in table 9.2.

Apart from variations in terminology and concepts, it is particularly interesting that half the subjects only used meta-commentary, such as in the second example above, and did not provide any concrete command, even though the experimenters prompted them multiple times. This may have been due to the study setup, but as can be seen in later parts, subjects did produce concrete example commands when it was clear to them what they could say.

Label Domain	fruit name 80%	"object" 20%		
Container Label	"bowl" 40%	"dish" 40%	none 20%	
Attributes Used	none 50%	Shape 40%	Color 30%	Size 10%
Subtask	none 70%	sorting 30%		
Commands Given	none 50%	"put *a* in *b* 20%	"put all..." 20%	"sort" 10%

Table 9.2.: Percent of subjects using a particular concept

Further, please note that 50% of the subjects did not use any attributes for object specifications, but those that did often used multiple attributes at once. Of these, shape was the most frequent, followed by color. Only one person used a size attribute. These frequencies cannot really be considered typical, because the sample size has been comparatively small, but the split between use/non-use of attributes might be a more general trend.

Description of grasping

One of the questions used during the trial has been "How do I grasp the 'object'?". The robot did not provide any indication on which aspect of grasping it wants described, hence this question is considerably more open than the others. The motivation underlying this question is twofold: Firstly, it is interesting to see how subjects react to unclear guidance and secondly, this provides an uninfluenced view on how subjects naturally describe grasping. Table 9.3 shows the aspects used (sometimes several aspects were given). In at least one respect, the results are very clear: Subjects took an average of 19 seconds to answer, compared to just 5 seconds for the label question. This indicates considerable more confusion, as expected.

Aspect Described	Percent of Subjects
Effector position relative to object	30%
Trajectory of effector	20%
Fingers to Use	40%
Force to Use	30%
Grasp point on object	20%

Table 9.3.: Aspect of Grasping described.

From the video recordings of the subjects, it is also apparent (again not surprisingly) that many subjects performed the action of grasping during the explanation.

Observations regarding user initiative

An example of user initiative can be observed in a situation where the robot fails to grasp the object. These utterances are syntactically more varied, particularly when users provide corrections, see table 9.4. However, they are conceptually much more straightforward than the initial descriptions and it appears promising that users do provide verbal commentary relating to grasp parameters, such as “rounder” or “softer”, which are complementary to visual demonstration.

Answer	% of Subjects
“Try again”	40%
“Grasp the ...”	20%
Grasp corrections (“rounder”, “both hands”, “softer”)	40%

Table 9.4.: User Commands after Failed Grasp

Reactions to system guidance

In contrast, answers to the “What is that?” question by the robot are considerably more consistent, as shown in table 9.5. Only three constructions are used in total and they are all slight variations of a single sentence. The subjects apparently found it easy to answer this question, as they needed only an average five seconds to answer (measured from end of question to end of answer). Only one subject required clarification.

In an error condition, where the system pointed at an empty spot, two variations occur, roughly in equal proportion: Asking for clarification and giving the name of the closest object. The latter were always accompanied by comments expressing that an error occurred and thus recognizably different from regular replies.

Situation	Answer	Percent of Subjects
"What is that?"	"That is a..."	70%
	"a ..."	20%
	"a yellow ..."	10%
empty pointing	"What do you mean?"	50%
	(pointing wrong) "That is a ..."	40%
	"nothing"	10%

Table 9.5.: Replies after System Initiative

Overall judgments

The last part of the evaluation had the subjects answer a few questions (cf. appendix E.1.1) that reflect their overall judgment of the system and the interaction seen. The results are summarized in figure 9.6.

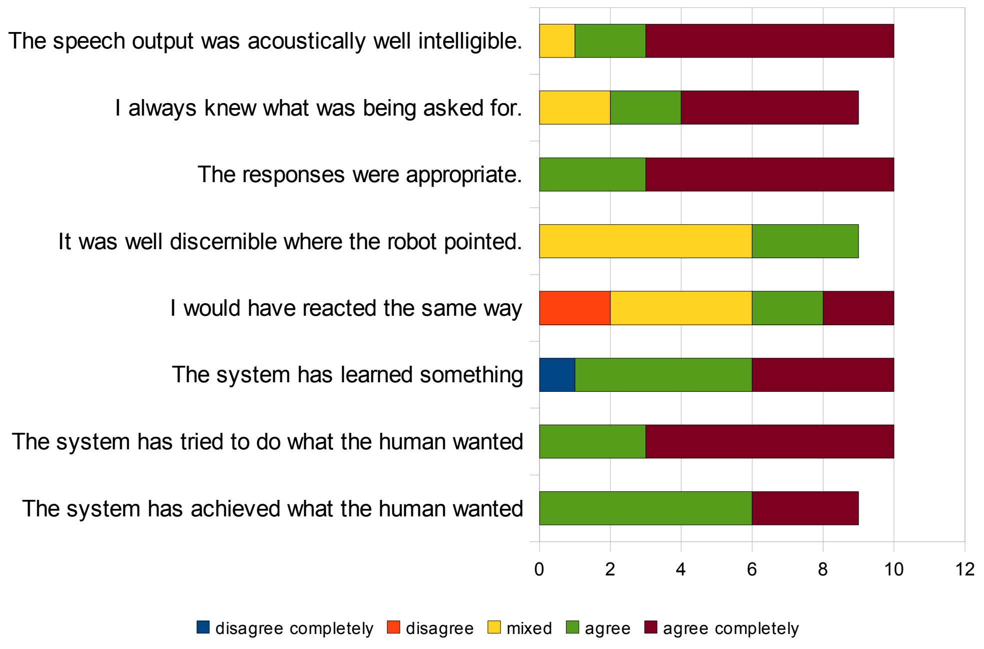


Figure 9.6.: Overall system value judgments. Also compare appendix E.1.1 for the exact questions, as the ones shown here have been slightly abbreviated.

The statements have been posed in the positive and overall, subjects mostly agree or strongly agree with the statements, with some mixed answers. This is to be expected[3], and should not be interpreted to mean that the subjects were completely satisfied with the system. The true information is more in the trend of the answers, e.g. which of them they answered more or less positively than the others.

Perhaps most notable is the response to the statement "I would have reacted the same way as the person in the video". It is the only question to receive disagreement at all (20%) and another 40% answered "mixed". This indicates that the interaction as shown still has quite a way to go before it can be considered intuitive. From an analysis of the transcripts, it appears that most of the disagreements were caused by differences in handling erroneous situations and the description of grasping (where most people would have demonstrated it).

In particular, several subjects commented that they would not have said "stop" or similar things on action failure, but would have repeated the previous command instead. This is an example of implicit feedback which most systems do not handle.

A slightly interesting response regards the discernibility of pointing: It received rather mixed results. This appears to be largely due to the video study setup, where people could not clearly see the pointing in depth. However, it may also have something to do with the distance from the finger to the object, particularly when objects are close together.

A last revealing judgment has been the (single) person that answered that the system has not learned anything: The person complained that the robot did not *say* what it has learned, even though it acted on its knowledge.

9.3.3. Discussion

Speculation behavior. From the initial speculations of the users, it appears that subjects tend to make judgments of the sort "because multiple colors appear, the system can differentiate colors", thus assuming capabilities that the system may not actually support. In the present case, they assumed object labels to be known, which has not been the case and would have been a problem if not for the system's guidance. This illustrates the (sometimes accidental) influence of appearances, and a dialog system should be prepared to address such preconceptions.

Detecting subject uncertainty. It is notable that subjects sometimes used meta-commentary ("I would have...") and sometimes gave very explicit answers, despite the same amount of prompting by the experimenters. It seems that subjects used meta-commentary when they would have been unsure of what to do in a real situation.

Furthermore, responses after guidance by the system are not only more explicit, but also show extreme consistency, almost to the point of being exact repetitions. Even reactions to errors are surprisingly consistent and corrections are provided without hesitation.

[3]When subjects have a neutral reaction, they are known to answer slightly towards what they believe the experimenter would want them to say.

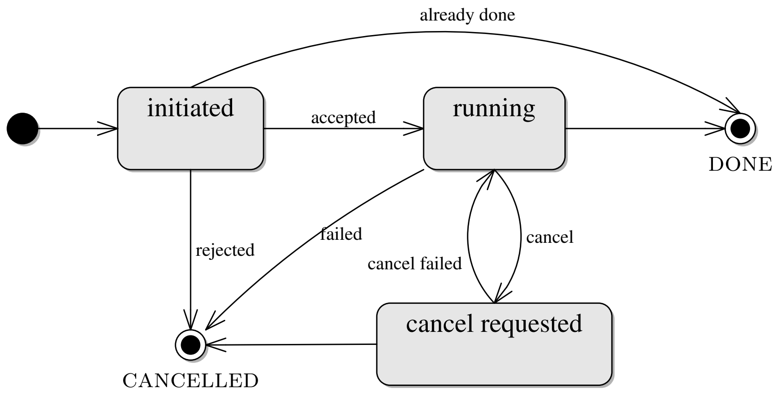


Figure 9.7.: State Machine for the initial life-cycle model

It is concluded that task-structuring by the robot is necessary and should include not just verbal help but also contextual constraints. The results indicate that the proposed method achieves this for object reference. Grasp descriptions, however, need more guidance, which is not surprising as this aspect was intended to be exploratory.

Discourse structuring Another result from the responses is that a dialog system is required and simple "question-reply" not sufficient: Requests for clarification occur frequently and user initiative plays an important role for error detection. Additionally, even though utterances are relatively consistent conceptually, there are still considerable syntactical variations present.

The responses by the test subjects also show that the interaction as currently implemented would not be their preferred mode in some cases. The preferred alternatives are relatively few and consistent, so that they can be implemented in the next iteration of the system.

An aspect that remains open is how to let users know when they may interrupt the system, e.g. with additional commentary or error feedback. The study design prompted them, but in a real situation, other cues are necessary. This is basically a social interaction issue and it would thus be interesting to add more social feedback mechanisms to the interaction.

9.4. Relation to proposed methods

The initial version of the system marks the starting point for many of the previous, method-oriented case studies. This means, most importantly, that the toolkits developed later have not been used, yet. However, the task-state has been applied based on custom implementations by the component developers for the action selection, dialog manager and HSM components.

The life-cycle model in use during this version is as shown in figure 9.7.

In comparison to the basic and general life-cycle models (cf. figure 4.2, respectively

4.10), this model marks a mid-way point. It does include cancellation of requests, but neither updates nor intermediate results.

From the experience in this version of the system, the task-state pattern is considered to be a good candidate for a general interaction protocol between diverse components. However, three issues were identified that should be improved.

Partial implementation

In the initial version, the components only implemented part of the life-cycle model. In particular, aborting was only supported by the HSM. The dialog could issue a cancel request to the HSM, but it could not react to such requests by the action selection component. The action selection component, in turn, did not issue cancellations. This meant, for example, that updated information from vision could not be taken into account.

It may be that part of the reason for this situation is that cancellation deviates from the standard procedure call semantics. Without it, the task-state pattern coordination degenerates to a slightly improved version of RPC, which has well-known semantics for most developers. If this is indeed the case, cancellation, and any other extensions such as updates that leave this well-known area, would increase the likelihood of partial implementations.

To be clear, it is not a target that all components support the full model. However, if a state is not supported, it should be refused explicitly to ensure that a submitter can safely try it and fall back. Otherwise, client components would have to include knowledge of what a particular handler component supports, which is not the goal of a general model.

Model extension

Furthermore, extensions to the life-cycle model are not currently possible in a backwards compatible way. For example, if a novel state is introduced, components that do not implement it must still answer in a way intelligible for the source component. This is, however, not possible without either a) versioning support, b) explicit changes to the components for each such extension, or c) the introduction of an adapter component. While all of these may be possible solutions for some cases, they each have their respective drawbacks.

While it is not suggested that the model be extended indiscriminately, some future extension should be possible. The favorable experience with the XCF middleware, which enables data structure extension in a backwards compatible way, further supports this view.

It is suggested that both of these problems can be solved through a service-level toolkit for implementing the task-state pattern. Thus, the the XCF Task Toolkit implementation described in the previous chapters was started.

The issue of three-way interaction

One other problematic aspect has been the interaction between the dialog manager and the object recognizer. As mentioned previously, the recognizer can simply observe dialog events to acquire label information. However, this label information must be associated with the visual appearance of the objects at the beginning of the interaction, which may not be the same as the one at the end, when the label is available. For this reason, it was decided that the object recognizer should store a visual snapshot in the interaction region representation used for task coordination. This is the same representation that is used for coordinating a task between the dialog and the HSM, resulting in a three-way interaction.

It is the opinion of the present author that this decision, while convenient at the time, has been a serious mistake, given the technical circumstances. The reason is that the semantics of the event-bus used are such that derived events *replace* the events they are derived from. Therefore, to keep the image information that the object recognizer has added, the dialog must use the updated representation when adding its label. This means that a change in the object object recognizer affects the dialog, too, which is not desirable.

The first, conceptual, problem is that the life-cycle model does not specify a transition for this and thus, it was performed while keeping the current state. Because of this, initial implementations did not count it as an update and ignored it. It would appear that this may be easily fixed through an extension of the model.

However, the harder problem is that of three-way interaction without explicit synchronization. This is already visible in the above: Because a third component issued the update, neither of the original participants in task-state coordination for this task were aware of it. Therefore, neither could detect that an update had been missed, which may not only occur due to bugs, but also due to timing races. In contrast, if only two components participate, a missed update is always detectable by one of them, because the previous version is present locally.

In practice, races does not occur in this situation, because it takes much longer for the text-to-speech service to synthesize the question (which has to happen before it can be spoken) than it takes the object recognizer to attach the image snapshots. Therefore, the update will happen before the human partner even has a chance of replying. Nevertheless, situations such as these must be avoided.

9.5. Extended / Alternative Scenarios

Based upon the present system and in the light of the user evaluation, two further iterations of the system have been created. The first of these was aimed at demonstration during a trade fair (2009's CeBIT) where it was presented for one week. It incorporates some of the evaluation's lessons already, particularly regarding feedback. However, it has also been significantly stripped down in other respects, to fit the fair context, and thus constitutes a branch in functionality.

In contrast, the second iteration is a direct continuation of the full object learning system, addressing concerns identified during the user evaluation. Furthermore, it has also been used for final evaluation of the proposed toolkits, as described in previous chapters.

9.5.1. CeBIT demonstrator

The CeBIT demonstrator has been an interesting diversion and an opportunity to expose the research system to a larger audience. In contrast to other robotic systems at CeBIT, which were only demonstrated by experts on a predefined schedule, the “Curious Robot” is explicitly targeted at naive users and thus it was decided to run it continuously. This meant, most importantly, that the system had to run 10 hours a day for seven days straight, which had never been tried before. It also meant severe constraints on the hardware and space available (cf. figure 9.8).

Besides the runtime challenge, it has also been a challenge to get across a scientific topic in such a context, with limited time and many competing exhibits. It is certainly a credit both to the scenario and to the demonstration personnel[4] that this has been achieved and the booth was always crowded.

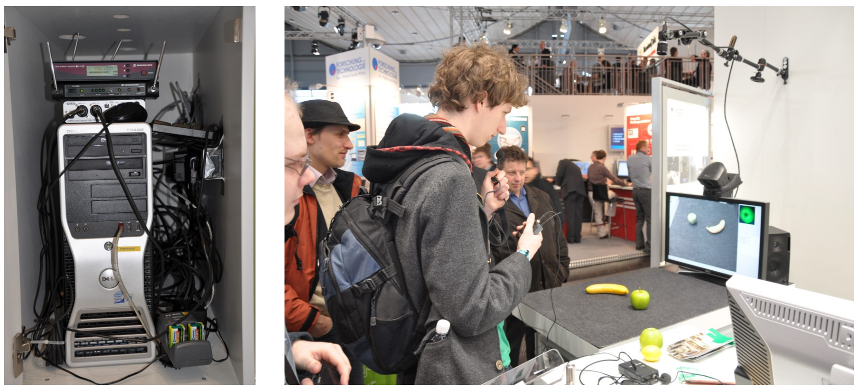

(a) cramped resources (b) trade fair guests

Figure 9.8.: CeBIT 2009 trade fair demonstrator

Scientifically, this demonstrator served as a case study on how quickly the system could be changed to accommodate the trade fair situation. This not only required stripping out parts that could not be brought but also adding replacement parts, so that the scenario as a whole was still interesting. The following changes have been carried out:

- Removal of manipulators and all grip-related behaviors.
- Use of a Pan-Tilt camera for feedback (in place of the humanoid robot).

[4]The assistance of Johnathan Maycock, Matthias Schöpfer, Sebastian Wrede, Patrick Holthaus, Florian Schmidt and Franz Kummert has been a tremendous help and is gladly acknowledged.

- Addition of a display to “point” through highlighting.
- Calibration of a moving camera to a static one, to integrate visual information from both cameras.
- Addition of information-query behaviors by which subjects could ask the robot for learned information.
- Addition of a situation-reporting behavior that listed the available objects.

Many of these additions have been simple to integrate, due to the task-state pattern. For example, the query and reporting behaviors could be added in a straightforward manner, by augmenting the dialog and adding a simple service component for the object models. These and other changes are similar to the one for the extended learning scenario, discussed in previous chapters.

Action selection: A case for declarative specifications

Removing grasping functionality could be achieved simply by disabling respective action targets. However, at the time, the action selection component had been only partially refactored. It did use the conditional dispatch prototype, but not the full FTS toolkit, yet, and in particular, no declarative component model. Therefore, disabling the actions required code edits, instead of just changing the graph specification file.

This example demonstrates why having a declarative specification for components as a matter of course can be beneficial. For instance, it would certainly have been easily possible to make the conditions for the various behaviors configurable – allowing them to be disabled without code edits. However, at the time the component was written, it did not appear necessary. When the need came, it was too late to make such a change.

In essence, reconfigurability of traditional components is an additional feature that takes additional development time. Moreover, it is specific to the component. In contrast, declarative graph specifications, in conjunction with the filter-transform-select decomposition of nodes, prepares for such changes, without additional, component-specific work.

User feedback from fair use

The trade fair situation did not permit a full study, despite the attraction of studying dozens or even hundreds of visitors: It would simply have hampered the demonstration to get permission from every user and, additionally, the situation could not be sufficiently controlled, because visitors talk a lot to each other about what they have seen and done.

However, a few subjective impressions may be interesting to report. Most importantly, the scenario seemed to be a big hit with the visitors. We attribute this largely to two factors: Firstly, visitors could try it out for themselves, in contrast to all the other robot exhibits. Secondly, and very unexpectedly, visitors always had fun with the image captured by the Pan-Tilt camera and shown on the display screen. In particular the situation where the camera looks up at the face, and visitors saw themselves generated many laughs. More importantly, after this situation, visitors became more engaged. It appeared as if recognizing the system as “active” changed their stance towards it.

9.5.2. Enhanced object learning system

The enhanced object learning system represents the final demonstrator built and it incorporates enhancements to the scenario in response to user evaluation. Moreover, the methods developed during this thesis have been fully applied in this version. In particular, the addition of the feedback sub-system (see section 9.6.3) has utilized both the task-state pattern (for observing interaction state changes) and the FTS toolkit (for creating the components).

Gaze cues The feedback sub-system already represents one of the additions made due to results from the user evaluation: It provides faster feedback and regulatory cues for interaction. Its necessity was not wholly surprising, as such cues have been proposed often (most notably by Breazeal (2004)), but from the multitude of possible cues, gaze was selected based as the most obvious turn-taking cue for object-related interaction.

Learning feedback. As outlined, subjects inquired about the systems knowledge during the interaction. Furthermore, they were unsure of what the system has learned. To address this, the dialog has been extended to provide information on what it learned upon request. This includes both labels, grip types and which objects the present scene contains. The language for these queries is based on a straightforward reversal of the questions the robot asks, e.g. “How do you grasp the apple?”.

One consequence of this change is that the possibility for the user to take initiative has been substantially enhanced. The dialog manager used supports nested interaction, so inquiries can be made at any point, including during motor activity. As such activities take a while, they provide a natural point of inquiry. The actions can also be aborted, as before.

One interesting aspect here is that the information necessary to answer the queries is not provided by the object recognizer component, but is based on records of previous (successfully completed) task interactions. These records are automatically kept by the “Active Memory” event bus.

Implicit interaction & guidance. One efficiency improvement made to the interaction has been to replace the explicit confirmation question with an implicit acknowledgment: Instead of asking “I understood X, is that correct?”, the robot now says “OK, X” and proceeds. The user can then correct the robot (“No, Y” or “A Y”, again), or accept and proceed. This removes one round-trip from the interaction, while still keeping the possibility to correct misunderstood words. This change has been based on the user feedback, where subjects answered that, in case of an error, they would repeat their previous utterance.

To realize implicit completion, both the dialog manager and the task-state life-cycle had to be extended. The latter extension has been the "result_available" state, which signifies that an intermediate result is available, but not the final one. This has been done to simplify rollback for the object recognizer in case of an erroneous label.

Hand posture classification. In the video study, subjects uniformly had trouble with describing the specifics of physical actions, such as the difference between different ways of grasping. They had no trouble actually demonstrating the grasp, however.

Therefore, to enable subjects to demonstrate grasps directly, a posture-sensing glove has been added. The posture is then classified into the three different grip types supported. Classification information is fused back into the input for the dialog system, to be processed in conjunction with verbal information.

No change to the interaction has been made to use this glove, but during user evaluation, subjects have been observed to instinctively perform grasping motions during their explanations (in the first trials, without wearing the glove). Thus, the classification is also performed during the subjects explanatory statement, and can be associated with any complementary verbal information.

Of course, it has to be admitted that putting on a posture-sensing glove is an obvious hint to subjects that the system will be able to interpret its input, so little further guidance is necessary. It is also a constraint, so further options are being considered.

One option is to recognize the hand-posture visually, but this is still an incredibly challenging problem. For the object shapes in use, automatic selection of grasp postures would actually be more feasible. This would change the interaction requirement to corrective feedback and this is currently being considered for the next iteration.

Toolkit and component changes This iteration uses the task toolkit in all components, except for those in motion control. Furthermore, the FTS toolkit has been used for the action select component as well as the new feedback behaviors, and the fusion component integrating the CyberGlove input.

These additions result in the extended set of components already shown in the scenario description, figure 3.1. The motor and interaction sub-systems are unchanged on the architecture level, but a new sub-system (for feedback) has been added. All three will be described jointly in the succeeding section.

9.6. Functional architecture

The functional architecture is the distribution of functionality amongst components, and the associations between these components in the integrated system. It determines the path of data through processing stages and constrains which components may interact in what way.

The main subject of this section will be to examine the interplay between architectural principles and system evolution. The visualizations in the following have been made according to the (sub-)system schemata shown in sections 3.3 and 3.4. However, these schemata were developed after the initial system and thus in themselves already represent an analysis step.

For the motion control and interaction subsystems, both the initial and the extended object learning system share the same components and functional architecture, depicted in figure 9.9. The motion control and interaction sub-systems are very loosely coupled externally, with tighter internal coupling. In the extended system, a third sub-system has been added, coupled to the interaction sub-system in the same way (see figure 9.10).

9.6.1. Interaction subsystem

The interaction subsystem adheres most closely to the schema, which is not surprising, as it is the one the present author was most directly involved in. The main notable fact relating to the schemata is that the dialog manager (DM) component is shown twice: In the checking and the sequencing layers. The DM is the only component in the sequencing layer and contains both functions, thus this is a straightforward change and one which does not violate the layering of the schema.

Object recognition integration

One other notable aspect in the interaction subsystem is the integration of the object recognition component. This component initially has no label information and once the user provides a label (determined by the dialog), the representation must be updated.

Due to the use of an event-based integration approach, this interaction can occur without explicit notification. The events generated by the Dialog Manager contain a reference to the object region and by “listening in” on these events, the object recognizer can get the necessary information.

The exact integration had good and bad aspects. The fact that object recognizer *modifies* a document shared by dialog manager and a third component, is considered problematic in hindsight (cf. 9.4). However, the ability to *purely observe* other component’s activities in this way is highly advocated and can result in a loosely coupled, yet well coordinated system.

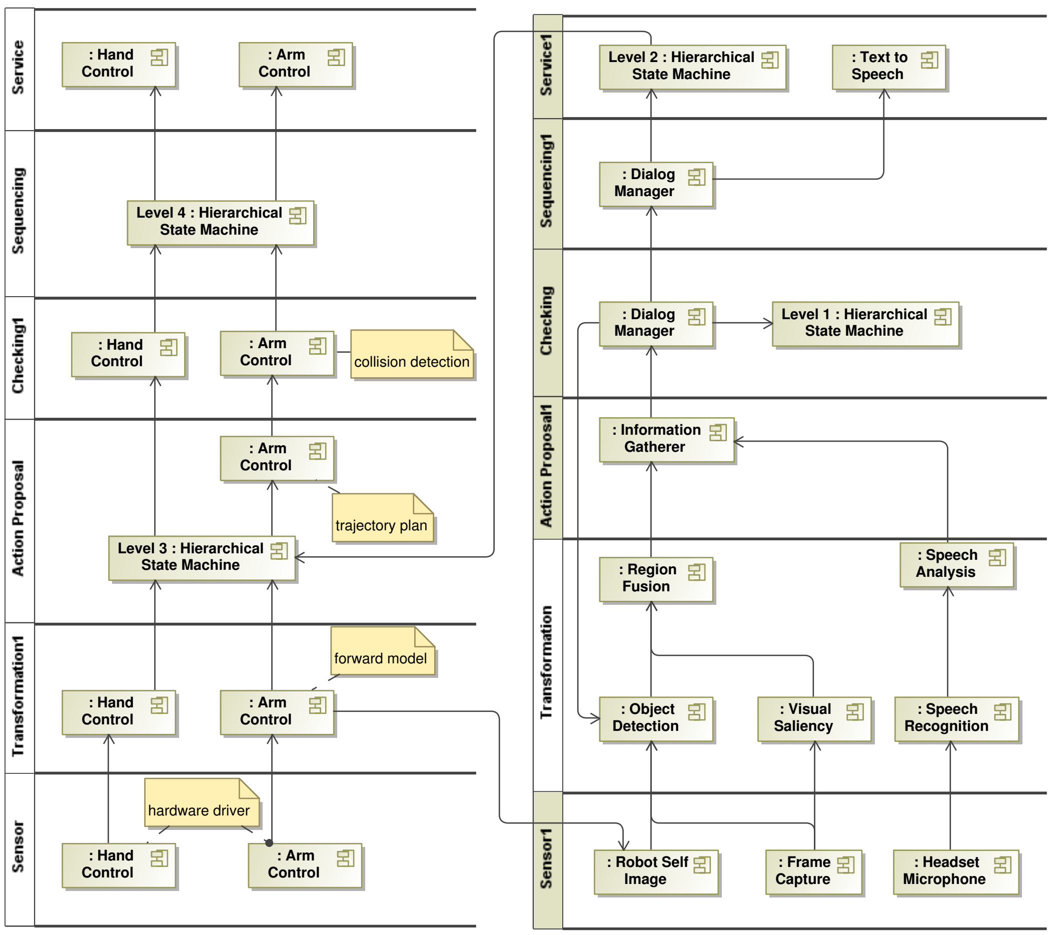


Figure 9.9.: Functional layers of the motion control (left) and interaction (right) sub-systems, and the components they contain.

9.6.2. Motion control subsystem

In the motion control sub-system, the picture is quite different at first blush. There are only three distinct components: The control components for the arm and the hand and the hierarchical state machine. The control components include both sensors, transformation, checking and service components internally. Transformation from joint angles to a robot posture is carried out by an internal kinematic model, collision checking is via an internal simulation engine and the service part implements the control drivers. Furthermore, the arm server also includes an internal action proposal component in the form of a trajectory planner.

It is also notable that the HSM, which has been shown as one component previously, is shown at three levels now. This analysis is the result of a question: How could one component perform so many different functions, without becoming hopelessly incoherent? The answer to that question is found in the hierarchical nature of the state-machine, each level of which has connections to different external components.

At first, this might be interpreted as a fundamentally different design. However, a closer look reveals that the interaction between the other components servers and the various levels of the hierarchical state machine maps quite well. There is a direct association between the hierarchy of the state machine and the functionality associated with the layers: Level 2 contains the mapping from goals to a sequence of steps to carry out and level 3 monitors the execution of these steps, and upon completion, proposes the next step to carry out. The highest level of the HSM maintains the current activity state and can thus check whether a new action would currently be possible without interruption.

This interaction reveals that not only does the "arm control" component internally contain the various functional components of the subsystem schema, their interaction is also sequenced by an external component, the HSM. This sequencing breaks up the execution of higher-level actions into functional stages with striking resemblance to the stages of the subsystem schema. It is still a different design, but not a radically different one.

In conclusion, while the motion subsystem contains all the individual parts specified by the subsystem schema, the implementation as a few, tightly integrated components has led to a different internal design. It should be noted that its design predates the current system and is also more concerned with real-time consideration than the rest of the system. Some of its aspects have been incorporated into the subsystem schema described earlier, whereas some other aspects are still under discussion for a future, potentially consolidated architecture.

9.6.3. Non-verbal feedback subsystem

The non-verbal feedback subsystem is a new one, created in response to user evaluation results. Its components are depicted in figure 9.10. The primary functional reason for its existence (in this system) is that it can be important to communicate that the system is attending (and to what) before the user's activities are fully analyzed.

As introduced in chapter 2, two types of feedback are currently produced:

- *Gaze direction.* The humanoid torso BARTHOC turns its head and eyes to be directed at either the human partner or the region being interacted with.
- *Motion pausing.* When the human starts to speak, the robot arm will slow down in motion, eventually halting, thus providing both an immediate reaction as well as more time to process speech.

The sub-system observes the selected interaction region, candidates for which are produced by "region fusion" (RF) and selected for use by the dialog manager (DM). As the DM selects the region, it may appear counter-intuitive to link from RF instead.

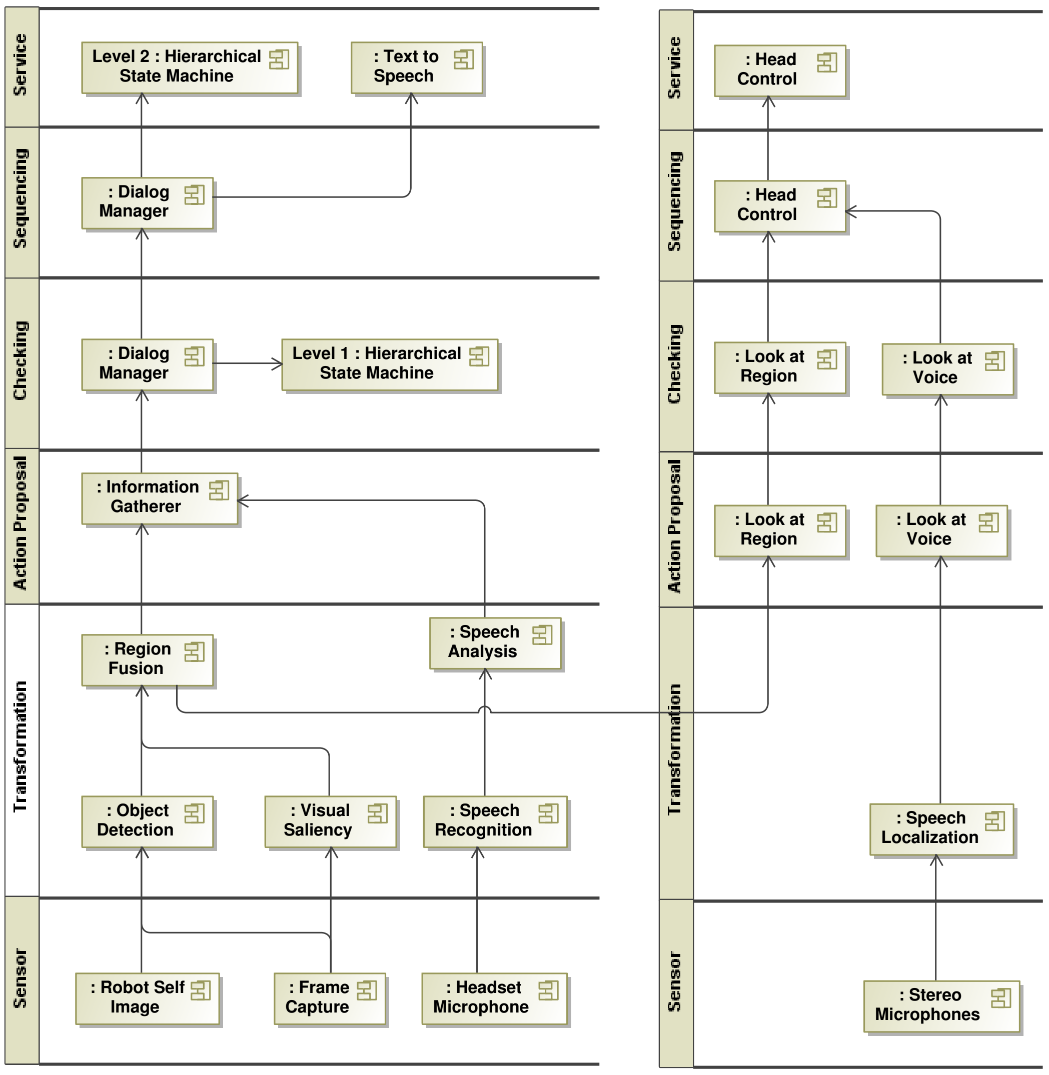


Figure 9.10.: Functional layers of the interaction (left) and feedback behavior (right) subsystems, and the components they contain. Connections are via data or life-cycle event notifications.

However, the link is intended to specify coupling. Because the DM uses the generic life-cycle protocol, there is no coupling specific to it, whereas the data format of the region description does introduce coupling. Hence, the sensor is data-coupled to RF, not to DM.

It should be noted that the non-verbal feedback as realized currently does not perform incremental processing, but instead works on new modalities compared to the initial version. For example, incremental speech recognition would still produce symbolic output. In contrast, speaker detection (one of the new sensors) simply detects voice activity, without regard to its content. Voice activity can be detected with comparatively high reliability and low error rates even while the first word (or even the first syllable) is still being spoken.

9.7. Related Work

Despite considerable work on individual aspects of interaction, learning and collaboration, fully integrated systems that interact with a human while performing a task are comparatively rare. Fortunately, however, the one domain that has seen work from several groups is close to the present work: Collaborative construction. Other relevant work can be found in several robot competitions, both at the AAAI Conference and in the new RoboCup@Home league, as well as in learning social robots.

9.7.1. Collaborative construction scenarios

Collaborative construction, in this context, refers to joint work by at least one robot and one human on a common task. Early work in this area typically has the human command the robot, whereas more recent work moves to a mixed initiative setting, with the robot also being able to elicit help.

Situated Artificial Communicators (SRC 360) (Knoll et al., 1997; Rickheit and Wachsmuth, 2006)

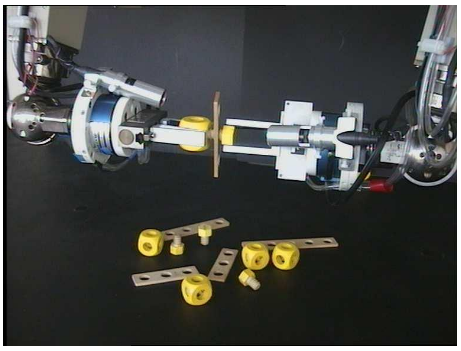


Figure 9.11.: SFB360 setup (Knoll et al., 1997).

An influential early system is the outcome of the special research area 360 “situated artificial communicators”, at Bielefeld University. In this system, human and robot collaborate on an assembly task, with the human instructing the robot in natural language, and the robot replying through actions and speech output. Typical commands include picking up objects and screwing together parts, as well as meta-commentary (“move the arm out of the way”).

The project has been a collaboration between computer science and linguistics departments and thus it is not surprising that an emphasis has been placed on natural dialog, which obeys interaction maxims of human interaction.

In particular, the system has turned away from the so-called "front-end" style, that treats the robot as a passive receptacle of commands. In earlier systems, the operator has provide a detailed, multi-step sequence of instructions, with unambiguous specifications of all parts. This is hard for the human partner and quite error-prone, because it requires to take all actions into account beforehand. In contrast, incremental dialog focuses on providing instructions one at a time, in their matching context, and in response to the current situation and/or errors.

One of the important dialog principles that follows from this approach is the concept of "situatedness". It refers to the use of the situational context (e.g. the objects present (Vorwerg et al., 2006), the configuration of the robot and the perspective of the human) to disambiguate deictic references (Kransted et al., 2006) and for concrete, human-interpretable error responses. While the former aspect has been the subject of many other systems, the latter aspect is often ignored, even though detailed, helpful feedback is an important cue for state transparency. The necessary combination of vision and language for interaction, while challenging, has also been a continuing influence (Bauckhage et al., 2006).

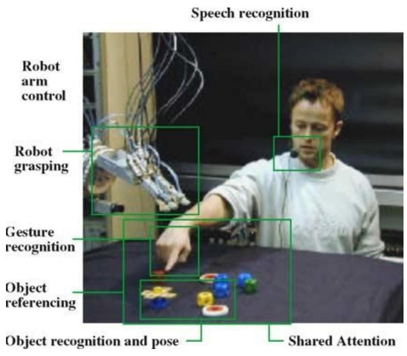
Compared to the present system, the SRC360 system assumes human experts, both with regard to the task at hand, as well as regarding the possible dialog commands. While an emphasis was placed on supporting typical human constructions, the dialog is still severely limited and no guidance about the dialog or the goal is provided at all. Furthermore, no regulatory cues are provided, and a strict one-for-one turn-taking model is enforced.

The system has also been limited to a number of predefined objects, manipulation actions, grasps and dialog. No learning of any kind has taken place. Despite this, the incremental approach towards interaction has shaped many subsequent systems, certainly including the present one.

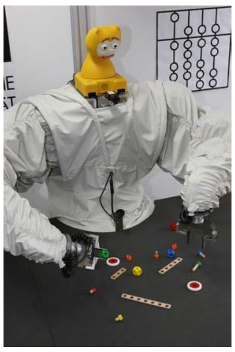
Extended interaction in the BAUFIX domain The BAUFIX objects have proven surprisingly persistent and can be found in several other systems, whose similarities suggest that the have been inspired at least in part by the SFB360 work, but clearly address the shortcomings of that work.

Learning grasps has been the focus of the GRAVIS system by Steil, Röthling, Haschke, and Ritter (2004) (cf. figure 9.12(a)). It uses a combination of visual observation of human demonstration (for initial posture) and physical simulation (for optimization). Similarly, Zhang (2004), focused on learning grasps purely through experimentation. Neither of these systems extends the original interaction capabilities, however.

Collaboration is targeted by the JAST scenario (Rickert et al. (2007); Foster et al. (2009a); and cf. figure 9.12(b)). Here, the human performs the construction with assistance by the robot, who instructs him/her and also passes objects that the human cannot reach. The respective action strategies of the robot have been examined in



(a) GRAVIS setup (Steil et al., 2004)


(b) JAST setup (Rickert et al., 2007)

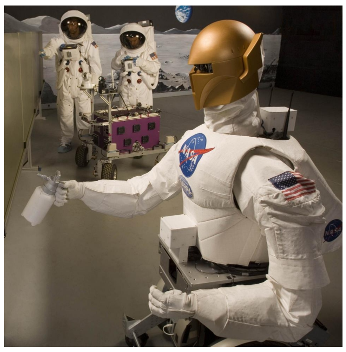

(c) ROBONAUT setup (Fong et al., 2006b)

Figure 9.12.: Collaborative construction robots.

more detail in several case studies: Foster et al. (2009b) has found a slightly positive effect of changing referring expressions when objects are in hand, as opposed to on the table and Foster et al. (2009a) has found that announcing the goal of the action significantly improves understanding by the human and reduces inquiries.

Architecturally, the original scenario as well as the extended versions use a hybrid approach, which is fairly obvious choice when integrating perception with symbolic natural language processing. Despite the fact that several of them use powerful dialog systems, decision making is central to one component. This probably reflects the fact that in all of these systems, dialog is exclusively driven by one side and the other side is restricted to clarification requests that cannot affect the ongoing task.

Robonaut (Fong et al., 2005, 2006a,b) The "Robonaut" is a proposal for a robotic assistant in space, to support astronauts in tasks they cannot do on their own. In common with the present system, the Robonaut uses a mixed-initiative dialog based on natural speech.

The present experiments with Robonaut are on the ground, in a seam welding task (cf. figure 9.12(c)). Two special purpose robots – one for welding, one for inspection – collaborate with a human who holds up the parts to weld. The Robonaut has ambitious goals for collaboration, in particular, the selection of available interaction partners should be automatic for both the robot and the human. User initiative is supported to request robot assistance, and robot initiative supports asking for help with weld inspection. Last, but not least, a particularly impressive aspect of the system is the support for resolution of spatial references, which supports taking the perspective of the user.

In comparison to the present system, both dialog structuring and status feedback are missing, i.e. exactly those aspects that are considered most important for the present work. Now, in contrast, Robonaut targets expert users, so one might expect that guidance is not necessary. However, to the contrary, a user study by Fong et al. (2006b) demonstrates that even with such highly trained personnel, these supporting measures would have been needed. In particular, Fong reports that astronauts did use incorrect commands and this was missed because the system has no speech feedback at all. Furthermore, a grave issue was reported to be the lack of status monitoring: Astronauts had difficulty assessing the status of the robots.

Architecturally, the Robonaut system uses a combination of various approaches: Tasks are described using the Task-Description Language (TDL) (Simmons and Apfelbaum, 1998), though communication occurs not via IPC (Simmons and James, 2001) (as usual for TDL) but through an agent framework, except for the dialog which communicates using the ICE middleware. This combination of approaches has been used because of existing legacy software (Simmons, 2010).

9.7.2. Interactive mobile robots

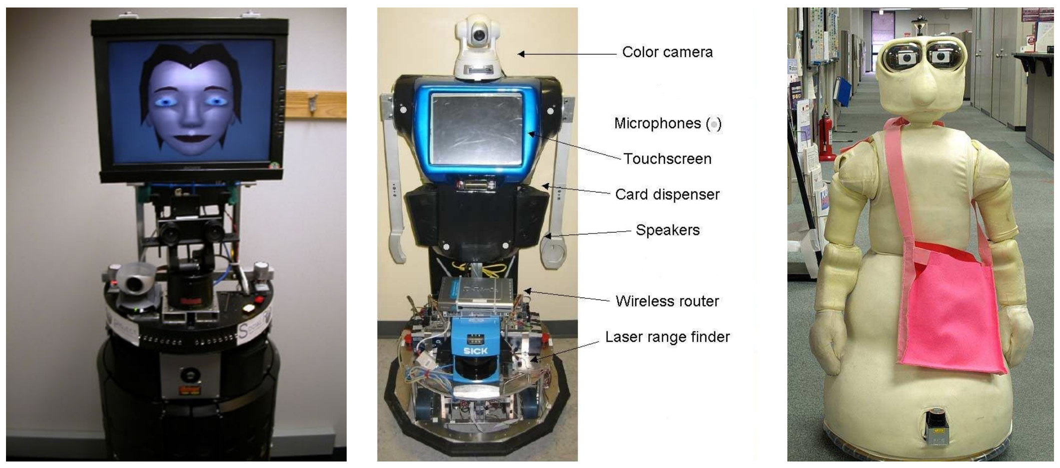
Mobile robots are one of the most popular areas of robotics research right now, and the scenario chapter has already used interaction with them as a motivating example (cf. chapter 2). However, most of of the work on them has concentrated on navigation and mobility, and recently also manipulation. A notable exception are two competition series, the AAAI robot challenge and the RoboCup, which both have acquired a HRI part in recent years. Therefore, representative robots from these competitions will be discussed. Furthermore, Ishiguro et al. (2001)'s ROBOVIE robot is widely used for interaction research in Japan.

General results from competitions Due to the emphasis on navigation issues, Human-Robot-Interaction is still comparatively limited in mobile robots and usually restricted to either commands or inquiries, i.e. one- or two-step exchanges.

Furthermore, because of the need to integrate a large variety of different functional modules, the results from competitions are overshadowed by one blatant result: The huge effort involved, which is accomplished primarily through lots of "elbow grease". In particular, recognized problems are the integration of decision making, how to cope with errors in perception of natural modalities in noisy and highly dynamic environments, and monitoring and reporting the robot's state (Gockley et al., 2004; Maxwell, 2007; Michaud et al., 2007).

One consequence of this is that only few, if any, robots actually manage to complete the challenge tasks completely. However, considerable progress is being made from year to year and the increased interest in such challenges is because they constitute one of the few means of comparing performance of whole systems. Therefore, their importance will likely increase.

Graduate Robot Attending Conference ConferencE (GRACE) A contestant in the AAAI robot challenge with a marked focus on integration is GRACE, with a maximum of five institutions collaborating in 2002. Later instances had fewer par-



(a) GRACE (Simmons et al., 2003) (b) SPARTACUS (Michaud et al., 2007) (c) ROBOVIE-IV (Mitsunaga et al., 2008)

Figure 9.13.: Interactive Mobile Robots

ticipants, but built on this base. Regarding integration of HRI, different approaches have been experimented with over the years.

The 2002 (Simmons et al., 2003) and 2005 (Michalowski et al., 2007) versions modeled interaction through FSMs and coupled it with a task-oriented architecture. The first used speech, the second a touch-screen, for input. Interaction of dialog with other ongoing tasks appears to be very limited. In contrast, the 2003 (Gockley et al., 2004) version used a much more elaborate interaction system, with a dedicated dialog and two different parsers (with the simple one winning out due to its robustness – a familiar result). It sported multiple interactions with other components that carried out longer-running tasks. Little is reported about the realization of these, but they are marked as two-way, which indicates a fairly tight integration.

The change in approaches illustrates that interaction is still much less of a "standard" component than other functional parts, probably owing to the variety of ways it can be used, and the problems with speech recognition in a noisy environment. More positively, the development over the years also illustrates that HRI is becoming a more confidently used tool to solve a robot's challenges.

SPARTACUS (Michaud et al., 2007) A contestant at the 2005 challenge with a very explicit focus on architectural concerns is represented by SPARTACUS. Its task was to register, go to the correct room and hold a talk (the standard conference guest task). SPARTACUS' basic architecture is behavior-based and utilizes a hierarchical priority-schema for arbitration. Furthermore, it emphasizes an adaptation based approach towards rapid integration, through the MARIE framework (Côté et al., 2006). Unfortunately, SPARTACUS could not successfully complete the full challenge due to the various issues, but it covered a few important bases.

To provide coordinated exchange of data between processes, the SPARTACUS architecture uses a so-called "Dynamic Task Workspace" (DTW), which contains a generic representation. This is similar in spirit to the generic task life-cycle. To adapt from the behavior-specific representation to this generic store, Michaud et al. (2007) proposes a "System Know-How" global component. In a later[5] paper, however, Michaud et al. (2006) do not show the SNOW module anymore and describe that it is "by exchanging information through the DTW that motivations are kept *generic*" (my emphasis). It is not entirely clear whether the SNOW module is gone or just not shown, but it is never mentioned throughout the whole paper, whereas the previous one shows it as a part of the high-level architecture. This indicates that the global adaptation module has been replaced with a generic representation, as is argued for here and, if this is indeed what happened, it would be gratifying to see that a system that *has* tried it apparently did change over to a general coordination protocol.

ROBOVIE-IV The most recent development of the robovie (Ishiguro et al., 2001) series, Robovie-IV (cf. figure 9.13(c)) is a mobile robot explicitly designed for social interaction, that tries to engage office workers in a dialog about themselves (Mitsunaga et al., 2006). Notably, the robot engages people in a dialog on its own, by saying "Hi" when it detects them, and it continues this behavior during the interaction. From a six-week study (Mitsunaga et al., 2008), it can be discerned that the robot could successfully engage people in interaction. Unfortunately, while a number of people were polled about their impressions, and thus the robot cannot be wholly unsuccessful, the overall interaction success has not been measured.

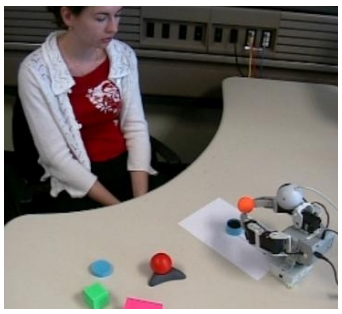
The architecture is based on "tasks" (behaviors) implemented in a scripting language, with the dialog task coordinating the others. It is not specified how this coordination takes place or how it is formalized. In any case, the robot does not perform longer running tasks that it could report on.

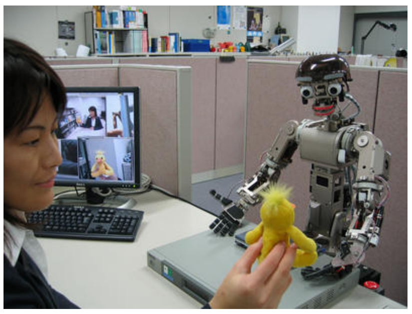
9.7.3. Social Robots

The study of social robots is particularly relevant for the scenario design, as it addresses how to use social cues to improve Human-Robot-Interaction. While there is usually a focus on paralinguistic cues, the goal of embedding guidance into the task interaction (as opposed to an "introduction speech" or just having the robot give commands) is shared with the current work.

Socially Guided Exploration Thomaz and Breazeal (2008) outline, in common with the present work, that most previous systems have been restricted to either tutoring or exploration and that it is useful to combine the two. In Thomaz and Breazeal (2008), they suggest interpreting human gaze and pointing to increase object saliency, which assumes human tutoring behavior and expects the human to know how to act.

[5]While published earlier, the one about the 2005 architecture has been delayed due to journal publication.


(a) Socially Guided Learning (Thomaz and Cakmak, 2009)


(b) Learning Attention (Nagai, 2005)

(c) Demonstration for BIRON

Figure 9.14.: Social Learning Robots

In contrast, in Thomaz and Cakmak (2009) the robot is autonomous and just uses its own gaze in a looking-back-and-forth way to indicate failure and thus induce the human to act.

This uses comparatively little instruction: Subjects are told the goal and not to move when the robot moves. The robot neither understands nor produces speech and the interaction is extremely limited, but it demonstrates the value of such simple cues. In comparison to the present work, there is still the requirement for instruction and, furthermore, there is no interactive dialog about the task, just action on the same objects.

Unfortunately, not much is known about the architecture of these systems, except that they are using the "C5M" architecture, an unpublished modification of Blumberg (1996). It is described as a behavior-based architecture.

Learning Joint Attention Nagai et al. (2003b) presents a system whereby the robot learns reading the partner's gaze direction in an unsupervised manner, thus achieving joint attention. Originally, this is based purely on statistical correlation. In Nagai et al. (2006), the same task is explored using feedback from the partner, both regarding correctness and to incrementally increase the difficulty of the task. In the latter, it could be demonstrated that learning is sped up.

This scenario demonstrates how to scale a learning task that would usually be extremely tedious to do through demonstration to an interactive scenario, by drawing from ideas of human development – the robot's capabilities are initially more limited, which paradoxically eases the learning task. Furthermore, it demonstrates how adaptation by the human can provide the necessary stimulus for improved performance. However, it again assumes the human to be an expert and, furthermore, uses a GUI to provide feedback, instead of integrating that into the scenario.

Human Augmented Mapping A system built in the COGNIRON project – which the author's working group at Bielefeld University has been involved in – is the

"BIRON" robot companion (Fink et al., 2004; Wrede et al., 2007). It is a companion robot, living in people's homes and its defining scenario is the so-called "Home Tour" in which the human shows the robot around the home and explains it (Li et al., 2004). In other words, the scenario is completely driven by the human.

In contrast to other such companions, however, and this is also the aspect closer to the current work, BIRON has later been extended towards mixed-initiative dialog (Booij et al., 2008; Hanheide and Sagerer, 2008). This is triggered by navigation: Based on the detection of transitions or the confidence level of localization, a clarification dialog can be entered. The goal here is quite different from the present system: The clarification does not provide the human with additional guidance, it merely serves the localization component. The dialog also already emphasized the role of state-reporting during the interaction, to improve usability (Wrede et al., 2007).

The integration requirements of this interaction led to implementation of the task-state pattern in an early form (Peltason et al., 2009a), which formed the basis of the formalization and toolkit by the present author described in chapter 4.

BIRON's architecture was planned to be a reactive-deliberative hybrid (Li et al., 2004), but the planner has never been implemented. Instead, the system used an FSM to specify the scenario, first directly implemented and today in a sequencing component (Spexard and Hanheide, 2009). The system also used RPC-style interaction throughout, which has been partially converted to a pub/sub and event-based style now, though the tight coupling and corresponding assumptions of the RPC interaction do make this a more involved undertaking. It can certainly be said that the integration experiences of the BIRON team, which the present author has heard about often, have been a decisive influence in designing the current, more loosely coupled, event-based architecture.

9.8. Summary

The initial autonomous object learning system has combined a flexible, robust dialog manager, a powerful manipulator subsystem and a saliency-based action selection component that proposes the next interaction goal. The integration has been successfully carried out using the XCF middleware toolkit for exchange of extensible representations. For high-level interaction, the task-state pattern has been used. This coordination model has been found to be a desirable abstraction, but its implementation posed some open questions, which led to a service-level toolkit that encapsulates communication complexity.

The system has been evaluated using the video study paradigm regarding the efficacy of the dialog structuring approach and the overall functionality. A clear difference in interaction consistency was observed between closed questions (very consistent behavior) and open questions, which demonstrates the value (or, indeed, the necessity) of dialog structuring support for complex HRI.

To improve the interaction, particularly to better teach grasping and give more immediate feedback, an extended version of the system has been constructed. Both a haptic input device with grip type classification and a third subsystem, for non-verbal feedback, have been added. Integration with the existing subsystem has, again, been performed using the task-state pattern. Furthermore, information giving capabilities of the interaction subsystem have been extended. Last, but not least, the feedback behaviors and additional data fusion components in the extended version have been rapidly constructed using the FTS toolkit.

10. Conclusion

This thesis has explored two architectural aspects for building robot software systems: i) the communication interface for component coordination, and ii) component composition for rapid construction and change. These aspects are considered to be of particular importance for building larger-scale systems in a principled fashion, with the first addressing inter-component construction and the second addressing within-component construction.

For each of them, initially a common approach has been realized as a toolkit, and integrated into a Human-Robot-Interaction scenario, dubbed "Curious Robot". While the integration itself also constitutes a contribution, the primary goal has been to enable empirical study of the architectural qualities resulting from the chosen approaches. Therefore, three major iterations of the scenario have been built by a team of developers, to allow comparative empirical analysis. Empirical analysis of these or comparable approaches is rare, and comparative analysis has not been done before at all. Moreover, the iterative improvement of the approaches has also led to some novel refinements and generalizations.

10.1. Communication for Coordination

Regarding the first aspect, communication for coordination, the task-oriented approach to coordination has been used as a basis. In this, a combination of an abstract state-machine and task-specific representations is used. The suggested advantage of this approach, a unified treatment of functionally different components, has been demonstrated through multiple case studies.

While task-oriented coordination is only one of several proposed approaches, through an analysis of the literature an important commonality with a number of other coordination methods could be established: The use of an abstract state-machine for communicating task execution progress. While the abstract nature of the state-machine is often obscured by inconsistent naming, the actual use and historical progression towards more abstraction suggests a pattern. The identification and formulation of this pattern, under the name "Abstract Task-State Pattern" forms the core of contributions in this area.

Earlier realizations of this pattern usually placed the mapping between task-specific states and the abstract state-machine in a dedicated coordination component. In this thesis, it is suggested that placing it in the executing component instead can lead to improved observability of task execution, and improved de-coupling on the system level. This has been demonstrated through a case-study of the third iteration of the "Curious Robot" where-in a new sub-system could be added, and its activity coordinated with the earlier sub-systems, through the use of observation alone.

Finally for this area, a toolkit-based implementation of the pattern has been explored, which could be shown to reduce implementation effort and reduce component complexity.

Outlook

Throughout the work presented, the Task-State-Pattern has emerged as a simplifying and reusable building block for system construction. Part of this success is due to the flexibility afforded by a replaceable life-cycle, which allows tailoring a toolkit to the system at hand. However, it is still unclear whether this flexibility is needed in full, or whether the same simplicity with room for growth could also be had from one, fully generic life-cycle. A potential drawback would be complexity, an advantage increased commonality.

One promising avenue to explore this question would be to apply the approach in systems of a different structure. The present system has derived many of its requirements from the tightly integrated Human-Robot-Interaction, so it might be insightful to explore tele-operated or fully-autonomous systems instead.

A different avenue of development would be exploring the integration of the pattern into communications middleware. It has increasingly become clear that it performs a similar function, but also that it derives part of its ability to simplify from having more domain knowledge available (in the form of the life-cycle). Therefore, keeping this advantage while moving to deeper layers would probably be an insightful challenge.

In the other direction, research on the pattern has so far side-stepped high-level coordination, such as hierarchical dependency structures. This is intentional – the pattern should be able to be used with several different coordination structures. However, it certainly has the consequence that even simple constructs, such as sequential or parallel execution, are not yet covered. As one of the goals of the pattern is to make transition to a full coordination package easier, it might be useful to explore whether providing simple combinations of patterns within the toolkit might facilitate "moving up" to more complex coordination approaches.

10.2. Rapid Component Composition

Regarding the second aspect, rapid component composition, the data-flow method of constructing components from a graph of smaller constituent nodes has been chosen as the base method. Due to the explicit connection model and externally specified node configuration, it lends itself well to rapid composition and change, and nodes are a potentially good level for reusing functionality. It was not clear, however, whether the approach is suitable for all kinds of components, as it has so far been primarily used in specialized domains such as signal processing.

Therefore, this thesis has explored the use of data-flow based component construction for components typical of robotic software systems. Through empirical analysis of the size and complexity of constructed graphs, it could be shown the that the approach is ideally suited for transformation-heavy components, such as data fusion.

Moreover, it was found that while data-flow is generally applicable also for implementing protocol driver implementations, such as those used for serial control of micro-controller-based robots, its suitability differs depending on the structure of the protocol. In particular, for protocols where global state must be maintained, data-flow construction appears to be less well suited and it has been suggested that a combination with an explicit state-tracking mechanism would be useful, which would leave the area of pure data-flow.

Classically, data-flow has been used at different levels of granularity, with a trade-off between the complexity of the structure and the re-usability of individual nodes. In general, re-use has an identification and learning cost that must be weighed against the size of the functionality gained. This favors larger nodes. An opposing force is that smaller nodes tend to incorporate fewer assumptions, which improves their transferability. To address this, the filter-transform-select principle has been proposed. It generally distinguishes filtering nodes (which incorporate decisions) from transformation nodes. Furthermore, to improve predictability of graph execution, it distinguishes filter nodes, which determine whether processing occurs at all, from selection nodes, which choose between alternative paths. By applying this principle at the highest level of a node, the node granularity can be decided.

Regarding the original goals of improved re-use and changeability, it has been found that the data-flow model can produce generic nodes which lend themselves well to re-use. Structural changes were also found to be possible on the model alone, e.g. to change resource-access models for hardware control, while keeping the protocol itself stable, or to add optional visualization to existing transformation-oriented components.

Outlook

There is both considerable existing work on component composition, as well as ongoing work into model-driven approaches, which has direct relations to the data-flow oriented component composition approach. One clear candidate for further work is on higher-level abstractions and domain-specific languages: The latter, in particular, are already being explored and promising early results are available. While they can already be translated to the regular graph specification, this would not result in a higher-level view. Thus, appropriate visualizations must be found. Further, work to combine the approach with Harel statecharts is ongoing, thus improving the representation of state in a principled way.

One looming issue in using data-flow with the various process models is a potential explosion of complexity: There is a reason why such approaches are not yet popular for general purpose applications and while the current implementation has been deliberately limited to a well understood model, care must be taken not to increase the complexity in the quest for more power. Therefore, similar to the way the present work has suggested a novel decomposition strategy, potential future work might lie in providing strategies to reduce and/or manage this complexity.

10.3. Integration in an HRI scenario

The present thesis considers the primary value of an architecture to be to facilitate system construction and maintenance in a team. Accordingly, all of the studies presented have been taken from successive iterations of a fully integrated system, the “Curious Robot”. The advantages shown during these studies are all architecturally related, and have demonstrated that the proposed principles can achieve the desired loose coupling and separation of concerns.

Loose coupling, however, is only an achievement when combined with the necessary functional integration. Here, the system has actually increased functional integration beyond the state of the art in a consistent fashion.

Firstly, the integrated systems demonstrate a particularly high amount of functional integration between the speech dialog and the acting components. This allows the dialog component to provide detailed feedback throughout action execution, in a generic manner based on the abstract task-states. Furthermore, the combination of abstract and task-specific information in passively observable state notifications also allows more specific verbal commentary, at a low coupling cost.

Secondly, placing the mapping from task-specific state to abstract state in the server components has increased observability of overall system activities. This has been demonstrated by adding a new sub-system solely based on observation.

A second goal has also been to improve team-based system construction, through a guiding architecture. Here, it has been argued that the common task-state pattern provides a re-usable and well-understood interaction pattern that reduces communication effort. By applying a common concept throughout the system, developer understanding is increased, and interfaces are simplified. This could also be supported by demonstrating that a toolkit-based approach decreases effort and complexity for server component developers.

Last, but certainly not least, the whole system has also been empirically tested through user experiments. This primarily took the form of video studies, wherein users judged the naturalness and understandability of the resulting interaction. As a non-rigorous complement, a simplified version of the system has also been used with great success by many naive users at a trade fair. This has demonstrated that the general architectural concepts are sound, but more evaluation would certainly be needed to draw more general conclusions regarding the Human-Robot-Interaction concepts.

Outlook

The integrated system obviously poses many open questions, and evaluating the full system in full user studies is a crucial next step, to validate the work done so far, and identify open issues. Work on this is currently ongoing.

From an architectural point of view, the interplay between components beyond the task-state pattern has already become more interesting. The over-arching question here is how to push recurring structure into the architecture, e.g. supporting toolkits, thus reducing the developers work, without constraining development too much.

Furthermore, the present author believes that empirical study of existing systems is crucial for gaining more insight into which approaches really are beneficial, and which are not. However, such empirical research is often hampered by a lack of tools, and a lack of comparability. Promising developments in this direction are the emergence of common robot software frameworks, such as OpenRTM[1], ROS[2], Yarp[3], and others. These provide at least a common communications infrastructure. Analysis of the interactions within these systems is still very hard, however, due to the required detailed knowledge of the components at hand. For one of the frameworks (ROS), the task-state pattern is already being used, and ongoing work has examined the applicability of analysis based on the pattern alone, to elucidate system dynamics, with promising first results. Broadening this approach to the other frameworks would therefore be one ingredient in improving cross-system comparability that the present author intends to explore.

[1]`http://www.openrtm.org/`
[2]`http://www.ros.org/`
[3]`http://eris.liralab.it/yarp/`

A. Software Metrics

A.1. Metrics vs. Case Studies

As Basili et al have pointed out early on, many issues of interest for software engineering, such as the benefit of a certain development technique or practice, are comparatively more expensive to determine manually, because long observation of human activity is required. Furthermore, the observation may disturb the process[1], thus shedding doubts on its validity. Therefore, metrics represent the attempt to create *automatable* measures that correspond to these issues (Basili and Reiter, 1979).

That the metrics do, in fact, correspond to the issues of interest is a matter of intense debate and, to this day, considerable research activity in the field of software engineering. The *predictive* power of metrics has been particularly criticized and it can probably be said that many metrics do not have as much predictive power as was hoped for, particularly when taken beyond their original context.

A well-studied example of use out of context is McCabe's Cyclomatic Complexity (McCabe, 1976), which counts the number of independent execution paths through a procedure. His formulation of the metric exactly corresponds to the number of distinct tests that must be created to exercise all paths, that is, it measures test effort. However, it also corresponds, albeit not as exactly, to the control flow complexity and further correlates fairly strongly with overall length. Length of the code, in turn, is known to be a weak predictor for defect density, so the idea that McCabe complexity might be a better predictor for defects was fairly obvious. After all, it measures not just length but also independent paths. However, this could not be shown and on the contrary, the available evidence indicates that it has weaker correlation with defect density than length of code (Shepperd, 1988; Fenton and Neil, 1999).

In this thesis, the metrics are always embedded in case studies and thus presented in context. Moreover, they have only been used for *comparison*, not for prediction. The concrete metrics used are now shortly summarized

Chidamber and Kemerer Object-Oriented Metrics

Chidamber and Kemerer's metrics were the first metrics to be proposed directly for object-oriented development. Like all metrics, their relevance to practical concerns is not without criticism, but both analytical and empirical evidence provide some support for their claims (Chidamber and Kemerer, 1994; Basili et al., 1996).

[1]Known as the "Hawthorne" effect, after (Endres and Rombach, 2003, p.232).

These metrics are concerned with coupling and cohesion properties, which are widely believed to have an impact on the maintainability of software. Six different metrics are proposed, with all of them computed per class:

1. *WMC: Weighted Methods per Class.* Enumerate the methods of a class as $M_1, ..., M_n$ and let $c_1, ..., c_n$ be the methods' complexity (thus their weight). Then WMC is simply the sum of all c_i.

   The weighting method to be used is left unspecified by Chidamber and Kemerer, to allow different classical complexity measures (or just 1, for a method count). Therefore, the metric is only comparable between tools with the same settings.

2. *DIT: Depth of Inheritance Tree.* This metric counts the number of superclasses of a class. In cases of multiple inheritance, the maximum of the path lengths is used.

3. *NOC: Number Of Children.* The number of immediate subclasses.

4. *CBO: Coupling Between Object classes.* The number of other classes that are coupled to the class in question. Two classes are coupled when methods of one class use methods or attributes defined by the other class.

5. *RFC: Response For a Class.* An unintuitively named metric, RFC is equal to the size of the "response set", which is defined as the "set of methods that can potentially be executed in response of a message received by an object of that class". This explanation is slightly ambiguous because obviously, methods executed may themselves execute other methods (including those of the original class) and it is not clear whether these methods would also contribute to the count.

   Fortunately Chidamber and Kemerer also provide a more formal definition, as follows. When $\{M\}$ is the set of methods of the class and $\{R_i\}$ is the set of methods called by method i, then the response set is defined as $RS = \{M\} \cup_{\text{alli}} \{R_i\}$. From this definition, we can discern that only the methods called *directly* from a method of the class are to be included in the count.

   One last item of note regarding the RFC metric is that it explicitly considers only "methods" called. In languages such as C++, which have both object methods and non-object functions, the latter may be excluded and this is apparently what Chidamber and Kemerer themselves have done (Chidamber and Kemerer, 1994, footnote 27). However, it could certainly be argued that functions also introduce external coupling and should therefore be included in the count.

6. *LCOM: Lack of COhesion in Methods.* For the methods $M_1, ..., M_n$ of a class, let $\{I_i\}$ be the set of instance variables used by method M_i. Now consider all possible pairs of sets and let $P = \{(I_j, I_j) | I_i \cap I_j = 0\}$, be the pairs which have

an empty intersection, and $Q = \{(I_j, I_j) | I_i \cap I_j \neq 0\}$ be the pairs with non-empty intersections. Then $LCOM = |P| - |Q|$ if $|P| > |Q|$ and 0 otherwise. In other words, this subtracts the number of pairs which share access to at least one instance variable from those pairs which do not share anything.

Some of these metrics may appear skewed – for example, the LCOM metric is very sensitive to the number of methods. Due to the number of pairs being of the order O(n^2), classes with many methods often have much larger LCOM values than classes with few methods, even when they have more methods that are cohesive in relative terms. Arguably, though, classes with many methods can be more difficult to grasp mentally, so a lack of cohesiveness may actually be worse in them and this would validate the way the LCOM metric is computed.

That said, the primary value of these metrics is irrespective of their absolute values: They can be used to quantify *changes* in cohesion and coupling.

Churcher and Sheppherd (Churcher et al., 2002) note that derived measures, such as Chidamber and Kemerers metrics, require precise definitions of underlying metrics such as the number of methods in a class.

A.1.1. Tools Used

To compute the source-code metrics, three tools have been used:

- *Chidamber Kemerer Java Metrics (CKJM) (Spinellis, 2007)* computes Chidamber and Kemerer Metrics for Java software. It works on Java bytecode and can thus be used for all Java programs used, regardless of whether source code is available or not. CKJM computes the WMC metric using 1 for all methods and the DIT metric is only accurate when all required JAR-files are on the class-path. In addition to the classical CK metrics, it also computes two additional but similar metrics:
  - *Ce: efferent Coupling.* Ce is the inverse of CBO, i.e. it measures outside references to a class
  - *NPM: number of public methods.* In contrast to WMC, this metric counts public methods only.
- *sloccount (Wheeler, 2004)* SLOCCount, for Source Lines of Code Count, determines the number of actual source lines, ignoring comments, empty lines, etc. It supports all languages in use in the present system.
- *Dependency Finder (Tessier, 2009)* is a suite of tools for analysis of Java programs. It offers a range of interesting functionality, including metrics, but in this thesis, it was used merely to trace dependencies and output them as XML for further analysis. The produced dependency information is on the method-level, i.e. for each method in the class-files analyzed, it produces a list of methods called and the classes they belong to. As all the naming information is present in string form, it is easy to post-process this information to compute all dependencies going to a particular package, for example.

B. Full Example Frames from Interaction

This appendix contains example frames from interaction sequences at larger size than possible within the main text. Several of these images are composited from video and application screen-shots.

B.1. "What is That?" Sequence

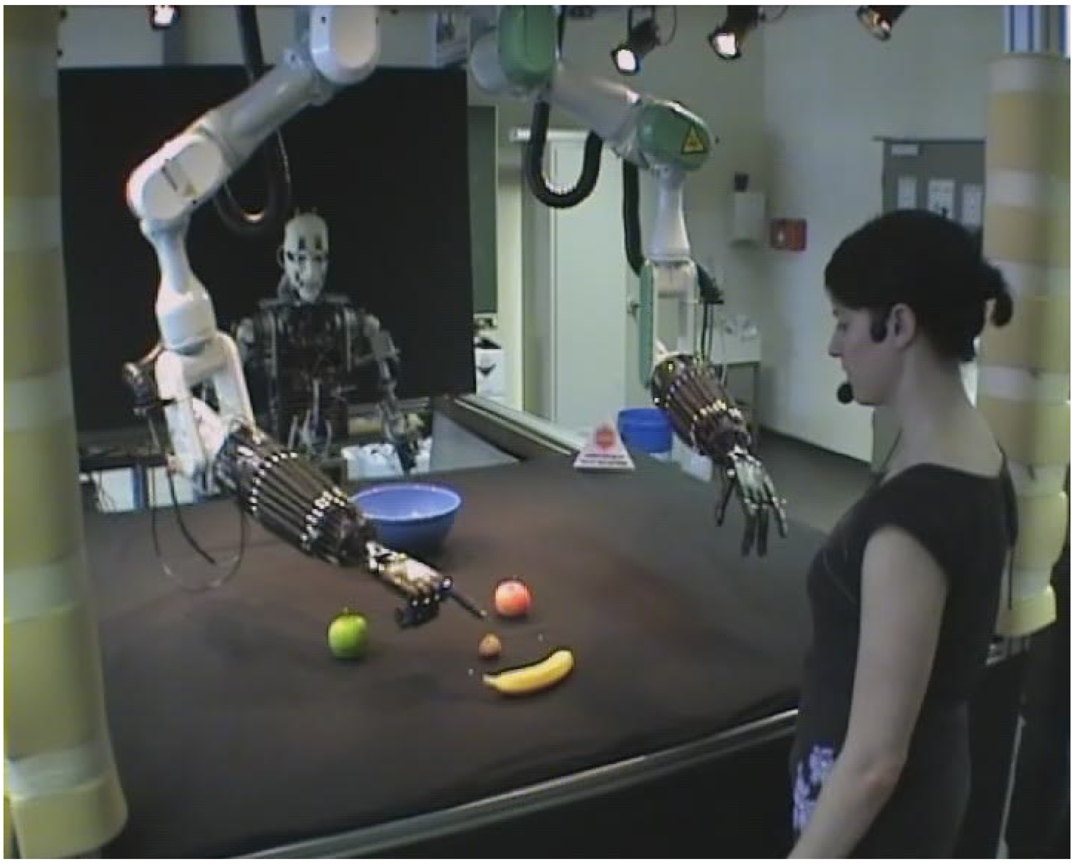


Figure B.1.: "What is that?"

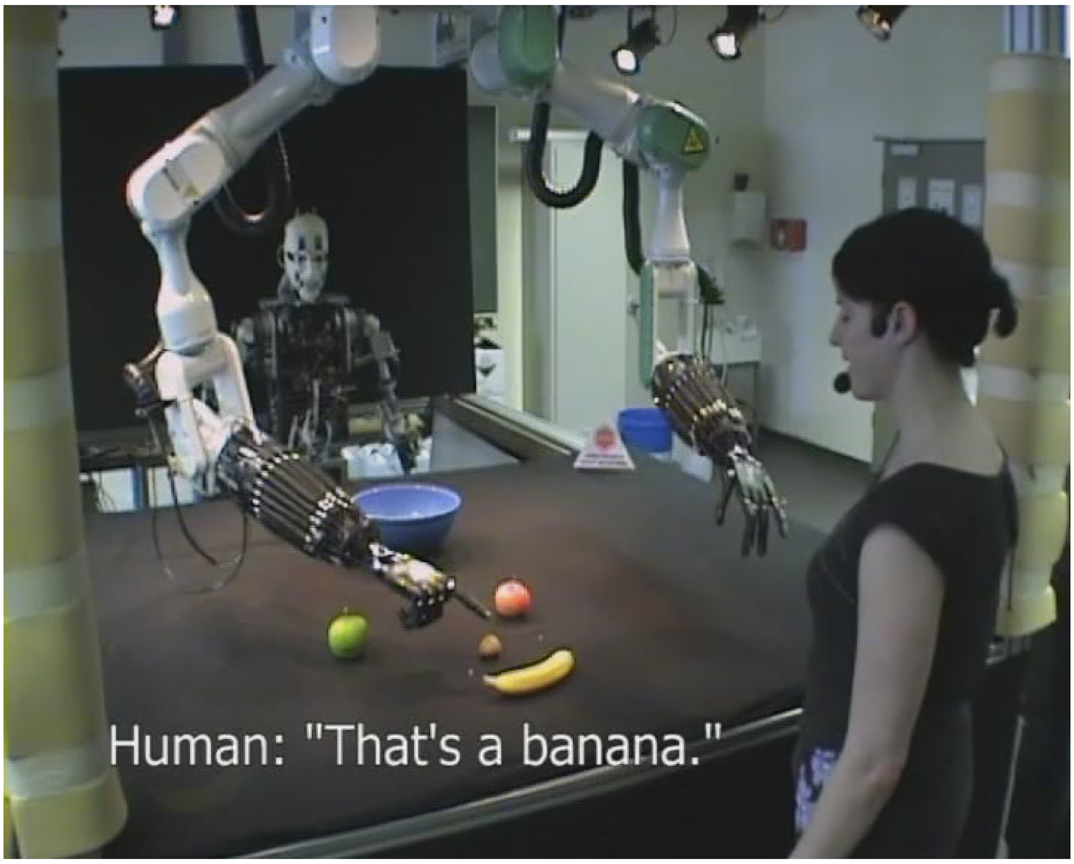


Figure B.2.: “That is a banana”

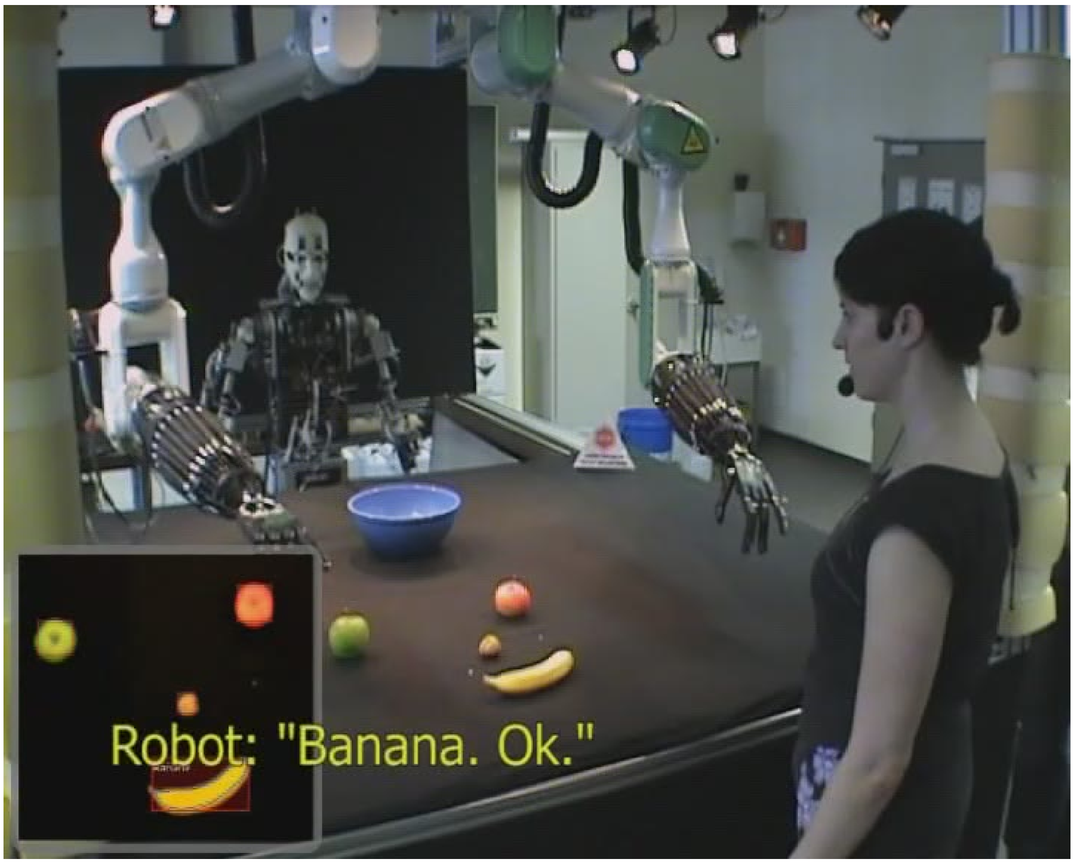


Figure B.3.: “Banana, OK.”

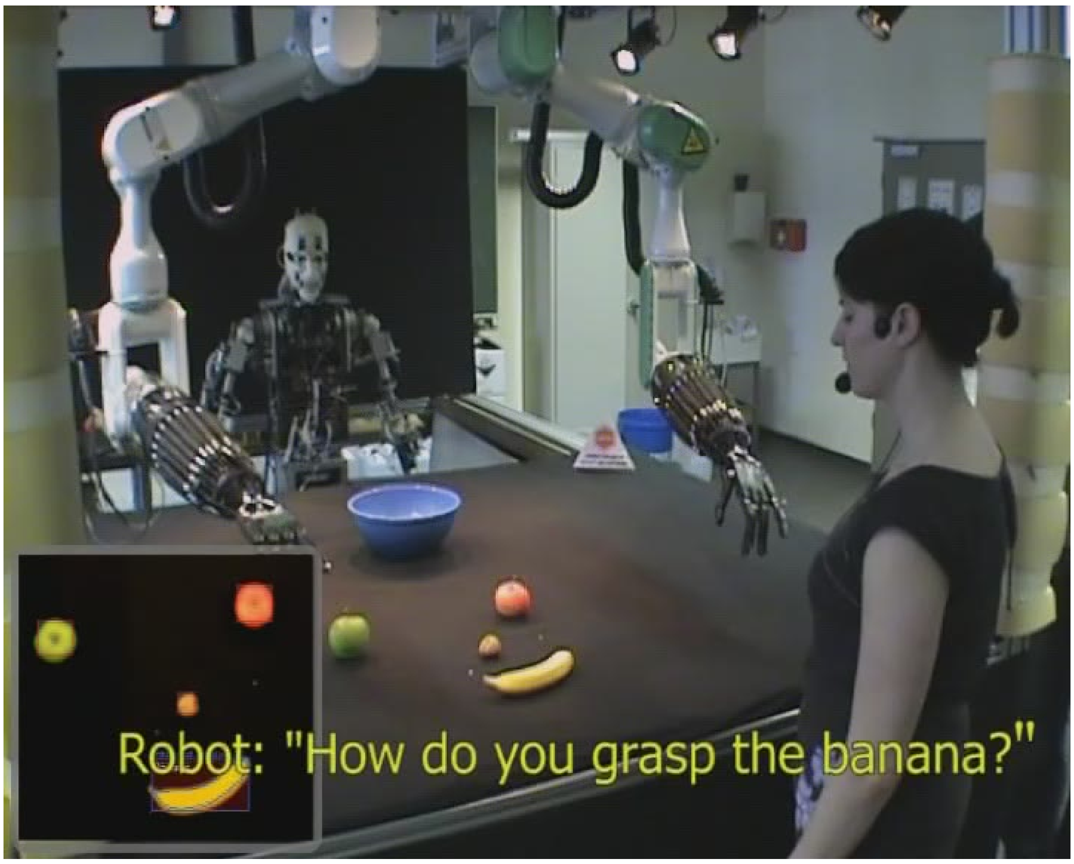


Figure B.4.: “How do I grasp that?”

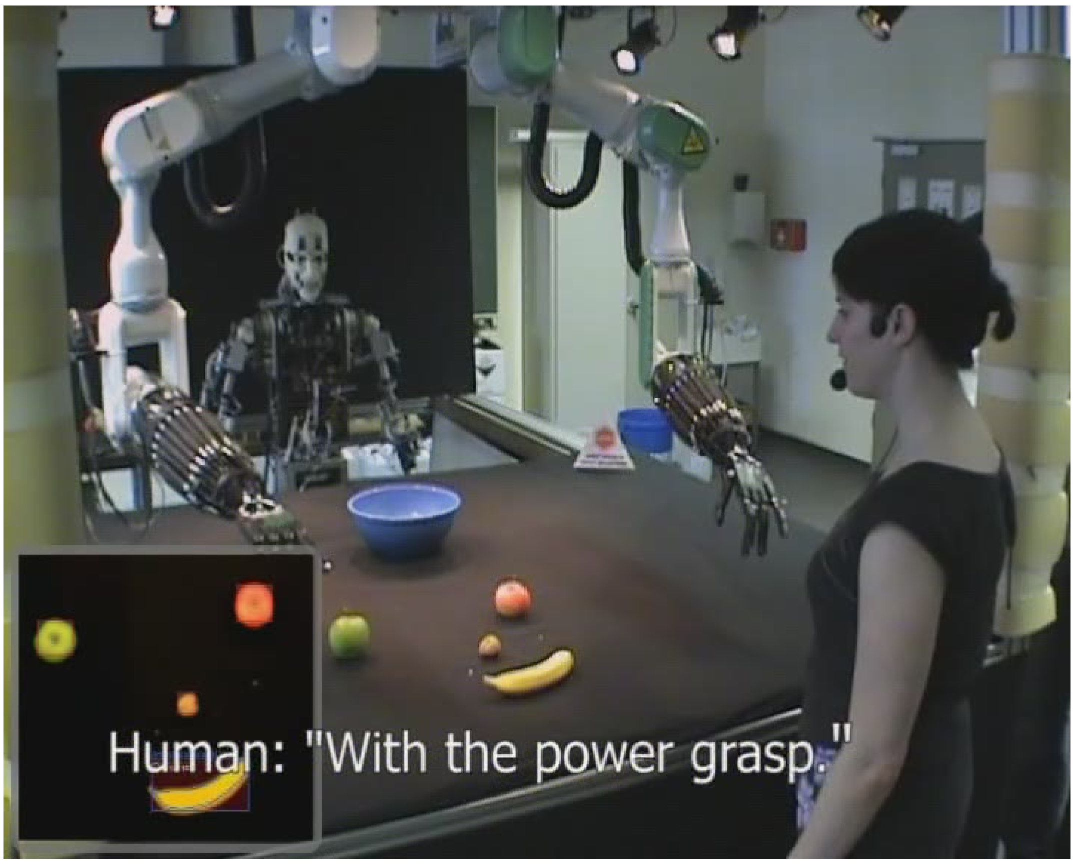


Figure B.5.: "With the power grasp"

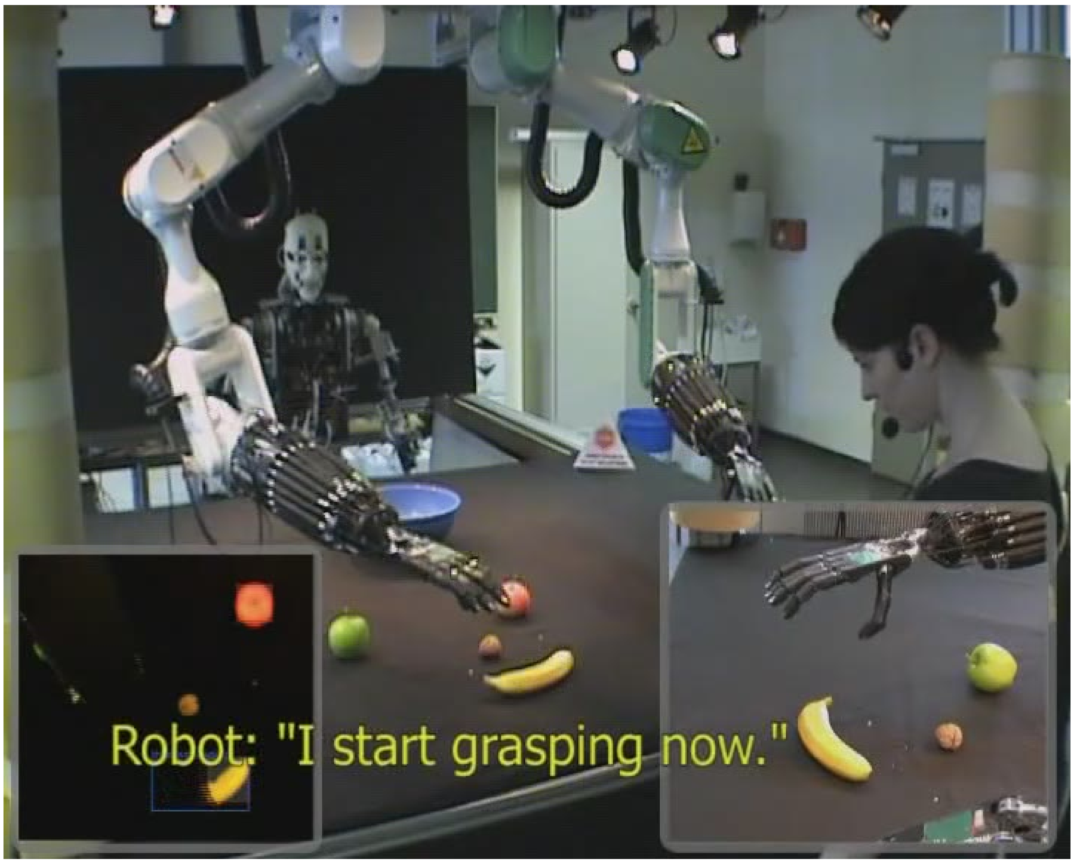


Figure B.6.: "I start grasping now"

C. Graph and Document Examples

C.1. Graph Specification Language

C.1.1. Document Type Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Copyright 2008, 2009 Bielefeld University
Copyright 2008, 2009 Ingo Luetkebohle <ingo@fargonauten.de>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, version 3.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
-->

<!--
    TODO define vocabulary identification data
    PUBLIC ID : -//vendor//vocabulary//EN
    SYSTEM ID : http://server/path/__NAME__
-->

<!ELEMENT __ROOT__ ANY>
<!ATTLIST __ROOT__ version CDATA #REQUIRED>
<!ELEMENT model ( node* | select* | fuse* | sync* | mark* |CDATA* )*>
<!ELEMENT node ( arg* )>
<!ATTLIST node
  type CDATA #REQUIRED
  name ID #IMPLIED
  source CDATA #IMPLIED
>
```

```
<!ELEMENT arg ANY>
<!ATTLIST arg
  type CDATA #REQUIRED
  name CDATA #IMPLIED
>
<!ELEMENT select (target+)>
<!ATTLIST select
  source CDATA #IMPLIED
  selectMax CDATA "1"
>
<!ELEMENT target (filter, node+)>
<!ELEMENT filter (arg*)>
<!ATTLIST filter
    type CDATA #REQUIRED
>
<!ELEMENT fuse EMPTY>
<!ATTLIST fuse
  sources CDATA #REQUIRED
  names CDATA #IMPLIED
  useOnceOnly (true|false) "true"
  checkComplete (true|false) "true"
  required CDATA #IMPLIED
>
<!ELEMENT sync EMPTY>
<!ATTLIST sync
  source CDATA #REQUIRED
  name CDATA #IMPLIED
  label CDATA #REQUIRED
  type CDATA #REQUIRED
  point CDATA #REQUIRED
>
<!ELEMENT mark EMPTY>
<!ATTLIST mark
  type CDATA #REQUIRED
  nodeName CDATA #REQUIRED
>
```

C.2. Example graph specifications

C.2.1. Curiosity Generation Specification

```
<?xml version="1.0" encoding="utf-8"?>
<model engine="BreadthFirst">
  <!-- saliencies -->
```

```
<node type="MemorySource">
  <arg type="StringChild">ShortTerm</arg>
  <arg type="StringChild">INSERT</arg>
  <arg type="StringChild">/saliencies</arg>
</node>
<node type="Event2Document"/>
<node type="StaticUnpack">
  <arg type="StringChild">SalientPointArray</arg>
</node>
<node type="NonEmptyArray"/>
<node type="ResolutionScale"/>
<node name="saliencies" type="Array2List"/>

<!-- handle reset -->
<node source="null" type="MemorySource">
  <arg type="StringChild">ShortTerm</arg>
  <arg type="StringChild">INSERT</arg>
  <arg type="StringChild">/SYSTEMRESET</arg>
</node>
<node name="gripClear" type="ListClear"/>

<!-- remember information on learned grips -->
<node source="null" type="MemorySource">
  <arg type="StringChild">ShortTerm</arg>
  <arg type="StringChild">REPLACE</arg>
  <arg
      type="StringChild">/InteractionRegion/STATUS[@value="completed"]</arg>
</node>
<node type="Event2Document"/>
<node name="gripAdd" type="ListAdd"/>
<node source="gripAdd,gripClear" name="gripList" type="ListCollector"/>

<!-- regionlist -->
<node source="null" type="MemorySource">
  <arg type="StringChild">ShortTerm</arg>
  <arg type="StringChild">INSERT</arg>
  <arg type="StringChild">/RegionList[not(InteractionGoal)]</arg>
</node>
<node name="regiondoc" type="Event2Document"/>
<node type="StaticUnpack">
  <arg type="StringChild">ObjectRegionArray</arg>
</node>
<node type="NonEmptyArray"/>
<node name="objectregions" type="Array2List"/>
```

```

<!-- rank regions -->
<fuse sources="saliencies,objectregions"
 names="regions.saliency,regions.objects"
 useOnceOnly="false" checkComplete="true"/>
<node type="RegionInfoRank"/>
<node type="PeriodicFilter">
  <arg type="Integer">500</arg>
</node>

<!-- start xml based steps -->
<node name="InteractionRegion" type="Pack">
        <arg type="StringChild">ObjectRegion</arg>
        <arg type="StringChild">InteractionRegion</arg>
</node>

<!-- fuse prior information -->
<fuse sources="InteractionRegion,gripList" names="fuse.target,fuse.background"
        useOnceOnly="false" checkComplete="false"/>
<node type="CompareAndFuse">
        <arg type="XPath">/InteractionRegion/Region/Object/@detectorLabel</arg>
  <arg type="XPath">/InteractionRegion/Region/Object/Grip</arg>
  <arg type="XPath">/InteractionRegion/Region/Object</arg>
</node>

<node type="XPathFilter"><arg type="StringChild">not(/InteractionRegion/STATUS)</arg></node>
<node type="XPathTransformSingle"><arg type="StringChild">/*</arg></node>
<node name="interestregion" type="AddChildElement">
  <arg type="StringChild">InteractionGoal</arg>
</node>

<!-- select next action -->
<select name="actionSelection" selectMax="1">
  <target>
    <filter type="XPathFilter">
      <arg type="StringChild">/InteractionRegion/Region/Object/Grip[@type != "unknown"]</arg>
    </filter>
    <node type="SetStaticAttribute">
```

```
      <arg type=”StringChild”>type</arg>
              <arg type=”StringChild”>grasp</arg>
    </node>
  </target>
  <target>
    <filter type=”XPathFilter”>
      <arg type=”StringChild”>/InteractionRegion/Region/Object[@userLabel
          or @detectorLabel][not(Grip)]</arg>
    </filter>
    <node type=”SetStaticAttribute”>
      <arg type=”StringChild”>type</arg>
              <arg type=”StringChild”>grip</arg>
    </node>
  </target>
  <target>
    <filter type=”XPathFilter”>
      <arg type=”StringChild”>/InteractionRegion/Region/Object[@userLabel
          or @detectorLabel]/Grip[@type = ”unknown”]</arg>
    </filter>
    <node type=”SetStaticAttribute”>
      <arg type=”StringChild”>type</arg>
              <arg type=”StringChild”>grip</arg>
    </node>
  </target>
  <target>
    <filter type=”XPathFilter”>
      <arg
          type=”StringChild”>/InteractionRegion/Region[not(Object/@detectorLabel)
          and not(Object/@userLabel)]</arg>
    </filter>
    <node type=”SetStaticAttribute”>
      <arg type=”StringChild”>type</arg>
              <arg type=”StringChild”>label</arg>
    </node>
  </target>
</select>

<!-- add timestamp -->
<node type=”DocumentFromNode”/>
<node name=”target” type=”CurrentTime”/>
<node type=”DocumentSerializer”/>
<node type=”TextFrame”><arg
    type=”StringChild”>InteractionGoal</arg></node>
  <!-- publish to memory -->
  <node source="target" type="Document2XOPData"/>
  <node type="TaskSubmission">
    <arg type="StringChild">ShortTerm</arg>
  </node>


</model>
```

C.2.2. Selection Visualization

```
<?xml version="1.0" encoding="utf-8"?>
<model>
  <!-- saliencies -->
  <node type="MemorySource">
    <arg type="StringChild">ShortTerm</arg>
    <arg type="StringChild">INSERT</arg>
    <arg type="StringChild">/saliencies</arg>
  </node>
  <node type="Event2Document"/>
  <node type="StaticUnpack">
    <arg type="StringChild">SalientPointArray</arg>
  </node>
  <node type="NonEmptyArray"/>
  <node type="ResolutionScale"/>
  <node type="SaliencyRegionColor">
    <arg type="Color" r="255" g="0" b="0" a="150"/>
  </node>
  <node name="saliencies" type="Array2List"/>

  <!-- regionlist -->
  <node source="null" type="MemorySource">
    <arg type="StringChild">ShortTerm</arg>
    <arg type="StringChild">INSERT</arg>
    <arg type="StringChild">/RegionList</arg>
  </node>
  <node name="regiondoc" type="Event2Document"/>
  <node type="StaticUnpack">
    <arg type="StringChild">ObjectRegionArray</arg>
  </node>
  <node type="NonEmptyArray"/>
  <node type="SingleColorRegion">
    <arg type="Color" r="255" g="255" b="255" a="100"/>
  </node>
```

```
<node name="objectregions" type="Array2List"/>

<!-- interaction region -->
<node source="null" type="MemorySource">
  <arg type="StringChild">ShortTerm</arg>
  <arg type="StringChild">REPLACE</arg>
  <arg type="StringChild">/InteractionRegion/STATUS[@value = "accepted"]</arg>
</node>
<node type="Event2Document"/>
<node type="XPathTransformSingle"><arg type="XPath">//Region</arg></node>
<node type="StaticUnpack">
  <arg type="StringChild">ObjectRegion</arg>
</node>
<node type="ColorRegion">
  <arg type="Color" r="0" g="255" b="0" a="128"/>
</node>
<node name="interactionregion" type="SingletonList"/>

<!-- put saliencies and regionlist together -->
<fuse sources="saliencies,objectregions,interactionregion"
               useOnceOnly="true" required="saliencies,objectregions"/>
<node name="regions" type="ConcatLists"/>

<!-- get base image -->
<node source="null" type="Subscriber">
  <arg type="StringChild">InputImage</arg>
  <arg type="Boolean">true</arg>
</node>
<node type="SingleAttachmentTransform">
  <arg type="StringChild">image</arg>
</node>
<node type="ImageDecoder"/>
<node name="image" type="Raw2BufferedImage"/>

<node type="ImageDimensionChangedFilter"/>
<node name="panel" type="PanelForImage"/>
<node type="SinglePanelFrame">
  <arg type="StringChild">Regions</arg>
</node>

<!-- fuse inputs -->
<fuse sources="panel,regions,image"
```

```
    names=”display.panel,display.regions,display.image”
        useOnceOnly=”false” checkComplete=”true”
        />
  <node type=”PeriodicFilter”><arg type=”Integer”>100</arg></node>

  <node type=”StatelessAugmentedDisplay”></node>

</model>
```

C.3. Example graph visualizations

C.4. Data format examples

C.4.1. Saliency computation output

```
<saliencies width=”40” height=”30” img_number=”1234”>
  <point>
    <coord ref=”image” kind=”relative” x=”0.125” y=”0.1”/>
    <coord ref=”image” kind=”absolute” unit=”px” x=”5” y=”3”/>
    <saliency saliency=”123456” colorV=”100” intensityV=”50”
        orientationV=”10”/>
  </point>
</saliencies>
```

C.4.2. Object detector output

```
<RegionList imageWidth=”640” imageHeight=”480”>
   <TIMESTAMP>
      <INSERTED value=”1211204126474”/>
      <UPDATED value=”1211204126702”/> required?
   </TIMESTAMP>

   <Region varianceFirstMajorAxis=”434” varianceSecondMajorAxis=”433”
       pixelCount=”4384”>
      <Object detectorLabel=”apple” confidence=”0.87”/>
      <coord ref=”image” kind=”relative” x=”” y=”” width=”0.4” height=”0.8”
          majorAxisAngle=”90”/>
      <coord ref=”image” kind=”absolute” unit=”px” x=”” y=”” width=”0.4”
          height=”0.8” majorAxisAngle=”90”/>
      <coord ref=”world” kind=”absolute” unit=”cm” x=”” y=”” z=””
          width=”0.4” height=”0.8” majorAxisAngle=”90”/>
   </Region>

   <!−− further regions −−>
   <Region>
```

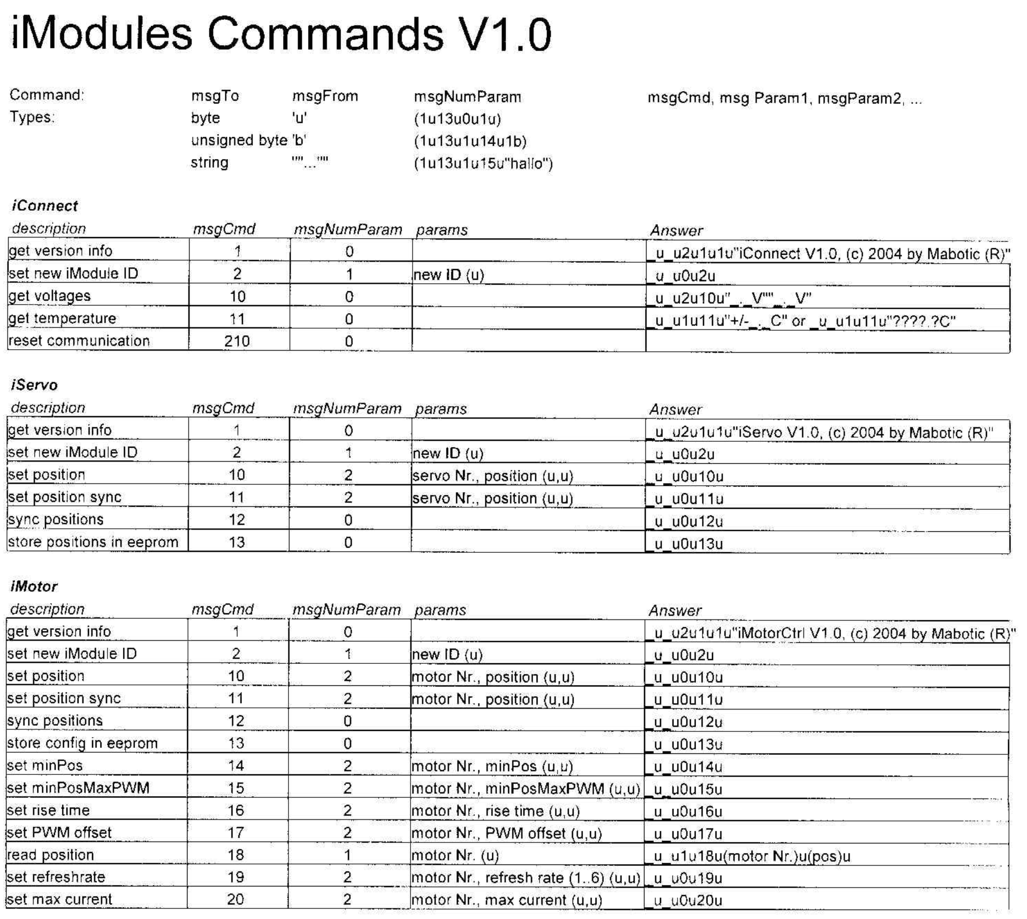
```
  ...
  </Region>
</RegionList>
```

C.4.3. Interaction region

```
<InteractionRegion>
  <TIMESTAMP>
    <INSERTED value=”192835719357”/>
    <UPDATED value=”012347912475”/>
  </TIMESTAMP>
  <Region varianceFirstMajorAxis=”434” varianceSecondMajorAxis=”433” pixelCount=”4384”>
    <coord ref=”image” kind=”relative” x=”” y=”” width=”0.4” height=”0.8” majorAxisAngle=”90”/>
    <coord ref=”image” kind=”absolute” unit=”px” x=”” y=”” width=”0.4” height=”0.8” majorAxisAngle=”90”/>
    <coord ref=”world” kind=”absolute” unit=”cm” x=”” y=”” z=”” width=”0.4” height=”0.8” majorAxisAngle=”90”/>
    <saliency saliency=”” colorV=”” intensityV=”” orientationV=””/>
    <Object detectorLabel=”apple” userLabel=”banana”>
        <Grip type=”TwoFingerSpecial” quality=”1”/>
    </Object>
  </Region>
  <InteractionGoal type=”label|grip|grasp”/>
  <STATUS value=”initiated|accepted|completed”/>
  <IMAGE uri=”imageName”/>
    <TIMESTAMPS/>
    ...
  </IMAGE>
</InteractionRegion>
```

D. Protocol Specifications

D.1. MABOTIC iModule Command Specification



iModules Commands V1.0

Command:	msgTo	msgFrom	msgNumParam	msgCmd, msg Param1, msgParam2, ...
Types:	byte	'u'	(1u13u0u1u)	
	unsigned byte	'b'	(1u13u1u14u1b)	
	string	""...""	(1u13u1u15u"hallo")	

iConnect

description	*msgCmd*	*msgNumParam*	*params*	*Answer*
get version info	1	0		_u_u2u1u1u"iConnect V1.0, (c) 2004 by Mabotic (R)"
set new iModule ID	2	1	new ID (u)	_u_u0u2u
get voltages	10	0		_u_u2u10u"_._V""_._V"
get temperature	11	0		_u_u1u11u"+/-_._C" or _u_u1u11u"????.?C"
reset communication	210	0		

iServo

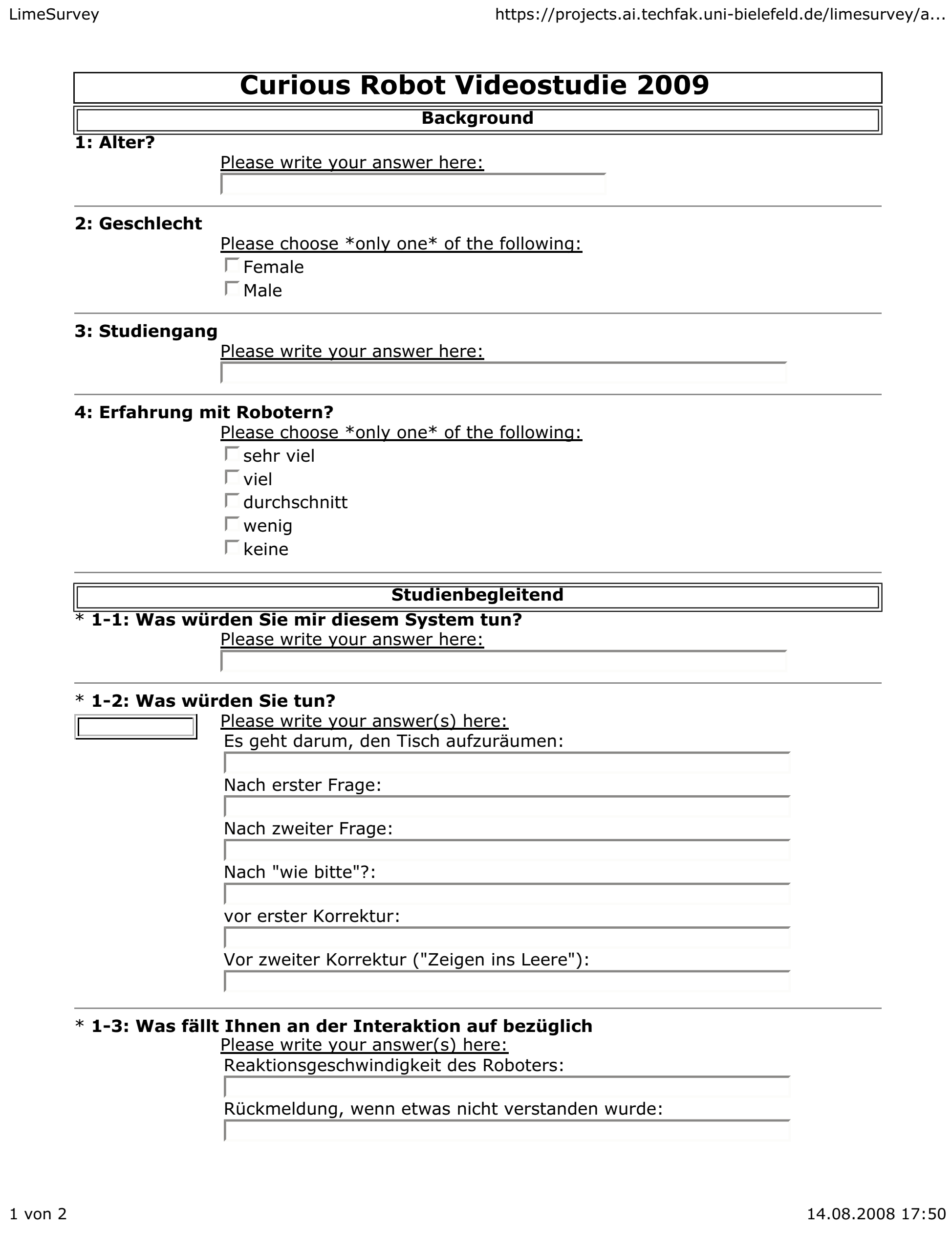
description	*msgCmd*	*msgNumParam*	*params*	*Answer*
get version info	1	0		_u_u2u1u1u"iServo V1.0, (c) 2004 by Mabotic (R)"
set new iModule ID	2	1	new ID (u)	_u_u0u2u
set position	10	2	servo Nr., position (u,u)	_u_u0u10u
set position sync	11	2	servo Nr., position (u,u)	_u_u0u11u
sync positions	12	0		_u_u0u12u
store positions in eeprom	13	0		_u_u0u13u

iMotor

description	*msgCmd*	*msgNumParam*	*params*	*Answer*
get version info	1	0		_u_u2u1u1u"iMotorCtrl V1.0, (c) 2004 by Mabotic (R)"
set new iModule ID	2	1	new ID (u)	_u_u0u2u
set position	10	2	motor Nr., position (u,u)	_u_u0u10u
set position sync	11	2	motor Nr., position (u,u)	_u_u0u11u
sync positions	12	0		_u_u0u12u
store config in eeprom	13	0		_u_u0u13u
set minPos	14	2	motor Nr., minPos (u,u)	_u_u0u14u
set minPosMaxPWM	15	2	motor Nr., minPosMaxPWM (u,u)	_u_u0u15u
set rise time	16	2	motor Nr., rise time (u,u)	_u_u0u16u
set PWM offset	17	2	motor Nr., PWM offset (u,u)	_u_u0u17u
read position	18	1	motor Nr. (u)	_u_u1u18u(motor Nr.)u(pos)u
set refreshrate	19	2	motor Nr., refresh rate (1..6) (u,u)	_u_u0u19u
set max current	20	2	motor Nr., max current (u,u)	_u_u0u20u

E. Video Study Material

E.1. Questionnaire for the video study



LimeSurvey https://projects.ai.techfak.uni-bielefeld.de/limesurvey/a...

Curious Robot Videostudie 2009

Background

1: Alter?

Please write your answer here:

2: Geschlecht

Please choose *only one* of the following:

☐ Female
☐ Male

3: Studiengang

Please write your answer here:

4: Erfahrung mit Robotern?

Please choose *only one* of the following:

☐ sehr viel
☐ viel
☐ durchschnitt
☐ wenig
☐ keine

Studienbegleitend

* **1-1: Was würden Sie mir diesem System tun?**

Please write your answer here:

* **1-2: Was würden Sie tun?**

Please write your answer(s) here:

Es geht darum, den Tisch aufzuräumen:

Nach erster Frage:

Nach zweiter Frage:

Nach "wie bitte"?:

vor erster Korrektur:

Vor zweiter Korrektur ("Zeigen ins Leere"):

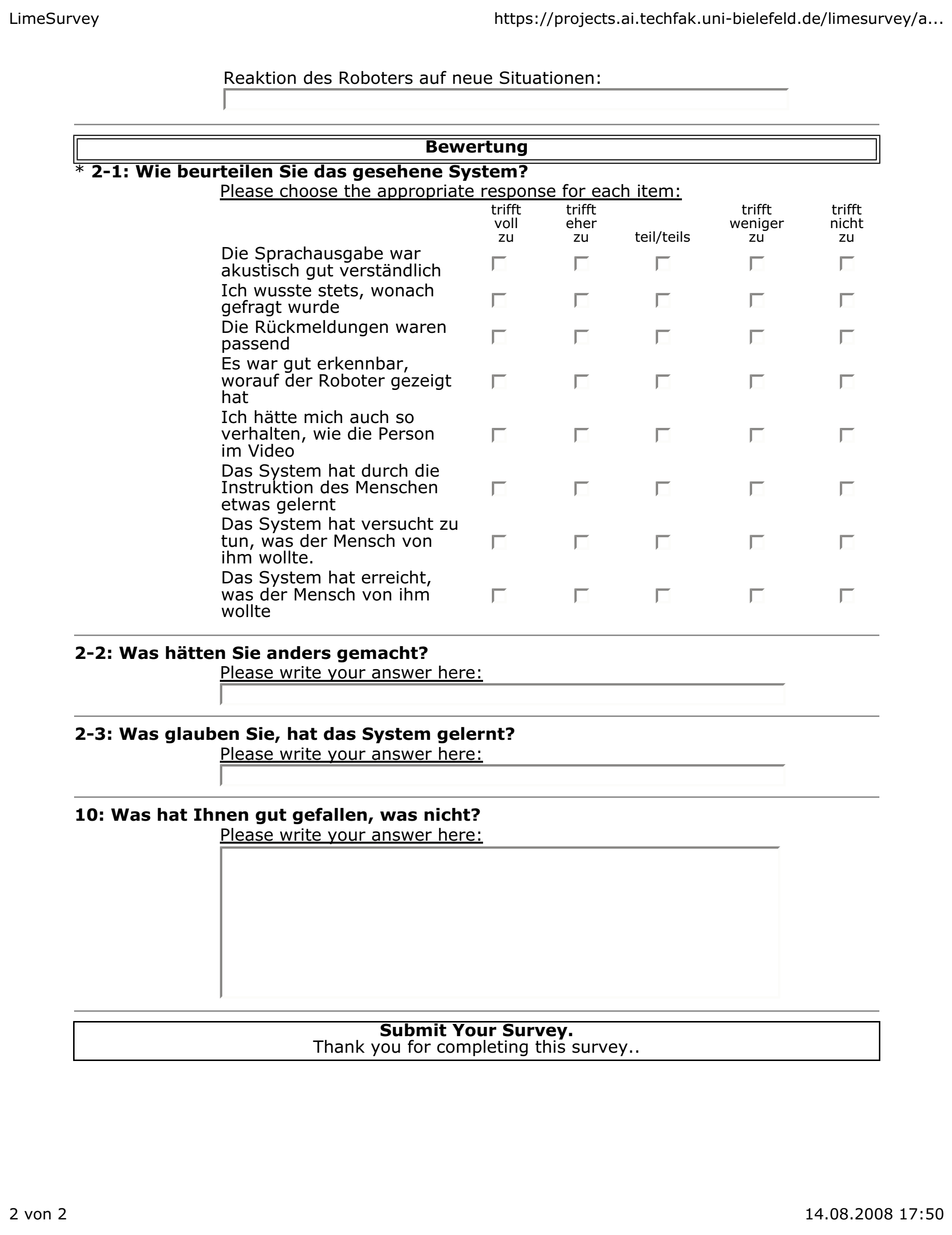
* **1-3: Was fällt Ihnen an der Interaktion auf bezüglich**

Please write your answer(s) here:

Reaktionsgeschwindigkeit des Roboters:

Rückmeldung, wenn etwas nicht verstanden wurde:

1 von 2 14.08.2008 17:50



LimeSurvey https://projects.ai.techfak.uni-bielefeld.de/limesurvey/a...

Reaktion des Roboters auf neue Situationen:

Bewertung

* **2-1: Wie beurteilen Sie das gesehene System?**

Please choose the appropriate response for each item:

	trifft voll zu	trifft eher zu	teil/teils	trifft weniger zu	trifft nicht zu
Die Sprachausgabe war akustisch gut verständlich	☐	☐	☐	☐	☐
Ich wusste stets, wonach gefragt wurde	☐	☐	☐	☐	☐
Die Rückmeldungen waren passend	☐	☐	☐	☐	☐
Es war gut erkennbar, worauf der Roboter gezeigt hat	☐	☐	☐	☐	☐
Ich hätte mich auch so verhalten, wie die Person im Video	☐	☐	☐	☐	☐
Das System hat durch die Instruktion des Menschen etwas gelernt	☐	☐	☐	☐	☐
Das System hat versucht zu tun, was der Mensch von ihm wollte.	☐	☐	☐	☐	☐
Das System hat erreicht, was der Mensch von ihm wollte	☐	☐	☐	☐	☐

2-2: Was hätten Sie anders gemacht?

Please write your answer here:

2-3: Was glauben Sie, hat das System gelernt?

Please write your answer here:

10: Was hat Ihnen gut gefallen, was nicht?

Please write your answer here:

Submit Your Survey.
Thank you for completing this survey..

2 von 2 14.08.2008 17:50

E.1.1. Translation of questionnaire

In the following, all text except for the *emphasized text* is a literal translation of the questionnaire. The emphasized text contains notes.

Background

1. Age
2. Gender
3. Course of study
4. Experience with robots (very much, much, average, little, none)

Study log

1. What would you do with this system?
2. What would you do now? *(cf. table 9.1 for timing)*
3. Was did you notice in the interaction regarding...
    - reaction time of the robot
    - response when something has not been understood
    - reaction of the robot to new situations

Judgements

1. How would you judge the system you just saw regarding...
    - The speech output was acoustically well intelligible.
    - I always knew what was being asked for.
    - The responses were appropriate.
    - It was well discernible where the robot pointed.
    - I would have reacted the same way as the person in the video.
    - The system has learned something through the instruction of the human.
    - The system has tried to do what the human wanted.
    - The system has achieved what the human wanted.

    Responses: Five point Likert scale (fully agree, partially agree, mixed, partially disagree, fully disagree)
2. What would you have done differently?
3. What do you think did the system?
4. What did you like, what not?

E.2. Transcription of subject recordings

E.2.1. Responses to free questions (German)

Responses are given exactly as the subjects wrote them down, one bullet point per subject. Note that not all subjects provided answers to all questions.

English translations are provided in the following section.

“Was hätten Sie anders gemacht?”

- Ich hätte mehr gestikuliert
- farbliche Unterscheidung zw. gleichen Gegenständen
- Beispiel “Nichts” $\rightarrow$ ich: Was siehst du? (mehr Versuch und Irrtum)
- Nur Stichworte, keine Sätze
- Ich hätte nicht “halt” gesagt, sondern “versuch es noch einmal”
- Farben von gleichen Objekten eingebracht
- Anweisungen in Details vorgeben
- Benennung der Griffe; keinen Neustart nach Fehlversuch; neuen Befehl gleich bei Fehlversuchen

“Was glauben Sie, hat das System gelernt?”

- bessere Anpassung der Hand an die Form (lang+rund)
- Griffweisen bzgl. unterschiedlicher Gegestände
- Griff $\leftrightarrow$ Gegenstand
- Handgriffe passend zu Gegenstand zu nutzen
- welche Frucht mit welchem Griff zu tragen ist!
- Zuordnung Obstart $\leftrightarrow$ Griff
- Greifen von verschiedenen Objekten, Objekte bestimmen, Unterschied zwischen Banane + Apfel
- Versch. Obstsorten (Formen)
- Bewegung von Dingen, Zugreifen und an andere Stelle legen
- Bezeichnung der Gegenstände, Motorik zum Greifen der Gegenstände

“Was hat Ihnen gut gefallen, was nicht?”

- Ungewöhnlich war die “Ruhestellung” der Hand mit ausgestreckten Fingern, d.h. Als VP wäre ich vielleicht etwas verwirrt, ob der Roboter noch etwas von mir will, oder nicht
- Gut: gute Reaktion des Roboters (Wahl der Sprache und Ausdruck) Weniger gut: teilweise eher lange Phasen, ehe die nächste Reaktion erfolgte
- Gut: Umgang mit Roboter mi einfacher Sprache Bestätigung des Instruierenden wiederholt Schlecht: Kein Feedback was der Roboter wahrnimmt bzw. “denkt” -gute Spracherkennung -Roboterarme erstaunlich wendig und “gefühlvoll” (hat das Obst nicht zerquetscht)
- das System ist sehr langsam! Der mechanische Aufbau der Hand ist sehr gut!
- Gute Stimme, gute Mechanik
- Einen Roboter mit Gesicht darzustellen- Hände zu kopieren+ Versuchsaufbau wurde geändert-
- Beeindrucken, wie ein Roboter agiert+ Nichterkennen der Nuss-
- Die technische Umsetzung zu beobachten sowie das Zusammenspiel von Mensch + Roboter -
- Reaktionen auf Sprache - Formulierungen sind zu “technisch”

E.2.2. Responses to free questions (English)

This is the literal translation of the original German responses from the previous section.

“What would you have done differently?”

- I would have gesticulated more
- color differences between same objects
- Example “Nothing” → I: what do you see (more trial and error)
- only keywords, no sentences
- I wouldn’t have said “halt” but “try again”
- color of same objects introduced
- Describe commands in detail
- Names of the grips; no restart after error; immediately new command on errors

"What do you think the system has learned?"

- better adaptation of the hand to the form (long+round)
- grip types regarding different objects
- grip $\leftrightarrow$ object
- use of hand grip types appropriate for object
- which fruit is picked up with which grip
- Mapping fruit type $\leftrightarrow$ grip
- gripping of different objects, discern objects, difference between banana and apple
- different types of fruit (shapes)
- motion of objects, grasping and putting at another place
- names of the objects, motor control for gripping of objects

"What did you like, what not?"

- the "rest position" of the hand was unconventional with extended fingers; that means, as a subject I might have been confused whether the robot still wants something from me or not
- good: good reaction of the robot (choice of speech and expression) not so good: partially somewhat long pauses before the next reaction
- good: interaction with robot with simple speech, feedback of the instructor, bad: no feedback what the robot perceives or "thinks". good speech recognition. robot arms surprisingly agile and "feeling" (didn't crush the fruit)
- the system is very slow! the mechanical construction of the hand is very good!
- good voice, good mechanical construction
- bad: showing a robot with a voice; good: copying hands; bad: experimental setting was changed
- good: impressive how a robot works; bad: not recognizing the nut
- good: seeing the technical realization and the interplay between human and robot
- reaction to speech: expressions are too "technical"

Bibliography

Corba notification service, version 1.1. Technical report, Object Management Group (OMG), Inc, October 2004. 66

NAOqi Framework Overview, 2011. URL http://users.aldebaran-robotics.com/docs/site_en/reddoc/framework/framework.html. Version 1.10.37 from May the 30th, 2011. 9

Documentation - ROS Wiki, 2011. URL http://www.ros.org/wiki/. 8, 9

P. E. Agre. Hierarchy and history in simon's "architecture of complexity". *Journal of the Learning Sciences*, 12(3):413–426, 2003. doi: 10.1207/S15327809JLS1203_4. 38

J. S. Albus. RCS: a reference model architecture for intelligent control. *Computer*, 25(5):56–59, May 1992. ISSN 0018-9162. doi: 10.1109/2.144396. 7, 35

C. Alexander, S. Ishikawa, and M. Silverstein. *Pattern Manual.* Berkeley, Nov 1967. 45

C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977. 45

J. E. Allen, C. I. Guinn, and E. Horvtz. Mixed-initiative interaction. *IEEE Intelligent Systems*, 14(5):14–23, September 1999. ISSN 1094-7167. doi: 10.1109/5254.796083. 16

J. F. Allen. Mixed-initiative interaction. *IEEE Intelligent Systems*, 14(5):14–23, 1999. ISSN 1541-1672. doi: 10.1109/5254.796083. 83

A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006. ISSN 1066-8888. doi: 10.1007/s00778-004-0147-z. 146

R. C. Arkin. Motor schema-based mobile robot navigation. *The International Journal of Robotics Research*, 8(4):92–112, August 1989. doi: 10.1177/027836498900800406. 31

K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *COMPUTER*, 39(02): 33–40, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.54. 109

D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A framework for event-based software integration. *ACM Trans. Softw. Eng. Methodol.*, 5(4):378–421, 1996. ISSN 1049-331X. doi: 10.1145/235321.235324. 105

V. R. Basili and R. W. Reiter. Evaluating automatable measures of software development. In *Workshop on Quantitative Software Models*, pages 107–116, New York, NY, October 1979. IEEE. 189

V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22 (10):751–761, October 1996. ISSN 0098-5589. doi: 10.1109/32.544352. 189

V. Baskaran, M. Dalal, T. Estlin, C. Fry, R. Harris, M. Iatauro, A. Jónsson, C. Pasareanu, R. Simmons, and V. Verma. Plan execution interchange language (plexil) version 1.0. Technical report, NASA Ames Research Center / Jet Propulsion Laboratory / Carnegie Mellon University, 2007. 64

C. Bauckhage, G. A. Fink, J. Fritsch, N. Jungclaus, S. Kronenberg, F. Kummert, F. Lömker, G. Sagerer, and S. Wachsmuth. *Integrated perception for cooperative human-machine interaction*, volume 166 of *Trends in Linguistics*. Mouton de Gruyter, March 2006. ISBN 311018897X. 175

K. Beck and W. Cunningham. Using Pattern Languages for Object-Oriented Programs. Technical Report CR-87-43, Tektronix, Inc, 1987. 45

G. A. Bekey. *Autonomous Robots: From Biological Inspiration to Implementation and Control*. Intelligent Robotics and Autonomous Agents. The MIT Press, June 2005. ISBN 0262025787. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0262025787`. 17, 18

S. S. Bhattacharyya, C. Brooks, E. Cheong, J. Davis, M. Goel, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Y. Zhao, and H. Zheng. Heterogeneous Concurrent Modeling and Design in Java: Introduction to Ptolemy II. Technical Report UCB/ERL M05/21, EECS, Berkeley, CA, USA, July 2005. 37, 123

A. Billard, S. Calinon, R. Dillmann, and S. Schaal. *Robot Programming by Demonstration*, pages 1371+. Springer Verlag, Berlin / Heidelberg Germany, 2008. ISBN 978-3-540-23947-4. 13, 14

B. M. Blumberg. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, Massachusetts Institute of Technology, 1996. 180

M. Boden. Autonomy: What is it? *Biosystems*, 91(2):305–308, February 2008. ISSN 03032647. doi: 10.1016/j.biosystems.2007.07.003. 18

B. W. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, January 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010193. 131

R. P. Bonasso. Integrating reaction plans and layered competences through synchronous control. In *Proceedings of the Twelfth International Conference on Artificial Intelligence (IJCAI)*, pages 1225–1233, 1991. 8

O. Booij, B. Kröse, J. Peltason, T. Spexard, and M. Hanheide. Moving from augmented to interactive mapping. In *Robotics: Science and Systems Conference*, Zurich, 2008. 69, 181

C. Breazeal, G. Hoffman, and A. Lockerd. Teaching and working with robots as a collaboration. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1030–1037, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 1-58113-864-4. doi: 10.1109/AAMAS.2004.258. 16

C. Breazeal, A. Takashini, and T. Kobayashi. *Social Robots that Interact with People*, pages 1349–1369. Springer Verlag, 2008. ISBN 978-3-540-23947-4. 13, 17

C. L. Breazeal. *Designing Sociable Robots.* The MIT Press, September 2004. ISBN 0262524317. 168

S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, July 1984. ISSN 0004-5411. doi: 10.1145/828.833. 110

R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, January 1986. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1087032`. 31, 108

R. A. Brooks. Intelligence without representation. *Artif. Intell.*, 47(1-3):139–159, January 1991. ISSN 0004-3702. doi: 10.1016/0004-3702(91)90053-M. 29

D. Brugali and M. E. Fayad. Distributed computing in robotics and automation. *Robotics and Automation, IEEE Transactions on*, 18(4):409–420, December 2002. doi: 10.1109/TRA.2002.802937. 32

D. Brugali and P. Scandurra. Component-based robotic engineering (part i) [tutorial]. *Robotics & Automation Magazine, IEEE*, 16(4):84–96, December 2009. doi: 10.1109/MRA.2009.934837. 29

D. Brugali, A. Brooks, A. Cowley, C. Côté, A. Domínguez-Brito, D. Létourneau, F. Michaud, and C. Schlegel. Trends in component-based robotics. In D. Brugali, editor, *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*, chapter 8, pages 135–142. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-68949-2. doi: 10.1007/978-3-540-68951-5_8. 29, 32

W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *ArtificialIntelligence*, 114(1-2), 2000. 13

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 1996. 45, 46

F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, volume 4 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 2007a. 45

F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*, volume 5 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 2007b. 45

CAN. Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. Standard ISO 11898-1:2003, International Organization for Standardization (ISO), 2003. 145

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, August 1994. ISSN 0098-5589. doi: 10.1109/32.295895. 189, 190

N. I. Churcher, M. J. Shepperd, S. Chidamber, and C. F. Kemerer. Comments on " a metrics suite for object oriented design". *Software Engineering, IEEE Transactions on*, 21(3):263–265, August 2002. doi: 10.1109/32.372153. 191

H. H. Clark and S. E. Brennan. *Grounding in communication.*, pages 127–149. 1991. ISBN 1557983763. 86

J. H. Connell. Sss: a hybrid architecture applied to robot navigation. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 2719–2724 vol.3, May 1992. doi: 10.1109/ROBOT.1992.219995. 31

C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud. Robotic software integration using marie. *International Journal of Advanced Robotic Systems*, 3(1): 55–60, 2006. ISSN 1729-8806. 178

N. Dahlbäck, A. Jönsson, and L. Ahrenberg. Wizard of oz studies: why and how. In *IUI '93: Proceedings of the 1st international conference on Intelligent user interfaces*, pages 193–200, New York, NY, USA, 1993. ACM. ISBN 0-89791-556-9. doi: 10.1145/169891.169968. 150

A. Damasio. *Descartes' Error*. G.P. Putnam, July 1994. ISBN 0399138943. 17

K. Dautenhahn. Getting to know each other – artificial social intelligence for autonomous robots. *Robotics and Autonomous Systems*, 16:333–356, 1995. ISSN 0921-8890. 17

A. Demers, J. Gehrke, M. Hong, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *Conference on Innovative Data Systems Research (CIDR)*, pages 412–422. VLDB, 2007. 146

B. Duffy. Anthropomorphism and the social robot. *Robotics and Autonomous Systems*, 42(3-4):177–190, March 2003. ISSN 09218890. doi: 10.1016/S0921-8890(02)00374-3. 23

A. Endres and D. Rombach. *Handbook of Software and Systems Engineering*. Pearson Education Ltd., 2003. ISBN 0-321-15420-7. 38, 189

T. Faison. *Event-Based Programming: Taking Events to the Limit*. Apress, 1 edition, May 2006. ISBN 1590596439. 34

N. Fenton. Viewpoint article: Conducting and presenting empirical software engineering. *Empirical Software Engineering*, 6(3):195–200, September 2001. ISSN 13823256. doi: 10.1023/A:1011449731678. 11

N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, August 1999. ISSN 0098-5589. doi: 10.1109/32.815326. 189

G. Fink. Developing HMM-Based Recognizers with ESMERALDA. In V. Matousek, P. Mautner, J. Ocelíková, and P. Sojka, editors, *Text, Speech and Dialogue*, volume 1692 of *Lecture Notes in Computer Science*, chapter 42, page 843. Springer Berlin Heidelberg, Berlin, Heidelberg, October 1999. ISBN 978-3-540-66494-9. doi: 10.1007/3-540-48239-3_42. 154

G. A. Fink, J. Fritsch, S. Hohenner, M. Kleinehagenbrock, S. Lang, and G. Sagerer. Towards multi-modal interaction with a mobile robot. *Pattern Recognition and Image Analysis*, 14(2):173–184, 2004. 181

R. J. Firby. Task networks for controlling continuous processes. In *In Proceedings of the Second International Conference on AI Planning Systems*, pages 49–54, 1994. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.1223`. 8, 28

P. Fitzpatrick, L. Natale, G. Metta, G. Spigler, A. Scalzo, A. van Rossum, D. Krieg, A. Bernardino, A. Gijsberts, R. Detry, J. Gomes, Z. Ji, M. Castelnovi, F. Nori, C. Beltran-Gonzalez, A. Mirza, U. Pattacini, M. Randazzo, J. Ruesch, M. P. Blow, M. Brunettini, L. Olsson, G. Massera, F. Magnusson, E. Mislivec, P. Fitzpatick, H. Kose-Bagci, and C. Castellini. Yet Another Robot Platform, 2011. URL `http://eris.liralab.it/yarp/`. Accessed on May 30th 2011. 9

T. Fong, I. Nourbakhsh, C. Kunz, L. Flückiger, J. Schreiner, R. Ambrose, R. Burridge, R. Simmons, L. M. Hiatt, A. Schultz, J. G. Trafton, M. Bugajska, and J. Scholtz. The peer-to-peer human-robot interaction project. In *In AIAA Space 2005*, pages 2005–6750, 2005. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.9651`. 176

T. Fong, C. Kunz, L. M. Hiatt, and M. Bugajska. The human-robot interaction operating system. In *HRI '06: Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, pages 41–48, New York, NY, USA, 2006a. ACM. ISBN 1-59593-294-1. doi: 10.1145/1121241.1121251. 5, 176

T. Fong, J. Scholtz, J. A. Shah, L. Fluckiger, C. Kunz, D. Lees, J. Schreiner, M. Siegel, L. M. Hiatt, I. Nourbakhsh, R. Simmons, R. Ambrose, R. Burridge, B. Antonishek, M. Bugajska, A. Schultz, and J. G. Trafton. A preliminary study of peer-to-peer human-robot interaction. In *Systems, Man and Cybernetics, 2006. SMC '06. IEEE International Conference on*, volume 4, pages 3198–3203, 2006b. doi: 10.1109/ICSMC.2006.384609. 176, 177

M. E. Foster, M. Giuliani, A. Isard, C. Matheson, J. Oberlander, and A. Knoll. Evaluating description and reference strategies in a cooperative human-robot dialogue system. In *IJCAI'09: Proceedings of the 21st international jont conference on Artifical intelligence*, pages 1818–1823, San Francisco, CA, USA, 2009a. Morgan Kaufmann Publishers Inc. URL `http://portal.acm.org/citation.cfm?id=1661737`. 175, 176

M. E. Foster, M. Giuliani, and A. Knoll. Comparing objective and subjective measures of usability in a human-robot dialogue system. In *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL-IJCNLP 2009)*, Singapore, Aug. 2009b. 176

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, July 1999. ISBN 0201485672. 138

G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, November 1987. ISSN 0001-0782. 13, 23

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. 45, 52

E. R. Gansner, E. Koutsofios, and S. North. *Drawing graphs with dot*, 2009. URL `http://www.graphviz.org/pdf/dotguide.pdf`. 121

E. Gat. *Three-layer architectures*, pages 195–210. MIT Press, Cambridge, MA, USA, 1998. ISBN 0-262-61137-6. URL `http://portal.acm.org/citation.cfm?id=292092.292130`. 8, 31

R. Gockley, R. Simmons, J. Wang, D. Busquets, C. DiSalvo, K. Caffrey, S. Rosenthal, J. Mink, S. Thomas, W. Adams, T. Lauducci, M. Bugajska, D. Perzanowski, and A. Schultz. Grace and george: Social robots at aaai. Technical report, AAAI, 2004. 177, 178

M. A. Goodrich and A. C. Schultz. Human-robot interaction: a survey. *Found. Trends Hum.-Comput. Interact.*, 1(3):203–275, January 2007. ISSN 1551-3955. doi: 10.1561/1100000005. 16

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The.* Addison Wesley, 3rd edition, June 2005. ISBN 0321246780. URL `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`. 11, 117, 118

M. Hackel, S. Schwope, J. Fritsch, B. Wrede, and G. Sagerer. A humanoid robot platform suitable for studying embodied interaction. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 56–61, Edmonton, Alberta, Canada, August 2005. IEEE, IEEE. 19, 139

M. Hanheide and G. Sagerer. Active memory-based interaction strategies for learning-enabling behaviors. In *International Symposium on Robot and Human Interactive Communication (RO-MAN)*, Munich, 01/08/2008 2008. 16, 23, 60, 62, 69, 70, 149, 181

D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. doi: 10.1016/0167-6423(87)90035-9. 108

D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-15181-8. URL `http://portal.acm.org/citation.cfm?id=101990`. 107

I. Herman and M. Marshall. Graphxml — an xml-based graph description format. In J. Marks, editor, *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, chapter 6, pages 33–66. Springer Berlin Heidelberg, Berlin, Heidelberg, May 2001. ISBN 978-3-540-41554-1. doi: 10.1007/3-540-44541-2_6. 121

M. Himsolt. Gml: A portable graph file format. Technical report, Universität Passau, 94030 Passau, Germany, 1999. URL `http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html`. Accessed March 10th, 2010. 121

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. 110

R. Holt, A. Schürr, S. E. Sim, and A. Winter. Gxl - graph exchange language. Online, July 2002. URL `http://www.gupro.de/GXL/index.html`. 121

S. Hüwel, B. Wrede, and G. Sagerer. Robust speech understanding for multi-modal human-robot communication. In *Proc. 15th Int. Symposium on Robot and Human Interactive Communication*, pages 45–50. IEEE Press, IEEE Press, 2006. 22, 154

H. Ishiguro, T. Ono, M. Imai, T. Maeda, T. Kanda, and R. Nakatsu. Robovie: an interactive humanoid robot. *Industrial Robot: An International Journal*, pages 498–504, 2001. ISSN 0143-991X. doi: 10.1108/01439910110410051. 177, 179

R. E. Johnson and B. Foote. *Designing Reusable Classes.* IEEE Computer Society Press, Los Alamitos, CA, USA, May 1991. ISBN 081868996X. 116

W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004. ISSN 0360-0300. doi: 10.1145/1013208.1013209. 8, 106, 138

G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing '77: Proceedings of IFIP Congress*, pages 993–998, Amsterdam, The Netherlands, 1977. North-Holland Publishing Co. 110

J. Karat, D. B. Horn, C. A. Halverson, and C. M. Karat. Overcoming unusability: developing efficient strategies in speech recognition systems. In *CHI '00: CHI '00 extended abstracts on Human factors in computing systems*, pages 141–142, New York, NY, USA, 2000. ACM. ISBN 1-58113-248-4. doi: 10.1145/633292.633372. 22

J. F. Kelley. An empirical methodology for writing user-friendly natural language computer applications. In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 193–196, New York, NY, USA, 1983. ACM. ISBN 0-89791-121-0. doi: 10.1145/800045.801609. 150

M. Kirchner and P. Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*, volume 3 of *Wiley Series in Software Design Patterns.* John Wiley & Sons, 2004. 45

A. Knoll, B. Hildenbrandt, and J. Zhang. Instructing cooperating assembly robots through situated dialogues in natural language. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 1, apr 1997. doi: 10.1109/ROBOT.1997.620146. 174

D. Kortenkamp and R. Simmons. *Robotic System Architectures and Programming*, chapter 8, pages 187–206. Springer-Verlag, 2008. 7, 27, 28, 31, 35

D. Kortenkamp, R. P. Bonasso, D. Ryan, and D. Schreckenghost. Traded control with autonomous robots as mixed-initiative interaction. In *AAAI Spring Symposium SS-97-04*, pages 89–94. AAAI, 1997. 16

D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems.* AAAI Press, 1st edition, March 1998. ISBN 0262611376. 6, 28

A. Kransted, A. Lücking, T. Pfeifer, H. Rieser, and I. Wachsmuth. *Deictic object references in task-oriented dialogue*, volume 166 of *Trends in Linguistics.* Mouton de Gruyter, March 2006. ISBN 311018897X. 175

G. J. Kruijff, J. D. Kelleher, G. Berginc, and A. Leonardis. Structural descriptions in human-assisted robot visual learning. In *HRI '06: Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, pages 343–344, New York, NY, USA, 2006. ACM. ISBN 1-59593-294-1. doi: 10.1145/1121241.1121307. 16

D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309, Dec. 2005. ISSN 0167-6423. doi: 10.1016/j.scico.2005.03.007. 114

E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987. 8, 111

E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83 (5):773–801, 1995. doi: 10.1109/5.381846. 8

D. R. Lefebvre and G. N. Saridis. A computer architecture for intelligent machines. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 3, pages 2745–2750, May 1992. doi: 10.1109/ROBOT.1992.219991. 6, 60, 61

S. Li. *Multi-modal Interaction Management for a Robot Companion.* Phd, Bielefeld University, Bielefeld, 2007. URL `http://bieson.ub.uni-bielefeld.de/volltexte/2007/1174/pdf/diss.pdf`. 86

S. Li, M. Kleinehagenbrock, J. Fritsch, B. Wrede, and G. Sagerer. "biron, let me show you something": Evaluating the interaction with a robot companion. In W. Thissen, P. Wieringa, M. Pantic, and M. Ludema, editors, *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics, Special Session on Human-Robot Interaction*, pages 2827–2834, The Hague, The Netherlands, October 2004. IEEE, IEEE. 181

C. LLC. Cyberglove ii. URL `http://www.cyberglovesystems.com/products/cyberglove-ii/overview`. Last checked 12th of February, 2010. 19

K. S. Lohan, A. L. Vollmer, J. Fritsch, K. Rohlfing, and B. Wrede. Which ostensive stimuli can be used for a robot to detect and maintain tutoring situations? In *IEEE International Workshop on Social Signal Processing*. IEEE, 2009. 16

M. Lohse. The role of expectations in HRI. In *New Frontiers in Human-Robot Interaction*, 2009. 23

D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Professional, May 2002. ISBN 0201727897. 146

F. Lömker, S. Wrede, M. Hanheide, and J. Fritsch. Building modular vision systems with a graphical plugin environment. In *Proc. of International Conference on Vision Systems*, St. Johns University, Manhattan, New York City, USA, January 2006. IEEE, IEEE. 154

I. Lütkebohle, J. Peltason, L. Schillingmann, C. Elbrechter, B. Wrede, S. Wachsmuth, and R. Haschke. The curious robot - structuring interactive robot learning. In *International Conference on Robotics and Automation*, Kobe, Japan, May 2009a. Robotics and Automation Society, IEEE. 4, 63, 130, 131, 154, 157

I. Lütkebohle, J. Schaefer, and S. Wrede. Facilitating re-use by design: A filtering, transformation, and selection architecture for robotic software systems. In *Software Development and Integration in Robotics*, Kobe, Japan, 2009b. 112, 115

I. Lütkebohle, F. Hegel, S. Schulz, M. Hackel, B. Wrede, S. Wachsmuth, and G. Sagerer. The bielefeld anthropomorphic robot head "flobi". In *2010 IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, 03/05/2010 2010. IEEE, IEEE. accepted. 138, 139, 145

I. Lütkebohle, J. Peltason, L. Schillingmann, C. Elbrechter, S. Wachsmuth, B. Wrede, and R. Haschke. Realizing a robot system for interactive online learning. In *Towards Service Robots for Everyday Environments*. Springer Verlag, 2011. submitted. 20

E. Marder-Eppstein and V. Pradeep. actionlib - ros wiki, 2010. URL `http://www.ros.org/wiki/actionlib`. 50, 63

M. J. Mataric. Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312, June 1992. ISSN 1042296X. doi: 10.1109/70.143349. 31

B. Maxwell. Building robot systems to interact with people in real environments. *Autonomous Robots*, 22(4):353–367, May 2007. ISSN 0929-5593. doi: 10.1007/s10514-006-9020-9. 177

T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4), 1976. 189

G. Metta and P. Fitzpatrick. Better vision through manipulation. *Adaptive Behavior*, 11(2):109–128, June 2003. doi: 10.1177/10597123030112004. 14

R. Meunier. *The pipes and filters architecture*, pages 427–440. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. ISBN 0-201-60734-4. URL `http://portal.acm.org/citation.cfm?id=218662.218694`. 8

M. Michalowski, S. Šabanović, C. DiSalvo, D. Busquets, L. Hiatt, N. Melchior, and R. Simmons. Socially distributed perception: Grace plays social tag at aaai 2005. *Autonomous Robots*, 22(4):385–397, May 2007. ISSN 0929-5593. doi: 10.1007/s10514-006-9015-6. 178

F. Michaud, D. Létourneau, M. Fréchette, Beaudry, and F. Kabanza. Spartacus, scientific robot reporter. Technical report, AAAI, 2006. 179

F. Michaud, C. Côté, D. Létourneau, Y. Brosseau, J. M. Valin, Beaudry, C. Raïevsky, A. Ponchon, P. Moisan, P. Lepage, Y. Morin, F. Gagnon, P. Giguère, M. A. Roux, S. Caron, P. Frenette, and F. Kabanza. Spartacus attending the 2005 aaai conference. *Autonomous Robots*, 22(4):369–383, May 2007. ISSN 0929-5593. doi: 10.1007/s10514-006-9014-7. 177, 178, 179

N. Mitsunaga, T. Miyashita, H. Ishiguro, K. Kogure, and N. Hagita. Robovie-iv: A communication robot interacting with people daily in an office. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5066–5072. IEEE, October 2006. ISBN 1-4244-0258-1. doi: 10.1109/IROS.2006.282594. 179

N. Mitsunaga, Z. Miyashita, K. Shinozawa, T. Miyashita, H. Ishiguro, and N. Hagita. What makes people accept a robot in a social environment - discussion from six-week study in an office -. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3336–3343. IEEE, September 2008. ISBN 978-1-4244-2057-5. doi: 10.1109/IROS.2008.4650785. 178, 179

G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems.* Springer, 1st edition, July 2006. ISBN 3540326510. 30, 34

R. R. Murphy, J. Casper, M. Micire, J. Hyams, Robin, R. Murphy, R. Murphy, R. R. Murphy, J. L. Casper, M. J. Micire, and J. Hyams. Mixed-initiative control of multiple heterogeneous robots for urban search and rescue. *Robotics and Automation*, 2000. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.4761`. 16

Y. Nagai. The role of motion information in learning human-robot joint attention. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, april 2005. 180

Y. Nagai, K. Hosada, A. Morita, and M. Asada. A constructive model for the development of joint attention. *Connection Science*, 15(4):211–229, December 2003a. doi: 10.1080/09540090310001655101. 150, 154

Y. Nagai, K. Hosoda, A. Morita, and M. Asada. A constructive model for the development of joint attention. *Connection Science*, 15(4):211–229, 2003b. doi: 10.1080/09540090310001655101. 130, 180

Y. Nagai, M. Asada, and K. Hosoda. Learning for joint attention helped by functional development. *Advanced Robotics*, 20(10):1165–1181, 2006. ISSN 0169-1864. doi: 10.1163/156855306778522497. 180

N. J. Nilsson. Shakey the robot. Technical report, SRI International, Menlo Park, CA, USA, 1984. collection of earlier reports. 13

D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972a. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/355602.361309. 29

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972b. ISSN 0001-0782. doi: 10.1145/361598.361623. 29

J. Peltason, 2008. Personal Interview during Task Toolkit Design. 71

J. Peltason, 2010. Personal Interview during XTT Evaluation. 100, 101

J. Peltason, F. H. Siepmann, T. P. Spexard, B. Wrede, M. Hanheide, and E. A. Topp. Mixed-initiative in human augmented mapping. In *International Conference on Robotics and Automation*, Kobe, Japan, 14/05/2009 2009a. IEEE, IEEE. 181

J. Peltason, F. H. K. Siepmann, T. P. Spexard, B. Wrede, M. Hanheide, and E. A. Topp. Mixed-initiative in human augmented mapping. In *International Conference on Robotics and Automation*, 2009b. 154

M. Piccoli. Gripping delicate objects, 2009. URL `http://www.willowgarage.com/blog/2009/08/04/breaking-eggs`. Video report. 15

K. Pitsch, A.-L. Vollmer, J. Fritsch, B. Wrede, K. Rohlfing, and G. Sagerer. On the loop of action modification and the recipient's gaze in adult-child interaction. In *Gesture and Speech in Interaction*, page 6, September 2009. 16

P. G. Plöger, K. Pervölz, C. Mies, P. Eyerich, M. Brenner, and B. Nebel. The desire service robotics initiative. *KI - Zeitschrift Künstliche Intelligenz*, 22(4):29–32, 2008. ISSN 0933-1875. 62, 69

L. Rabiner and B.-H. Juang. *Historical Perspective of the Field of ASR/NLU*. Springer, 2008. ISBN 978-3-540-49125-5. 22, 23

M. Radestock and S. Eisenbach. Coordination in evolving systems. In *TreDS '96: Proceedings of the International Workshop on Trends in Distributed Systems*, pages 162–176, London, UK, 1996. Springer-Verlag. ISBN 3-540-61842-2. URL `http://portal.acm.org/citation.cfm?id=695228`. 29

M. Rickert, M. Foster, M. Giuliani, T. By, G. Panin, and A. Knoll. Integrating language, vision and action for human robot dialog systems. In C. Stephanidis, editor, *Universal Access in Human-Computer Interaction. Ambient Interaction*, volume 4555 of *Lecture Notes in Computer Science*, chapter 108, pages 987–995. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73280-8. doi: 10.1007/978-3-540-73281-5_108. 175, 176

G. Rickheit and I. Wachsmuth, editors. *Situated Communication*, volume 166 of *Trends in Linguistics.* Mouton de Gruyter, March 2006. ISBN 311018897X. 174

H. Ritter, R. Haschke, F. Röthling, and J. J. Steil. Manual Intelligence as a Rosetta Stone for Robot Cognition. In *International Symposium on Robotics Research (ISRR)*, Hiroshima, 26/12/07 2007. International Foundation of Robotics Research. 154

K. J. Rohlfing, J. Fritsch, B. Wrede, and T. Jungmann. How can multimodal cues from child-directed interaction reduce learning complexity in robots? *Advanced Robotics*, 20(10):1183–1199, 2006. ISSN 0169-1864. doi: 10.1163/156855306778522532. 16

P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009. ISSN 1382-3256. doi: 10.1007/s10664-008-9102-8. 10, 84

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition).* Prentice Hall, 2 edition, December 2002. ISBN 0137903952. 27

S. Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6):233–242, June 1999. ISSN 13646613. doi: 10.1016/S1364-6613(99)01327-3. 15

W. N. Scherer, D. Lea, and M. L. Scott. Scalable synchronous queues. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 147–156, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: 10.1145/1122971.1122994. 114

D. Schmidt, M. Stal, H. Rohnert, and F. Buschman. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2 of *Wiley Series in Software Design Patterns.* John Wiley & Sons, 2000. 45, 52

D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *COMPUTER*, 39(02):25–31, 2006. doi: 10.1109/MC.2006.58. 108, 109

M. Schröder and J. Trouvain. The German Text-to-Speech Synthesis System MARY: A Tool for Research, Development and Teaching. *International Journal of Speech Technology*, 6(4):365–377, October 2003. ISSN 13812416. doi: 10.1023/A:1025708916924. 79, 154

S. Schwope. Re: Nachtrag: Re: Duplexübertragung, July 2008. Personal Communication. 140

M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996. ISBN 0131829572. 28

M. Shepperd. A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.*, 3(2):30–36, 1988. ISSN 0268-6961. URL http://portal.acm.org/citation.cfm?id=48322. 189

R. Simmons, 2010. Personal communication. 177

R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proc. of Conference on Intelligent Robotics and Systems*, 1998. 6, 60, 61, 72, 177

R. Simmons and D. James. *Inter Process Communication (IPC) – A reference manual.* Robotics Institute, Carnegie Mellon University, Pittsburg, PA, USA, 2001. URL `http://www.cs.cmu.edu/~ipc/`. The manual is for version 2.6. In April 2010, the IPC software is at version 2.8.5. 61, 177

R. Simmons, D. Goldberg, A. Goode, M. Montemerlo, N. Roy, B. Sellner, C. Urmson, A. Schultz, M. Abramson, W. Adams, A. Atrash, M. Bugajska, M. Coblenz, M. MacMahon, D. Perzanowski, I. Horswill, R. Zubek, D. Kortenkamp, B. Wolfe, T. Milam, and B. Maxwell. Grace: an autonomous robot for the aaai robot challenge. *AI Mag.*, 24(2):51–72, 2003. ISSN 0738-4602. URL `http://portal.acm.org/citation.cfm?id=960150.960159`. 178

R. G. Simmons. Structured control for autonomous robots. *Robotics and Automation, IEEE Transactions on*, 10(1):34–43, February 1994. doi: 10.1109/70.285583. 6, 60, 61, 69, 73

W. D. Smart. Is a common middleware for robotics possible? In *Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*, 2007. 28

Command list – Intelligent Communication Color Video Camera EVI-D30/D31. Sony Corporation, 4-16-1, Okata, Atsugi-shi, Kanagawa-ken, 243-0021 Japan, v1.21, english edition, 1999. 139, 142

D. M. Sotirovski and P. B. Kruchten. Implementing dialogue independence. *IEEE SOFTWARE*, 12(06):6170, 1995. doi: http://doi.ieeecomputersociety.org/10.1109/52.469761. 32

T. P. Spexard and M. Hanheide. System integration supporting evolutionary development and design. In H. Ritter, G. Sagerer, R. Dillmann, and M. Buss, editors, *Human Centered Robot Systems*, volume 6, chapter 1, pages 1–9. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-10402-2. doi: 10.1007/978-3-642-10403-9_1. 181

D. Spinellis. Chidamber and Kemerer Java Metrics, version 1.8, 2007. URL `http://www.spinellis.gr/sw/ckjm/`. last checked February 12th, 2010. 191

J. Steffen, S. Klanke, S. Vijayakumar, and H. J. Ritter. Realising dextrous manipulation with structured manifolds using unsupervised kernel regression with structural hints. In *ICRA 2009 Workshop: Approaches to Sensorimotor Learning on Humanoid Robots*, Kobe, Japan, 2009. 14

J. J. Steil, F. Röthling, R. Haschke, and H. Ritter. Situated robot learning for multimodal instruction and imitation of grasping. *Robotics and Autonomous Systems*, 47(2-3):129–141, June 2004. ISSN 09218890. doi: 10.1016/j.robot.2004.03.007. 175, 176

D. B. Stewart and P. K. Khosla. Rapid development of robotic applications using component-based real-time software. *Intelligent Robots and Systems, IEEE/RSJ International Conference on*, 1:465+, 1995. doi: 10.1109/IROS.1995.525837. 32

java.util.concurrent Package Summary. Sun Microsystems, Inc, 2009a. URL `http://java.sun.com/javase/6/docs/api/java/util/concurrent/package-summary.html`. Accessed March 10th, 2010. 63

API Documentation for ServiceLoader class. Sun Microsystems, Inc, 2009b. URL `http://java.sun.com/javase/6/docs/api/java/util/ServiceLoader.html`. Accessed March 10th, 2010. 123

Java™ Platform, Standard Edition 6 API Specification. Sun Microsystems, Inc., 2009c. URL `http://java.sun.com/javase/6/docs/api/`. accessed 20-Jan-2010. 75

J. Tessier. Dependency Finder, version 1.2.1-beta3, 2009. URL `http://depfind.sourceforge.net/`. last checked February 12th, 2010. 191

A. Thomaz and C. Breazeal. Experiments in socially guided exploration: lessons learned in building robots that learn with and without human teachers. *Connection Science*, 20(2):91–110, June 2008. ISSN 0954-0091. doi: 10.1080/09540090802091917. 179

A. L. Thomaz and M. Cakmak. Learning about objects with human teachers. In *HRI '09: Proceedings of the 4th ACM/IEEE international conference on Human robot interaction*, pages 15–22, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-404-1. doi: 10.1145/1514095.1514101. 180

M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Technical report, 2011. URL `http://code.google.com/p/disruptor`. 115

UML2.0. Unified modeling language: Superstructure version 2.0. Technical report, Object Management Group (OMG), Inc, 2005. 127, 128

VERBMOBIL. VERBMOBIL – Erkennung, Analyse, Transfer, Generierung und Synthese von Spontansprache, 2000. URL `http://verbmobil.dfki.de/`. Last accessed 23rd of March, 2010. 154

A.-L. Vollmer, K. Lohan, K. Fischer, Y. Nagai, K. Pitsch, J. Fritsch, K. Rohlfing, and B. Wrede. People modify their tutoring behavior in robot-directed interaction for action learning. In *International Conference on Development and Learning*, volume 8, Shanghai, China, June 2009. IEEE, IEEE. 16

R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The claraty architecture for robotic autonomy. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 1, pages 1/121–1/132 vol.1, 2001. doi: 10.1109/AERO.2001.931701. 31

C. Vorwerg, S. Wachsmuth, and G. Socher. *Visually Grounded Language Processing in Object Reference*, volume 166 of *Trends in Linguistics*. Mouton de Gruyter, March 2006. ISBN 311018897X. 175

S. Wachsmuth and G. Sagerer. Bayesian networks for speech and image integration. In *Proc. of 18th National Conf. on Artificial Intelligence (AAAI-2002)*, pages 300–306, Edmonton, Alberta, Canada, 2002. 22

S. Wachsmuth, S. Wrede, and M. Hanheide. Coordinating interactive vision behaviors for cognitive assistance. *Computer Vision and Image Understanding*, 108:135–149, 2007. 62

D. A. Wheeler. SLOCCount version 2.26. A set of tools for counting physical Source Lines of Code, 2004. URL `http://www.dwheeler.com/sloccount/`. last checked February 12th, 2010. 191

S. N. Woods, M. L. Walters, K. L. Koay, and K. Dautenhahn. Methodological issues in HRI: A comparison of live and video-based methods in robot to human approach direction trials. In *Proceedings of the 15th IEEE International Symposium on Robot and Human Interactive Communication*, pages 51–58. IEEE, 2006. 157

B. Wrede, M. Kleinehagenbrock, and J. Fritsch. Towards an integrated robotic system for interactive learning in a social context. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems - IROS 2006*, Bejing, 2007. 181

S. Wrede. *An Information-Driven Architecture for Cognitive Systems Research.* PhD thesis, Bielefeld University, 2008. 11, 111, 131, 154, 156

S. Wrede, M. Hanheide, S. Wachsmuth, and G. Sagerer. Integration and coordination in a cognitive vision system. In *International Conference on Computer Vision Systems (ICVS)*, St. Johns University, Manhattan, New York City, USA, 2006. IEEE. International Conference on Computer Vision Systems 2006. 6, 34, 60, 62, 63, 70, 73

J. Zhang. Self-valuing learning and generalization with application in visually guided grasping of complex objects. *Robotics and Autonomous Systems*, 47(2-3):117–127, June 2004. ISSN 09218890. doi: 10.1016/j.robot.2004.03.006. 175

J. Zhang and B. Rössler. *Grasp learning by active experimentation using continuous B-spline model*, pages 353–372. Physica-Verlag GmbH, Heidelberg, Germany, Germany, 2003. ISBN 3-7908-1546-2. URL `http://portal.acm.org/citation.cfm?id=860250`. 14