Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model

Christian Schlegel, Andreas Steck, Alex Lotz
Computer Science Department
University of Applied Sciences Ulm, Germany

1 Introduction

Vital functions of robots are provided by software and software dominance is still growing. Mastering the software complexity is not only a demanding but also indispensable task towards an operational robot. The importance of identifying and developing software principles for robotics has achieved increased awareness within the robotics community (SDIR, 2011; JOSER, 2011) and is being considered strategic (EUROP, 2009).

Best practices in robotics, robotics requirements, software engineering and implementation technologies like, for example, middleware systems are strongly interrelated. We explicate important aspects of the subtle relationship between robotics and software technology allowing the identification of stable structures. These are taken as basis for a service-oriented robotic software component model to enable robotics to benefit from Component-Based Software Engineering (CBSE) (Heineman & Councill, 2001). CBSE shifts the emphasis in system-building from traditional requirement analysis, system design and implementation to composing software systems from a mixture of reusable off-the-shelf and custom-built components.

Identified stable structures and established & relevant solutions of an application domain also form the input for a sound model-based representation of core domain concepts. Consequently, we represent the service-oriented robotic software component model as an abstract meta-model. This paves the way towards Model-Driven Software Development (MDSD) (Beydeda et al., 2005) in robotics. MDSD decouples robotics domain concepts valid in the long term from the short update cycles of implementational technologies. MDSD in general is becoming increasingly accepted as approach to handle the increasing complexity of large software systems. Consistency and traceability of artifacts in the development process are two of the major advantages of a model driven development process (Cadenas et al., 2010).

We outline relationships between robotics and software engineering, motivate and illustrate the link between component based software engineering and a service-oriented robotic software component model and close the circle from CBSE to MDSD in robotics. Selected details are illustrated by means of the SMARTSOFT approach (Schlegel et al., 2009).
2 Software Engineering in Robotics

Software for autonomous robots is typically embedded, concurrent, real-time, distributed and data-intensive and must meet specific system requirements, such as safety, reliability, and fault tolerance. From this point of view, software requirements of autonomous robots are similar, to a large extent, to those of software systems in other domains, such as avionics, automotive, factory automation, telecommunication and even large scale information systems. In these domains, it can be observed a strong move towards the application of software engineering principles to significantly reduce the effort to develop new software applications.

In contrast, most robotics research and development is still based on proprietarily designed software architectures and software systems invented from scratch each time. Although robotic applications are typically developed to solve a specific class of problems, one would like to reuse existing and matured software building blocks in order to reduce development time and costs, increase robustness and take advantage from specialized or second source vendors. Up to now, this is not possible due to the lack of an appropriate design abstraction for software in robotics.

As result of a missing design abstraction, tremendous code-bases (libraries, middleware, etc.) coexist without any chance of interoperability and each tool has attributes that favors its use. A huge corpus of software applications, which implement the entire spectrum of robot functionality, algorithms, and control paradigms, are potentially available in robotic research laboratories but are often not reusable even in slightly different application scenarios because they are tight to specific robot hardware, processing platforms, and communication infrastructures and because the assumptions and constraints about tasks, operational environments, and robotic hardware are hidden and hard coded in the software implementation.

The robotics domain imposes specific requirements on software systems. Software approaches for robotic systems have to assist in building a robotic system and to provide a software architecture without enforcing a particular robot architecture. Thus, one has to identify how and where to start with software engineering in robotics and how to tailor software engineering concepts to best match robotic needs. In particular in robotics, there are many demanding functional and non-functional requirements and specifics (like the context and situation dependent configuration of skills to behaviors at runtime) different to other domains (e.g. automotive, avionics and distributed embedded systems in general) introducing additional complexity.

2.1 Software Components to Master System Complexity

As already successfully demonstrated in many disciplines, complexity can be mastered by component based approaches. These split a complex system into several independent units with well-formed interfaces. Complexity is reduced by restricting the focus on a single component when going into details. Fitting of components is ensured by standards for their external appearance and behavior. Examples are COTS (commercial off-the-shelf) mechanical components, electrical components, computer and network components and others.

The same holds true for complex software systems. Component-Based Software Engineering (CBSE) (Heineman & Councill, 2001) is an approach that has arisen in the software engineering community in the last decade. It aims to shift the emphasis in system-building from traditional requirement analysis, system design and implementation to composing software systems from a mixture of reusable off-the-shelf and custom-built components. In the software domain, component based approaches can reduce complexity by decoupling implementations of services from interfaces to access them. Making component dependencies explicit by well-formed interfaces supports reuse of components. A compact and widely accepted definition of a software component is the following one:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be developed independently and is subject to composition
by third parties.” (Szyperski, 2002).

Note that there is an important difference between objects in object-oriented approaches and software components. The main difference is the coarser level of granularity of components. The definition of objects is purely technical and does not include notions of independence or late composition. Although these can be added to objects, components explicitly consider reusable pieces of software that have well-specified public interfaces, can be used in unpredictable combinations and are stand-alone entities.

2.2 User Roles and Requirements

From the user view, we can distinguish several roles that all put a different focus on complexity management and on component based software engineering:

**End users** operate applications based on the provided user interface. They focus on the functionality of readily provided systems. They do not care on how the application has been built and mainly expect reliable operation and easy usage.

**Application builders** assemble applications out of components. Systems should be composable out of off-the-shelf, standardized and thus reusable components. Application builders customize components by adjusting parameters and sometimes even fill in application dependent parts at so-called hot spots. They expect the framework to ensure clearly structured and consistent component interfaces for easy assembling of approved off-the-shelf components.

**Component builders** focus on the specification and implementation of a single component. They expect the framework to provide the infrastructure which supports their implementation effort in such a way that it is finally compatible with other components without being restricted too much in respect of component internals. They want to focus on algorithms and component functionality without bothering with integration issues.

**Framework builders** design and implement the framework such that it matches the manifold requirements at its best and that the above types of users can focus on their role.

2.3 Requirements from a Technical Point of View

From a technical point of view, robotics software always requires to cope with the inherent complexity of concurrent activities, the deployment of software components on networked computers ranging from embedded systems to personal computers, a bunch of platforms, operating systems and programming languages, the requirement to hide distribution aspects by a middleware mechanism, timing and resource constraints and of course also with organizational challenges of distributed development processes and issues of integration of independently developed components. Thus, the following requirements are of particular interest in robotics.

**Dynamic wiring** of components can be considered as the pattern in robotics. Dynamic wiring allows changes to connections between components to be made at runtime. It is the basis for making both, the control flow and data flow configurable at runtime. That is the basis for situated and task-dependent composition of skills to behaviors.

**Component interfaces** have to be defined at a reasonable level of granularity. A reasonable level avoids fine-grained intercomponent interactions and supports loosely coupled components but with a stringent and standardized interface semantics.
Asynchronicity: Patterns of a robotics framework should make use of asynchronicity wherever possible. A system consists of loosely coupled components that run asynchronously and communicate with each other. The usage of asynchronous interactions by the component builder should be as simple as possible.

Component internal structures: Components at different levels of a robot system can follow completely different designs. Component builders therefore ask for as few restrictions as possible with regard to component internal structures. For example, processing chains for computer vision filters are often based on a data-flow architecture, task coordination often prefers asynchronous event-based approaches and task planning or learning mechanisms require separate threads for long-running computations.

Transparency: A framework has to provide a certain level of transparency by hiding details to reduce complexity. Fully hiding all distribution aspects is unpromising since that often not only results in a decrease in performance but also prevents predictability of the time needed for communication and of the use of system resources. In robotics, as in embedded systems in general, explicit assignment of resources to components is needed (for example, bandwidth requirements in vision systems).

Legacy code: A tremendous body of robotics knowledge is available in form of implementations, either standalone applications, libraries or frameworks. A component approach should allow for reuse of existing software libraries inside components. A component has to provide the container for system level interoperability and composition.

Easy usage: Acceptance of a framework is increased by offering obvious additional value not only in the longterm but also incrementally related to the effort put into learning its concepts. It should avoid requiring huge learning efforts prior to be able to use even parts of it. A framework provides additional value only if its usage is much simpler than matching all the requirements without using a framework.

Stable interfaces: A framework should provide stable interfaces that are motivated by the requirements of the application domain and not by the capabilities of certain software technologies and implementation technologies. Stable interfaces allow to decouple the domain concepts from the software implementation. While domain concepts are stable over longer periods of time, implementation technologies change much faster and thus, a decoupling is mandatory.

State-of-the-art software technology: Progress in software technology is driven mostly from outside robotics. Providing access to state-of-the-art software technology without requiring every robot component builder to be a software engineering expert allows robotics experts to focus on algorithms and relieves them from the burden of software integration. A framework can be migrated onto latest software technology as soon as this provides advantages with respect to the robotics needs and without affecting the stable interfaces towards the domain experts.

2.4 Stable Structures versus Variation Points

At its heart, all the above requirements ask for the identification of stable structures versus variation points (Webber & Gomaa, 2004). A suitable approach has to provide guidance via stable structures where these exist across different robotic software systems and it has to support variability where diversity in robotic software systems is needed. The distinction between stable structures and variation points is of concern at all levels of robot software systems ranging from operating system interfaces and library interfaces over component internal structures to provided and required services of components.

The major approach is to follow the idea of freedom from choice instead of supporting freedom of choice (Lee, 2011). In fact, identified and enforced stable structures come along with restrictions for the users. However,
one has to notice that well thought out limitations are not a universal negative. Actually, appropriately restricting freedom of choice for a component developer gives him guidance and assurance of properties beyond his responsibilities. For example, although component-based software engineering is an approach to make a shift from implementation to composition, it does not per se ensure that independently developed components finally fit together. The reason is that general purpose component based approaches still provide far too much freedom and alternatives with respect to defining and implementing component interfaces. In terms of freedom from choice, a developer can expect assistance by strictly reducing the number of offered alternatives such that he can rely on system level conformance of his contributions as long as he sticks to the imposed restrictions.

Freedom of choice, which often is considered the best, has severe drawbacks as can be illustrated by a middleware example. Nearly all robotic systems depend on some kind of middleware. Of course, there are many matured middleware systems already available like CORBA (OMG CORBA, 1997) and DDS (OMG DDS, 2007), for example. However, they are designed to support as many different styles of programming and as many use-cases as possible. As result, there is an overwhelming number of ways on how to implement even a simple two-way communication using one of these general purpose tools. Unfortunately, the various options result in completely different behaviors at the system architecture level. For example, it makes a significant difference whether a communication is buffered or not, what the buffer size is and what kind of threading model is activated (per object, per request or thread pool).

Even most experienced developers face a real challenge when they have to sort out the effects of different styles of interaction used at the component level with respect to the side-effects at the overall system level. At design time, a component developer cannot know (and even is not allowed to make any assumptions) with which other components his component has to interact. Even reasonable arrangements at a bilateral component level might result in inadequate structures at a system level (for example, undesired effects in a circular arrangement like blocking of services).

2.5 Structure of a Component Based Software Approach

A component based approach relies on stable structures at various levels. The basic structure of a component based approach is shown in figure 1.

At the system level (S), provided and required service ports of a component form a stable interface for the application builder. In an ideal situation, all relevant properties of a component are made explicit to support a black box view. Hence, system level properties like resource conformance of the component mapping to the computing platform can be checked during system composition and deployment.

At the component level (C), the component builder wants to rely on a stable interface to the component framework. In an ideal situation, the component framework can be considered as black box hiding all operating system and middleware aspects from the user code. The component framework adds the execution container to the user code such that the resulting component is conformant to a black box component view.

At the framework level (F), two stable interfaces exist: (i) between the framework and the user code of the component builder and (ii) between the framework and the underlying middleware & operating system. The stable interface ensures that no middleware and operating system specifics are unnecessarily passed on to the component builder. The stable interface ensures that the framework can be mapped onto different implementational technologies (middleware, operating systems) without reimplementing the framework in its entirety. The framework builder maintains the framework which links the stable interfaces and maps the framework on different implementational technologies via the interface.
2.6 Model-Driven Software Development

Identifying stable structures and variation points of an application domain provides the ground for a model-based representation of the relevant structures and solutions of a domain. Although those structures can be used directly by a component developer or a system builder, they can also be represented in an abstract and formalized form.

Model-Driven Software Development (MDSD) provides a design abstraction as illustrated in figure 2. Abstractions are provided by models (Beydeda et al., 2005). Abstraction is the most basic principle of software engineering.

"A model is a simplified representation of a system intended to enhance our ability to understand, predict and possibly control the behavior of the system." (Neelamkavil, 1987).

Whereas problem analysis or project requirements are non-computational, models for model-driven software development are computational. Modeling and model transformation constitute the core of model-driven development. Models can be refined and finally be transformed into a technical implementation. MDSD-models are no paperwork, they are the solution which can be translated into code via tools.

MDSD is a technology that introduces significant efficiencies and rigor to the theory and practice of software development. In MDSD, models are used for many purposes, including reasoning about problem and solution domains and documenting the stages of the software life cycle; the result is improved software quality & time-to-value and reduced costs (IBM, 2006).

Figure 3 gives an overview on the model-driven software development process and the overall model transformation steps. Starting with an idea, the model is enriched during development time until finally it becomes

---

**Figure 1:** Stable structures and roles in a component based software approach.
Figure 2: Design abstraction of model-driven software development.

Figure 3: Model-Driven Software Development at a glance.

Executable in form of deployed software.

The *first* step is to describe the system in a model-based representation (*platform independent model, PIM*). In this design phase, the architecture is specified independently of the underlying framework, middleware structures, operating system and programming language. At this stage, for example, the tasks and services of a software component are specified while its properties like task execution times are only known as requirements.

The *second* step is to transform the PIM into a *platform specific model (PSM)* based on further information provided by the *platform description model (PDM)*. Model checks can test whether all required parameters for this transformation step are consistently specified. For example, the platform independent representations of tasks and services of a component can be transformed into the appropriate elements of the platform specific metamodel. These elements already represent characteristics of the underlying environment (middleware structures, operating system, framework, additional structures needed to map the concept of a periodic task onto a specific operating system etc.).

The *third* step is to transform the PSM into the *platform specific implementation (PSI)* exploiting further parts of the PDM. This step replaces the meta-model elements by concrete code elements and, for example, sets up header files etc. Developers add their algorithms and libraries (user code) under guidance of the toolchain. However, some parameters like worst case execution times of tasks or device names of particular interfaces can still be unknown and are added not until the deployment of the component.

The *fourth* step is to prepare the PSI for deployment. The capabilities and characteristics of the target platforms are again provided by the PDM. Further model checkings are performed to verify the component’s constraints against the capabilities of the target platforms (e.g. is executable only on a certain type of hardware, needs one serial port etc.). The result are deployable executables with parameter and initialization files.
The above workflow is supported by tools like Eclipse Modeling Project (Eclipse Modeling Project, 2010). These provide means to express model-to-model and model-to-code transformations. They import standardized textual XMI representations of the models and can parse them according to the used meta-model. Thus, one can easily introduce domain specific extensions like additional stereotypes, tagged values and constraints to forward information from the model-level to the model transformation and code generators. Tools like Papyrus (PAPYRUS UML, 2010) allow for graphical representations of the various models and can export them in the XMI format. Thus, there is a complete toolchain for graphical modeling and model-to-model and model-to-code transformations available that can be tailored for domain specific needs.

2.7 The Relationship between CBSE and MDSD

CBSE and MDSD have much in common in adopting many concepts from systems engineering and benefit from cross-fertilization (Törngren et al., 2005a; Törngren et al., 2005b). Both address complexity management: CBSE separates the component development process from the system development process and aims at component reusability. Thus, substitutability and composability are important concerns. MDSD separates domain knowledge (formally specified by domain experts) from how it is being implemented (defined by software experts using model transformations). Formal models to capture domain knowledge, separation of functionality and implementation and stepwise refinements from reusable models to (also different) implementations are important concerns.

It is important to understand that MDSD is much more than just code generation for different platforms to address the technology change problem and to make development more efficient by automatically generating repetitive code. The benefits of MDSD are manifold (Stahl & Völt, 2006; Völt, 2006): (i) models are free of implementation artefacts and directly represent reusable domain knowledge including best practices, (ii) domain experts can play a direct role and are not requested to translate their knowledge into software representations, (iii) design patterns, sophisticated & optimized software structures and approved software solutions can be made available to domain experts and enforced by embedding them in templates for use by highly optimized code generators such that even novices can immediately take advantage from a coded immense experience, (iv) parameters and properties of components required for system level composition and the adaptation to different target systems are explicated and can be modified within a model-based toolchain.

2.8 CBSE and MDSD in Robotics

An overview on CBSE in robotics and on design principles to enable the development of reusable and maintainable software building blocks is given in (Brugali & Scandurra, 2009; Brugali & Shakhimardanov, 2010). Up to now, many fundamental requirements on CBSE and MDSD are not fulfilled by currently wide-spread robotics software frameworks.

ROS (Quigley et al., 2009) is a currently widely-used framework and typical representative of the current situation in robotics software. ROS provides a huge and valuable codebase of (open source) drivers and libraries for a wide variety of robotic systems including, for example, navigation, perception, simulation and visualization. ROS aims at supporting the reuse of different libraries by means of a shared build infrastructure and means for node communication. Although ROS sees its nodes as components, ROS lacks a pivotal property of a component based approach. CBSE requires identified stable structures which provide an execution container and guide the component developer such that he ends up with system level conformance for composability. Instead, ROS supports side-by-side existence of all kinds of overlapping concepts without an abstract representation of the core features and properties. ROS lacks a component model representing its node concept independently of a particular implementation. In a nutshell, all software stacks compile in the same way and co-exist, but up to now there exist no guidelines towards achieving system level conformance.

For example, the core communication of ROS provides two mechanisms: (i) a publish/subscribe inter-
action called *topic* and (ii) a synchronous request/response interaction called *service*. In addition, users are allowed to introduce and use a variety of further communication mechanisms provided as add-on libraries (e.g. *actionlib*) and different styles of using communication mechanisms are encoded as part of the user code. As consequence, node-builders (role (C) in figure 1) not only bind their user code to different styles and flavours of non-interoperable communication mechanisms but they even use them across components thus violating the concept of stable and interoperable component interfaces (1, 2 in figure 1). The apparent effect for *ROS* users of a missing clear separation of user code and framework code are frequently changing *APIs*. As no clear definition of the different communication mechanisms is available, an abstract component model which is independent of code fragments is not available. Components with individual communication mechanisms become non-separable and cannot be reused separately. Providing components with unprecisely specified interfaces makes it difficult for the application builder (S) to come up with systems with explicated properties like resource requirements. As such frameworks do not provide a precisely defined component model, they cannot adequately be integrated into a model-driven approach.

The ongoing *BRICS* project (BRICS, 2011) specifically aims at exploiting *model-driven engineering (MDE)* as enabling approach to reducing the development effort of engineering robotic systems by making best practice robotics solutions more easily reusable. The 3-View Component Meta-Model $V^3CMM$ (Alonso et al., 2010) is one of the few model-driven tool-chains for robotics software development enabling component-based platform-independent design modeling and platform-specific code generation by means of model transformations. $V^3CMM$ comprises three complementary views (structural view, coordination view, algorithmic view) to model component-based robotics systems. It supports structural and behavioural variability modeling at design-time. The *GenoM3* (Mallet et al., 2010) project tries to harmonize different robotic frameworks by providing component templates that are specific for each framework. The user code and libraries are separated from the framework details and the interface 2 is provided by the generic *GenoM* component model. However, there is no explicited meta-model available and the single-step transformation in form of component templates generating against a framework does not conform with best practices in MDSD. It lacks, for example, the well-founded and established abstraction level of a *PSM*. EasyLab (Geisinger et al., 2009) is a software tool for modeling, simulation, code generation and debugging with the primary focus of mechatronic systems. The strength of EasyLab is a consistent implementation of model-driven development of robot systems at different levels. Currently, it is used to program smart sensors and actuators.

The *Object Management Group (OMG)* standardized the Robotics Technology Component (RTC) Specification (OMGRTC, 2008) comprising a robotics component model definition and description. This initiative illustrates the high value of providing an abstract component model which can be implemented by different stakeholders using different implementation technologies. Three important implementations of that standard are available: (i) Gostai RTC (Gostai, 2011), (ii) OpenRTM-aist (Ando et al., 2005) and (iii) OPRos (Song et al., 2008). The implementations all adhere to the same component model but set different priorities in implementing it. Unfortunately, not all implementations are already matured enough with respect to adhering to the defined abstraction levels. For example, the *CORBA* based OpenRTM-aist implementation does not fully hide the *CORBA* middleware details from the component builder (C). Thus, the user code contains *CORBA* code fragments and has therefore a binding to this specific implementation of the specification.

The *RTC* specification is strongly influenced by use-cases requiring a data-flow architecture. Thus, its component model in the current stage is strongly influenced by a strict internal automaton structure that is tightly coupled with the activity model inside a component. For example, it does not easily allow multiple tasks inside a component. Nevertheless, providing an abstract component model is the only way to discuss and compare different robotic concepts and component models with the overall aim of harmonizing the various models without...
getting stuck in implementation details and at the level of code fragments. The RTC specification is considered being the most advanced concept of MDSD in robotics.

3 The SMARTSOFT-Approach

The basic idea of SMARTSOFT (Schlegel, 2004; Schlegel, 2006; Schlegel, 2007) is to master the component hull. The component hull provides the interface between the internal structure of a component and its outside view (see ①, ② in fig. 1). A component hull also provides the link towards the underlying operating system that provides the resources required to run the component (② in fig. 1). By mastering the component hull, one gains control over all aspects relevant to address the challenges explained in sections 2.2 and 2.3.

*Freedom from choice* means that a component developer can compose a component hull only out of exactly specified building blocks. This gives the freedom to fill in component internals according to his preferred approach while not being able to present arbitrary structures outside the component. Thus, component internal structures cannot impose constraints on the externally visible interface (which would have the potential to result in dependencies that span across components).

3.1 The SMARTSOFT-Component

The structure of a SMARTSOFT component is shown in figure 4. All *required* and *provided services* of a component are built on top of a small set of communication patterns. A communication pattern connects the externally visible service (stable interface to other components) with the internally visible set of access methods for this service (stable interface to user code inside a component). Communication patterns provide a completely middleware independent view on the component ports and on the communication interfaces visible to the user. Thus, communication patterns achieve the required abstraction from implementation technologies with respect to middleware systems.

![Figure 4: Overview on the structure of a SMARTSOFT component.](image-url)

User code inside a component can access any part of the stable interface inside a component. The internally visible stable interface to the user code also provides operating system independent representations of the most often needed artefacts like threads, timer, synchronization mechanisms, memory access etc. Since these are based on ACE (Schmidt, 2009; Schmidt & Huston, 2002; Schmidt & Huston, 2003), they are also stable across nearly
all relevant operating systems in use. User code can comprise any number of threads and can include any kind of existing libraries. Of course, as soon as user code introduces bindings to a specific platform, the component gets bound to this particular platform.

**Components:** A component can contain several threads and interacts with other components via predefined communication patterns that seamlessly overcome process and computer boundaries. Components can be dynamically wired at runtime. A component needs not to be a process: multiple components can share a process and the communication patterns can even be used inside components to structure the interaction of subparts of a component.

**Communication Patterns** assist the component builder and the application builder in building and using distributed components in such a way that the semantics of the interface is predefined by the communication patterns, irrespective of where they are applied. A communication pattern defines the communication mode, provides predefined access methods and hides all the communication and synchronization issues. It always consists of two complementary parts named *service requestor* and *service provider* representing a *client/server*, *master/slave* or *publisher/subscriber* relationship.

**Communication Objects** parameterize the communication pattern templates. They represent the content to be transmitted via a communication pattern. They are always transmitted *by value* to avoid fine grained intercomponent communication when accessing an attribute. Furthermore, object responsibilities are much simpler to manage with locally maintained objects than with remote objects. Communication objects are ordinary objects decorated with additional member functions for use by the framework. Universal applicability of the approach is achieved by using arbitrary and individual communication objects to instantiate communication patterns.

**Service:** Each instantiation of a communication pattern provides a service. Generic communication patterns become services by binding the templates by types of communication objects.

The service based view comes along with a specific granularity of a component based approach. Services are not as fine grained as arbitrary component interfaces since they are self-contained and meaningful entities. Major characteristics are as follows.

- Communication patterns provide the only link of a component to its external world. Communication patterns decouple structures used inside a component from the external behavior of a component. Decoupling starts with the specific level of granularity of component interfaces enforced by the communication patterns which avoids too fine grained interactions and ends with the message oriented mechanisms used inside the patterns.

- Using communication patterns with given access modes prevents the user from puzzling over the semantics and behavior of both, component interfaces and usage of services. One can neither expose arbitrary member functions as component interface nor can one dilute the precise interface semantics and the interface behavior. Given member functions provide predefined user access modes and hide concurrency and synchronization issues from the user and can exploit asynchronicity without teasing the user with such details.

- Arbitrary communication objects provide diversity and ensure genericity with a very small set of communication patterns. Individual member functions are moved from the externally visible interface to communication objects.
• **Dynamic wiring** of intercomponent connections at runtime supports context and task dependent assembly of components. Reconfigurable components are modular components with the highest degree of modularity. Most important, they are designed to have replacement independence. Many component approaches only provide a deployment tool to establish component connections before the application is started.

• Services need to be self-contained which means, a service requestor is not allowed to make any assumptions about the state of the service provider when invoking it. Each communication object must be designed such that it contains all the necessary information to be used as stand-alone entity without requiring a particular additional context. For example, it makes sense to provide time and pose stamps with many communication objects in robotics to be able to process the information decoupled from the time of reception or the current pose of the robot.

Since component interactions are mapped onto predefined communication patterns, all component interfaces are composed out of the same set of patterns. Therefore, looking at the external interface of a component immediately opens up the provided and required services, and looking at the communication pattern underlying a service immediately opens up the usage and semantics of this service. Mastering the component hull (which mediates between externally visible services of a component and their component internal implementation) by a set of predefined communication patterns

• effectively enforces the component architecture and the appropriate level of abstraction & decoupling of the externally visible services (stateless and self-contained services),
• supports explication of non-functional properties (quality of service attributes, resource-awareness) for services and components,
• provides solutions for robotics specific requirements (state and activity management, dynamic wiring of services at runtime), and
• provides a suitable starting point for a meta-model (structures and semantics) which represents the characteristics of the component model independently of the underlying implementation technology.

![Figure 5: Composability of components due to standardized component interfaces.](image-url)

A set of navigation components is shown in figure 5. At the component hull, one can identify required and provided services and one can also see on which particular communication pattern a port is based on. For example, the *path planning* component gets a map update from the *map building* component. That service is based on a *push newest* pattern and transmits a *map* communication object. The semantic of the service provider port at the *map building* component is completely specified by the server part of the *push newest* pattern. The semantic of the service requestor port at the *path planning* component is completely specified by the client part of the *push newest* pattern.
3.2 The SMARTSOFT Communication Patterns

Restricting all component interactions to given communication patterns requires a set of patterns that is sufficient to cover all communicational needs. Of course, one also wants to find the smallest such set for maximum clarity of the component interfaces and to avoid unnecessary implementational efforts for the communication patterns. One the other hand, one has to find a reasonable trade-off between minimality and usability. In principle, a request/response service and a push service would be sufficient to provide all the above interactions. However, that would not be very comfortable to the robotics user. The goal is to keep the number of communication patterns as small as possible without restricting ease of use. Table 1 shows the set of SMARTSOFT communication patterns. They are underpinned by numerous use-cases. A service provider can handle any number of clients concurrently.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Relationship</th>
<th>Communication Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
<td>client/server</td>
<td>one-way communication</td>
</tr>
<tr>
<td>query</td>
<td>client/server</td>
<td>two-way request/response</td>
</tr>
<tr>
<td>push newest</td>
<td>publisher/subscriber</td>
<td>1-to-n distribution</td>
</tr>
<tr>
<td>push timed</td>
<td>publisher/subscriber</td>
<td>1-to-n distribution</td>
</tr>
<tr>
<td>event</td>
<td>client/server</td>
<td>asynchronous conditioned notification</td>
</tr>
<tr>
<td>state</td>
<td>master/slave</td>
<td>activate/deactivate component services</td>
</tr>
<tr>
<td>wiring</td>
<td>master/slave</td>
<td>dynamic component wiring</td>
</tr>
</tbody>
</table>

Table 1: The set of SMARTSOFT communication patterns.

Communication patterns make several communication modes explicit like a oneway or a request/response interaction. Push services are provided by the push newest and the push timed pattern. Whereas the push newest pattern can be used to irregularly distribute data to subscribed clients whenever updates are available, the latter distributes updates on a regularly basis. The event pattern is used for asynchronous notifications if an event condition becomes true under the activation parameters. The wiring pattern covers dynamic wiring of components at runtime.

3.3 Use-Cases of the Communication Patterns

The send pattern implements a client initiated one-way communication. It transmits a communication object from a client (service requestor) to the server (service provider). It is used, for example, to control the velocity or the steering angle of a mobile platform. It is also used to implement a data driven processing chain of components.

The query pattern implements a client initiated two-way communication. A client (service requestor) sends a request containing individual parameters and receives an individual result from the server (service provider). The query pattern is also used if a service is needed at a very low rate compared to the cycle time of the service. To save communication bandwidth, it makes more sense to perform a query if new data is needed instead of being overrun by not needed updates. The query pattern is, for example, used to request a particular part of a map where the request specifies the size and the origin of the map patch that is then returned by the answering communication object.

The push patterns (push newest, push timed) provide a publish/subscribe mechanism for data distribution. Each client gets the same data as soon as new data is available at the server without requiring polling. The communication is initiated by the server and not by the clients. Compared to the query pattern, a push service is superior to polling in case several clients need the same data or in case the update rate is dictated by the server. The push patterns are, for example, used to distribute laser range scans to various components and to provide an
updated local map patch as soon as something has changed (which is known at the server side).

The basis of the *push timed* pattern is the same as that of the *push newest* pattern. The *push timed* pattern, however, distributes data on a regular basis and therefore provides an additional mechanism to regularly trigger data distribution. The *push timed* pattern is triggered periodically by the framework to acquire new data and distributes the data on a regular basis with individual client update intervals. It relieves the component builder from handling timing issues. The *push timed* pattern is, for example, used to regularly provide a base state including pose estimates.

The *event* pattern supports asynchronous notifications. An event activation provides a parameter set for the event condition. The event condition is checked at the server side. An event activation fires as soon as the event condition becomes true under its individual activation parameters. The returned communication object can contain any required information about what caused the event to fire. As with the *push* pattern, the server is the active part and the server decides on when to check the event condition. The *event* pattern, however, does not distribute the same data to every subscribed client but provides individual answers for each firing activation.

Events are, for example, used for vertical communication between components. They are used to synchronize continuous task execution progress with a discrete description of task plots (for example, based on state charts or task nets). Events are also used to monitor battery voltage, for example. The server side knows when to check the activated events. Each event activation is based on the same predicate (condition to be checked) but can comprise its individual parameterization. For instance, one activation can check whether the battery voltage drops below a warning threshold, another activation of the same event can check whether the voltage level is below a critical threshold and another activation can be parameterized such that it reports when voltage is again in the expected range.

### 3.4 The Structure of Communication Objects and their Interfaces

Figure 6 shows the basic structure of a communication object by the example of a *laser range scan*. Communication objects are regular objects decorated by additional member functions for use by the framework. The user interface does not expose any middleware specific data structures to the user and is thus stable with respect to changes in the underlying middleware. Marshalling is transparent to the user since it is done inside the communication objects.

**Figure 6:** The basic structure of a communication object and its middleware interface.
Communication objects are transmitted by-value as illustrated in figure 7. This appears as moving a copy of an object between components and works like a copy constructor or assignment operator across component boundaries executed via the communication patterns. This preserves the loose coupling between components by not spanning object life cycles across components. The framework transmits objects by solely transmitting the data content of a communication object which keeps the network traffic low. The get-method of the framework interface extracts the relevant data content and marshalls it into a platform independent representation for transmission. The set-method demarshalls the platform independent representation and fills the data content of the object instance at the receiving side. The identifier-method is used by the framework to uniquely identify the type of a communication object. These member functions connect to the underlying middleware and are the only place where its marshalling mechanisms are visible (and only to the builder of communication objects).

The internal data structure in figure 6 consists solely of a laser range scan in polar coordinates but the user interface also supports cartesian coordinates. The required conversion is performed locally in the member function and does not result in a remote access with network load. In case further access methods (even with very special purpose data structures) are required within a component, one can locally derive from the communication object and extend its member functions. This is important since it provides freedom inside a component to best match the internally used data structures while neither affecting all the other components using that communication object nor resulting in fat interfaces visible outside a component.

The internal data structure in figure 6 holding the laser range scan in polar coordinates consists of a std::list. Two different implementations of the interface are shown, one for ACE/SMARTSOFT (Schlegel & Lotz, 2010; ACE/SmartSoft, 2009) and one for CORBA/SMARTSOFT (CORBA/SmartSoft, 2004). Different implementations of the interface can co-exist within a communication object and are selected / switched according to the underlying framework implementation. It is important to note that the interface decouples and from any middleware specific data types (ACE-types are not middleware specific but are completely platform independent abstractions). Explicit get/set-methods provide means to (de)marshall arbitrary data types like a std::list or even Boost library data structures without restrictions of, for example, a CORBA IDL compiler.

### 3.5 The Life-Cycle-Automaton of a Component

A SMARTSOFT component comprises a standardized life-cycle automaton to manage its internal activities like startup, shutdown, error conditions and user activities (Schlegel et al., 2011). Figure 8 shows this standardized life-cycle automaton by the example of the mapper component. The generic part consists of the main states Init, FatalError, Shutdown and the pseudo state Alive. The pseudo state Alive can be extended by component specific user-defined states but always comprises the main state Neutral.

The Init state indicates that the component is currently performing its startup, the Shutdown state indicates that it is going down. In both states, the provided services of the component are not operational. They are either not yet announced or are already withdrawn. The FatalError state indicates severe problems inside a component.
component that cannot be handled internally and require external assistance (e.g. even the basic initialization failed or absolutely mandatory resources are not available). It is important to note that problems do not result in a *FatalError* state as long as the provided services are up and can at least distribute communication objects with an *invalid* flag requiring the service requestors to properly react on the reduced quality of service. In the *Neutral* state, a component consumes as less resources as possible, rejects subscriptions of its services and responds to requests just by indicating its neutral state. In the *Neutral* state, parameterizations and wirings can be safely modified.

The state automaton is managed via the *state slave pattern* and is accessible from outside the component via the *state master pattern*. From outside a component, only *main states* are visible. *Main states* can be commanded to control the component’s activity. Figure 9 shows the master-slave relationship of the *state pattern*. A state master can connect to an arbitrary number of state slaves to gain control over them. The *state pattern* is, for example, used by a task coordination to manage activities within components, to control their life-cycle, to perform configuration changes while they are in safe states and to save resources by switching off components when they are not needed etc.

In the example in figure 8, the main states visible from outside are the user-defined main states *BuildCurrentMap*, *BuildLongtermMap*, *BuildBothMaps* and the generic system-defined main states *Init*, *Neutral*, *FatalError*, *Shutdown*. A main state is a mask overal several substates. The substates are visible only inside a component. In the example, these are the user-defined substates *buildCurrentMap*, *buildLongtermMap* and the system-defined substates *init*, *fatalerror*, *shutdown*, *neutral*, *nonneutral*. The *CurrMapTask* to build the current map runs only if the substate *buildCurrentMap* is available (the component needs to be in one of the main states *BuildCurrentMap*, *BuildBothMaps*) and the *LtmMapTask* runs only if the substate *buildLongtermMap* is available (the component needs to be in one of the main states *BuildLongtermMap*, *BuildBothMaps*). The advantage of this approach is that the component internal knowledge on which activities are compatible and are allowed to run at the same time needs not be exposed to the outside. Instead, the outside view just consists of main states between which one can arbitrarily switch. This decouples the stateful component internal structure from the outside and allows for standardized and still user-extensible state structures at the external configuration interface.

As shown in figure 9, an activity inside a component can lock a substate to inhibit state changes at critical sections. That prevents a component activity from being interrupted by state change requests from outside at an unsuitable point of execution. In the mapper example, the *CurrMapTask* regularly acquires and releases the *buildCurrentMap* substate, the *LtmMapTask* the substate *buildLongtermMap*.

---

**Figure 8:** Life-cycle and state automaton inside the mapper component.
The state pattern gives the master precedence over the slave for state changes. As soon as a request to change the main state is received from the master, the slave rejects component internal locks for substates that are not compatible to the pending request of the master. The requested main state change of the master is executed by the slave as soon as all locks for substates affected by the main state change are released. A slave just follows its regular course of activities, releases substate locks and gets blocked either in case a required substate is inactive or is pending for deactivation.

A main state change can also be enforced by the master. In that case, all blocking calls to communication patterns do not block anymore but return immediately with an appropriate status code. Thus, the component rushes through as fast as possible (following its regular course of activities) until it releases the substate lock which allows the slave pattern to perform the mainstate change request. Therefore, the state pattern closely interacts with all the communication patterns and cannot be implemented independently of the communication patterns.

User provided handlers at the slave allow for cleanup tasks with a state change. Enforcing state changes is extremely useful for performing state changes of multiple components in any order as is needed by the task sequencing layer. Otherwise, inappropriate orders of state change requests might result, for example, in pending queries that prevent a component from making progress towards releasing its locked state which then prevents the component from performing its state change. It is important to note that all components still follow their regular course of activities and thus perform all required housekeeping activities, it is not an interruption or exception.

Figure 10 shows how the startup code of a component is attached to the state automaton. The component builder defines after which initialization steps he wants the component to switch from Init to Alive (commanding the pseudo state Alive results in the main state Neutral by default) and what conditions require a switch into the FatalError state or what kind of activities have to be performed under protection by the Shutdown state.

3.6 Dynamic Wiring and Online Reconfiguration

The wiring pattern provides a consistent mechanism for dynamic wiring of client parts of communication patterns from outside a component. The service requesting part of a communication pattern can expose itself as wireable from outside the component by enrolling at the component’s wiring slave object. It becomes a managed client port and can then be connected to an appropriate service provider from outside the component via the master part of the wiring pattern. Dynamic wiring is mainly used by task coordination layers which compose different behaviors out of a set of skills taking into account current situation and context. This requires to change the data flow between components at runtime.

Figure 11 shows an application of the dynamic wiring pattern. The CDL component (motion control) can receive its intermediate goals either from a path planner or a person tracker and its distance information from either the laser or the PMD component. Dynamic wiring allows to reuse the CDL component within many different settings and behaviors. It allows at runtime to temporarily replace the physical robot component by a robot simulator component (Gazebo (Gazebo, 2006; Gerkey et al., 2003) with physics engine). This online
Figure 10: Linking component startup code to the life-cycle automaton inside a component.

Figure 11: Online reconfiguration of the dataflow by means of the dynamic wiring pattern.

Simulation is used at runtime to check certain parameters like maximum velocity prior to commanding them to the physical robot.

3.7 The SMARTMARS Meta-Model

SMARTMARS (Modeling and Analysis of Robotic Systems) (Schlegel et al., 2009) represents the above concepts independently of any implementational technology. It covers two different views: (i) it is a completely abstract meta-model for modeling and analysis of robotic systems and (ii) it is a concrete reference implementation in form of a UML profile (Fuentes-Fernández & Vallecillo-Moreno, 2004).

Figures 12 and 13 show core elements of the meta-model and the representation of the SMARTSOFT component model. The meta-model lays out the concepts behind the SMARTSOFT component model, its elements and how they are related to each other. The meta-model answers the question how a well-formed SMARTSOFT component has to be built that is it defines which elements it has to contain, how these have to be arranged etc.
Figure 12: Excerpt of the SMARTMARS meta-model.

Figure 13: The communication patterns within the SMARTMARS meta-model.

For example, a component can contain any number of tasks and interaction patterns. However, as interaction patterns only the small set of communication patterns is allowed etc.

4 SMARTSOFT and CBSE

4.1 Building a SMARTSOFT-Component as Component Builder

The first step of a component builder is to specify the component hull. A component hull can be reused in order to provide an alternative implementation of an already existing component. A component hull specifies the required and provided services (composed out of a communication pattern and the used communication objects). Services like, for example, providing a laser range scan on a regularly basis (based on a laser range scan communication object and a push timed pattern) are stable across many robotics applications and thus, component hulls are typically composed out of existing services as far as possible. New communication objects are introduced extremely carefully since too many different manifestations make services incompatible and prevent components from be-
The reuse of generic communication objects is supported by (i) the granularity of services as imposed by the communication patterns, (ii) the requirement on services to be self-contained and stateless and (iii) the option of adding further methods to communication objects locally and inside a component by derivation without resulting in fat interfaces outside components.

### 4.2 Glueing User Code to Service Ports as Component Builder

Figure 14 illustrates how existing libraries are integrated into the user space of a component hull and how the glue logic looks like to link existing libraries (or even code generated by tools) to the component hull. It is important to note that there is no need to modify the data structures of existing libraries for reuse within a component hull. The local extensibility of communication objects to adapt to locally required data structures significantly simplifies the reuse of existing code while not exposing those specific data structures to the outside of a component or to other users of the communication object.

![Figure 14: Glueing a service port with library code inside a SMARTSOFT component.](image)

The example shows part of a SMARTSOFT component hull wrapping the Player/Stage simulator (Gerkey et al., 2003). The Player base state (pose, velocity) is regularly provided via the SMARTSOFT push-timed pattern. The interface A is the stable user interface of the push-timed server consisting of an upcall into the user code. Part C denotes the access from the user code to the Player library. The base state reported from Player is being put into the communication object of the SMARTSOFT service (B) and handed over to the push-timed server (A).

The stable interface to the user of the communication patterns (A) is illustrated by means of the query pattern. Both parts of a communication pattern can provide completely different access modalities since both parts are not only forwarding method calls (as is the case with proxies) but are standalone entities.

The client side of the query pattern (figure 15 on the right) provides both, synchronous (query) and asynchronous (queryRequest, queryReceive, queryReceiveWait, queryDiscard) access modalities. These can be used at the same time in any order and from the same or different threads. The component developer can select that access modality which best fits to his internal structure and needs. The access methods are completely independent of the underlying middleware, do not contain any middleware-related data structures and do not show any middleware-related behaviors. It is the framework builder who maps the communication patterns onto a specific middleware and thereby ensuring that the specified semantics is being fulfilled.

The server side of the query pattern provides an asynchronous handler-based interface (figure 15 on the left). Each incoming request results in an upcall to a registered handler. A handler can provide any kind of
process the request. This is done by creating a separate handler upcall method to provide the result to the query server. For example, in the case of a processing pipeline, this allows the upcall to immediately return after it forwarded a request to the processing pipeline. The last processing step of the pipeline then puts the result into the query server. A synchronous handler upcall would have to wait until the result is available before returning. Thus, it would either block the upcalling thread or require a thread per upcall just for waiting. The asynchronous handler-based approach moves the threading model from inside a communication pattern to the user level again providing much better control on assigned resources.

4.3 Migrating Communication Patterns as Framework Builder

For the framework builder, the communication patterns of SMARTSOFT provide a stable interface to the middleware layer. SMARTSOFT uses the message-based connection-oriented split protocol (Schlegel, 2004, section 5.6.6) that is based on oneway messages. It only requires that messages between a particular client and a particular server keep their initial order and never pass each other (Schlegel, 2004, page 137). It works with a reliable send, a delivered and even a processed policy. Therefore, the message-based protocol of SMARTSOFT can be mapped onto all kinds of different middleware systems with all kinds of different characteristics. The CORBA-based mapping is detailed in (Schlegel, 2004) and the ACE (Schmidt, 2009) based mapping in (Schlegel & Lotz, 2010).

5 SMARTSOFT and MDSD

5.1 The SMARTSOFT MDSD Process at a Glance

Figure 16 gives an overview on the model-driven software development process based on the SMARTSOFT component model.

At the PIM level, the component hull and the internal structures of the component ranging from the service...
ports over the basic components structures (component state automaton) to the tasks are specified based on the concepts provided by the meta-model. For example, the concept of a SmartTask is completely independent of its final implementation but already explicates which parameters and attributes finally need to be bound. In the example, a task is specified as periodic with a period of 1000ms and hard realtime with a wcet (worst case execution time) of 100ms. Of course, all timings are requirements as long as the target platform is not selected. The priority can also not be known until the characteristics of all tasks that become mapped onto the same processor are known.

At the PSM level, platform-specific information is added. In the example, the target platform is Linux with the RTAI realtime extension. Based on general parameters of Linux platforms, the wcet becomes estimated as 80ms. The generic SmartTask is refined into a RTAITask due to its parameters periodic and hard realtime.

In the next step, the deployment is specified. The component is mapped onto a specific computer with IP 192.168.100.2 and a P8700 CPU. At this stage, one knows the complete details of the target platform and also which other components are mapped onto this target system. Thus, the wcet becomes known and all hard realtime tasks on this CPU can be assigned priorities based on rate monotonic scheduling (RMS), for example.
5.2 Toolchain-Support of the MDSD Process for the Component Builder

Figure 17: The PIM (platform independent model) of the mapper component.

Figure 17 shows the view of the component builder on the platform independent model (PIM) of the mapper component. On the right side, one can see the palette of elements that are available to the component builder to setup its component hull and internal structure. The left side shows the components available within the MDSD toolchain with a particular mapper component being openend. In the middle, the PIM of the SmartMap-PerGridMap is shown. This component builds various grid maps and provides several services on grid maps. Various parameters for the current map and the longterm map are explicated in the model.

Figure 18 shows the platform specific model (PSM) of the mapper component after the CORBA based implementation of the SMARTSOFT framework has been selected. The generic concepts are transformed into the CORBA-specific representations. In this example, the component builder neither needs to go into the details of the PSM nor has he to adjust anything at this abstraction level. The PSM level is used by the toolchain as detailed in section 5.4.

Figure 19 shows the platform specific implementation (PSI) of the mapper component. Depending on the services specified in the PIM, the various code fragments are composed to form the code of the component hull with explicated sections to be filled-in by the user. The example shows the query handler which responds to requests for the longterm map. In that example, the user code just assigns the most recent version of the longterm map to the communication object which is then put into the query server via the answer method.

The user added code needs to be protected from modifications by the code generator. A common way to weave manually written code into generated software artifacts is provided by the generation gap pattern (Vlissides, 1996). The general approach is shown in figure 20. An abstract base class introduces behaviour and methods specific to the model and will be overwritten each time the generator produces artifacts. Concrete classes allow manual extensions of generated code without being hit by output of the code generator.

As soon as inserted user code imposes further constraints on where the component can run (e.g. runs only...
Figure 18: The PSM (platform specific model) of the mapper component.

on RTAI-Linux due to using RTAI-specific APIs in the user code, included libraries that are not portable across platforms etc.), this needs to be made explicit by the component builder. The component builder can indicate user code constraints by free-form tags on the component representation. At deployment, these are matched against the platform description model (PDM) and result in hints to the toolchain user.

5.3 Toolchain-Support of the MDSD Process for the Application Builder

Figure 21 shows off-the-shelf navigation components assembled by an application builder via a deployment diagram. The deployment diagram comprises the initial wiring of the component services and filled-in parameters of the component instances. For example, several components allow the specification of cycle times of their tasks. The application builder can finetune the settings at the system level.

The key advantage of the model-based approach is the explicit representation of fundamental properties of robotic systems that so far have not been made detailed enough nor explicit in most of the robotic software systems. As shown in figure 22, parameters can be extracted from the overall model and handed over to a schedulability analysis tool. The parameters of all hard realtime tasks mapped onto the same processor independently to which component they belong are forwarded to CHEDDAR (Singhoff et al., 2004) for timing analysis. The
5.4 The Model Transformation Steps inside the MDSD toolchain

Figure 23 shows an example of a model-to-text transformation template as used within the MDSD toolchain. The template specifies how a query server handler in the CORBA-based PSM has to be transformed into the PSI of a query server handler. The toolchain selects and arranges predefined code snippets guided by this template to end up with compilable code. These templates are neither visible to the component builder nor the application builder. They are provided by toolchain experts based on the input of the framework developers.

Figure 24 shows a major advantage of the concept of abstraction levels and transformation steps. The platform independent model of a SmartTask covers periodic and non-periodic as well as hard-realtime and best-effort tasks. Some platforms like RTAI-Linux directly support periodic tasks whereas other platforms require additional structures to emulate the behavior of periodic tasks. However, these structures are themselves generic across many platforms. The PIM-to-PSM transformation step introduces these structures where needed but still in a generic form. Only afterwards, the PSM-to-PSI transformation expands these into platform specific code. Thus, the transformation steps can reuse many and even abstract structures for code generation across different
platforms which significantly reduces the effort to maintain the toolchain and adapt it to new targets.

The major refinement steps of developing a SMARTSOFT component are depicted in figure 25. The component builder focuses on modeling the component hull which comprises, for example, service ports and tasks - without any implementation details in mind. Due to the stable interface to the user code, algorithms and libraries can be integrated independently of any middleware structures. The component hull can be generated for different platforms but provides both, a stable interface towards the user code (inner view of the component) and a stable interface towards the other components (outer view of component). The user parts keep the same and can be reused independently of the implementation technology hidden by the component hull.
6 Results and Conclusion

Engineering the software development process in robotics is one of the basic necessities towards industrial-strength robot systems. The benefits of model-driven software development for robotics are manifold. In the long term perspective, one can get rid of hand-crafted single unit lab demonstrators and compose them out of standardized off-the-shelf components with explicited properties. Solutions expressed at a model level can be reused, modified and migrated to different implementation technologies. One can take advantage from the knowledge of software engineers that is encoded in code transformators. Furthermore, properties of components and properties of systems composed out of components can be verified (or at least conformance checks can be done). In particular, a model-based approach allows to address resource awareness and gives the perspective towards addressing non-functional properties like safety.

The SMARTSOFT approach and the SMARTMDSD-toolchain have been used to build numerous components needed for service robotics scenarios (CORBA/SmartSoft Components, 2010). The components and the toolchain are available on sourceforge (http://smart-robotics.sf.net).

A typical scenario build by the presented approach is the cleanup the table scenario shown in figure 26.
Figure 25: Refinement steps of the MDSD-based component development.

Figure 26: The cleanup the table scenario.

(and also available on YouTube (Robotics@HS Ulm Youtube Channel, 2010)). The robot moves to the person and asks via speech output what should be done next. The robot can be commanded via speech input to cleanup the table. It then moves to the table, scans the table, detects objects and grasps them. The cleanup task is performed by throwing the cups into the trashbin. The candle is not known to the robot and is thus just considered as an obstacle for the manipulator path planning. The objects can be placed arbitrarily on the table and can be cleaned up as long as they are reachable by the manipulator. Persons are allowed within the scenario and can interact with the robot, block passages etc. The deployment diagram of that complex overall setting is shown in figure 27. Most of the components are also used within a Robocup@Home-scenario (Robocup@Home, 2010).

As of today, the SMARTSOFT communication patterns have been used continuously for more than 10 years within all kinds of robotic projects and even outside robotics (Thirde et al., 2006). The latest release is called ACE/SMARTSOFT (ACE/SmartSoft, 2009; Schlegel & Lotz, 2010) and runs on Windows, Linux (x86, PowerPC) and QNX (x86, PowerPC). Although the communication patterns have been mapped onto various operating systems and different middleware systems meanwhile, the user interface has been stable up to now without requiring any modifications. This gives strong evidence that the structuring and granularity of the communication patterns is in line with basic and stable structures of complex software systems.

The SMARTSOFT component model first has been based on the communication patterns and has been extended next by a generic state automaton to handle the life cycle of a component in a standardized way (Schlegel et al., 2011). Currently, it is being extended by monitoring and diagnosis capabilities (Lotz, 2010). Again, these structures are based on many successful implementations and use-cases and can therefore be considered as best
practices.

Hence, we made the step towards representing the stable structures as meta-models and independently of implementational technologies. This provided the foundations to exploit model-driven software development for robotics. It proved to be essential that the SMARTSOFT component model is strict with respect to the component hull but provides freedom inside a component. This allows reuse of existing libraries and legacy code while benefitting from system level architectural support of the component model.

Providing a model-driven software development toolchain significantly boosted the development of robotics applications in our lab and in our projects (http://www.zafh-servicerobotik.de) in several ways:

- Application builders have been able to easily and successfully compose systems out of prebuild robotics software components by a deployment diagram. This has been substantially exploited by students for Robocup@home scenarios. They reuse existing components in new configurations and compose them for the different challenges.

- Algorithm developers have been able to specify and generate component hulls, integrate their libraries and use their component in combination with already existing components and services within an extremely short time without requiring a tedious learning curve on how to deal with the component model, its software implementation etc. A component hull can be specified and generated from scratch within one hour.

- The reuse of existing libraries (even of complex ones like OpenRave (Diankov, 2010)) proved to be simple since there is a strict separation between the library and the component hull. The component hull provides a mean to successfully exploit the tremendous code-base in robotics and make them collaborate as components.

The model-driven software development process is seen as a suitable approach towards the overall vision of a robotics software component shelf. It allows to explicate component properties that are required for system level composition without the need to give insights into their internal implementation. This is essential towards system
composability while at the same time protecting intellectual property. Component-based and tool-supported composition of systems is seen as enabler towards wide-spread use of robotics technology. The current situation in software for robotics can be compared with the early times of the web where computer engineers designed content. The World Wide Web turned into a universal medium by the availability of tools which allow domain experts to provide content without bothering with the technical details. A model-driven software development approach for robotics can make robotics technology accessible to domain experts without requiring them to become a robotics expert in order to boost the wide-spread use of robotics technology.

References


CORBA/SmartSoft Components (2010).
http://smart-robotics.sf.net/corbaSmartSoft/components.php.


Robotics@HS Ulm Youtube Channel (2010). http://www.youtube.com/user/RoboticsAtHsUlm.


