23

Robotic Software Systems: From Code-Driven to Model-Driven Software Development

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

1. Introduction

Advances in robotics and cognitive sciences have stimulated expectations for emergence of new generations of robotic devices that interact and cooperate with people in ordinary human environments (robot companion, elder care, home health care), that seamlessly integrate themselves into complex environments (domestic, outdoor, public spaces), that fit into different levels of system hierarchies (human-robot co-working, hyper-flexible production cells, cognitive factory), that can fulfill different tasks (multi-purpose systems) and that are able to adapt themselves to different situations and changing conditions (dynamic environments, varying availability and accessibility of internal and external resources, coordination and collaboration with other agents).

Unfortunately, so far, steady improvements in specific robot abilities and robot hardware have not been matched by corresponding robot performance in real-world environments. On the one hand, simple robotic devices for tasks such as cleaning floors and cutting the grass have met with growing commercial success. Robustness and single purpose design is the key quality factor of these simple systems. At the same time, more sophisticated robotic devices such as *Care-O-Bot 3* (Reiser et al., 2009) and *PR2* (Willow Garage, 2011) have not yet met commercial success. Hardware and software complexity is their distinguishing factor.

Advanced robotic systems are systems of systems and their complexity is tremendous. Complex means they are built by integrating an increasingly larger body of heterogeneous (robotics, cognitive, computational, algorithmic) resources. The need for these resources arises from the overwhelming number of different situations an advanced robot is faced with during execution of multitude tasks. Despite the expended effort, even sophisticated systems are still not able to perform at an expected and appropriate level of overall quality of service in complex scenarios in real-world environments. By quality of service we mean the set of system level non-functional properties that a robotic system should exhibit to appropriately operate in an open-ended environment, such as robustness to exceptional situations, performance despite of limited resources and aliveness for long periods of time.

Since vital functions of advanced robotic systems are provided by software and software dominance is still growing, the above challenges of system complexity are closely related to the need of mastering software complexity. Mastering software complexity becomes pivotal towards exploiting the capabilities of advanced robotic components and algorithms. Tailoring modern approaches of software engineering to the needs of robotics is seen as decisive towards significant progress in system integration for advanced robotic systems.

2. Software engineering in robotics

Complex systems are rarely built from scratch but their design is typically partitioned according to the variety of technological concerns. In robotics, these are among others mechanics, sensors and actuators, control and algorithms, computational infrastructure and software systems. In general, successful engineering of complex systems heavily relies on the *divide and conquer* principle in order to reduce complexity. Successful markets typically come up with precise role assignments for participants and stakeholders ranging from component developers over system integrators and experts of an application domain to business consultants and end-users.

Sensors, actuators, computers and mechanical parts are readily available as commercial off-the-shelf black-box components with precisely specified characteristics. They can be re-used in different systems and they are provided by various dedicated suppliers. In contrast, most robotics software systems are still based on proprietarily designed software architectures. Very often, robotics software is tightly bound to specific robot hardware, processing platforms, or communication infrastructures. In addition, assumptions and constraints about tasks, operational environments, and robotic hardware are hidden and hard-coded in the software implementation.

Software for robotics is typically embedded, concurrent, real-time, distributed, data-intensive and must meet specific requirements, such as safety, reliability and fault-tolerance. From this point of view, software requirements of advanced robots are similar to those of software systems in other domains, such as avionics, automotive, factory automation, telecommunication and even large scale information systems. In these domains, modern software engineering principles are rigorously applied to separate roles and responsibilities in order to cope with the overall system complexity.

In robotics, tremendous code-bases (libraries, middleware, etc.) coexist without being interoperable and each tool has attributes that favors its use. Although 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 and second source suppliers, up to now this is not possible. Typically, experts for application domains need to become experts for robotics software to make use of robotics technology in their domain. So far, robotics software systems even do not enforce separation of roles for component developers and system integrators.

The current situation in software for robotics is caused by the lack of separation of concerns. In consequence, role assignments for robotics software are not possible, there is nothing like a software component market for robotic systems, there is no separation between component developers and system integrators and even no separation between experts in robotics and experts in application domains. This is seen as a major and serious obstacle towards developing a market of advanced robotic systems (for example, all kinds of cognitive robots, companion systems, service robots).

The current situation in software for robotics can be compared with the early times of the *World Wide Web (WWW)* where one had to be a computer engineer to setup web pages. The *WWW* turned into a universal medium only since the availability of tools which have made it accessible and which support separation of concerns: domain experts like journalists can now easily provide content without bothering with technical details and there is a variety of specialized, competing and interoperable tools available provided by computer engineers, designers and others. These can be used to provide and access any kind of content and to support any kind of application domain.

Based on these observations, we assume that the next big step in advanced robotic systems towards mastering their complexity and their overall integration into any kind of environment and systems depends on separation of concerns. Since software plays a pivotal role in advanced robotic systems, we illustrate how to tailor a service-oriented component-based software approach to robotics, how to support it by a model-driven approach and according tools and how this allows separation of concerns which so far is not yet addressed appropriately in robotics software systems.

Experienced software engineers should get insights into the specifics of robotics and should better understand what is in the robotics community needed and expected from the software engineering community. *Experienced roboticists* should get detailed insights into how model-driven software development *(MDSD)* and its design abstraction is an approach towards system-level complexity handling and towards decoupling of robotics knowledge from implementational technologies. *Practitioners* should get insights into how separation of concerns in robotics is supported by a service-oriented component-based software approach and that according tools are already matured enough to make life easier for developers of robotics software and system integrators. *Experts in application domains* and *business consultants* should gain insights into maturity levels of robotic software systems and according approaches under a short-term, medium-term and long-term perspective. *Students* should understand how design abstraction as recurrent principle of computer science applied to software systems results in *MDSD*, how *MDSD* can be applied to robotics, how it provides a perspective to overcome the vicious circle of robotics software starting from scratch again and again and how software engineering and robotics can cross-fertilize each other.

2.1 Separation of concerns

Separation of concerns is one of the most fundamental principles in software engineering (Chris, 1989; Dijkstra, 1976; Parnas, 1972). It states that a given problem involves different kinds of concerns, promotes their identification and separation in order to solve them separately without requiring detailed knowledge of the other parts, and finally combining them into one result. It is a general problem solving strategy which breaks the problem complexity into loosely-coupled subproblems. The solutions to the subproblems can be composed relatively easily to yield a solution to the original problem (Mili et al., 2004). This allows to cope with complexity and thereby achieving the required engineering quality factors such as robustness, adaptability, maintainability, and reusability.

Despite a common agreement on the necessity of the application of the separation of concerns principle, there is not a well-established understanding of the notion of concern. Indeed, *concern* can be thought of as a unit of modularity (Blogspot, 2008). Progress towards separation of concerns is typically achieved through modularity of programming and encapsulation (or *transparency* of operation), with the help of information hiding. Advanced uses of this principle allow for simultaneous decomposition according to multiple kinds of (overlapping and interacting) concerns (Tarr et al., 2000).

In practice, the principle of separation of concerns should drive the identification of the right decomposition or modularization of a problem. Obviously, there are both: (i) generic and domain-independent patterns of how to decompose and modularize certain problems in a suitable way as well as (ii) patterns driven by domain-specific best practices and use-cases.

In most engineering approaches as well as in robotics, at least the following are dominant dimensions of concerns which should be kept apart (Björkelund et al., 2011; Radestock & Eisenbach, 1996):

Computation provides the functionality of an entity and can be implemented in different ways (software and/or hardware). Computation activities require communication to access required data and to provide computed results to other entities.

Communication exchanges data between entities (ranging from hardware devices to interfaces for real-world access over software entities to user interfaces etc.).

Configuration comprises the binding of configurable parameters of individual entities. It also comprises the binding of configurable parameters at a system level like, for example, connections between entities.

Coordination is about when is something being done. It determines how the activities of all entities in a system should work together. It relates to orchestration and resource management.

According to (Björkelund et al., 2011), this is in line with results published in (Delamer & Lastra, 2007; Gelernter & Carriero, 1992; Lastra & Delamer, 2006) although variations exist which split *configuration* (into *connection* and *configuration*) or treat *configuration* and *coordination* in the same way (Andrade et al., 2002; Bruyninckx, 2011).
It is important to recognize that there are cross-cutting concerns like *quality of service (QoS)* that have instantiations within the above dimensions of concerns. Facets of *QoS for computation* can manifest with respect to time (best effort computation, hard real-time computation) or anytime algorithms (explicated relationship between assigned computing resources and achieved quality of result). Facets of *QoS for communication* are, for example, response times, latencies and bandwidth.
It is also important to recognize that various concerns need to be addressed at different stages of the lifecycle of a system and by different stakeholders. For example, *configuration* is part of the *design phase* (a component developer provides dedicated configurable parameters, a system integrator binds some of them for deployment) *and* of the *runtime phase* (the task coordination mechanism of a robot modifies parameter settings and changes the connections between entities according to the current situation and task to fulfill).
It is perfectly safe to say that robotics should take advantage from insights and successful approaches for complexity handling readily available in other but similar domains like, for example, automotive and avionics industry or embedded systems in general. Instead, robotics often reinvents the wheel instead of exploiting cross-fertilization between robotics and communities like software engineering and middleware systems. The interesting question is whether there are differences in robotics compared to other domains which hinder roboticists from jumping onto already existing and approved solutions. One should also examine whether or not these solutions are tailorable to robotics needs.

2.2 Specifics in robotics

The difference of robotics compared to other domains like automotive and avionics is neither the huge variety of different sensors and actuators nor the number of different disciplines being involved nor the diversity of hardware-platforms and software-platforms. In many domains, developers need to deal with heterogeneous hardware devices and are obliged to deploy their software on computers which are often constrained in terms of memory and computational power.
We are convinced that differences of robotics compared to other domains originate from the need of a robot to cope with open-ended environments while having only limited resources at its disposal.

Limited resources require decisions: when to assign which resources to what activity taking into account perceived situation, current context and tasks to be fulfilled. Finding adequate solutions for this major challenge of engineering robotic systems is difficult for two reasons:

- the *problem space* is huge: as uncertainty of the environment and the number and type of resources available to a robot increase, the definition of the best matching between current situation and correct robot resource exploitation becomes an overwhelming endeavour even for the most skilled robot engineer,
- the *solution space* is huge: in order to enhance overall quality of service like robustness of complex robotic systems in real-world environments, robotic system engineers should master highly heterogeneous technologies, need to integrate them in a consistent and effective way and need to adequately exploit the huge variety of robotic-specific resources.

In consequence, it is impossible to statically assign resources in advance in such a way that all potential situations arising at runtime are properly covered. Due to open-ended real-world environments, there will always be a deviation between *design-time optimality* and *runtime optimality* with respect to resource assignments. Therefore, there is a need for dynamic resource assignments at runtime which arises from the enormous sizes of the problem space and the solution space.
For example, a robot designer cannot foresee how crowded an elevator will be. Thus, a robot will need to decide by its own and at runtime whether it is possible and convenient to exploit the elevator resource. The robot has to trade the risk of hitting an elevator's user with the risk of arriving late at the next destination. To match the level of safety committed at design-time, the runtime trade-off has to come up with parameters for speed and safety margins whose risk is within the design-time committed boundaries while still implementing the intent to enter the elevator.

2.2.1 Model-centric robotic systems

The above example illustrates why we have to think of engineering advanced robotic systems differently compared to other complex systems. A complex robotic system cannot be treated as design-time finalizable system. At runtime, system configurations need to be changeable according to current situation and context including prioritized assignments of resources to activities, (de)activations of components as well as changes to the wiring between components. At runtime, the robot has to analyze and to decide for the most appropriate configuration. For example, if the current processor load does not allow to run the navigation component at the highest level of quality, the component should be configured to a lower level of navigation quality. A reasonable option to prepare a component to cope with reduced resource assignments might be to reduce the maximum velocity of the robot in order to still guarantee the same level of navigation safety.
In consequence, we need to support design-time reasoning (at least by the system engineer) as well as runtime reasoning (by the robot itself) about both, the problem space and the solution space. This can be achieved by raising the level of abstraction at which relevant properties and characteristics of a robotics system are expressed. As for every engineering endeavour, this means to rely on the power of models and asks for an overall different design approach as illustrated in figure 1:

- The solution space can be managed by providing advanced design tools for robot software development to design reconfigurable and adaptive robotic systems. Different stakeholders involved in the development of a robotic system need the ability to formally

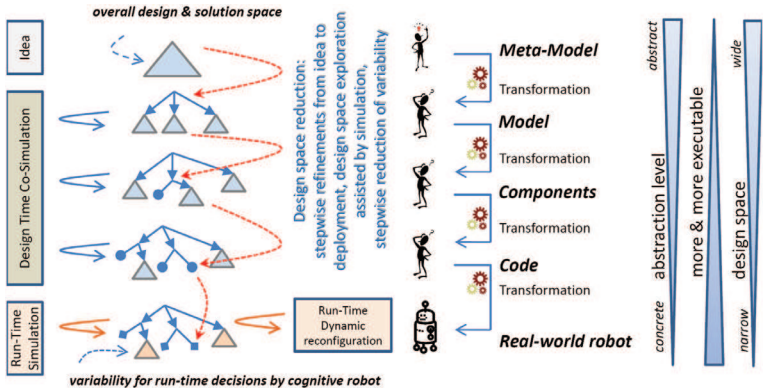


Fig. 1. Novel workflow bridging design-time and runtime model-usage: at design-time variation points are purposefully left open and allow for runtime decisions (Schlegel et al., 2010).

model and relate different views relevant to robotic system design. A major issue is the support of separation of concerns taking into account the specific needs of robotics.

- The problem space can be mastered by giving the robot the ability to reconfigure its internal structure and to adapt the way its resources are exploited according to its understanding of the current situation.

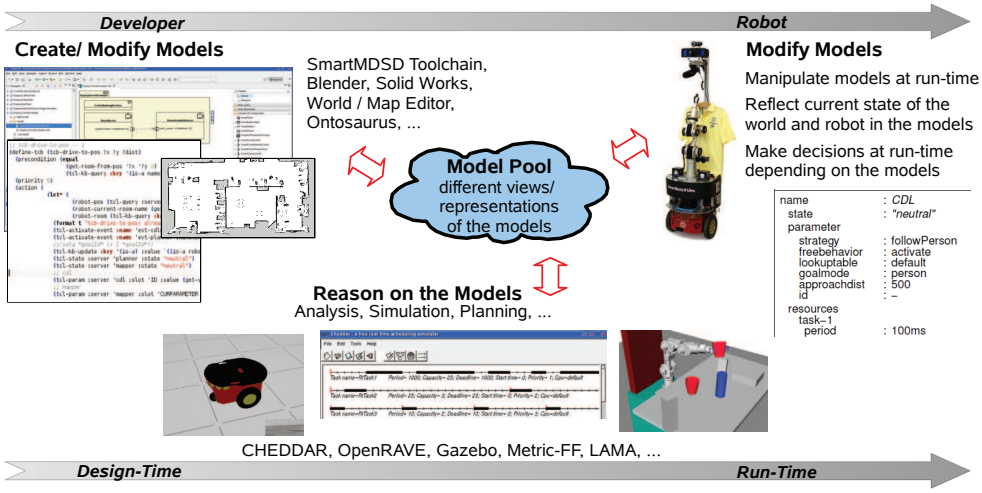


Fig. 2. Separation of concerns and design abstraction: models created at design-time are used and manipulated at runtime by the robot (Steck & Schlegel, 2011).

We coin the term *model-centric robotic systems* (Steck & Schlegel, 2011) for the new approach of using models to cover and support the whole life-cycle of robotic systems. Such a model-centric view puts models into focus and bridges design-time and runtime model-usage.

During the whole lifecycle, models are refined and enriched step-by-step until finally they become executable. Models comprise variation points which support alternative solutions. Some variation points are purposefully left open at design time and even can be bound earliest at runtime after a specific context and situation dependent information is available. In consequence, models need to be interpretable not only by a human designer but also by a computer program. At design-time, software tools should understand the models and support designers in their transformations. At runtime, adaptation algorithms should exploit the models to automatically reconfigure the control system according to the operational context (see figure 2).
The need to explicitly support the design for runtime adaptability adds robotic-specific requirements on software structures and software engineering processes, gives guidance on how to separate concerns in robotics and allows to understand where the robotics domain needs extended solutions compared to other and at first glance similar domains.

2.2.2 User roles and requirements

Another strong influence on robotic software systems besides technical challenges comes from the involved individuals and their needs. We can distinguish several user roles that all put a different focus on complexity management, on separation of concerns and on software engineering in robotics:

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, easy usage and reasonable value for money.

Application builders / system integrators assemble applications out of approved, standardized and reusable off-the-shelf components. Any non trivial robotic application requires the orchestration of several components such as computer vision, sensor fusion, human machine interaction, object recognition, manipulation, localization and mapping, control of multiple hardware devices, etc. Once these parts work together, we call it a *system*. This part of the development process is called, therefore, *system integration*. Components can be provided by different vendors. Application builders and system integrators consider components as black boxes and depend on precise specifications and explications of all relevant properties for smooth composition, resource assignments and mappings to target platforms. Components are customized during system level composition by adjusting parameters or filling in application dependent parts at so-called *hot spots* via plug-in interfaces. Application builders expect support for system-level engineering.

Component builders focus on the specification and implementation of a single component. They want to focus on algorithms and component functionality without being restricted too much with respect to component internals. They don't want to bother with integration issues and expect a framework to support their implementation efforts such that the resulting component is conformant to a system-level black box view.

Framework builders / tool providers prepare and provide tools that allow the different users to focus on their role. They implement the software frameworks and the domain-specific add-ons on top of state-of-the-art and standard software systems (like middleware systems), use latest software technology and make these available to the benefit of robotics.

The robotics community provides domain-specific concepts, best practices and design patterns of robotics. These are independent of any software technology and implementational technology. They form the *body of knowledge of robotics* and provide the domain-specific ground for the above roles.

The essence of the work of the *component builder* is to design reusable components which can seamlessly be integrated into multiple systems and different hardware platforms. A component is considered as a black box. The developer can achieve this abstraction only if he is strictly limited in his knowledge and assumptions about what happens outside his component and what happens inside other components.
On the other hand, the methodology and the purpose of the *system integrator* is opposite: he knows exactly the application of the software system, the platform where it will be deployed and its constraints. For this reason, he is able to take the right decision about the kind of components to be used, how to connect them together and how to configure their parameters and the quality of service of each of them to orchestrate their behavior. The work of the system integrator is rarely reusable by others, because it is intrinsically related to a specific hardware platform and a well-defined and sometimes unique use-case. We don't want the system integrator to modify a component or to understand the internal structure and implementation of the components he assembles.

2.2.3 Separation of roles from an industrial perspective

This distinction between the development of single components and system integration is important (figure 3). So far, reuse in robotics software is mainly possible at the level of libraries and/or complete frameworks which require system integrators to be component developers and vice versa. A formal separation between *component building* and *system integration* introduces another and intermediate level of abstraction for reuse which will make it possible to

- create commercial off-the-shelf (COTS) robotic software: when components become independent of any specific robot application, it becomes possible to integrate them quickly into different robotic systems. This abstraction allows the component developer to sell its robotic software component to a system integrator;
- overcome the need for the system integrator to be also an expert of robotic algorithms and software development. We want companies devoted to system integration (often SMEs) to take care of the Business-to-Client part of the value chain, but this will be possible only when their work will become less challenging;
- establish dedicated system integrators (specific to industrial branches and application domains) apart from experts for robotic components (like navigation, localization, object recognition, speech interaction, etc.);
- provide plug-and-play robotic hardware: so far the effort of the integration of the hardware into the platform was undertaken by the system integrator. If manufacturers start providing ready-to-use drivers which work seamlessly in a component-driven environment, robotic applications can be deployed faster and become cheaper.

This separation of roles will eventually have a positive impact in robotics: it will potentially allow the creation of a robotics industry, that is an ecosystem of small, medium and large enterprises which can profitably and symbiotically coexist to provide business-to-business

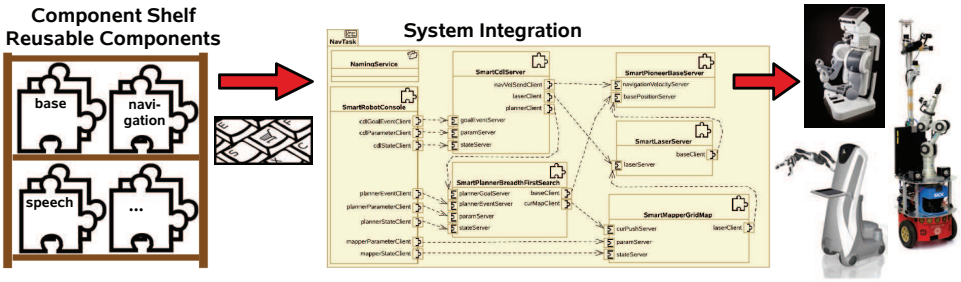


Fig. 3. Building robotic systems out of readily-available and reusable software components: separation of the roles of component development and system integration.

and business-to-client products such as hardware parts, software applications and complete robots with a specific application and deliver a valuable product to the customer.
To understand better what a *robotics industry* means, we draw an analogy to the personal computer industry. Apart from very few exceptions, we can identify several companies involved in the manufacturing of a single and very specific part of the final product: single hardware components (memories, hard drives, CPU, mother boards, screens, power supplies, graphic cards, etc.), operating systems (Windows, commercial Linux distributions), software applications (CAD, word processing, video games, etc.) and system integrators which provide ready-to-use platforms to the end user.

2.3 Service-oriented software components to master system complexity

Software engineering provides three major approaches that help to address the above challenges, that is *component-based software engineering (CBSE)*, *service-oriented architectures (SOA)* and *model-driven software development (MDSD)*.
CBSE separates the component development process from the system development process and aims at component reusability. *MDSD* separates domain knowledge (formally specified by domain experts) from how it is being implemented (defined by software experts using model transformations). *SOA* is about the right level of granularity for offering functionality and strictly separates service providers and consumers.

2.3.1 Component-based software engineering

CBSE (Heineman & Councill, 2001) is an approach that has arisen in the software engineering community in the last decade. It shifts the emphasis in system-building from traditional requirements analysis, system design and implementation to composing software systems from a mixture of reusable off-the-shelf and custom-built 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).

Software components explicitly consider reusable pieces of software including notions of independence and late composition. *CBSE* promises the benefits of increased reuse, reduced production cost, and shorter time to market. In order to realize these benefits, it is vital to have components that are easy to reuse and composition mechanisms that can be applied systematically. Composition can take place during different stages of the

lifecycle of components that is during the design phase (design and implementation), the deployment phase (system integration) and even the runtime phase (dynamic wiring of data flow according to situation and context). *CBSE* is based on the explication of all relevant information of a component to make it usable by other software elements whose authors are not known. The key properties of *encapsulation* and *composability* result in the following seven criteria that make a good component: "(i) may be used by other software elements (clients), (ii) may be used by clients without the intervention of the component's developers, (iii) includes a specification of all dependencies (hardware and software platform, versions, other components), (iv) includes a precise specification of the functionalities it offers, (v) is usable on the sole basis of that specification, (vi) is composable with other components, (vii) can be integrated into a system quickly and smoothly" (Meyer, 2000).

2.3.2 Service-oriented architectures

Another generally accepted view of a software component is that it is a software unit with *provided services* and *required services*. In component models, where components are architectural units, *services* are represented as *ports* (Lau & Wang, 2007). This view puts the focus on the question of a proper level of abstraction of offered functionalities. Services "combine information and behavior, hide the internal workings from outside intrusion and present a relatively simple interface to the rest of the program" (Sprott & Wilkes, 2004). The (CBDI Forum, 2011) recommends to define *service-oriented architectures (SOA)* as follows:

> **SOA** are "the policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface" (Sprott & Wilkes, 2004).

Service is the key to communication between providers and consumers and key properties of good service design are summarized as in table 1. *SOA* is all about style (policy, practice, frameworks) which makes process matters an essential consideration. A *SOA* has to ensure that services don't get reduced to the status of interfaces, rather they have an identity of their own. With *SOA*, it is critical to implement processes that ensure that there are at least two different and separate processes - for providers and consumers (Sprott & Wilkes, 2004).

reusable	use of service, not reuse by copying of code/implementation
abstracted	service is abstracted from the implementation
published	precise, published specification functionality of service interface, not implementation
formal	formal contract between endpoints places obligations on provider and consumer
relevant	functionality is presented at a granularity recognized by the user as a meaningful service

Table 1. Principles of good service design enabled by characteristics of *SOA* as formulated in (Sprott & Wilkes, 2004).

2.3.3 Model-driven software development

MDSD is a technology that introduces significant efficiencies and rigor to the theory and practice of software development. It provides a design abstraction as illustrated in figure

4. Abstractions are provided by models (Beydeda et al., 2005). Abstraction is a core 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).

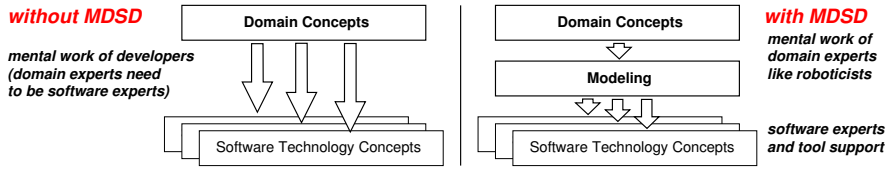


Fig. 4. Design abstraction of model-driven software development.

In *MDSD*, models are used for many purposes, including reasoning about problem and solution domains and documenting the stages of the software lifecycle; the result is improved software quality, improved time-to-value and reduced costs (IBM, 2006).

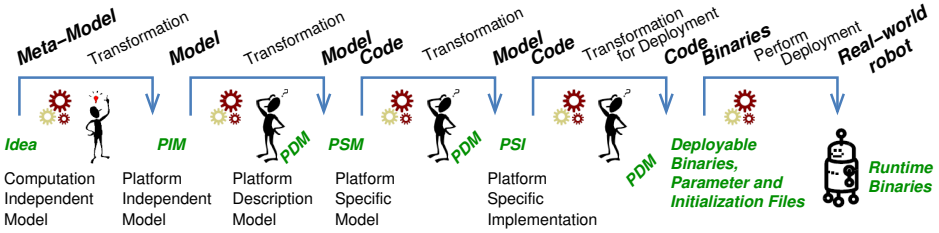


Fig. 5. Model-driven software development at a glance.

The standard workflow of a model-driven software development process is illustrated in figure 5. This workflow is supported by tools like the *Eclipse Modeling Project* (Eclipse Modeling Project, 2010) which 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 concepts to forward information from a model-level to the model transformations and code generators. Tools like *Papyrus* (PAPYRUS UML, 2011) allow for a graphical representation of the various models and can export them into the *XMI* format. Overall, there is a complete toolchain for graphical modelling and transformation steps available that can be tailored to the domain specific needs of robotics.

MDSD is much more than 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 & Völter, 2006; Völter, 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.

2.4 Stable structures and freedom from choice

In robotics, we believe that the cornerstone is a *component model* based on *service-orientation* for its provided and required interactions represented in an abstract way in form of *models*.
A *robotics component model* needs to provide *component level* as well as *system level* concepts, structures and building blocks to support separation of concerns while at the same time ensuring composability based on a composition theory: (i) building blocks out of which one composes a component, (ii) patterns of how to design a well-formed component to achieve system level conformance, (iii) guidance towards providing a suitable granularity of services, (iv) specification of the behavior of interactions and (v) best practices and solutions of domain-specific problems. *MDSD* can then provide toolchains and thereby support separation of concerns and separation of roles.
The above approach asks for the identification of *stable structures* versus *variation points* (Webber & Gomaa, 2004). A robotics component model has to provide guidance via stable structures where these are required to support separation of concerns and to ensure system level conformance. At the same time, it has to allow for freedom wherever possible. The distinction between stable structures and variation points is of relevance at all levels (operating system interfaces, library interfaces, component internal structures, provided and required services etc.). In fact, identified and enforced stable structures come along with restrictions. However, one has to notice that well thought out limitations are not a universal negative and *freedom from choice* (Lee & Seshia, 2011) gives guidance and assurance of properties beyond one's responsibilities in order to ensure separation of concerns.
As detailed in (Schlegel et al., 2011), stable structures with respect to a service-oriented component-based approach can be identified. These are illustrated in figure 6.

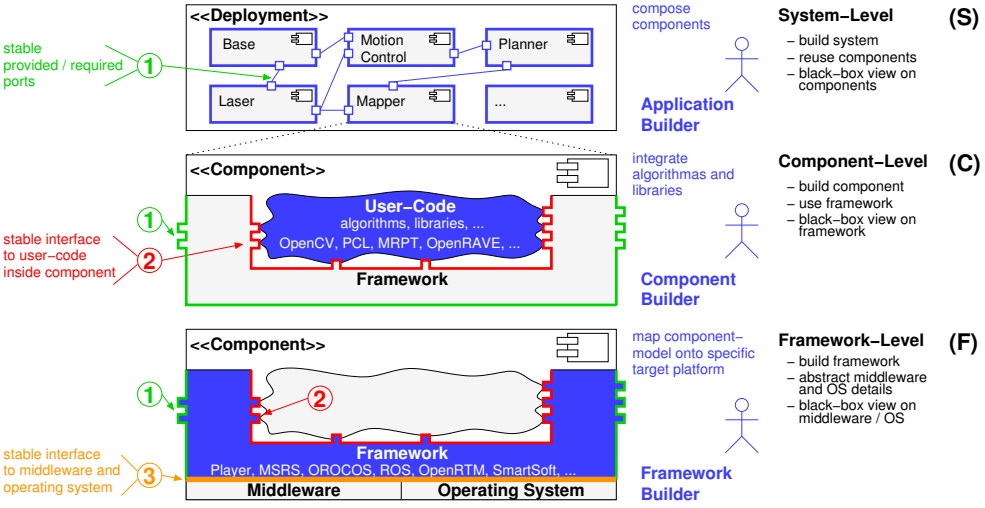


Fig. 6. Stable structures and different roles in a component-based software approach.

At the *system level (S)*, *provided* and *required* service ports ① 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 ②. 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 ② and (ii) between the framework and the underlying middleware & operating system ③. The stable interface ② ensures that no middleware and operating system specifics are unnecessarily passed on to the component builder. The stable interface ③ 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 ② and ③ and maps the framework onto different implementational technologies via the interface ③.

3. The SMARTSOFT-approach

The basic idea behind SMARTSOFT (Schlegel, 2011) is to master the component hull and thereby achieve *separation of concerns* as well as *separation of roles*. Figure 7 illustrates the SMARTSOFT component model and how its component model links the stable interfaces ①, ② and ③.

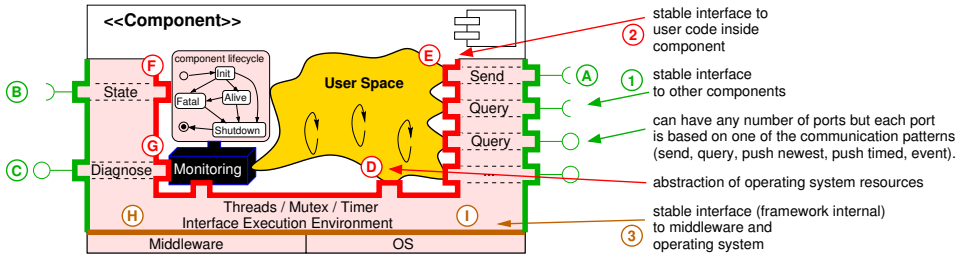


Fig. 7. The structure of a SMARTSOFT component and its stable interfaces.

Pattern	Description
send	one-way communication
query	two-way request
push newest	1-to-n distribution
push timed	1-to-n distribution
event	asynchronous notification

Service	Description
param	component configuration
state	activate/deactivate component services
wiring	dynamic component wiring
diagnose	introspection of components
(internally based on communication patterns)	

Table 2. The set of patterns and services of SMARTMARS.

The link between ① and ② is realized by *communication patterns*. Binding a communication pattern with the type of data to be transmitted results in an externally visible service represented as port. The small set of generic and predefined communication patterns listed in the left part of table 2 are the only ones to define externally visible services. Thus, the behavior and usage of a service is immediately evident as soon as one knows its underlying communication pattern.

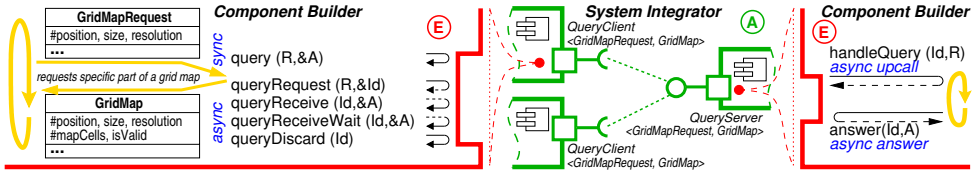


Fig. 8. The views of a component builder and a system integrator on services by the example of a grid map service based on a query communication pattern.

Figure 8 illustrates this concept by means of the *query* communication pattern which consists of a *query client* and a *query server*. The query pattern expects two *communication objects* to define a service: a request object and an answer object. Communication objects are transmitted *by-value* to ensure decoupling of the lifecycles of the client side and the server side of a service. They are arbitrary objects enriched by a unique identifier and get/set-methods. Hidden from the user and inside the communication patterns, the content of a communication object provided via *E* gets extracted and forwarded to the middleware interface *H*. Incoming content at *H* is put into a new instance of the according communication object before providing access to it via *E*.

In the example, the system integrator sees a provided port based on a *query server* with the communication objects *GridMapRequest* and *GridMap*. The map service might be provided by a map building component. Each component with a port consisting out of a *query client* with the same communication objects can use that service. For example, a path planning component might need a grid map and expose a required port for that service. The *GridMapRequest* object provides the parameters of the individual request (for example, the size of the requested map patch, its origin and resolution) and the *GridMap* returns the answer. The answer is self-contained comprising all the parameters describing the provided map. That allows to interpret the map independently of the current settings of the service providing component and gives the service provider the chance to return a map as close to but different from the requested parameters in case he cannot handle them exactly.

A component builder uses the stable interface *E*. In case of the client side of a service based on the query pattern, it always consists of the same synchronous as well as asynchronous access modes independent from the used communication objects and the underlying middleware. They can be used from any number of threads in any order. The server side in this example always consists of an asynchronous handler upcall for incoming requests and a separated answer method. This separation is important since it does not require the upcall to wait until the answer is available before returning. We can now implement any kind of processing model inside a component, even a processing pipeline where the last thread calls the answer method, without blocking or wasting system resources of the upcall or be obliged to live with the threading models behind the upcall.

In the example, the upcall at the service provider either directly processes the incoming *GridMapRequest* object or forwards it to a separate processing thread. The requested map patch is put into a *GridMap* object which then is provided as answer via the *answer* method.

It can be seen that the client side is not just a proxy for the server side. Both sides of a communication pattern are completely standalone entities providing stable interfaces *A* and *E* by completely hiding all the specifics of *H* and *I* (see figure 7). One can neither expose arbitrary member functions at the outside component hull nor can one dilute the semantics and behavior of ports. The different communication patterns and their internals are explained in detail in (Schlegel, 2007).

Besides the services defined by the component builder (*A*), several predefined services exist to support system level concerns (Lotz et al., 2011). Each component needs to provide a *state service* to support system level orchestration (outside view *B*: activation, deactivation, reconfiguration; inside view *F*: manage transitions between service activations, support housekeeping activities by entry/exit actions). An optional *diagnostic service* (*C, G*) supports runtime monitoring of the component. The optional *param service* manages parameters by name/value-pairs and allows to change them at runtime. The optional *wiring service* allows to wire required services of a component at runtime from outside the component. This is needed for task and context dependent composition of behaviors.

3.1 The SMARTMARS meta-model

All the stable interfaces, concepts and structures as well as knowledge about which ingredients and structures form a well-formed SMARTSOFT component and a well-formed system of SMARTSOFT components are explicated in the SMARTMARS meta-model (figure 9). The meta-model is abstract, universally valid and independent from implementation technologies (e.g. UML profile (Fuentes-Fernández & Vallecillo-Moreno, 2004), eCore (Gronback, 2009)). It provides the input for tool support for the different roles (like component developer, system integrator etc.), explicates separation of concerns and can be mapped onto different software technologies (e.g. different types of middleware like CORBA, ACE (Schmidt, 2011) and different types of operating systems).

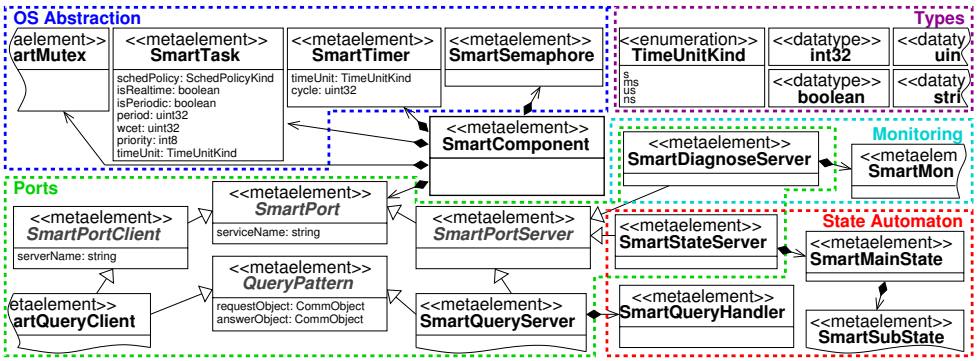


Fig. 9. Excerpt of the SMARTMARS meta-model.

3.2 Policies and strategies behind SMARTSOFT services

A major part of the SMARTSOFT approach are the policies and strategies that manifest themselves in the structure of the component model, explain its building blocks and guide their usage.

Separation of the roles of a component developer and a system integrator requires to control the interface between the inner part of a component and its outer part and to control this boundary. As soon as one gains control over the component hull, one can make sure that all relevant properties and parameters needed for the black box view of the system integrator become explicated at the component hull. One can also make sure that a component developer has no chance to expose component internals to the outside. SMARTSOFT achieves this via predefined communication patterns as the only building blocks to define externally visible services and by further guidelines on how to build good services.

A basic principle is that clients of services are not allowed to make any assumptions about offered services beyond the announced characteristics and that service providers are not allowed to make any assumptions about service requestors (like e.g. their maximum rate of requests).

This principle results in simple and precise guidelines of how to apply the communication patterns in order to come up with well-formed services. As long as a service is being offered, the service provider has to accept all incoming requests and has to respond to them according to its announced quality-of-service parameters.

We illustrate this principle by means of the *query pattern*. As long as there are no further quality-of-service attributes, the service provider accepts all incoming requests and guarantees to answer all accepted requests. However, only the service provider knows about its resources available to process incoming requests and clients are not allowed to impose constraints on the service provider (a request might provide further non-committal hints to the service provider like a request priority). Thus, the service provider is allowed to provide a nil answer (the flag *is valid* is set to false in the answer) in case he is running out of resources to answer a particular request. In consequence, all service requestors always must be prepared to get a nil answer. A service requestor is also not allowed to make any assumptions about the response time as long as according quality-of-service attributes are not set by the service provider. However, if a service provider announces to answer requests within a certain time limit, one can rely on getting at least a nil answer before the deadline. If a service requestor depends on a maximum response time although this quality-of-service attribute is not offered by the service provider, he needs to use client-side timeouts with his request. This overall principle ensures (i) loose coupling of services, (ii) prevents clients from imposing constraints on service providers and (iii) gives service providers the means to arbitrate requests in case of limited resources.

It now also becomes evident why SMARTSOFT offers more than just a request/response and a publish/subscribe pattern which would be sufficient to cover all communicational needs. The *send* pattern explicates a one-way communication although one can emulate it via a query pattern with a void answer object. However, practical experience proved that a much better clarity for services with this characteristic is achieved when offering a separate pattern. The same holds true for the *push newest* and the *push timed* pattern. In principle, the push timed pattern is a push newest pattern with a regular update. However, in case of a push newest pattern, service requestors rely on having the latest data available at any time. This is different from a push timed pattern where the focus is on the service provider guaranteeing a regular time interval (in some cases even providing the same data). Although one could cover some of these aspects by quality-of-service attributes, they also have an impact on the kind of perception of its usage by a component developer. Again, achieving clarity and making the characteristics easily recognizable is of particular importance for the strict separation of the roles of component developers and system integrators. This also becomes obvious with the *event* pattern. In contrast to the push patterns, service requestors get informed only in case a server side event predicate (service requestors individually parametrize each event activation) becomes true. This tremendously saves bandwidth compared to publishing latest changes to all clients since one then always would have to publish a snapshot of the overall context needed to evaluate the predicate at the client side instead of just the information when an event fired.

3.3 A robotics example illustrating the SMARTSOFT concepts

Figure 10 illustrates how the SMARTSOFT component model and its meta-elements provided by SMARTMARS structure and partition a typical robotics use-case, namely the navigation of a mobile platform. Besides access to sensor data and to the mobile base, algorithmic building blocks of a navigation system are map building, path planning, motion execution and self localization. Since these building blocks are generic for navigation systems independently of the used algorithms, it makes sense to come up with an according component structure and services (or expect readily available components and services).

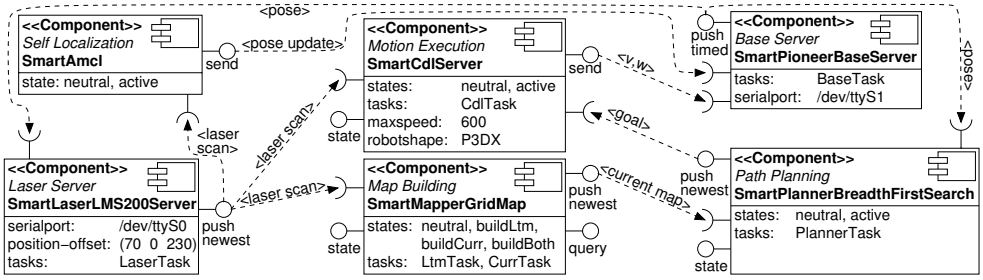


Fig. 10. Structure of a navigation task based on the SMARTSOFT component model.

The SmartLaserLMS200Server component provides the latest laser scan via a push newest port. Thus, all subscribed clients always get an update as soon as a new laserscan is available. It is subscribed to the pose service of the robot base to label laser scans with pose stamps. The component comprises a SmartTask to handle the internal communication with the laser hardware. This way, the aliveness of the overall component and its services is not affected by flaws on the laser hardware interface. Parameters like position-offset and serialport are used to customize the component to the target robotic system. These parameters have to be set by the application builder during the deployment step. The SmartMapperGridMap component requires a laser scan to build the longterm and the current map. The current map is provided by a push newest server port (as soon as a new map is available, it is provided to subscribed clients which makes sense since path planning depends on latest maps) and the longterm map by a query server port (since it is not needed regularly, it makes sense to provide it only on a per-request basis). The state port is used to set the component into different states depending on which services are needed in the current situation: build no map at all (neutral), build the current map only (buildCurr), build the longterm map only (buildLtm) or build both maps (buildBoth). The push newest server publishes the current map only in the states buildCurr and buildBoth. Requests for a longterm map are answered as long as the component and its services are alive but with an invalid map in case it is in the states neutral or buildCurr (valid flag of answer object set to false). Accordingly, the SmartPlannerBreadthFirstSearch component provides its intermediate waypoints by a push newest server (update the motion execution component as soon as new information is available). The motion execution component regularly commands new velocities to the robot base via a send service. The motion execution component is also subscribed to the laser scan service to be able to immediately react to obstacles in dynamic environments. This way, the different services interact to build various control loops to combine goal directed and reactive navigation while at the same time allowing for replacement of components.

3.4 State-of-the-art and related work

The historical need in robotics to be responsible for creation of the application logic and to be at the same time the system integrator generated a poor understanding in the robotics community that these two roles ought to be separated. In consequence, most robotics frameworks don't make this distinction and consequently they don't offer any clear guideline to the developer on how to achieve separation of roles.

For example, *ROS* (Quigley et al., 2009) is a currently widely-used framework in robotics providing a huge and valuable codebase. However, it lacks guidance for component developers to ensure system level conformance for composability. Instead, its focus is on side-by-side existence of all kinds of overlapping concepts without an abstract representation of its core features and properties in a way independent of any implementation.

The only approach in line with the presented concepts is the *RTC Specification* (OMG, 2008) which is considered the most advanced concept of *MDSD* in robotics. However, it is strongly influenced by use-cases requiring a data-flow architecture and they do not yet considerably take into account requirements imposed by runtime adaptability.

4. Reference implementation of the SMARTMDSD TOOLCHAIN

The reference implementation of the SMARTMDSD TOOLCHAIN implements the SMARTMARS meta-model within a particular MDSD-toolchain. It is used in real world operation to develop components and to compose complex systems out of them. The focus of this section is on technical details of the implementation of a meta-model. Another focus is on the role-specific view and the support a MDSD-toolchain provides. We illustrate the reference implementation of the toolchain along the different roles of the stakeholders and their views on the toolchain.

4.1 Decisions and tools behind the reference implementation - framework builder view

The reference implementation of our SMARTMDSD TOOLCHAIN is based on the *Eclipse Modeling Project (EMP)* (Eclipse Modeling Project, 2010) and *Papyrus UML* (PAPYRUS UML, 2011).

Papyrus UML is used as graphical modeling tool in our toolchain. Therefore, it is customized by the framework builder for the development of SMARTSOFT Components (component builder) and deployments of components (application builder). This includes for example a customized wizard to create communication objects, components as well as deployments. The modeling view of *Papyrus UML* is enriched with a customized set of meta-elements to create the models. The model transformation and code generation steps are developed with *Xpand* and *Xtend* (Efftinge et al., 2008) which are part of the *EMP*. These internals are not visible to the component builder and the application builder. They just see the graphical modeling tool to create their models and use the *CDT Eclipse Plugin* (Eclipse CDT, 2011) to extend the source code and to compile binaries. The SMARTMARS meta-model is implemented as a *UML Profile* (Fuentes-Fernández & Vallecillo-Moreno, 2004) using *Papyrus UML*.

The decision to use *UML Profiles* and *Papyrus UML* to implement our toolchain is motivated by the reduced effort to come up with a graphical modeling environment customized to the robotics domain and its requirements by reusing available tools from other communities. Although some shortcomings have to be accepted and taken into account we were not caught in the huge effort related to implementing a full-fledged *GMF*-based development environment. This allowed us to early come up with our toolchain and to gain deeper insights

and more experience on the different levels of abstraction. However, the major drawbacks of *UML Profiles* are:

- *UML* is a general purpose modeling language covering aspects of several domains and is thus complex. Using profiles, it is only possible to enrich *UML*, but not to remove elements.
- Deployment and instantiations of components are not adequately supported.
- *UML Profiles* provide just a lightweight extension of *UML*. That means, the structure of *UML* itself cannot be modified. The elements can be customized only by stereotypes and tagged values.

To counter the drawbacks of *UML Profiles*, we only support the usage of the stereotyped elements provided by SMARTMARS to create the models of the components and deployments. Directly using pure *UML* elements in the diagrams is not supported. Thus, the models are created using just the meta-elements provided by SMARTMARS. Restricting the usage to SMARTMARS meta-elements, a mapping to another meta-model implementation technology like *eCore* (Gronback, 2009) is straightforward. The stereotyped elements can be mapped onto *eCore* without taking into account *UML* and its structure. In the current implementation of our toolchain, the restriction to only use SMARTMARS meta-elements is enforced with *check* (Efftinge et al., 2008), the *EMP* implementation of *OCL* (Object Management Group, 2010). In the model transformation and code generation steps of our toolchain pure *UML* elements are ignored. Another approach would be to customize the diagrams by removing the *UML* elements from the palette (see fig. 12) and thus restricting their usage. The latter approach is on the agenda of the *Papyrus UML* project and will be supported by future releases.

4.2 Development of components – component builder view

Figure 11 illustrates the roles of the framework builder and the component builder. The component builder creates a model of the component using the Eclipse based toolchain, focusing on the component hull. Pushing the button he receives the source files where to integrate the business logic (algorithms, libraries) of the component. During this process the component builder is supported and guided by the toolchain. The internals of the model transformation and code generation steps implemented by the framework builder are not visible to the component builder.

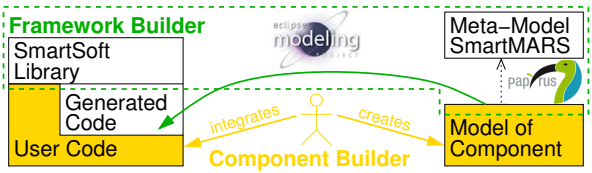


Fig. 11. The component builder models a component, gets the source code of its overall structure (component hull, tasks, etc.) generated by the toolchain and can then integrate user-code into these structures.

The view of the component builder on the toolchain is depicted in figure 12. It is illustrated by a face recognition component which is a building block in many service robotics scenarios as part of the human-robot interface (detection, identification and memorization of persons). In its active state, the component shall receive camera images, apply face recognition algorithms and report detected and recognized persons. Thus, besides the standard ports for setting

states (active, neutral) and parameters, we need to specify a port to receive the latest camera images (based on a push newest client) and another one to report on the results (based on an event server). The component shall run the face recognition based on a commercially available library within one thread and optional visualization mechanisms within a second and separated thread. Thus, we need to specify two tasks within the component.

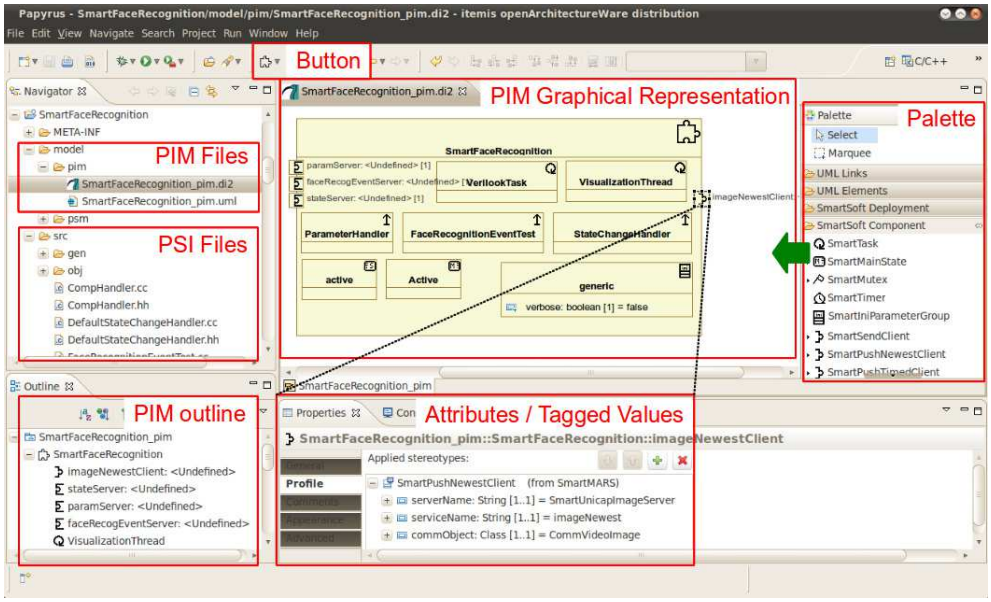


Fig. 12. Screenshot of our toolchain showing the view of the Component Builder.

To create the model the component builder uses the SMARTMARS meta-elements offered in the *palette*. The elements of the created model can be accessed either in the outline view or directly in the graphical representation. Several of the meta-element attributes (tagged values) can be customized and modified in the properties tab (e.g. customizing services to ports, specifying properties of tasks, etc.). The model is stored in files specific to *Papyrus UML*. Pushing the button, the workflow is started and the *PSI* (source) files are generated. The user code files are directly accessible in the *src* folder. The component builder integrates his business logic into these files (in our example, the interaction with the face recognition library). The generated files the component builder must not modify are stored in the *gen* folder. These files are generated and overwritten each time the workflow is executed. For the further processing of the source files, the *Eclipse CDT plugin* is used (*Makefile Project*). The makefile is also generated by the workflow specific to the model properties. User modifications in the makefile can be done inside of *protected regions* (Gronback, 2009).

4.3 Development of components – framework builder view

Taking a look behind the scenes of the toolchain, the workflow (fig. 13) appears as a two step transformation according to the *OMG MDA* (Object Management Group & Soley, 2000). The *Platform Independent Model (PIM)*, which is created by the component builder using the meta-elements provided by the *PIM UML Profile*, specifies the component independently of the implementation technology. The first step in the workflow is the model-to-model

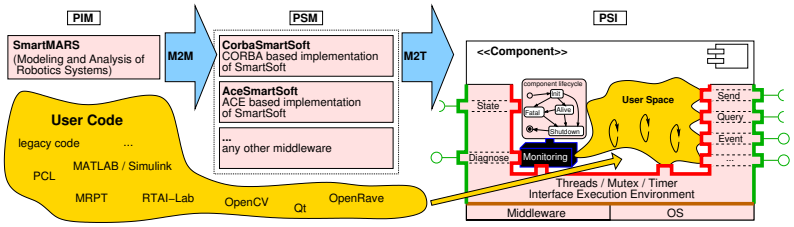


Fig. 13. Two step transformation workflow: Framework Builder view.

(*M2M*) transformation (encoded with *Xtend*) from the *PIM* into a *Platform Specific Model (PSM)*. In this step the elements of the *PIM* are transformed into corresponding elements of the *PSM* according to the selected target platform. The second step is the model-to-text (*M2T*) transformation (encoded with *Xpand* and *Xtend*) from the *PSM* into a *Platform Specific Implementation (PSI)*. This transformation is based on customizable code templates.

4.3.1 The SmartMARS UML profiles (PIM/PSM)

The abstract SMARTMARS meta-model is implemented by the framework builder as *UML Profile* using *Papyrus UML*. Therefore, standard *UML* elements (e.g. *Component*, *Class*, *Port*) are extended by stereotypes (e.g. SMARTCOMPONENT, SMARTTASK, SMARTQUERYSERVER) to give the meta-elements a new meaning according to the SMARTMARS concept. To distinguish and highlight the new element, it has its own icon attached. Tagged values are used to enrich the meta-element by new attributes which are not provided by the base *UML* element.
In fact there are two *UML Profiles*: one for the *PIM* and one for the *PSM*. The *PIM UML Profile* is visible to the component builder and is used by him to create the models of the components. For each SMARTSOFT implementation (e.g. *CORBA*, *ACE*), a *PSM UML Profile* has to be provided covering the specifics of the implementation. For example, the *CORBA*-based *PSM* supports *RTAI* linux to provide hard realtime tasks. This is represented by the meta-element *RTAITask*. The *PSM UML Profile* is not visible to the component builder and only used by the transformation steps inside the toolchain.

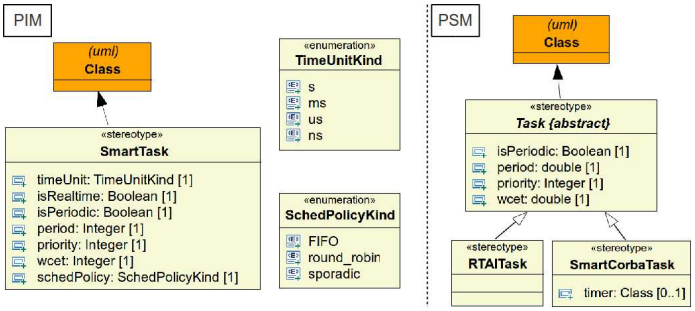


Fig. 14. Screenshots of excerpt of the UML Profiles created with *Papyrus UML* showing the metaelements dedicated to the SMARTTASK. *Left: PIM; Right: PSM* with the two variants (1) standard task and (2) *RTAI* task.

An excerpt of the *UML Profiles* is illustrated in figure 14. In the *UML Profile* for the *PIM*, the SMARTTASK extends the *UML class* and enriches it with attributes (tagged values) like *isPeriodic*, *isRealtime*, *period* and *timeUnit*. For the *timeUnit* an enumeration (*TimeUnitKind*)

is used to specify the unit in which time values are annotated. In the *UML Profile* for the *CORBA-based PSM*, an abstract task is specified (cannot be instantiated) and the two variants (1) standard task and (2) realtime task are derived from it. They are both not abstract and can thus be instantiated by the component builder to create the model. The standard task adds an optional attribute referencing to a SMARTTIMER meta-element. This is used to emulate periodic non-realtime tasks which are not natively supported by standard tasks of the *CORBA*-based SMARTSOFT implementation.

4.3.2 Model transformation and code generation steps

The *M2M* transformation maps the platform independent elements of the *PIM* onto platform specific elements of the selected target platform. Such a mapping is illustrated by the example of the *SmartTask* (fig. 15 *left*) and the *CORBA*-based *PSM*. The SMARTTASK comprises several elements which are necessary to describe a task behavior and its characteristics.

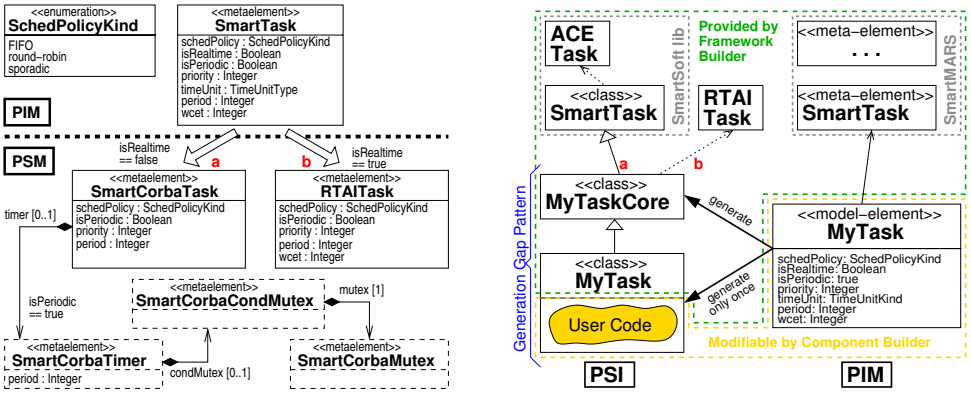


Fig. 15. Model transformation and code generation steps illustrated by the example of the SMARTTASK. *Left:* Transformation of the PIM into a PSM. *Right:* Code generation and Generation Gap Pattern.

task_mutex.ext

```
create uml::Class this addSmartTask(SmartMARS::SmartTask tsk, uml::Component cmp) :
    cmp.packagedElement.add(this) ->
    this.setName(tsk.name) ->
    if( tsk.isRealtime == true) then
    {
        this.applyStereotype("CorbaSmartSoft::RTAITask") ->
        setTaggedValue(this, "CorbaSmartSoft::RTAITask", "isPeriodic", tsk.isPeriodic) ->
        setTaggedValue(this, "CorbaSmartSoft::RTAITask", "wcet", tsk.wcet.toSecond(tsk.timeUnit.name)) ->
        setTaggedValue(this, "CorbaSmartSoft::RTAITask", "period", tsk.period.toSecond(tsk.timeUnit.name)) ->
        setTaggedValue(this, "CorbaSmartSoft::RTAITask", "priority", tsk.priority)
    }
    else
    {
        this.applyStereotype("CorbaSmartSoft::SmartCorbaTask") ->
        setTaggedValue(this, "CorbaSmartSoft::SmartCorbaTask", "isPeriodic", tsk.isPeriodic) ->
        setTaggedValue(this, "CorbaSmartSoft::SmartCorbaTask", "wcet", tsk.wcet.toSecond(tsk.timeUnit.name)) ->
        setTaggedValue(this, "CorbaSmartSoft::SmartCorbaTask", "period", tsk.period.toSecond(tsk.timeUnit.name)) ->
        setTaggedValue(this, "CorbaSmartSoft::SmartCorbaTask", "priority", tsk.priority) ->
        if( tsk.isPeriodic == true ) then
        {
            setTaggedValue(this, "CorbaSmartSoft::SmartCorbaTask", "timer", cmp.addTimer(tsk.name, tsk.period, tsk.timeUnit.name))
        }
    };
```

Fig. 16. PIM to PSM model transformation of the SMARTTASK depending on the attribute *isRealtime*.

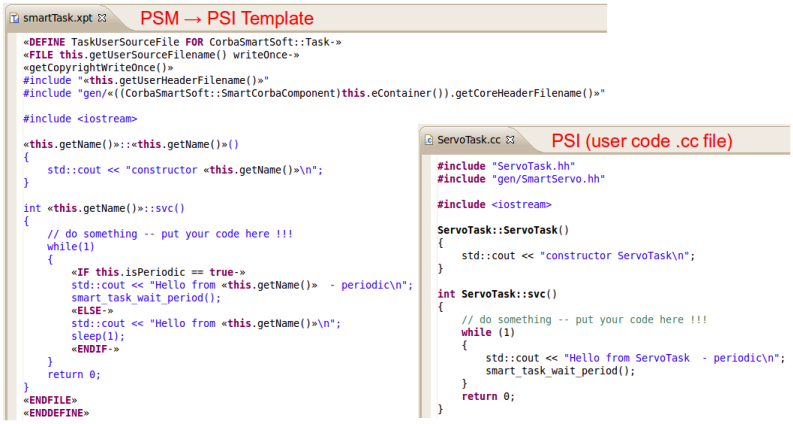


Fig. 17. *PSM* to *PSI* transformation of the SMARTTASK. *Left:* Excerpt of the transformation template (*xPand*) generating the PSI of a standard task. *Right:* The generated code where the user adds the business logic of the task.

Depending on the attribute *isRealtime* the SMARTTASK is either mapped onto a RTAITASK or a non-realtime SMARTCORBATASK[1]. The *Xtend* transformation rule to transform the *PIM* SMARTTASK into the appropriate *PSM* element is depicted in figure 16.
In case the attributes specify a non-realtime, periodic SMARTTASK, the toolchain extends the *PSM* by the elements needed to emulate periodic tasks (as this feature is not covered by standard tasks). In each case the user integrates his algorithms and libraries into the stable interface provided by the SMARTTASK (component builder view) independent of the hidden internal mapping of the SMARTTASK (generated code). Figure 17 depicts the *Xpand* template to generate the user code file for the task in the *PSI*. The figure shows the template on the left and the generated code on the right.
The *PSI* consists of the SMARTSOFT library, the generated code and the user code (fig. 15 *right*). To be able to re-generate parts of the component source code according to modified parameters in the model without affecting the source code parts added by the component builder, the generation gap pattern (Vlissides, 2009) is used. It is based on inheritance – the user code inherits from the generated code[2]. The source files called *generated code* are generated each time the transformation workflow in the toolchain is executed. These files contain the logic which is generated behind the scenes according to the model parameters and must not be modified by the component builder. The source files called *user code* are just generated if they do not already exist. They are intended for the component builder to add the algorithms and libraries. The generation of the user code files is more for the convenience of the component builder to have a code template as starting point. These files are in the full responsibility of the component builder and are never modified or overwritten by the transformation workflow of the toolchain. In this context *generate once* means that the file is only generated if it does not already exist. This is typically the case if the workflow is executed for the first time. The clear separation of generated code and user code by the generation gap pattern allows on the one hand to reflect modifications of the model in the generated source

[1] *Corba* in element names indicates that the element belongs to the *CORBA* specific *PSM*.
[2] The pattern could also be used in the opposite inheritance ordering so that the generated code inherits from the user code.

code without overwriting the user parts. On the other hand it gives the user the freedom to structure his source code according to his needs and does not restrict the structure as would be the case with, for example, *protected regions*. Consequently, the component builder can modify the *period*, *priority* or even the *isRealtime* attribute of the task in the model, re-generate and compile the code without requiring any modification in the user code files. The modification in the model just affects the generated code part of the *PSI*.

4.4 Deployment of components – application builder view

The deployment is used to compose a robotic system out of available components. The application builder imports the desired components and places them onto the target platform. Furthermore, he defines the initial wiring of the components by connecting the ports with the meta-element *Connection*. Figure 18 illustrates the composition of navigation

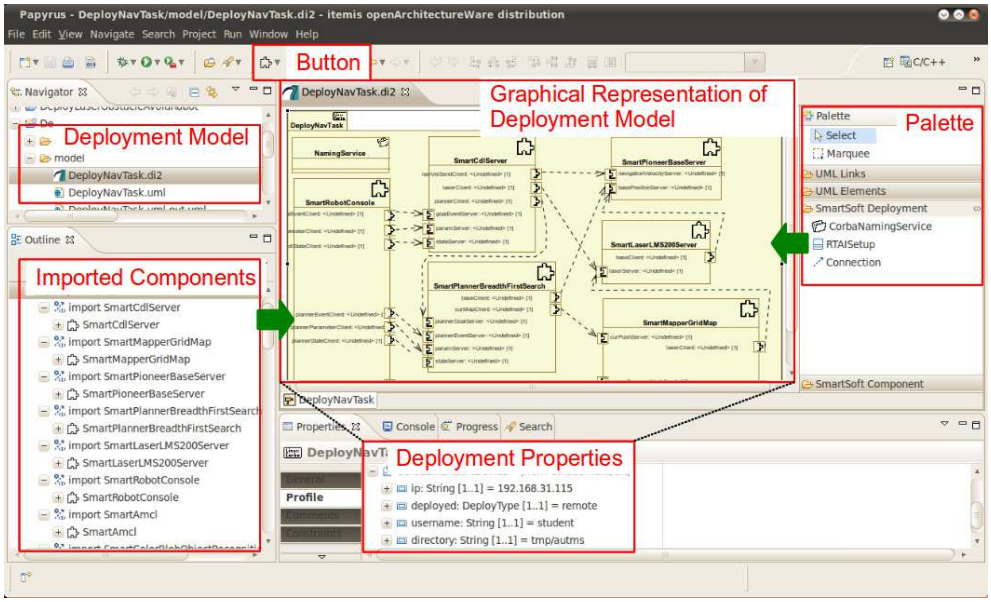


Fig. 18. Screenshot of our toolchain showing the deployment of components to build a robotic system.

components. In this example, the application builder (system integrator) imports components specific to a particular robot platform (SmartPioneerBaseServer) and specific to a particular sensor (SmartLaserLMS200Server). The navigation components (SmartMapperGridMap, SmartPlannerBreadthFirstSearch, SmartCDLServer) can be used across different mobile robots. The SmartRobotConsole provides a user interface to command the robot.

The components are presented to the application builder as black boxes with dedicated variation points. These have to be bound during the deployment step and can be specified according to system level requirements. For example, a laser ranger component might need the coordinates of its mounting point relative to the robot coordinate system. One might also reduce the maximum scanning frequency to save computing resources. Parameters also need to be bound for the target system. For example, in case *RTAI* is used inside of a component, the *RTAI* scheduler parameters (timer model underlying RTAI: periodic, oneshot) of the target

RTAI system have to be specified. If the application builder forgets to bind required settings, this absence is reported to him by the toolchain.
The application builder can identify the provided and required services of a component via its ports. He can inspect its characteristics by clicking on the port icon which opens a property view. That comprises the communication pattern type, the used communication objects and further characteristics like service name and also port specific information like update frequencies. The initial wiring is done within the graphical representation of the model. In case the application builder wants to connect incompatible ports, the toolchain refuses the connection and gives further hints on the reasons.
If the *CORBA*-based implementation of SMARTSOFT is used, the *CORBA* naming service properties *IP*-address and *port*-number have to be set. Furthermore, the deployment type (*local*, *remote*) has to be selected. For a remote deployment, the *IP*-address, *username* and *target folder* of the target computer have to be specified. The deployed system is copied to the target computer and can be executed there. In case of a local deployment, the system is customized to run on the local machine of the application builder. This is, for example, the case if no real robot is used and the deployed system uses simulation components (e.g. *Gazebo*). Depending on the initial wiring, parameter files are generated and also copied into the deployment folder. These parameter files contain application specific adjustments of the components. In addition, a shell script to start the system is generated out of the deployment model.

4.5 Deployment of components – framework builder view

To implement the deployment of components, some meta-elements are added by the framework builder to the *UML Profile* (fig. 19). This section focuses on the *CORBA*-based deployment.

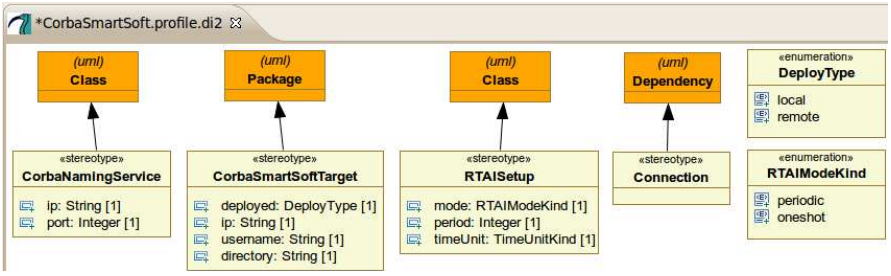


Fig. 19. Meta-elements to support the deployment of components.

The deployment model contains relevant information like the initial wiring between components (*Connection*), naming service properties (*CorbaNamingService*), scheduler properties (*RTAISetup*) and parameters about the deployment itself (*CorbaSmartSoftTarget*). The models of the components are made available to the deployment model using the *UML import* mechanism. This allows to access the internal structure of the components. Out of the deployment model the parameter files and a start script are generated (*M2T*) using *Xpand* and *Xtend* in a similar way as these transformation languages are used to generate code for the components. Based on the deployment model several analysis and simulation models can be generated to get feedback from 3rd-party tools. For example, one can extract parameters of all realtime tasks mapped onto a specific processor to perform hard realtime schedulability analysis (CHEDDAR (Cheddar, 2010)) (Schlegel et al., 2010).

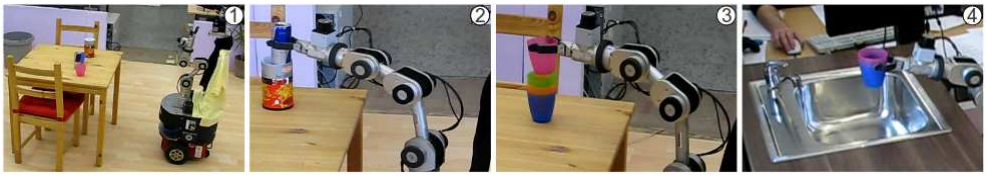


Fig. 20. The clean-up scenario. (1) Kate approaches the table; (2/3) Kate stacks objects into each other; (4) Kate throws cups into the kitchen sink.

As deployments and especially instantiations of components are not sufficiently supported by *UML*, a few workarounds are necessary as long as the SMARTMARS meta-model is implemented as *UML Profile*. For example, a robot with two laser range finders (front, rear) requires two instances of the same component. Each laser instance requires its individual parameters (e.g. serial port, pose on robot). These parameters are assigned to the deployment model by the application builder specifically for each component. In the implementation based on the *UML Profile*, we hence work on copies of components. Individual instances with their own parameter sets are considered in the abstract SMARTMARS meta-model and are also covered in the SMARTSOFT implementation, thus switching to a different meta-model implementation technology would allow for instances. This has not yet been done due to the huge manpower needed compared to just reusing *UML* tools.

5. Example / scenario

The work presented has been used to build and run several real-world scenarios, including the participation at the RoboCup@Home challenge. Among other tasks our robot "Kate" can follow persons, deliver drinks, recognize persons and objects and interact with humans by gestures and speech.

In the clean-up scenario [3] (fig. 20) the robot approaches a table, recognizes the objects which are placed on the table and cleans the table either by throwing the objects into the trash bin or into the kitchen sink. There are different objects, like cups, beverage cans and different types of crisp cans. The cups can be stacked into each other and have to be thrown into the kitchen sink. Beverage cans can be stacked into crisp cans and have to be thrown into the trash bin. Depending on the type of crisp can, one or two beverage cans can be stacked into one crisp can. After throwing some of the objects into the correct disposal the robot has to decide whether to drive back to the table to clean up the remaining objects (if existing) or to drive to the operator and announce the result of the cleaning task. The robot reports whether all objects on the table could be cleaned up or, in case any problems occurred, reports how many objects are still left.

Such complex and different scenarios can neither be developed from scratch nor can their overall system complexity be handled without using appropriate software engineering methods. Due to their overall complexity and richness, they are considered as convincing stress test for the proposed approach. In the following the development of the cleanup example scenario is illustrated according to the different roles.

The **framework builder** provides the tools to develop SMARTSOFT components as well as to perform deployments of components to build a robotic system. In the described example this includes the *CORBA*-based implementation of the SMARTSOFT framework

[3] http://www.youtube.com/roboticsathsulm#p/u/0/xtLK-655v7k

and the SMARTMDSD toolchain which are both available on *Sourceforge* (`http://smart-robotics.sourceforge.net`).
The component builder view of the SMARTMDSD toolchain supports **component builders** to develop their components independently of each other, but based on agreed interfaces. These components are independent of the concrete implementation technology of SMARTSOFT. Component builders provide their components in a component shelf. The models of the components include all information to allow a black-box view of the components (e.g. services, properties, resources). The explication of such information about the components is required by the application builder to compose robotic systems in a systematic way. To orchestrate the components at run-time, the task coordination language SMARTTCL (Steck & Schlegel, 2010) is used. Therefore, SMARTTCL is wrapped by a SMARTSOFT component and is also provided in the component shelf. The SMARTTCL component provides reusable action plots which can be composed and extended to form the desired behavior of the robot.
The **application builder** uses the application builder view of the SMARTMDSD toolchain. He composes already existing components to build the complete robotic system. In the above described cleanup scenario, 17 components (e.g. mapping, path planning, collision avoidance, laser ranger, robot-base) are reused from the component shelf. It is worth noting that the components were not particularly developed for the cleanup scenario, but can be used in the cleanup scenario due to the generic services they provide. The SMARTTCL sequencer component is customized according to the desired behavior of the cleanup scenario. Therefore, several of the already existing action plots can be reused. Application specific extensions are added by the application builder.
At run-time the SMARTTCL sequencer component coordinates the software components of the **robot** by modifying the configuration and parametrization as well as the wiring between the components. As SMARTTCL can access the information (e.g. parameters, resources) explicated in the models of the components at run-time, this information can be taken into account by the decision making process. That allows the robot not only to take the current situation and context into account, but also the configuration and resource usage of the components. In the described scenario, the sequencer manages the resources of the overall system, for example, by switching off components which are not required in the current situation. While the robot is manipulating objects on the table and requires all available computational resources for the trajectory planning of the manipulator, the components for navigation are switched off.

6. Conclusion

The service-oriented component-based software approach allows separation of roles and is an important step towards the overall vision of a robotics software component shelf. The feasibility of the overall approach has been demonstrated by an Eclipse-based toolchain and its application within complex Robocup@Home scenarios. Next steps towards model-centric robotic systems that comprehensively bridge design-time and runtime model usage now become viable.

7. References

Andrade, L., Fiadeiro, J. L., Gouveia, J. & Koutsoukos, G. (2002). Separating computation, coordination and configuration, *Journal of Software Maintenance* 14(5): 353–369.
Beydeda, S., Book, M. & Gruhn, V. (eds) (2005). *Model-Driven Software Development*, Springer.

Björkelund, A., Edström, L., Haage, M., Malec, J., Nilsson, K., Nugues, P., Robertz, S. G., Störkle, D., Blomdell, A., Johansson, R., Linderoth, M., Nilsson, A., Robertsson, A., Stolt, A. & Bruyninckx, H. (2011). On the integration of skilled robot motions for productivity in manufacturing, *Proc. IEEE Int. Symposium on Assembly and Manufacturing*, Tampere, Finland.

Blogspot (2008). Discussion of Aspect oriented programming(AOP).
URL: *http://programmingaspects.blogspot.com/*

Bruyninckx, H. (2011). Separation of Concerns: The 5Cs - Levels of Complexity, Lecture Notes, Embedded Control Systems.
URL: *http://people.mech.kuleuven.be/ bruyninc/ecs/LevelsOfComplexity-5C-20110223.pdf*

CBDI Forum (2011). CBDI Service Oriented Architecture Practice Portal - Independent Guidance for Service Architecture and Engineering.
URL: *http://everware-cbdi.com/cbdi-forum*

Cheddar (2010). A free real time scheduling analyzer.
URL: *http://beru.univ-brest.fr/ singhoff/cheddar/*

Chris, R. (1989). *Elements of functional programming*, Addison-Wesley Longman Publishing Co, Boston, MA.

Delamer, I. & Lastra, J. (2007). Loosely-coupled automation systems using device-level SOA, *5th IEEE International Conference on Industrial Informatics*, Vol. 2, pp. 743–748.

Dijkstra, E. (1976). *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ.

Eclipse CDT (2011). C/C++ Development Tooling for Eclipse.
URL: *http://www.eclipse.org/cdt/*

Eclipse Modeling Project (2010). Modeling framework and code generation facility.
URL: *http://www.eclipse.org/modeling/*

Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., Schönbach, P., Eysholdt, M. & Reinisch, S. (2008). openArchitectureWare User Guide 4.3.1.

Fuentes-Fernández, L. & Vallecillo-Moreno, A. (2004). An Introduction to UML Profiles, *UPGRADE* Volume V(2): 6–13.

Gelernter, D. & Carriero, N. (1992). Coordination languages and their significance, *Commun. ACM* 35(2): 97–107.

Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, Addison-Wesley, Upper Saddle River, NJ.

Heineman, G. T. & Councill, W. T. (eds) (2001). *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley Professional.

IBM (2006). Model-Driven Software Development, *Systems Journal* 45(3).

Lastra, J. L. M. & Delamer, I. M. (2006). Semantic web services in factory automation: Fundamental insights and research roadmap, *IEEE Trans. Ind. Informatics* 2: 1–11.

Lau, K.-K. & Wang, Z. (2007). Software component models, *IEEE Transactions on Software Engineering* 33: 709–724.

Lee, E. A. & Seshia, S. A. (2011). *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*, ISBN 978-0-557-70857-4.
URL: *http://LeeSeshia.org*

Lotz, A., Steck, A. & Schlegel, C. (2011). Runtime monitoring of robotics software components: Increasing robustness of service robotic systems, *Proc. 15th Int. Conference on Advanced Robotics (ICAR)*, Tallinn, Estland.

Meyer, B. (2000). What to compose, *Software Development* 8(3): 59, 71, 74–75.

Mili, H., Elkharraz, A. & Mcheick, H. (2004). Understanding separation of concerns, *Proc. Workshop Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, Vancouver, Canada, pp. 75–84.

Neelamkavil, F. (1987). *Computer simulation and modeling*, John Wiley & Sons Inc.

Object Management Group (2010). Object Constraint Language (OCL). URL: *http://www.omg.org/spec/OCL/*

Object Management Group & Soley, R. (2000). Model-Driven Architecture (MDA). URL: *http://www.omg.org/mda*

OMG (2008). Robotic Technology Component (RTC). URL: *http://www.omg.org/spec/RTC/*

PAPYRUS UML (2011). Graphical editing tool for uml. URL: *http://www.eclipse.org/modeling/mdt/papyrus/*

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules, *Communications of the ACM* 15(12).

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R. & Ng, A. (2009). ROS: An open-source Robot Operating System, *ICRA Workshop on Open Source Software*.

Radestock, M. & Eisenbach, S. (1996). Coordination in evolving systems, *Trends in Distributed Systems – CORBA and Beyond*, Springer-Verlag, pp. 162–176.

Reiser, U., Connette, C., Fischer, J., Kubacki, J., Bubeck, A., Weisshardt, F., Jacobs, T., Parlitz, C., Hägele, M. & Verl, A. (2009). Care-O-bot 3 – Creating a product vision for service robot applications by integrating design and technology, *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (ICRA)*, St. Louis, USA, pp. 1992–1997.

Schlegel, C. (2007). Communication patterns as key towards component interoperability, *in* D. Brugali (ed.), *Software Engineering for Experimental Robotics, STAR 30*, Springer, pp. 183–210.

Schlegel, C. (2011). SMARTSOFT – Components and Toolchain for Robotics. URL: *http://smart-robotics.sf.net/*

Schlegel, C., Steck, A., Brugali, D. & Knoll, A. (2010). Design abstraction and processes in robotics: From code-driven to model-driven engineering, *2nd Int. Conf. on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Springer LNAI 6472, pp. 324–335.

Schlegel, C., Steck, A. & Lotz, A. (2011). Model-driven software development in robotics: Communication patterns as key for a robotics component model, *Introduction to Modern Robotics*, iConcept Press.

Schmidt, D. (2011). The ADAPTIVE Communication Environment. URL: *http://www.cs.wustl.edu/ schmidt/ACE.html*

Sprott, D. & Wilkes, L. (2004). CBDI Forum. URL: *http://msdn.microsoft.com/en-us/library/aa480021.aspx*

Stahl, T. & Völter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management*, Wiley.

Steck, A. & Schlegel, C. (2010). SmartTCL: An Execution Language for Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots, *Int. Workshop on DYnamic languages for RObotic and Sensors systems (DYROS/SIMPAR)*, Germany, pp. 274–277.

Steck, A. & Schlegel, C. (2011). Managing execution variants in task coordination by exploiting design-time models at run-time, *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Francisco, CA.

Szyperski, C. (2002). *Component-Software: Beyond Object-Oriented Programming*, Addison-Wesley Professional, ISBN 0-201-74572-0, Boston.

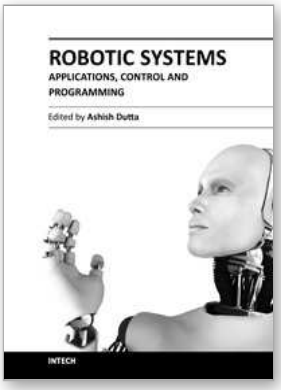
Tarr, P., Harrison, W., Finkelstein, A., Nuseibeh, B. & Perry, D. (eds) (2000). *Proc. of the Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, Limerick, Ireland.

Vlissides, J. (2009). Pattern Hatching – Generation Gap Pattern.
URL: *http://researchweb.watson.ibm.com/designpatterns/pubs/gg.html*

Völter, M. (2006). MDSD Benefits - Technical and Economical.
URL: *http://www.voelter.de/data/presentations/mdsd-tutorial/02_Benefits.pdf*

Webber, D. L. & Gomaa, H. (2004). Modeling variability in software product lines with the variation point model, *Science of Computer Programming - Software Variability Management* 53(3): 305–331.

Willow Garage (2011). PR2: Robot platform for experimentation and innovation.
URL: *http://www.willowgarage.com/pages/pr2/overview*



Robotic Systems - Applications, Control and Programming
Edited by Dr. Ashish Dutta

ISBN 978-953-307-941-7
Hard cover, 628 pages
Publisher InTech
Published online 03, February, 2012
Published in print edition February, 2012

This book brings together some of the latest research in robot applications, control, modeling, sensors and algorithms. Consisting of three main sections, the first section of the book has a focus on robotic surgery, rehabilitation, self-assembly, while the second section offers an insight into the area of control with discussions on exoskeleton control and robot learning among others. The third section is on vision and ultrasonic sensors which is followed by a series of chapters which include a focus on the programming of intelligent service robots and systems adaptations.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Christian Schlegel, Andreas Steck and Alex Lotz (2012). Robotic Software Systems: From Code-Driven to Model-Driven Software Development, Robotic Systems - Applications, Control and Programming, Dr. Ashish Dutta (Ed.), ISBN: 978-953-307-941-7, InTech, Available from: http://www.intechopen.com/books/robotic-systems-applications-control-and-programming/robotic-software-systems-from-code-driven-to-model-driven-software-development

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2012 The Author(s). Licensee IntechOpen. This is an open access article distributed under the terms of the Creative Commons Attribution 3.0 License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.