Service Robot Control Architectures for Flexible and Robust Real-World Task Execution: Best Practices and Patterns

Matthias Lutz, Dennis Stampfer, Alex Lotz, Christian Schlegel

Fakultät Informatik
Hochschule Ulm
Prittwitzstrasse 10
89075 Ulm
{lutz, stampfer, lotz, schlegel}@hs-ulm.de

Abstract: Successfully building complex service robotic systems that robustly operate in real-world environments gives a lot of insights into what are valuable patterns in architecting such systems. In this paper, we describe some of our insights with respect to robot control architectures and robotics software systems engineering. The focus of the paper is not on discussing specific functionalities but to explain some best practices that evolved out of our experience in building complex software intensive robotic systems. These behind-the-scenes insights are in particular relevant in moving forward towards a robotics business ecosystem with separation of roles and based on model-driven approaches for handling systems of systems.

1 Introduction and Motivation

Robotic systems are complex, software intensive and heterogeneous composite systems. Robotics as a science of integration depends on structures that guide the overall system design, the system integration and that even support run-time adaptation according to the executed task, the current context and the available resources. Such structures called architectural principles for robotics do not appear out of nowhere but should reflect and explicate practices and techniques developed and matured over many years of experience within all kinds of projects and with involvement of all kinds of stakeholders.

We consider architectural principles good if they provide partitioning schemes that organize different and various views on a robotics system such that one can effectively and efficiently cope with the complexity of the whole lifecycle of such a system (design, composition, deployment, operation, evolution, adaptation etc.).

There are partitioning schemes that are generally considered a good engineering practice and that are independent of the robotics domain. For example, *separation of concerns* [Chr89][Dij76] [Par72][BEH+11][RE96] is one of the most fundamental principles in software engineering and plays a vital role in robotics software systems as well [SSL12b][VKB14]. It aims at identifying orthogonal concerns in order to address com-

plexity by decoupling. It should drive the identification of the right decomposition of a problem. *Separation of roles* [SSL12b] is another partitioning scheme that proved to be a very successful approach for organizing business ecosystems where stakeholders have mutual benefits in collaborating and competing and where they share efforts and risks. Nevertheless, there is still the need to *identify* and to *explicate* behind-the-scenes insights related to achieving *separation of concerns* and of *separation of roles* in robotics.

The focus of the paper is not on discussing specific functionalities but to explain some best practices that evolved out of our experience in building complex software intensive robotic systems [Ulma] [Ulmb] (figure 1).

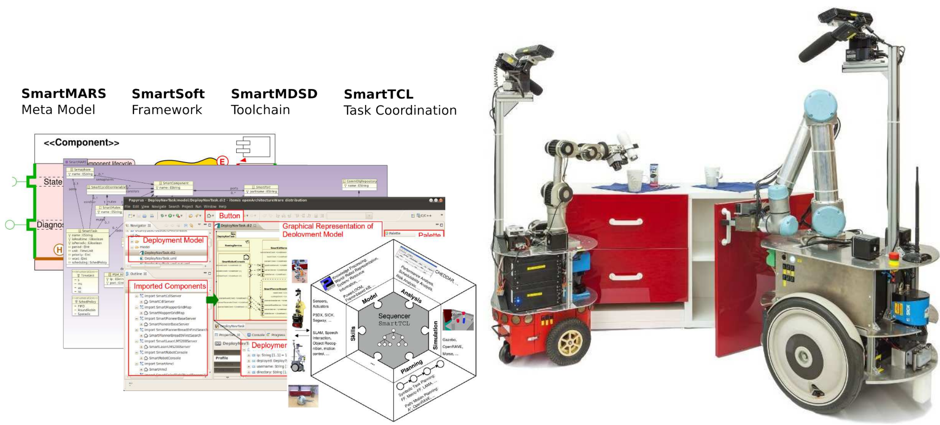

Figure 1: SMARTSOFT toolchain (left), service robots Kate and Larry (right)

2 Freedom from Choice vs. Freedom of Choice

We consider a generic robotics reference architecture that is refined in a top-down approach to end up with more and more specific architectures for dedicated robotic systems as illusive and unrewarding. A specific robotic control architecture always is a trade-off between various conflicting requirements. Thus, different co-existing architectural principles and their implementations represent different trade-offs of the interplay between e.g. modularity, adaptability, performance, dependability, resources etc. For a lawn-mower robot in a high-volume market, it might make perfect sense to loose extensibility and modularity in its software in order to save memory and processing power. However, it might also make sense to preserve some modularity in order to be able to support e.g. customization by user-downloadable apps.

The interesting question is what kind of architectural patterns form the sweet spot in supporting *separation of concerns* and *separation of roles*. That immediately comes along with the question of how much structure is needed for which aspects and is beneficial in which settings and how to support conformance to these structures.

One approach is called *freedom of choice*. One tries to support as many different schemes as possible and then leaves it to the user to decide which one best fits his needs. However, that requires huge expertise and discipline at the user side in order to avoid mixing non-interoperable schemes. Typically, academia tends towards preferring this approach since it seems to be as open and flexible as possible. However, the price to pay is high since there is no guidance with respect to ensuring composability and system level conformance.

Freedom from choice gives clear guidance with respect to selected structures and can ensure composability and system level conformance. However, there is a high responsibility in coming up with the appropriate structures such that they do not block progress and future designs.

An appropriate sweet spot between *freedom of choice* and *freedom from choice* can only be found by first agreeing on paramount objectives that give guidance. In our view, these are the need for *separation of roles* and *separation of concerns*. In line with the subsidiarity principle, architectural patterns should impose structures only as far as these cannot be achieved at a more local level. For example, structures and patterns that ensure *composability* restrict freedom of choice in order to allow *separation of roles* between component developers and system integrators. Typically, industry tends towards *freedom from choice* by agreeing on standards that is the minimum set of structures required to establish a business ecosystem.

An example for *freedom of choice* is ROS. One of the main design rules the ROS founders follow is *"We do not wrap your main."* [CGCG10]. What they mean by this is, among others, that they do not want to enforce any architectural design decisions for developers using ROS. As consequence, with ROS every developer can use his own personally preferred architecture. It then is very likely that the implemented architecture is in conflict with those defined by other parties. As also stated in [CGCG10], this can lead to confusions as this inevitably leads to the need for every ROS user to first understand the architectural decisions of each individual component before being able to reuse them in their own systems. The proposed solution in [CGCG10] for this problem is just to extensively document each ROS component on the ROS web portal. However, this does not circumvent the need to extensively analyse and understand the source code in order to adjust it or to implement workarounds in order to somehow make components compatible and reusable.

It is obvious that ROS (according to its overall design philosophy) does not yet give enough structure in an appropriate format in order to better address *separation of roles* and *separation of concerns* as is needed for a robotics business ecosystem. The minimum needed structures are a sound component model (in order to separate at least the role of component developer and system integrator and still ensure composability). Furthermore, it has to be formalized for use in model-driven tools in order to support *separation of concerns* (e.g. to maintain semantics independent of the OS / middleware mapping) and to assist the different roles in conforming to superordinated structures. The component model and model-driven toolchains of SMARTSOFT are following the approach of *freedom from choice* and explicate those structures[SSL12b].

3 Best Practices in Designing System Architectures

The design of the overall system architecture is an important part of the development process. The challenge is to break down the overall system into smaller units that can be solved by the individual stakeholders / project partners. The goal is to enable them to focus on their individual parts of the system by providing building blocks (software components) which can later be composed and integrated to the final application and target system.

Figure 2 illustrates the workflow showing our best practice in developing service-oriented component based architectures and applications. In a first step, services are defined (1), then they are reused and aggregated to components in order to provide components and groups of components (2). The outcome is a set of building blocks out of which applications can be composed (3).

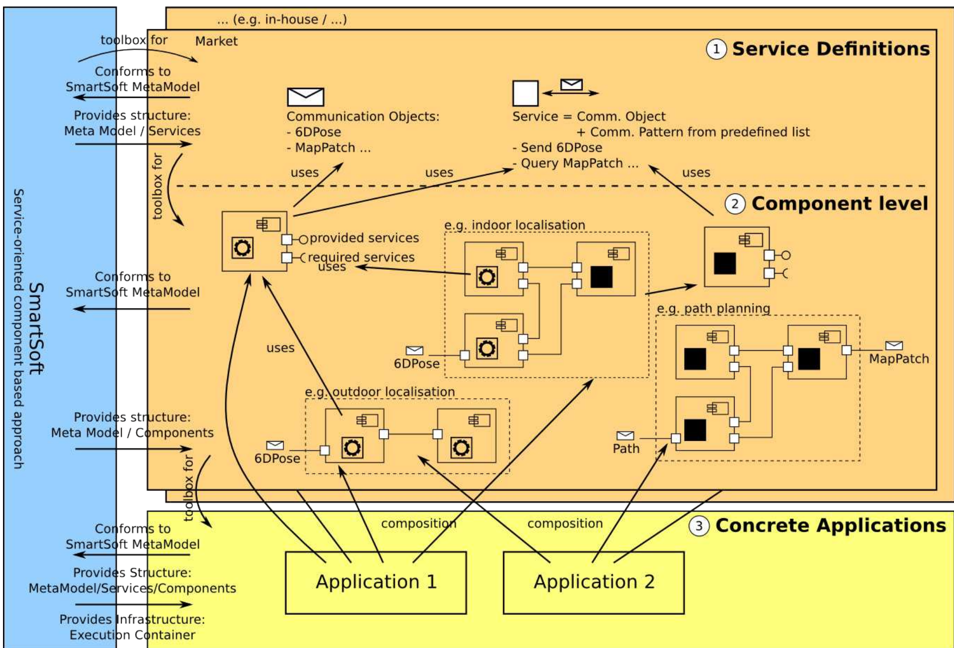

Figure 2: Best practices workflow for designing system architectures

The service-oriented component-based approach SMARTSOFT provides structure, infrastructure and tool support at all levels and steps (Figure 2, left). Amongst others, it includes:

- SmartMARS MetaModel. It defines the structure for communication objects, a set of communication patterns, the structure of services and components, the structure for deployment.
- SMARTSOFT Framework and implementation. In its current state, two exchangeable implementations (ACE- and CORBA-based middlewares). Execution containers for several platforms and operating systems.

- SMARTSOFT MDSD Toolchain. An integrated MDSD toolchain for development that supports the *separation of roles*. The toolchain covers the development process of modelling communication objects, components and systems.

The first level **Service Definitions** (Figure 2: (1)) uses the structure provided by SMARTSOFT to define *Services* that might be provided (made available) or required (relevant) in an application. Services are the basic architectural entities. They guarantee that supplied components can be integrated into concrete applications. At the same time, they keep the architecture and implementation containers flexible due to the service-level and component-level abstractions. Further, they allow to identify white spots in the architecture as early as possible (required services that no-one provides or provided services that no-one requires).

Services consist of communication objects [SSL12a] (data structure for communication: attributes, data types, access methods) and one selected communication pattern [SSL12a] (how a communication object is being communicated). A communication object together with one selected communication pattern (out of a set of communication patterns defined in SMARTSOFT) becomes a service.

For example, at this level the user defines what a ”location” for the robotics system actually is, i.e. whether it is represented as full 6D cartesian pose or with geographical position coordinates with or without uncertainty, orientation, etcetera. He can even decide on separate communication objects for both types of representations if required in the domain. It is also defined that e.g. a service providing regular location updates every 0.2s (push timed pattern for location communication object with period 0.2s) is required in the considered domain.

The second level **Component Level** focuses on aggregating services to components or groups of components (Figure 2: (2)). They provide or require the services from (1). There might be competing alternative components providing the same services from different suppliers with different characteristics. Both open source and closed source implementations can be used since one can rely on service descriptions. For example, the common denominator for the core functionality of localisation could have been defined in (1) as the communication object ”6DPose” (x, y, z, yaw, pitch, roll) that needs to be pushed to subscribers. Now there might be alternative components or groups of components for different localisation scenarios and accuracy requirements which consist of a different structure of components and are provided by 3rd party component suppliers in a market. These alternatives become alternatives only because they provide the same service (”periodically publish 6D pose”) that makes them exchangeable. Developers can choose between alternatives in the design phase and alternatives might even be swapped at runtime. Again, this level conforms to the SMARTSOFT MetaModel which provides structure in the form of components that can provide and require services from (1) and therefore provide further structure to realize / implement them.

Finally, there are **Concrete Applications** in the last step (Figure 2: (3)). When users develop new applications, these applications can rely on service definitions in order to either implement these services or reuse 3rd party implementations of these services. Compatibility is given by the definition of services (1). SMARTSOFT also contributes to this

level by providing an execution container for components. It maps the component hull [SSL12b] to the execution environment (operating system and processor architecture).

From our experience, the best practice is to regard services as the most important building blocks for designing the architecture. Even though components implement functionalities and provide/require these services, the granularity of components is arbitrary and not decisive for the architecture. Component granularity might differ from application to application, whereas the granularity of services can remain the same. For example, the services ”localization” and ”navigationCommand” might be provided by two components from two suppliers or by one single component as long as it provides the defined services. The service-oriented component model naturally supports competing alternatives as building blocks for systems while still ensuring composability. Thus, openness with respect to software service definitions is in no way in conflict with the progress in implementations.

Services as stable entities in the design of systems ensure system level conformance, define responsibilities and allow to identify white spots that have to be covered in a very early stage of the workflow. The strength of this structured workflow is that repeating architectural patterns as well as project or application specific structures can be identified and defined very early and in parallel to the implementation of functionalities. This decoupling from architecture design and implementation has proven successful in the later integration where the system parts (components) just fit together. To an extend, this design is also independent from the hardware architecture: SmartSoft provides abstractions that leave space for integrating hardware by either providing the execution containers to (computational) hardware platforms or by transforming hardware specific sensor values into standardized services.

The goal during service discussions is to converge to the least common denominator of communication objects and services that are relevant in the individual domain. The smaller this set is, the better is the composability that can be achieved with building blocks (software components) that provide and require these services. However, one can at the same time keep specific communication objects and specific services in order to best exploit unique abilities and features of the software components. Finally, it is about finding the right balance between ”too specific” and ”too general” in order to support composability and reuse while not loosing unique characteristics. The illustrated workflow helps in this respect at a very early stage.

Decoupling of services and components brings the decisive flexibility that is a must-have for system composition and establishing a software business ecosystem, for both in-house building blocks and towards a ”market” of building blocks for robotics. At the same it time is a structured approach that ensures system level conformance.

Applying this workflow and discussing the services of a system with project stakeholders at a very early stage also helps each of them: by thinking in detail about what services they require or what services they provide, stakeholders reported that the workflow is very helpful for their own components and expectations.

4 Communication Objects as Self-Contained Entities

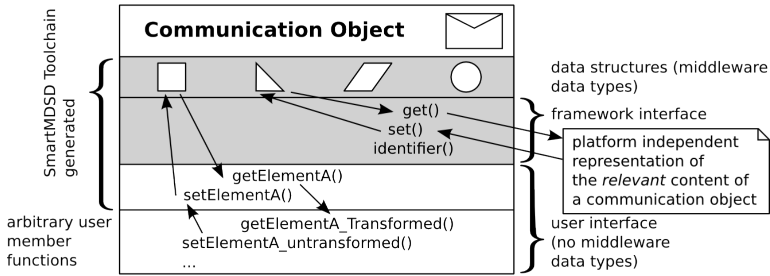

Figure 3: Communication Object with its interfaces for user and framework.

Communication objects [SSL12a], as shown in figure 3, define the data structures (figure 3, grey upper) to be transmitted via a communication pattern between components. Communication objects are C++-like objects decorated with additional member functions (figure 3, grey lower) for internal use by the framework. Besides attributes for communication, communication objects can define user access functions (figure 3, bottom).

It is our best practice to define communication objects as self-contained local entities, communicated *by value* between components via services. All the information needed to process communication objects within a component should be contained within that communication object and should not need additional communication in order to get that information. Communication objects should not only transport data, but also provide a locally extensible interface (user member functions: figure 3, white elements) to access the data, without influencing the overall system.

The self-containment of communication objects avoids fine grained intercomponent communication when processing them, thus helps to achieve one of the most important goals of service-oriented component based approaches: decoupling the sphere of influence between the components (*separation of concerns*). Loosely coupled components enable the *separation of roles*, reuse and easy system coordination.

For example a communication object for sensor data is tagged with a position (itself a communication object) where it was recorded. User member functions of the communication object perform the transformations from the sensor data into different formats, units, coordinate systems, etcetera. This is done without the need of further communication and coordination, keeping the effects limited to the local entity (component), as well as without reimplementing this logic over and over again. Regardless of the point of time when the communication object (e.g. camera image) is used, the information within the object (e.g. capture pose) forms a valid snapshot of the relevant system state in relation to the communicated data.

Communication objects that are not self-contained would require further intercomponent communication that would add to further system and coordination complexity. Other components would have to be activated and configured to get the additional information. Self-

contained communication objects typically also reduce the total amount of required inter-component connections, which again reduces system and coordination complexity. From an architectural point of view this best practice supports an early and expressive design of the components services, as all the relevant information is explicated and collected in the design phase of a system.

User member functions of the communication object provide access to data structures in those data types that are required by the user in the individual components. Commonly used functionality to access the data (e.g. transformation of their representation) can be implemented once and are provided with the communication object. Local extensions of communication objects, extending data types or user member functions, are implemented in locally inherited communication objects and do not spread across the system.

The user access functions are also used to prevent the communication middleware data types from polluting the user space. Using the SmartMDSD Toolchain, the interface functions (figure 3, white upper) to those communication middleware data structures are generated to make the user access function in the communication object independent of the underlying communication middleware, following the principle of *separation of concerns* in the same way as the SMARTSOFT components [SLL+13]. This ensures user code (figure 3, white lower) that is independent of the underlying communication middleware.

5 System Orchestration

System orchestration deals with coordinating the individual parts (components) of the system in such a way that the overall system performs a concrete task. It consists of plenty of challenges and the following will concentrate on best practices that enable *separation of concerns* and roles.

A robot system is separated into multiple layers by distinguishing hierarchical task decomposition and situation-driven task execution from fast reactive control algorithms on the skill layer. The *sequencer* bridges continuous processing and event-driven task execution. It orchestrates the software components in the system and assigns decision spaces to components. The components on the skill layer execute all kinds of algorithms, hardware drivers or control loops needed to perform the task guided by the *sequencer*. Details of the multi-layered system architecture can be found in [IRLVCS12] [SLL+13].

5.1 Coordination of Skill Components

Figure 4 shows an excerpt of the coordination within the collaborative robot butler scenario [Ulmb]. Given such a complex real world scenario (44 active components, open ended environment), the need for systematic coordination, reuse and separation of concerns becomes obvious.

The particularly interesting point is how the interface between *sequencer* and the skill

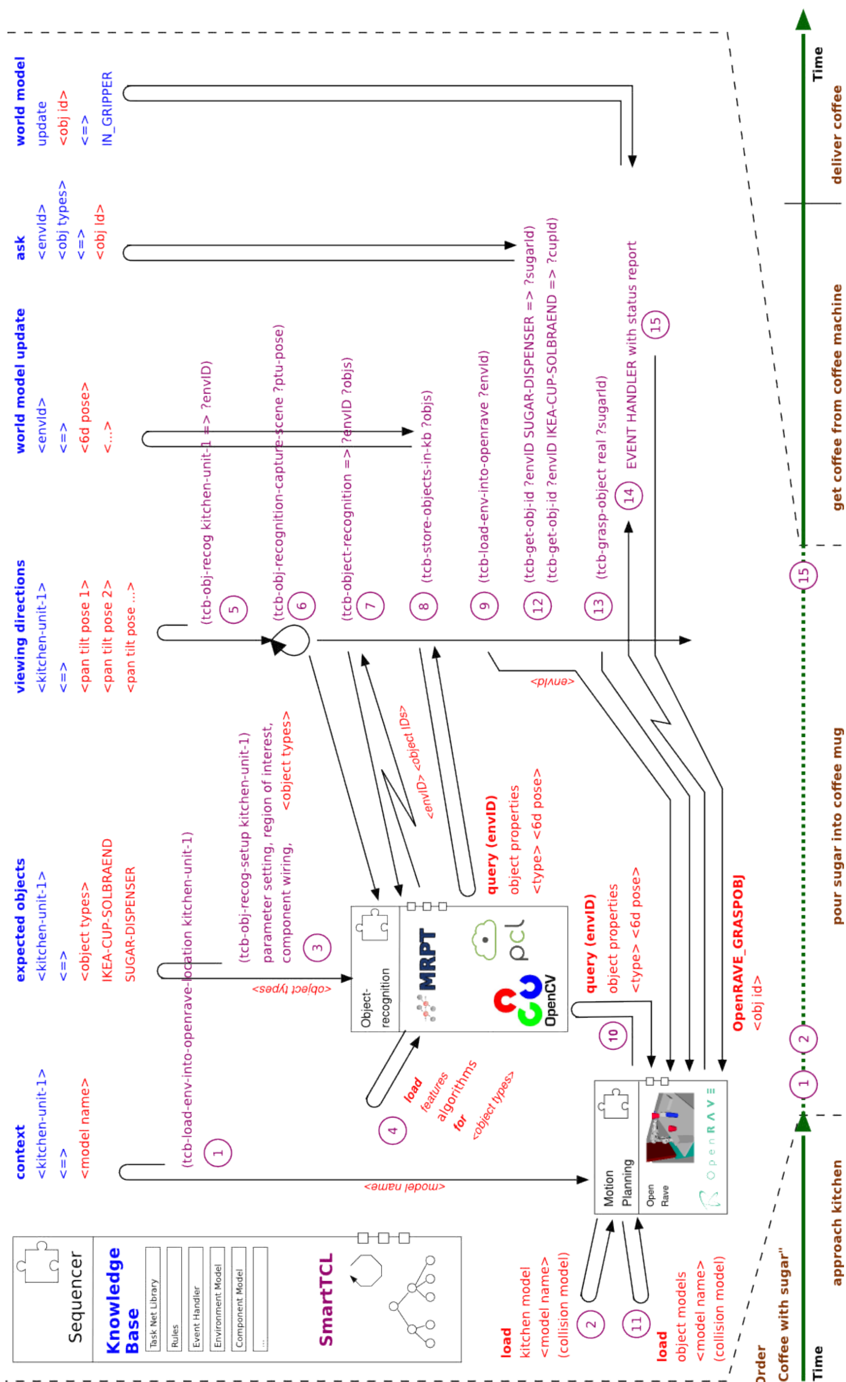

Figure 4: System coordination within the collaborative robot butler scenario. [Ulmb]

components is organized and how it distinguishes from the interfaces between each of the skill components. The interfaces within the skill components mostly consist of data (sub-symbolic) needed for reactive control algorithms such as sensor data, trajectories or sub-goals to approach, for example. The interface between *sequencer*, and skill components is used for coordination and typically includes symbolic data. One reusable pattern that is stable across all our systems is the orchestration cycle of the skill components driven by the *sequencer*: first the configuration of the components, then the execution or trigger of the actions in the components, events send to the *sequencer* driving it and fainally and optionally fetching resulting data from the components.

Given a task and the current context, the *sequencer* configures skill components according to the actions to be performed, using the SMARTSOFT parameter communication pattern. E.g. the object recognition component is configured to search for the object types *SUGAR-DISPENSER* and *IKEA-CUP-SOLBRAEND* (figure 4: ③). Once configured, the components are activated for continuous unsing state communication patter or single operation using a trigger (parameter communication pattern). In figure 4: ⑦), the object recognition component is triggered to perform recognition (once - triggered) given the configuration done before. The components run independent of further interaction with the *sequencer* as long as the skill components are able to operate within the configured decision spaces. If either the configured goal is achieved or components are not able to achieve the goal, an event using the corresponding SMARTSOFT event communication patter is send back from the individual skill component to the *sequencer*. The object recognition component fires an event received by the *sequencer* telling that objects with their corresponding *ids* where found (figure 4: ⑦), for example. It is the *sequencer's* responsibility to activate those events from the skill components that are necessary to perform the task given the current context. Dependent on the event, the *sequencer* might then fetch the result of the performed action in symbolic representation from the skill component. The object recognition component is queried for the recognized object using the beforehand via event transmitted *ids* (figure 4: ⑧), for example.

The separation of high-level task coordination and low-level control loops as well as the above described interface enforces the *separation of roles* and concerns in the development of complex systems. Explicating the configurations of components as well as their response (events), enables a systematic system orchestration and a hand over from the component developer to the role of the system integrator.

5.2 Distributed (Local) Knowledge Representation

At the *sequencer* level, information is stored and processed almost entirely in symbolic form. The sub-symbolic information and interaction is located at the level of the skill components. For many algorithms at skill level, reference data, models or configuration sets, etc. are required. The object recognition component for example needs all kind of modeled data to feed the different recognition algorithms, while other components such as the motion planning component again needs other views on the models. The individual models are best located close by the components using them (*separation of concerns*).

The coordination of the system however requires a common understanding of objects, locations, rooms etc. among the system parts.

Therefore, it is our best practice to connect the individual local views on the models and data with system-wide symbols, without spreading the local sub-symbolic information of component through the system. Those symbols are defined and linked to the modeled information at the *sequencer* level. Object types, for example are modeled at *sequencer* level and system wide uniquely identified by symbols (e.g. *IKEA-CUP-SOLBRAEND*). The object recognition component needs to get a shape model, whereas for manipulation planning a grasping model is needed. Both components get the reference to the modeled object type during the configuration using the unique symbol (figure 4: (4) and (11)).

This separation reduces the system complexity due to less intercomponent communication and keeps the responsibility for the local data bound to the role of the component developer while still being able of system wide coordination in the role of the system integrator.

6 Conclusions

All the described best practices and architectural principles were found and applied during many years of robotics software systems engineering. The SMARTSOFT approach has been used in various areas, by several users for many years. It has been used for education by students, e.g in the Robocup@Home competition. In our lab, every year a new team of students, without prior robotics knowledge, continues to work with the system of the previous team with no overlap. The black-box view of the components is the only way for the teams to continue and reuse the complex system parts. The SMARTSOFT approach has been used in several research projects, such as the *ZAFH Servicerobotik*, *FIONA* and *iserveU* and builds the stable foundation for many project partners in academia and industry. It is further used in intralogistic scenarios (e.g. [Ulmb]) in projects with partners from industry using the Robotino®3 robot.

Finding generic best practices in robot control architectures is about finding the sweet spot between those structures needed and those providing artificial restrictions. Structure is needed to ensure composability of the system parts which is vital for a robotics business ecosystem. Following the objectives of *separation of roles* and *separation of concerns* allows to guide which structures are needed. Following those objectives we presented some insights in those architectural structures and processes that helped us to organize and build a number of complex real world robotic software systems [Ulmb].

Our next steps are to further exploit model-driven techniques and enhance our model-driven toolchain in order to provide support to all roles of the development process with our best practices.

Acknowledgements

The research leading to these results has been partially funded by BMBF
(iserveU 01IM12008B and FIONA 01IS13017C).

References

[BEH+11] A. Bjorkelund, L. Edstrom, M. Haage, J. Malec, K. Nilsson, P. Nugues, S.G. Robertz, D. Storkle, A. Blomdell, R. Johansson, M. Linderoth, A. Nilsson, A. Robertsson, A. Stolt, and H. Bruyninckx. On the integration of skilled robot motions for productivity in manufacturing. In *Assembly and Manufacturing (ISAM), 2011 IEEE International Symposium on*, pages 1–9, May 2011.

[CGCG10] S. Cousins, B. Gerkey, K. Conley, and W. Garage. Sharing Software with ROS [ROS Topics]. *IEEE Robotics Automation Magazine*, 17(2):12–14, June 2010.

[Chr89] R. Chris. *Elements of functional programming*. Addison-Wesley Longman Publishing Co, Boston, MA., 1989.

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ., 1976.

[IRLVCS12] Juan F. Inglés-Romero, Alex Lotz, Cristina Vicente-Chicote, and Christian Schlegel. Dealing with Run-Time Variability in Service Robotics: Towards a DSL for Non-Functional Properties. In Emanuele Menegatti, editor, *3rd Int. Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob-12), Proc. of SIMPAR 2012 Workshop*, Tsukuba, Japan, November 2012. http://arxiv.org/pdf/1303.4296.

[Par72] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[RE96] M. Radestock and S. Eisenbach. Coordination in evolving systems. In *Trends in Distributed Systems CORBA and Beyond*, volume 1161 of *Lecture Notes in Computer Science*, pages 162–176. Springer Berlin Heidelberg, 1996.

[SLL+13] Christian Schlegel, Alex Lotz, Matthias Lutz, Dennis Stampfer, Juan F. Inglés-Romero, and Cristina Vicente-Chicote. Model-Driven Software Systems Engineering in Robotics: Covering the Complete Life-Cycle of a Robot. In *Informatik 2013, Workshop Roboter-Kontrollarchitekturen*. Springer LNI der GI, Koblenz, September 2013.

[SSL12a] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. In Daisuke Chugo and Sho Yokota, editors, *Introduction to Modern Robotics*. iConcept Press Ltd., 2012. Available from: http://www.iconceptpress.com/download/paper/101215045543.pdf.

[SSL12b] Christian Schlegel, Andreas Steck, and Alex Lotz. Robotic Software Systems: From Code-Driven to Model-Driven Software Development. In Dr. Ashish Dutta, editor, *Robotic Systems - Applications, Control and Programming*, chapter 23. InTech, 2012. Available from: http://www.intechopen.com/books/robotic-systems-applications-control-and-programming/robotic-software-systems-from-code-driven-to-model-driven-software-development.

[Ulma] Service Robotics Ulm. Website. http://www.servicerobotik-ulm.de.

[Ulmb] Service Robotics Ulm. Youtube Channel. http://youtube.com/user/RoboticsAtHsUlm.

[VKB14] Dominick Vanthienen, Markus Klotzbücher, and Herman Bruyninckx. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *Jorunal of Software Engineering for Robotics (JOSER)*, 5(1):17–35, May 2014.