Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development

Andreas Steck and Christian Schlegel
University of Applied Sciences Ulm
Department of Computer Science, Prittwitzstr. 10, 89075 Ulm, Germany
email: {steck, schlegel}@hs-ulm.de

***Abstract*— Engineering the software development process in robotics is one of the basic necessities towards industrial-strength service robotic systems. A major challenge is to make the step from code-driven to model-driven systems. This is essential to replace hand-crafted single-unit systems by systems composed out of components with explicitly stated properties. Furthermore, this fosters reuse by separating robotics knowledge from short-cycled implementational technologies. Altogether, this is one but important step towards "able" robots.**

This paper reports on a model-driven development process for robotic systems. The process consists of a robotics meta-model with first explications of *non-functional properties*. A model-driven toolchain based on *Eclipse* provides the model transformation and code generation steps. It also provides design time analysis of resource parameters (e.g. schedulability analysis of realtime tasks) as a first step towards overall *resource awareness* in the development of integrated robotic systems. The overall approach is underpinned by several real world scenarios.

I. Introduction

Nowadays, implementing complete robotic systems is still more of an art than a systematic engineering process. Integrating the various libraries currently is more like plumbing. Essential properties are mostly hidden in the software structures. In particular, *non-functional properties* and *Quality of Service (QoS)* parameters are not explicated. Although, these *non-functional properties* are considered being mandatory for embodied and deployed robots in real world, they are not yet addressed in a systematic way in most robotics software development processes. Thus, these properties cannot be taken into account during the design process, the system deployment and the dynamically changing runtime configurations.

The relevance of *non-functional properties* and *QoS* parameters arises from at least two observations of real world operation: (i) some components have to guarantee *QoS* (e.g. adequate response times to ensure collision avoidance), (ii) the huge number of different behaviors needed to handle real world contingencies (behaviors cannot all run at the same time and one depends on situation and context aware configuration management and resource assignment). Instead of allocating all resources statically, the system should dynamically adapt itself. Thereby, appropriate *QoS* parameters and resource information are to be taken into account.

For example, if the current processor load does not allow to run the navigation component at the highest quality level, the component should switch to a lower quality level, resulting in a reduced navigation velocity, but still ensuring safe navigation.

II. Motivation

Service robots are expected to fulfill a whole variety of tasks under very different conditions. We need various components providing the desired capabilities to build a full-fledged robotic system. Even the software components should be *COTS* components to foster reuse and ensure maturity and robustness. This kind of composability inevitably depends on assured services with explicated *QoS* parameters and required resources. That is even more important in service robotics where resources are strictly limited. Nevertheless, these are per se not explicated and are thus not accessible in most service robotic systems.

The desired approach should compose systems out of building blocks with assured services, both enriched with *QoS* parameters and resource information.

This would, for example, allow to perform realtime schedulability analysis, performance analysis and would also allow to check whether response times of services can be guaranteed. Furthermore, this allows to check whether the desired mapping of components to processors and communication busses provides enough computational and memory resources and bandwidth.

Depending on the executed task and the current state of the environment, the component interactions and configurations are different. Therefore, we cannot simply design a robotic software system as static system by considering only a small number of different configurations. In fact, we depend on dynamic system configurations at runtime. That also requires to take resource limitations into account.

Therefore, *resource awareness* will be mandatory in robotics at all phases of design, development, deployment and even at runtime.

As a consequence, one should establish a development process which can cope with *QoS* parameters and resource information at all development phases of the entire system. Essential for such a process is to describe the system in an abstract formal representation. *Model-driven engineering (MDE)* is the only known approach to make these relevant parameters explicit and accessible. Models abstract from unnecessary details and give the developers a domain-specific

view on the system [1].

Another significant benefit of *model-driven software development* is the decrease of development time and effort while increasing flexibility, reuse and the overall system quality. Design patterns, best practices and approved solutions can be made available within the code generators such that even novices can immediately take advantage from that coded and immense experience. A demanding challenge in making the step from code-driven to model-driven systems is the definition of an appropriate meta-model.

III. Related Work

Driven by the avionics and automotive industries, *distributed realtime and embedded (DRE)* systems denote an established research community, which already deals with questions relevant in robotics as well.

For example, the *Artist2 Network of Excellence on Embedded Systems Design* [2] addresses highly relevant topics concerning realtime components and realtime execution platforms. Furthermore, many symposia and conferences directly tackle the problem of component architectures for embedded systems and service oriented architectures in embedded systems [3].

The automotive industry is trying to establish the *AUTOSAR* standard [4] for software components and model-driven design of such components. *AUTOSAR* will provide a software infrastructure for automotive systems, based on standardized interfaces. Related to *AUTOSAR*, the ongoing *RT-Describe* project [5] addresses resource aspects. To adapt the system during runtime, *RT-Describe* relies on self-description of the components. Software components shall be enabled to autonomously reconfigure themselves, for example, by deactivating functions that are currently not needed. *RT-Describe* models are based on *UML* [6], *SysML* [7] and *MARTE* [8].

The *OMG MARTE* [8] activity provides a standard for modeling and analysis of realtime and embedded systems. They provide a huge number of *non-functional properties (NFPs)* to describe both, the internals and externals of the system. Mappings to analysis models (*CHEDDAR*, *RapidRMA*) are available[1] to perform scheduling analysis of *MARTE* models. This part of the *MARTE* profile is of interest to the robotics community.

The major differences that arise in robotics compared to other domains are *not* the huge number of different sensors and actuators or the various hardware platforms. Instead, the differences are the context and situation dependant reconfigurations of interactions and a prioritized assignment of limited resources to activities even at runtime – again depending on context and situation. Thus, mastering the huge amount of different configurations in robotic systems is far more complex than in common *DRE* systems or cars.

Robotic frameworks [9], like *Player/Stage* [10], *ROS* [11] and *MSRS* [12] are widely in use and many of them offer a rich support for common robotic hardware. All posses design tools, ranging from simple ones up to complex ones like visual programming in *Microsoft Robotics Developer Studio* [12]. The same holds true for middleware systems. The *OMG Data Distribution Service for Realtime Systems (DDS)* [13] standard provides the concepts of a publish-subscribe middleware structure with several integrated *QoS* parameters. These tools and libraries coexist, without any chance of interoperability. Each tool and framework has attributes that favors its use. They all do not make the step towards *MDSD*. However, only this step provides the semantics to become independent of specifics of equally suitable, but not easily replaceable, frameworks. The essential benefit would be to finally enable productive use of all those codebases due to a *MDE* approach.

An introduction into robotics component-based software engineering (*CBSE*) can be found in [14], [15]. Several important design principles and implementation guidelines that enable the development of reusable and maintainable software building-blocks are motivated in detail.

Within the robotics community, a model-based approach is meanwhile considered valuable and the interest of framework developers becomes focused on *model-driven software development (MDSD)*.

The *BRICS* project specifically aims at exploiting *MDE* as enabling approach to reducing the development effort of engineering robotic systems by making best practice robotic solutions more easily reusable. They plan to create a *BRICS* Integrated Development Environment, which will be based on Eclipse and will provide the robotics community with an MDE toolchain. [16]

The *OROCOS* [17] project conforms very well to our ideas. They currently work on the integration of their framework into a model-driven toolchain. Their focus is on hard realtime motion control applications. *OROCOS* provides *hotspots* to be filled in by the component developer.

GenoM3 [18] proposes scripting mechanisms to bind component skeleton templates. This does not reach the abstraction level of meta-models and severely restricts the application architecture to the given narrow template structure.

The *OMG Robotics Domain Task Force* [19] develops a standard to integrate robotic systems out of modular components. As in our work the components comprise an internal state automaton and interact via service ports. However, these ports are defined in a very generic way. The concept of execution contexts decouple the business logic from the thread of control. During the deployment the user has to assign appropriate execution contexts to the components. In our approach the user does not necessarily need any knowledge about the internal structure of the components during deployment. But as the elements and parameters are made explicit they can be accessed by analysis tools, for example. Unfortunately the reference implementation (OpenRTM) requires user-code to directly interact with the PSI-level even for core model parts, like communication. The user-code gets far to strong bound to middleware (*CORBA*) specifics.

[1] http://www.omgmarte.org/Tools.htm

An interesting approach which is similar to our work is presented in [20]. The *3-View Component Meta-Model (V^3CMM)* comprises three complementary views (*structural view*, *coordination view* and *algorithmic view*) to model component-based systems independent of the execution platform. V^3CMM encourages describing the component's algorithms (as an inherent part of the systems model) in a manner which is similar to *UML activity diagrams*. In comparison to that, we model the component hull "only". Inside of our components, the developers can integrate their algorithms and libraries (*OpenCV*, *Qt*, *OpenRAVE*, etc.) without any restrictions. In V^3CMM the components interact by calling the mutually provided interface operations. In our meta-model we provide strictly enforced interaction patterns to decouple the sphere of influence and thus partition the overall complexity of the system. This ensures reusability and interchangeability of the components.

It is worth highlighting that we focus on defining an abstract meta-model ensuring the modeling and analysis of robotic systems. Hence, we extend our current component model iteratively and express it in an abstract way (SMARTMARS meta-model) underpinned by different but replaceable implementational technologies.

IV. Model-Driven Software Development for Robotic Systems

Crucial to make the step from code-driven to model-driven engineering is to define a meta-model and a matured *model-driven software development* process which considers the properties explicated in the models. The challenge is to identify appropriate levels of abstraction and to handle the properties, attached to the elements of the model, in these different levels. As a result of our work, we propose a development process and a meta-model for robotic systems.

A. The Development Process

The development process (fig. 1) aims *not* to exhaustively model a robotic system at all views such that afterwards one has just to push the button of a toolchain and to await the executables. We are convinced that this will remain unrealistic. However, there is already a huge benefit if such a process can help to master the complexity of our systems and in particular, allows to manage and explicate *non-functional properties*. Starting with an idea, the model will be enriched during development time until it finally gets executable in form of deployed software components.

The major steps are, firstly, to describe the system in a model-based representation (*platform independet model - PIM*) which is independent of the underlying framework, middleware structures, operating system and programming language. In this design phase, several system properties are either unknown or only known as requirements.

The second step is to transform the *PIM* into a *platform specific model (PSM)*. After model checks, the platform independent elements are transformed into the appropriate elements of the platform specific meta-model. These elements represent the characteristics of the underlying environment

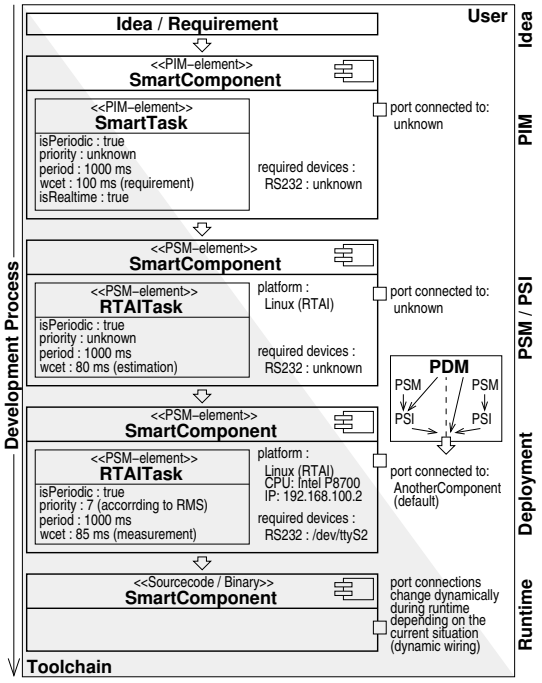

Fig. 1. The development process at-a-glance.

(middleware structures, operating system, framework, etc.). The *PSM* is enriched with properties which are known due to the knowledge about the platform specific characteristics. However, some parameters can still be unknown and are added not until the deployment of the component. Furthermore, the *PSM* is transformed into the platform specific implementation (*PSI*). The developers add their algorithms and libraries (*user-code*) with the guidance of the toolchain. The user code is protected from modifications made by the code generator due to the *generation gap pattern* [21] which is based on inheritance.

The third step is to deploy the components to the different platforms of the robotic system. The capabilities and characteristics of the target platforms are defined by the *platform description model (PDM)*. In this phase, the model is enriched by the knowledge about the target platforms. Further model checkings are performed to verify the constraints attached to the components against the capabilities of the system (e.g. is executable only on a certain type of hardware, needs one serial port, etc.). The *QoS* parameters of the interaction patterns, for example, can be cross-checked whether they can be satisfied. Furthermore, special analysis models can be generated out of the deployment model. These special models can be used to perform realtime schedulability analysis, for example.

The fourth step is to run the system according to the deployment model. Certain properties can still be unknown and will be reasoned during runtime. Additionally, the wiring between the components can change during runtime depending on the current situation of the robot. Even if it would be possible to calculate the required resources for a worst-case and rare-event situation in advance, it would not be

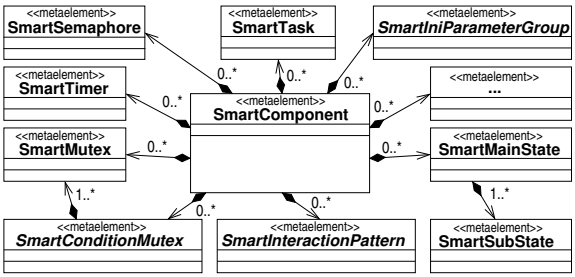

Fig. 2. Excerpt of the SMARTMARS meta-model.

TABLE I
THE SET OF INTERACTION PATTERNS.

Pattern	Relationship	Communication Mode
send	client/server	one-way communication
query	client/server	two-way request
push newest	publisher/subscriber	1-to-n distribution
push timed	publisher/subscriber	1-to-n distribution
event	client/server	asynchronous notification
state	master/slave	activate/deactivate component services
wiring	master/slave	dynamic component wiring

possible and efficient to provide these resources on a standard service robot. Dynamic adaptions and *resource awareness* are mandatory to be able to fulfill a variety of different tasks even with limited overall resources.

Absolutely essential for the development process is the definition of an abstract meta-model to describe robotic systems independently of the implementational technology.

B. The SMARTMARS meta-model

As a promising starting point to define the meta-model, we adapt and adopt well-established concepts of the SMARTSOFT framework [22], which has been continuously extended and used in building robotic systems for more than a decade now. The SMARTSOFT concepts are independent of the implementation technology and scale from 8-bit microcontrollers up to large scale systems [23]. Two reference implementations are available on sourceforge [24]. The first implementation is based on *CORBA* and the second one on *ACE* [25] only.

Consequently, we propose SMARTMARS (Modeling and Analysis of Robotic Systems) (fig. 2) as an abstract meta-model, which addresses modeling and analysis of robotic systems. The basic concept behind the meta-model are loosely coupled components offering/requiring services. These services are based on strictly enforced interaction patterns (table I) that transmit communication objects [22] [26]. They provide a precisely defined semantics and describe the outer view of a component, independent of its internal implementation. The patterns decouple the sphere of influence and thus partition the overall complexity of the system. Internal characteristics can never span across components. The interaction patterns provide stable interfaces towards the user code inside of the component and towards the other components independent of the underlying middleware

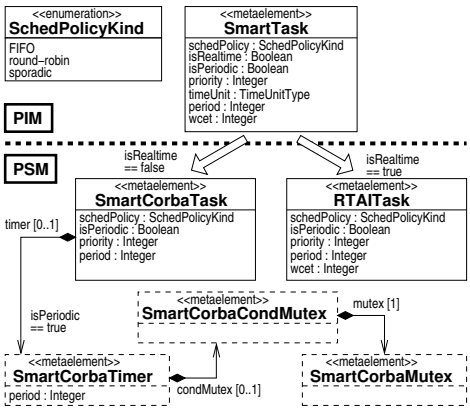

Fig. 3. The transformation of the *PIM* and the *PSM* by means of the *SmartTask* metaelement.

structure (fig. 4). The interfaces can be used in completely different access modalities as they are not only forwarding method calls but are standalone entities. The *query* pattern, for example, provides both, synchronous and asynchronous access modalities at the client side and a handler based interface at the server side. Interaction patterns are annotated with *QoS* parameters (e.g. cycle times for push timed pattern, timeouts for query and event pattern). Dynamic reconfiguration of the components at runtime is supported by a *param* port to send name-value pairs to the components, a *state* port to activate/deactivate component services and *dynamic wiring* to change the connections between the components.

The *state pattern* is used by a component to manage transitions between service activations, to support housekeeping activities (entry/exit actions) and to hide details of private state relationships (appears as stateless interface to the outside).

Dynamic wiring is the basis for making both, the control flow and the data flow configurable. This is required for situated and task dependent composition of skills to behaviors.

The *wiring pattern* supports dynamic wiring of services from outside (and inside) a component by exposing service requestors of a component as ports. The wiring pattern allows to connect service requestors to service providers dynamically at run time. A service requestor is connected only to a compatible service provider (same pattern and communication object). Disconnecting a service requestor automatically performs all housekeeping activities inside a communication pattern to sort out not yet answered and pending calls, for example, by iterating through the affected state automatons (inside interaction patterns) and thus properly unblocking method calls that otherwise would never return. For example, the wiring mechanism already properly sorts out effects of a server being destroyed while clients are in the process of connecting to it. Of course, this relieves a component builder from a huge source of potential pitfalls.

SMARTMARS covers two different views: (i) it is a completely abstract meta-model for modeling and analysis of robotic systems and (ii) it is a concrete reference implementation implemented as an *UML profile*.

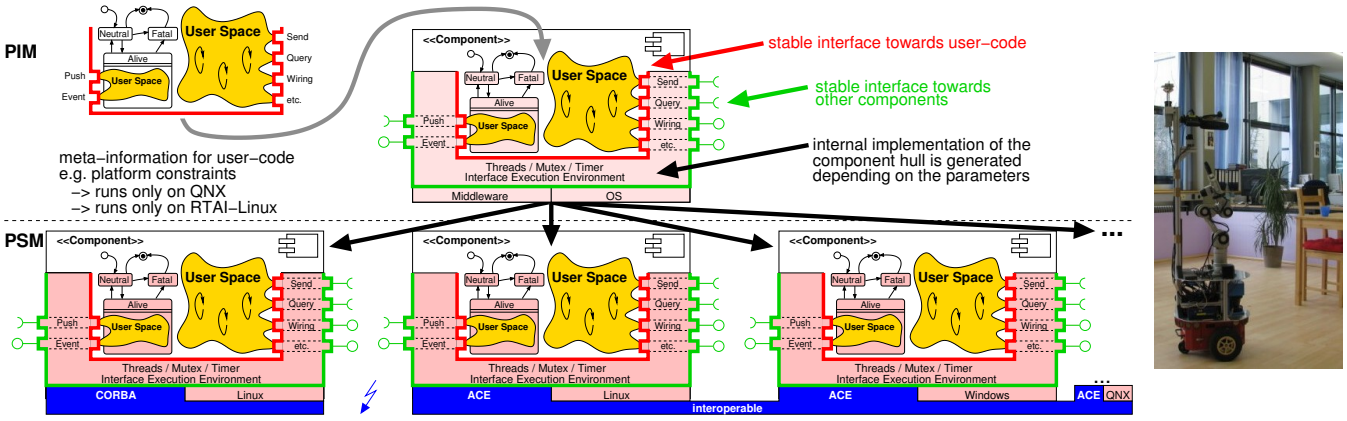

Fig. 4. *Left:* Refinement steps of component development. *Right:* Our robot Kate.

V. The SmartSoft MDSD Toolchain

As proof of concepts and to gain more experience on how to further investigate on the development of SmartMARS, we designed, implemented and provided the SmartSoft MDSD Toolchain [24] which is based on the Eclipse Modeling Project [27].

A. Mapping of Abstract Concepts

Fig. 3 shows the transformation of the *PIM* and the *PSM* by the example of the *SmartTask* metaelement and the *CORBA* based *PSM*. The *SmartTask* (*PIM*) comprises several parameters which are necessary to describe the task behavior and its characteristics independent of the implementation.

Depending on the `isRealtime` attribute and the capabilities of the target platform (*PDM*) the *SmartTask* is either mapped onto a *RTAITask* or a non-realtime *SmartCorbaTask*. If hard realtime capabilities are required and are not offered by the target platform, the toolchain reports this missing property. To perform realtime schedulability tests, the attributes (`wcet, period`) of the *RTAITasks* can be forwarded to appropriate analysis tools.

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` (user view) independent of the hidden internal mapping of the `SmartTask` (generated code).

B. Development of Components

The major steps to develop a SmartSoft component are depicted in figure 4 on the left. The component developer models the component in a platform independent representation using the SmartMARS *UML profile*. He focuses on the component hull, which comprises, for example, service ports and tasks – without any implementation details in mind (fig. 5). Pushing the button, the toolchain verifies the model (*oAW check*), transforms it into an appropriate *PSM* (*oAW xTend*) and generates the *PSI* (*oAW xPand*). Accordingly, the developer integrates his algorithms and libraries without

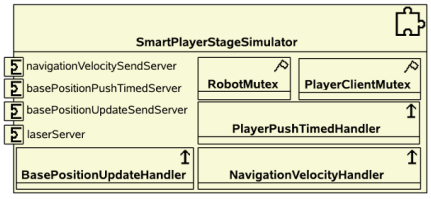

Fig. 5. Model of the SmartPlayerStageSimulator component modeled with the SmartSoft MDSD Toolchain

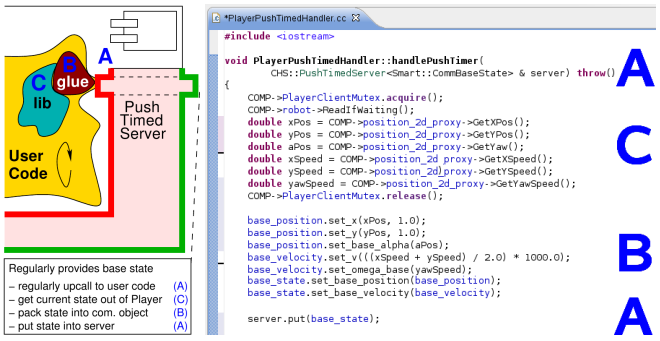

Fig. 6. The service port regularly provides a base state (pose, velocity etc.) by means of the *push timed* pattern.

any restrictions, but with the guidance of the toolchain (*oAW recipes*). The *oAW recipes* assist, for example, the handling and usage of the interaction patterns in the user part of the source code. Tags are used to indicate user code constraints (e.g. runs only on RTAI-Linux) and need to be set in the model by the user.

Fig. 6 illustrates how existing libraries are easily integrated into the user space of the component hull and how the glue logic looks like to link existing libraries (or code generated by other tools) to the generated component hull.

C. Deployment of Components

To create a system deployment the developer imports the components needed for the scenario into the deployment model. He maps the components onto the desired target platform and defines, for example, the initial wiring between the components.

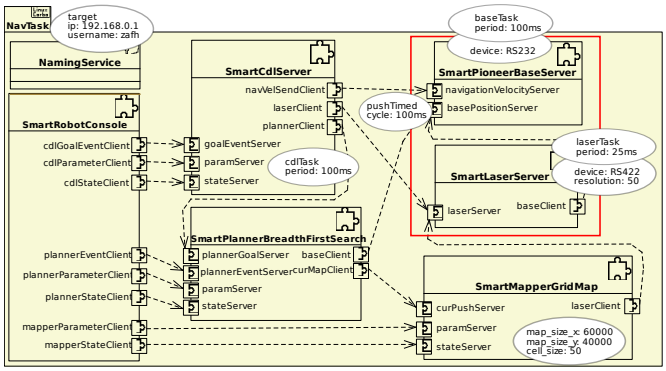

Fig. 7. Deployment diagram of a navigation task. Switching to simulation is done by replacing services (e.g. the Player/Stage component provides services of P3DX and LRF). Ellipses show selected non-functional properties defined in the models.

VI. Example

The model-driven toolchain has been used to build numerous robotic components (navigation, manipulation, speech, person detection and recognition, simulators) reusing many already existing libraries within the component hulls (*OpenRAVE, Player/Stage, GMapping, MRPT, Loquendo, VeriLook, OpenCV*, etc.). These components are reused in different arrangements (fig. 7) to implement, for example, *Robocup@Home* scenarios (*Follow Me*, *Shopping Mall*, *Who-is-Who*).

The *non-functional properties* specified in the different modeling levels enable analysis of resource usage and verification of resource constraints. Hard realtime schedulability analysis, for example, is performed by a model-to-model transformation from a deployment model into a *CHEDDAR* [28] specific analysis model.

VII. Conclusion and Future Work

The proposed development process and meta-model allows the explication, management and analysis of *non-functional properties*. We made those parameters accessible during development and deployment to check for guarantees and resource awareness in a systematic way. The next steps are to extend the SmartMARS meta-model and to further make use of it for analysis, verification and simulation at design-time as well as at run-time.

VIII. Acknowledgments

This work has been conducted within the *ZAFH Servicerobotik* (http://www.servicerobotik.de/). The authors gratefully acknowledge the research grants of state of Baden-Württemberg and the European Union.

We thank Dennis Stampfer for his extraordinary support in implementing the SmartSoft MDSD Toolchain.

References

[1] J. Bézivin, R. F. Paige, U. Aßmann, B. Rumpe, and D. C. Schmidt, "Perspectives Workshop: Model Engineering of Complex Systems (MECS)," Dagstuhl Seminar Proceedings, 2008. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2008/1604

[2] ARTIST, "Network of excellence on embedded system design," 2010, http://www.artist-embedded.org/, visited on June 15th 2010.

[3] ISORC09, "12th IEEE Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing," 2009, http://www.dcl.info.waseda.ac.jp/isorc2009/, visited on June 15th 2010.

[4] AUTOSAR, "Automotive open system architecture," 2010, http://www.autosar.org/, visited on June 15th 2010.

[5] RT-Describe, "Iterative Design Process for Self-Describing Real Time Embedded Software Components," 2010, http://www.esk.fraunhofer.de/EN/press/pm0911RTDescribe.jsp, visited on June 15th 2010.

[6] OMG UML, "Unified Modeling Language (UML) Superstructure specification v2.2, formal/09-02-02," February 2009.

[7] OMG SysML, "OMG Systems Modeling Language (SysML) specification v1.1, formal/2008-11-02," November 2008.

[8] OMG MARTE, "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2, ptc/2008-06-08," June 2008.

[9] Robot Standards and Reference Architectures (RoSTa), "Coordination Action funded under the European Unions Sixth Framework Programme (FP6)," February 2010. [Online]. Available: http://wiki.robot-standards.org/index.php/Middleware

[10] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proc. of the 11th Int. Conf. on Advanced Robotics (ICAR)*, Coimbra, Portugal, June 2003, pp. 317–323.

[11] ROS, "Robot operating system," 2010, http://www.ros.org/, visited on June 15th 2010.

[12] Microsoft, "Microsoft Robotics Developer Studio," 2010, http://msdn.microsoft.com/en-us/robotics/default.aspx, visited on June 15th 2010.

[13] OMG DDS, "Data Distribution Service for Real-time Systems (DDS) v1.2, formal/2007-01-01," January 2007.

[14] D. Brugali and P. Scandurra, "Component-Based Robotic Engineering (Part I)," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, Dezember 2009.

[15] D. Brugali and A. Shakhimardanov, "Component-Based Robotic Engineering (Part II)," *IEEE Robotics & Automation Magazine*, vol. 17, no. 1, pp. 100–112, March 2010.

[16] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, J. Broenink, D. Brugali, and N. Tomatis, "BRICS – Best practice in robotics," in *Proc. of the Joint 41st International Symposium on Robotics and the 6th German Conference on Robotics*, 2010, pp. 968–975.

[17] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, vol. 2, 2003, pp. 2766–2771.

[18] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "GenoM3: Building middleware-independent robotic components," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2010.

[19] OMG Robotics, "OMG Robotics Domain Task Force," 2010, http://robotics.omg.org/, visited on June 15th 2010.

[20] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Álvarez, "V^{3}CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development," *Journal of Software Engineering for Robotics*, vol. 1, no. January, pp. 3–17, 2010.

[21] J. Vlissides, "Pattern Hatching – Generation Gap Pattern," http://researchweb.watson.ibm.com/designpatterns/pubs/gg.html, visited on June 15th 2010.

[22] C. Schlegel, "Communication Patterns as Key Towards Component-Based Robotics," *Int. Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 49 – 54, 2006.

[23] C. Schlegel, T. Haßler, A. Lotz, and A. Steck, "Robotic software systems: From code-driven to model-driven designs," in *International Conference on Advanced Robotics (ICAR)*, June 2009.

[24] SmartSoft, http://smart-robotics.sf.net/, visited on June 15th 2010.

[25] D. Schmidt, "The ADAPTIVE Communication Environment," http://www.cs.wustl.edu/~schmidt/ACE.html, visited on June 15th 2010.

[26] C. Schlegel, "Navigation and execution for mobile robots in dynamic environments – an integrated approach," Ph.D. dissertation, Faculty of Computer Science, University of Ulm, 2004.

[27] E. M. Project, http://www.eclipse.org/modeling/, visited on June 15th 2010.

[28] CHEDDAR, "The cheddar project," http://beru.univ-brest.fr/~singhoff/cheddar/, visited on June 15th 2010.