Model-Driven Software Systems Engineering in Robotics: Covering the Complete Life-Cycle of a Robot

Christian Schlegel[1], Alex Lotz[1], Matthias Lutz[1], Dennis Stampfer[1],
Juan F. Inglés-Romero[2], Cristina Vicente-Chicote[3]

[1] University of Applied Sciences Ulm, Germany
{schlegel, lotz, lutz, stampfer}@hs-ulm.de
[2] Universidad Politécnica de Cartagena, Spain
juanfran.ingles@upct.es
[3] QSEG, Universidad de Extremadura, Spain
cristinav@unex.es

Abstract: Robotics experts still mostly work at the code-level when it comes to software integration for advanced robotic systems like service robots. Compared to other high-tech industries, software development and systems integration in robotics still lacks the processes and structures to come up with a software business ecosystem in robotics. In this paper, we outline how we can make the step towards a software business ecosystem in robotics by means of model-driven software development and by means of model-driven software systems integration. We summarize our current work on model-driven software development in robotics and put a focus on challenges still being considered as open and worth being addressed next.

1 Motivation

Vital functions of advanced robotic systems are provided by software and software dominance is still growing. Mastering the software complexity becomes pivotal towards exploiting the capabilities of advanced robotic components and algorithms. As long as the software part implies risks and efforts that are difficult to manage (see the Fraunhofer EFFIROB-study [HBK11]), the software challenge will very likely be a show-stopper towards the next level of robotic applications.

Although the challenges of implementing software for advanced robotic systems are tremendous, there is still a lack of applying state-of-the-art software development processes and software systems engineering approaches in robotics. Interestingly enough, for these most complex systems the robotics community does even not just reuse software design and development processes already established in other and similar domains like automotive and avionics. In the best case, there must be something different in robotics that requires at least enhancements.

In our understanding there is the following major difference of robotics compared to other domains where model-driven software development is already in use: the robot itself de-

pends on run-time exploitation of model-based variation-points in order to manage its (typically scarce) resources to face open-ended environments and to meet non-functional requirements.

Since the overall number of different situations in open-ended environments cannot be foreseen even by the most-skilled robotics engineer in advance, we need the robot to be able to make appropriate decisions at run-time. This is also mandatory since even trying to explicate all possible combinations of situations, proper resource assignments and reactions at design-time is quite inefficient due to a combinatoric explosion. It is much more appropriate to explicate the overall policies that should be applied in particular situations and then leave it to the robot at run-time to make the according decisions. The then available situation-dependent information significantly reduces the search space for an appropriate reaction. Based on its models, an advanced robotic system should trade-off at run-time how to use its capabilities and resources such that it successfully completes its tasks in open-ending environments with at least the expected and/or required quality-of-service.

2 The Need for Model-Driven Approaches in Robotics

We are convinced that the step towards model-driven software and systems engineering is mandatory in robotics in order to make the decisive evolution towards the next level of advanced robotic systems. Model-driven software development and systems integration is also considered as decisive towards a business ecosystem for robotics software.

2.1 Towards a Software Business Ecosystem in Robotics

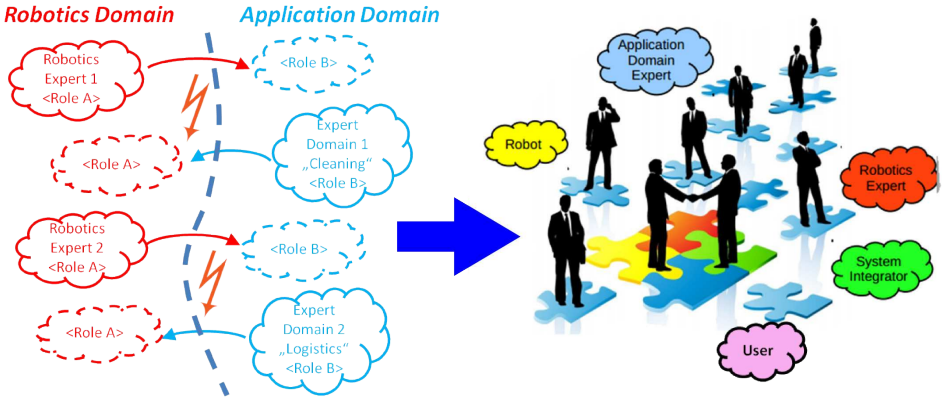

Figure 1: Towards a robotics business ecosystem.

As in every successful business ecosystem, *separation of roles* should achieve a symbiotic coexistence based on dedicated expertise of the various stakeholders (robotics expert, application domain expert, system integrator, framework and tool developers, professional users and consumers and the robot itself). This would allow to share and lower risks and efforts, to reduce costs, development time and time-to-market, to take advantage from specialized and second source suppliers, and to increase robustness and quality of products and services.

At present, *application domain experts* need to become *robotics experts* and vice versa (figure 1). This has not been a problem for implementing and operating lab prototypes in academia, but the lack of support for *separation of roles* in the development processes is a major hurdle when it comes to developing a service robotics market.

In order to achieve *separation of roles*, we need to support a black-box view of the software building blocks. A black-box view comes with explicated interfaces, properties and variation points. That is needed for system composition, proper configuration and hand-over to another role without requiring knowledge of inside details of the black-box as long as you are not responsible for that component.

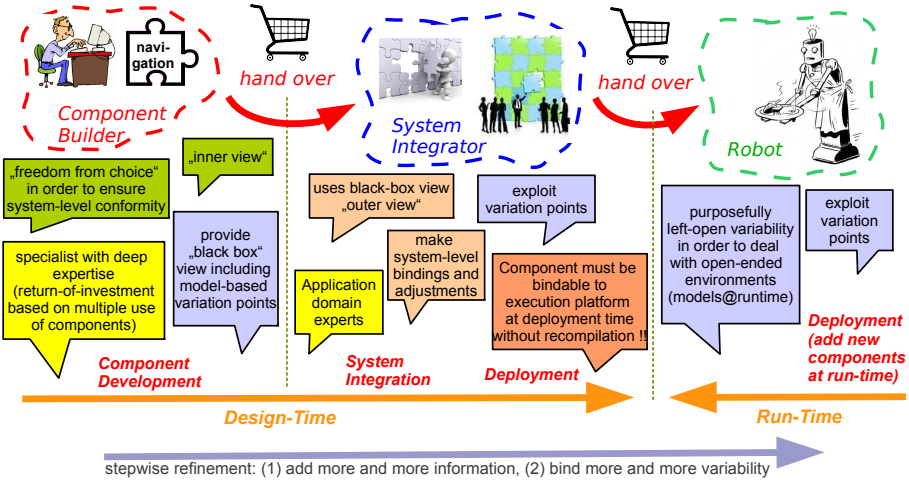

Figure 2: MDSD to manage the hand-over from one role to another role as key ingredience towards a robotics business ecosystem: use models for the entire life-cycle of a robot. We remove complexity from the designer: we make it as simple as possible to express variability at design-time by DSLs (domain-specific languages). We remove complexity from the robot's run-time decision by providing models and mechanisms for how to bind left-open variability.

This can be achieved by a *model-driven software development process* (figure 2). It supports the tasks of the specific roles without bothering them with the details that are not relevant for them. For example, a component builder provides a black-box component with explicated variation points [SSL12] as off-the-shelf component. A system integrator can pick-up this component and binds left-open variability and harmonizes the settings according to system level requirements.

2.2 Quality-of-Service in Open-Ended Environments

Figure 2 also shows the handover of design-time models for run-time usage by the robot itself and the stepwise refinement of the models along the workflow. Models allow for explicated variation points. That is important at design-time (support for a black-box view of a component with dedicated and purposefully left-open bindings in order to support the system integrator) as well as at run-time. At run-time, it is the robot itself that binds left-open variability based on the then available information such that it best fulfills the to be applied policies.

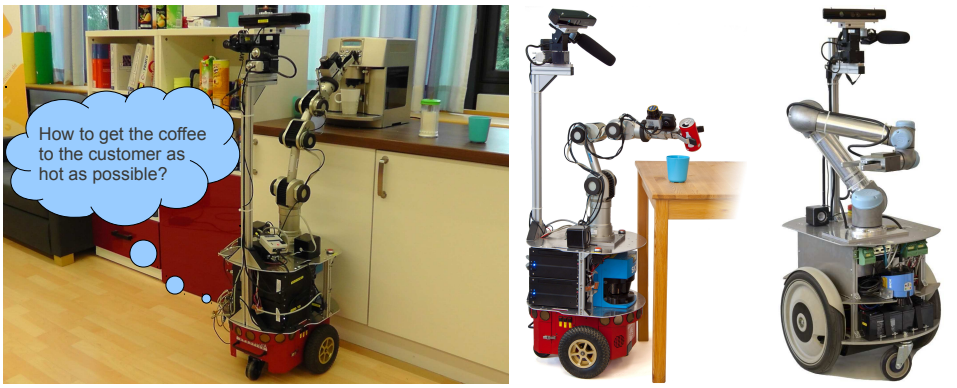

Figure 3: The robot *Kate* decides on non-functional properties at run-time (left). On the right, you see our service robots *Kate* and *Larry*.

An example for deciding on non-functional properties at run-time is shown in figure 3. The robot decides at run-time on an appropriate velocity by trading-off energy consumption, coffee temperature at delivery time and safety.

3 Method

The overall system architecture implemented on our servicerobots is shown in figure 4. The *sequencer* plays the master role in our multi-layered architecture. The sequencer bridges between continuous processing and event-driven task execution. It orchestrates the software components in the system and assigns decision spaces to components. The sequencer involves dedicated experts for run-time binding of designed variability [IRLVCS12].

The overall model-driven software development approach is summarized in figure 5. All the software components are based on the service-oriented SMARTSOFT software component model. The software component model is formalized as SMARTMARS meta-model and available within an Eclipse-based model-driven software development toolchain. The component model and its execution container abstracts away the operating system and the middleware.

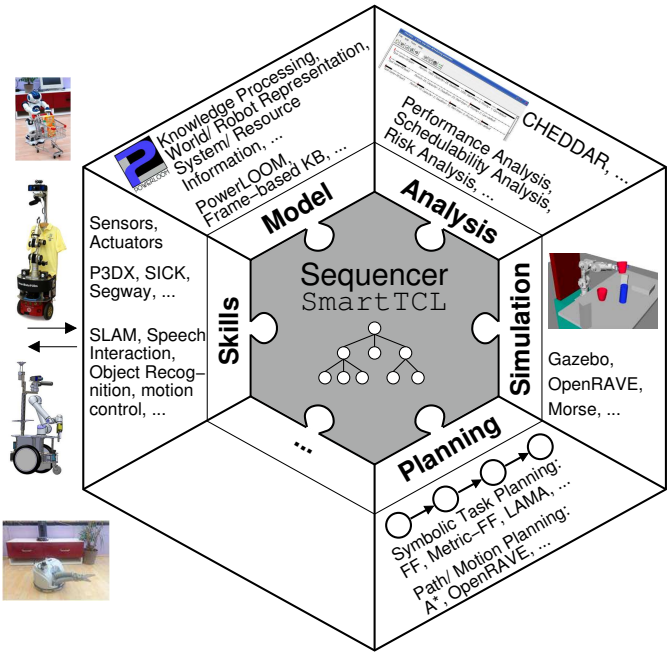

Figure 4: The overall system architecture implemented on our servicerobots. The sequencer orchestrates the system (sets parameters/configurations; switches components on/off; changes the wiring between components; queries information; waits for events in order to initiate the next execution step or modifies the task refinements) and involves experts like symbolic planners or simulation where appropriate for task refinement.

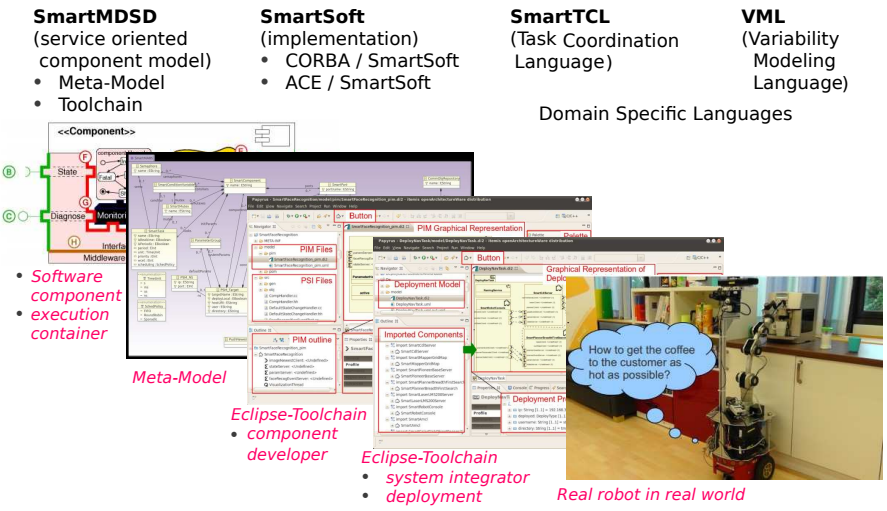

Figure 5: Overview on the SmartSoft-MDSD approach and its Eclipse-based toolchain.

3.1 SmartSoft and SmartMDSD: A Robotics Software Component Model

Besides *separation of roles*, another important aspect is *separation of concerns*: computation (functionality), communication (exchange data between entities), configuration (parameters at component and system level, initial wiring between components) and coordination (orchestration, resource management, dynamic wiring at run-time).

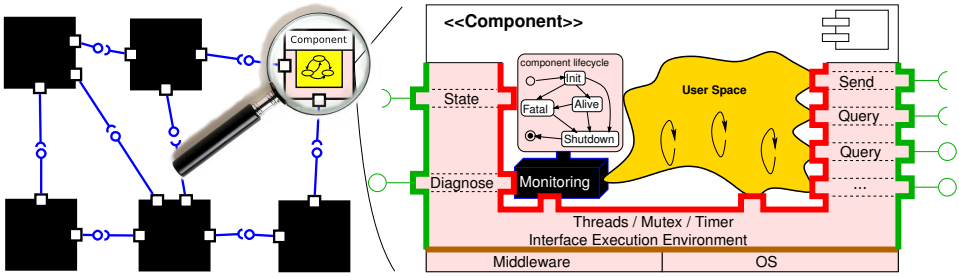

Figure 6: SmartSoft supports a black-box view based on a service-oriented software component model.

The approach behind SMARTSOFT is to gain control over the component hull of the software components that is mastering the link between the component inside view and the component outside view (see figure 6). Instead of allowing for any kind of ports at the component hull, SMARTSOFT requires all ports to be based on a small set of predefined communication patterns. The communication patterns think service oriented rather than message-centric: a SOA (service-oriented architecture) has to ensure that services don't get reduced to the status of interfaces, rather they have an identity of their own. Each communication pattern binds a reasonable and consistent configuration (communication policy: request/response, publish/subscribe, event; mechanism; protocol etc.). This results in a precise semantics of the services at the component hull and strictly separates the component inside structures, mechanisms and policies from the outside visible ones. Gaining control over the component hull is also decisive towards a black-box view where all relevant properties and parameters need to be explicated at the component hull in order to support separation of roles.

As illustrated in figure 7, the stable interfaces visible to the component developer allow for any kind of access methods to services. The variety inside a component eases the job of the component developer and gives him the freedom to use the desired and preferred access methods (synchronous, asynchronous; invocation, upcall) and gives him the freedom to install the desired processing (passive, thread-pool, pipeline, buffers, etc.). That variety is not presented outside the component where it affects system integration. Thus, the communication patterns avoid complexity of combinatorial explosion of policies and mechanisms etc. They ensure system level conformance (avoid distributed system deadlocks etc.) and they avoid incompatible port variants of the same service. It is also important to notice that the stable interfaces inside the component and outside the component are independent from the used middleware. Thus, this approach supports a late linking

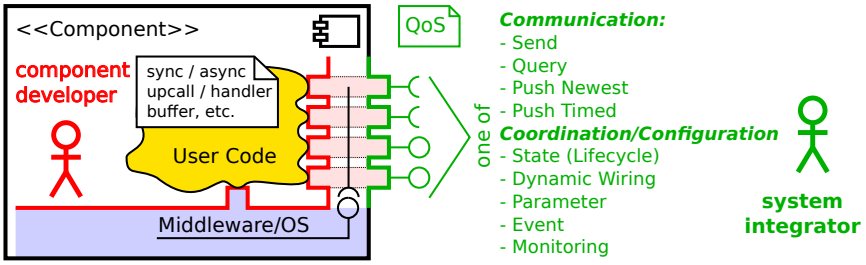

Figure 7: Mastering the link between component inside / outside view by communication patterns.

to the implementation of the execution container (operating system) and a late binding / exchange of the middleware system. The SMARTSOFT component model is represented as SMARTMARS meta-model and is being used in the MDSD Toolchain (see figure 5).

3.2 The Task Coordination Language (SmartTCL)

SMARTTCL is a domain-specific language to model hierarchical task decompositions (figure 8). It describes *how* to coordinate the robots actions and thus models *variability in operation* (sequencing and refinement of actions). SMARTTCL provides robustness to contingencies and maintains a high success rate in task fulfillment. The situation-driven execution and refinement of the SMARTTCL task-nets is performed by the *sequencer*.

Task Coordination Block (TCB)

signature		
	tcb-name	: *<name>*
	input variable	: *<list of variables>*
	output variable	: *<list of variables>*
	precondition	: *<condition>*
	priority	: *<integer>*
	rules	: *<list of rule-names>*
	action	: *<action description>*
	plan	: *<list of TCB-names>*

Event-Handler

name	: *<name>*
action	: *<action definition>*

Rule

rule-name		: *<name>*
ref. TCB	tcb	: *<tcb-name>*
	in-vars.	: *<list of variables>*
	out-vars.	: *<list of variables>*
return value		: *<return message>*
action		: *<action definition>*

```
(define-tcb (cleanup-table ?location)
  (rules (
    rule-action-cleanup-table-failed
    rule-action-cleanup-table-empty
    rule-action-cleanup-say-success ))
  (plan (
    (tcb-approach-location ?location)
    (tcb-say "I have reached the table,
     and will look for objects to clean up.")
    (tcb-cleanup-dinner-table)
  ))
)
```

Figure 8: The left and middle parts give an overview on the structure of SmartTCL. The right part is an example of the concrete task control block “cleanup-table”.

Figure 9 illustrates the required sequence of actions (via numbers) and task block expansions for the task to clean up a dinner table. Task blocks can use various sources of information such as the knowledge base (see (3) in figure 9), external experts like the symbolic planner (see (9) in figure 9), or events from components in the system. A task block can return different values upon which the next task refinements depend. In order to avoid repeating at each task block how to respond to certain outcomes, one can express these strategies to handle contingencies via rules and assign them to task blocks.

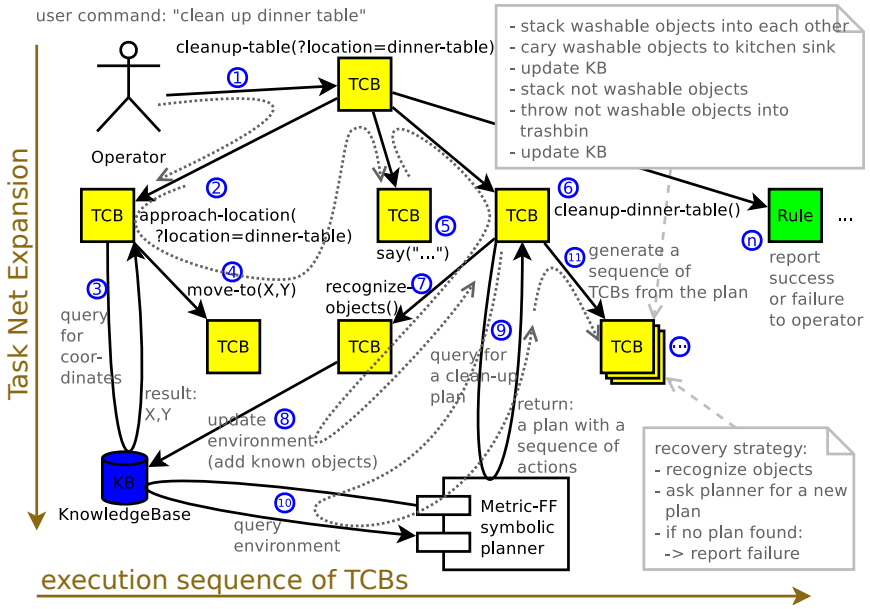

Figure 9: Snapshot of a possible task tree expanded for the command "cleanup-table(dinner-table)" (task control block described in figure 8).

3.3 The Variability Modeling Language (VML)

VML is a domain-specific language to model *variability in quality* that is to define *what is a good way (policy)* of achieving a task (optimizing non-functional properties). *VML* focuses on improving the overall execution performance of the robot under changing situations and limited resources.

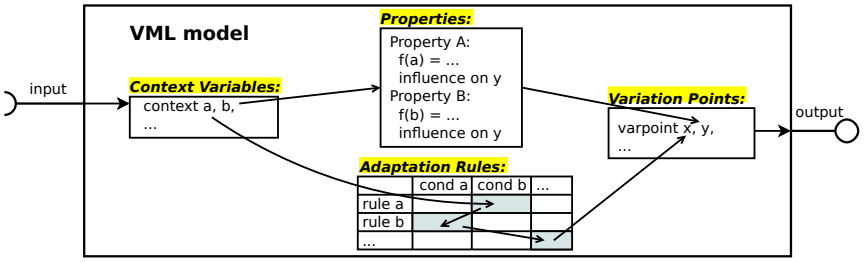

Figure 10: The overall structure of VML.

Context variables (see figure 10) define the input (selected parts of the current robot state and the environment) and *variation points* the output for *VML* models. *VML* allows to use discrete adaptation rules (event-condition-action rules that directly constrain the possible

values of variation points) and properties (continuous functions) to bind variation points. The execution environment is a constraint solver which optimizes the variation points to improve the overall system quality.

Figure 11: Coffee delivery scenario.

Figure 11 illustrates the example (introduced in section 2.2) in more detail. The corresponding *VML* model is shown in listing 1.

```
/* Contexts variables */
context ctx_battery               : number [0:1:100];
context ctx_distanceMachine_A     : number [0:0.1:20];
context ctx_distanceMachine_B     : number [0:0.1:20];
context ctx_waitingTimeMachine_A  : number [10:1:300];
context ctx_waitingTimeMachine_B  : number [10:1:300];
/* Auxiliary variables */
var timeMachine_A := ctx_waitingTimeMachine_A + ctx_distanceMachine_A/600;
var timeMachine_B := ctx_waitingTimeMachine_B + ctx_distanceMachine_B/600;
/* ECA rules */
rule lowBattery_NearMachineA :  ctx_battery < 15 and ctx_distanceMachine_A < ctx_distanceMachine_B
   => coffeeMachine = @coffeeMachine.COFFEE_MACHINE_A;
rule lowBattery_NearMachineB : ctx_battery < 15 and ctx_distanceMachine_A >= ctx_distanceMachine_B
   => coffeeMachine = @coffeeMachine.COFFEE_MACHINE_B;
rule high_EFF_coffeeMachA : ctx_battery >= 15 and timeMachine_A  > timeMachine_B
   => coffeeMachine = @coffeeMachine.COFFEE_MACHINE_A;
rule high_EFF_coffeeMachB : ctx_battery >= 15 and timeMachine_A <= timeMachine_B
   => coffeeMachine = @coffeeMachine.COFFEE_MACHINE_B;
/* Properties to optimize */
property performance : number[0:1:100] maximized {
  priorities: f(ctx_battery) = max(exp(-1*ctx_battery/15), 10 sec) - exp(-1*ctx_battery/15);
  definitions: f(maximumVelocity) = maximumVelocity;
}
property powerConsumption : number[0:1:100] minimized {
  priorities: f(ctx_battery) = exp(-1 * ctx_battery/15);
  definitions: f(maximumVelocity) = exp(maximumVelocity/150);
}
/* Variation points */
varpoint maximumVelocity : number [100:10:600];
varpoint coffeeMachine   : enum { COFFEE_MACHINE_A, COFFEE_MACHINE_B };
```

Listing 1: VML model for choosing a coffee machine and proper maximum velocity.

The *ECA rules* relate to the variation point *coffee machine* and describe which coffee machine to use: if the battery level is below a threshold, just take the nearest coffee machine; if the battery level is high enough, then take the coffee machine with the lowest waiting time.

The *properties* relate to the variation point *maximum velocity* and describe how to select an appropriate velocity value balancing *performance* (to be maximized) and *power consumption* (to be minimized).

3.4 Integration

Both variability management mechanisms (*variability in operation*: SMARTTCL, *variability in quality: VML*) need to be integrated such that they work together without interfering. The approach for integration follows the *subsidiarity principle* where we assign orthogonal decision spaces down the currently active control hierarchy [LIRVCS13].

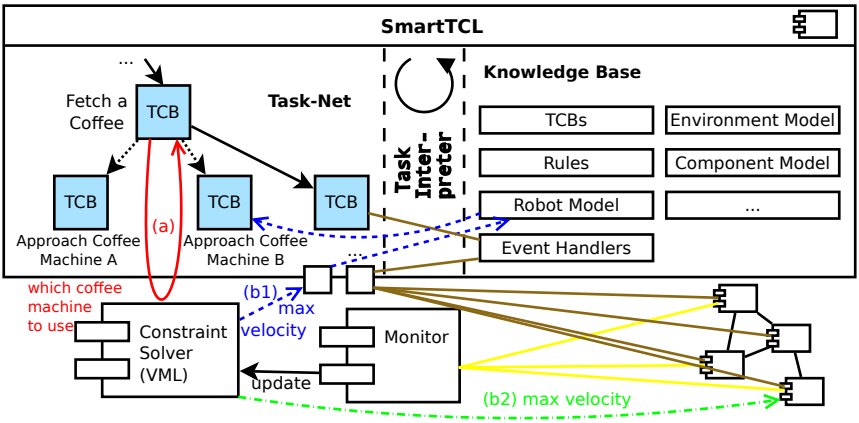

Figure 12: SmartTCL and VML interaction possibilities.

This results in three different ways of how SMARTTCL and *VML* interact (see figure 12): (a) *VML* as a service on demand: SMARTTCL uses the *VML* component as expert and asks for advice before making the decision of how to expand the task net. (b1) *VML* as a continuous service: SMARTTCL configures the *VML* component to monitor parameters that are managed by the sequencer and to give regular recommendations or notifications. SMARTTCL takes these updated values into account in its decisions and configurations. (b2) *VML* as a continuous service: SMARTTCL configures and wires the *VML* component such that it directly updates settings in components (orthogonal responsibilities and decision spaces according to the subsidiarity principle).

3.5 Component Reuse and Systems Integration

The benefits of a model-driven approach become obvious when migrating software components from one robot platform to another one or when composing another system out

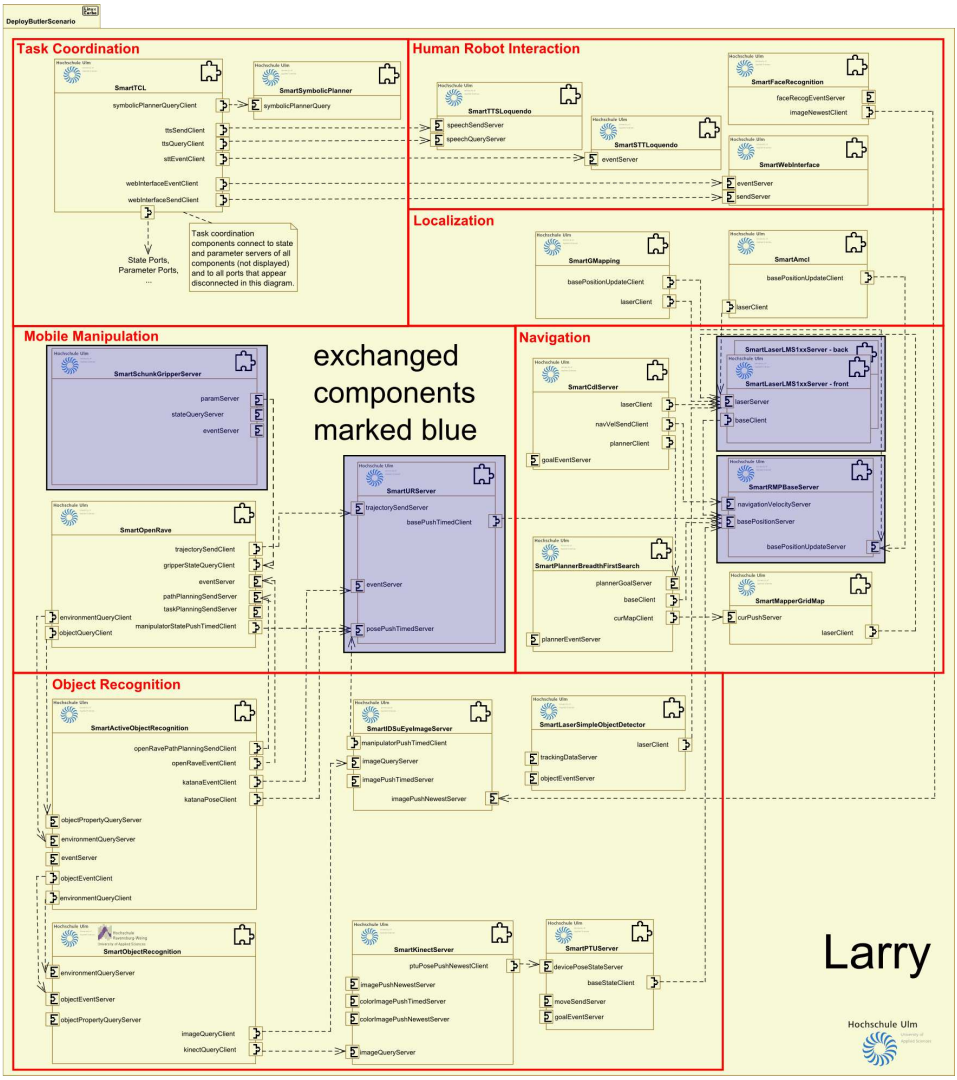

Figure 13: Migrating software components from robot *Kate* to robot *Larry*. The blue boxes indicate modified software components, the others have been reused as-is. It is the view from within the SmartMDSD-toolchain, only the red and blue boxes have been added afterwards for clarification.

of the same set of software components. Our service robot *Kate* is based on a P3DX-platform with a Katana 5-DoF manipulator while our service robot *Larry* is based on a RMP50-platform with a 6-DoF UR-manipulator. As shown in figure 13, modifications are related to exchanging models (the kinematic model of the manipulator) or software components that link services to the robot's hardware (base server). Meanwhile, many components are also in use with the FESTO RobotinoXT platform without any modifications. We were also able to seamlessly replace the *CORBA*-based implementation of the SMARTSOFT component model by the *ACE*-based implementation. The components have also been used successfully with the Fraunhofer IPA Care-O-bot system.

The SMARTMDSD toolchain and the SMARTSOFT software component model proved to support separation of roles and separation of concerns in various distributed projects (industrial collaborations, industry-academia joint projects) with different partners (experienced, novice): way before even knowing or implementing any algorithmic detail, it allowed for agreements on responsibilities and solid service definitions. The definition of the component hull and the required and provided services assigns clear responsibilities and enforces the strict separation of the component internal structures from the outside presentation of the services. Reuse takes place at the level of existing libraries (that can be wrapped by a component hull), by component models (reusing the component hull in order to pick up proven granularities of services and components) or at the level of black-box components.

The power of separation of roles based on a MDSD-approach is also evaluated within our Robocup@Home-Team. Each year, a new team of master students is formed that takes over and reuses software components, develops and adds new components and composes new scenarios out of the available robot skills. Based on the MDSD-toolchain, they successfully fill different roles and compose systems out of black-box software building blocks.

4 Related Work and State-of-the-Art

This paper summarizes the work presented in [SSL12, LIRVCS13, SS11]. We therefore do not cover the huge body of relevant related work and state-of-the art in this paper. We just refer to a small set of selected works in order to prepare for the open challenges in the next section.

In the last decade, there has been large achievements in producing software for robotics systems worldwide. For example, ROS [QCG+09] 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.

On the other hand there are domains which already master successfully the software complexity by applying model driven software engineering (MDSE) approaches. For example the AUTOSAR initiative [AUT] tries to establish standards related to software com-

ponents and interfaces for the automotive industry. Related to AUTOSAR, the ongoing RT-Describe project [RD10] addresses resource aspects. The OMG MARTE [OMG08a] activity provides a standard for modeling and analysis of realtime and embedded systems (including non-functional properties).

The OMG RTC [OMG08b] specification is one of the first initiatives to standardize a robotics component model. In Europe the BRICS component model (BCM) [BKH+13], the PROTEUS RobotML [DKS+12] and SMARTMARS [SSL12] are some of the popular initiatives towards unifying software structures in robotics. We believe that now is the time to explicate and bring together these expertises in a next generation robotics meta-model.

We also believe, that improving only the design-time development process in isolation is insufficient, and it is rather necessary to take run-time aspects like situation- and resource-awareness into account from the very beginning. One step is to use modeling languages for task coordination such as SMACH [BC10]. However, in addition it is necessary to express variability (e.g. with DSLs such as SMARTTCL [SS11] and *VML* [LIRVCS13]) which is purposefully left open at design-time and is bound at run-time taking the then available information about resources, environment, situation, etc., into account. Overall, this improves task execution quality, optimizes robot performance and cleverly arranges complexity and efforts between design-time and run-time.

5 Open Challenges and Future Work

Quality-of-Service: Overall, we consider the step towards a model-driven approach as the only chance to achieve Quality-of-Service (QoS) within robotic systems. Since robotic systems always only have limited resources, all their decisions at run-time are about achieving the required QoS in executing their tasks. More technically, addressing QoS needs to become central within robotics system design. By QoS, you can express whether and where you need hard real-time (and can check based on the model-driven approach at design-time already whether these parts can match their timings), you can also express how reducing the processor share reduces the maximum velocity of your collision avoidance (and thus gives the robot the chance at run-time to properly arrange its resources) and many more.

Black-box handover from one role to the next: Open challenges are related to *variability modeling* (transformation from design-time models to models exploitable at run-time), *resource modeling and QoS modeling* (relationships between the QoS-settings and variability explicated at the black-box component hull, e.g. when the system integrator doubles the maximum speed, he is notified that this e.g. requires three times the processing power etc.).

Link between S/W model (component settings, resources) and robot behavioral model (task nets): There is a subtle relationship between both which is not yet systematically addressed. While *software models* describe the variation points of a software component, the possible settings and the according resource requirements, *robot behavior models* describe how to achieve certain tasks. However, at some point, the robot behavioral model maps

into concrete configurations of the software components. Generic task nets (how to deliver a coffee?) relate to skills (which component wiring and settings implement the move-to behavior on that robot?) and these settings should match the software model. Right now, most of these links are done without any tool support and are thus extremely error-prone.

Workflow for MDSD in robotics: In order to better support separation of roles, we have to move towards a stepwise refinement approach. It is not just a linear workflow from a PIM (platform independent model) over a PSM (platform specific model) to a PSI (platform specific implementation) but it is a workflow from the component developer over the the system integrator to the robot itself. Each role binds variability at the level of a PIM, a PSM and a PSI. In robotics, there is a need for early partial binding of H/W (closed source library requirements, sensor mounting) and there is a need for late binding of the underlying execution platform (operating system, middleware, etc.) as object libraries (in order to allow for delivery of black-boxes without source code, see the generation gap pattern used in the SMARTSOFT templates).

Deployment: Deployment typically comprises the design-time mapping of software components onto a target hardware. At that time, there is a need to balance and trade-off the settings of a software component with the properties offered by the target hardware. Deployment is about mapping and/or matching software resource requirements with hardware platform models. In robotics, this ranges from processor performance requirements to sensors mounted within a specific height interval. Supporting the deployment step in robotics by MDSD is considered as a major challenge.

Acknowledgement

The project work at University of Applied Sciences Ulm is supported by BMBF (German Federal Ministry of Education and Research; grant number 01IM12008B / "iserveU"). The collaboration is jointly funded by the German Academic Exchange Service (DAAD; Project ID: 54365646; Project leader: Prof. Schlegel) and the General Directorate of International Projects of the Spanish Ministry of Economy and Competitiveness (MINECO; Project leader: Dr. Vicente-Chicote; Project ID: PRI-AIBDE-2011-1094). Juan F. Inglés-Romero thanks Fundación Séneca-CARM for a research grant (Exp. 15561/FPI/10).

References

[AUT] AUTOSAR. Automotive Open System Architecture. `http://www.autosar.org/`.

[BC10] Jonathan Boren and Steve Cousins. The SMACH High-Level Executive. *IEEE Robotics Automation Magazine*, 17(4):18 –20, dec. 2010.

[BKH+13] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The BRICS component model: a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1758–1764, New York, NY, USA, 2013. ACM.

[DKS+12] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In *SIMPAR*, pages 149–160, 2012.

[HBK11] Martin Hägele, Nikolaus Blümlein, and Oliver Kleine. EFFIROB-Studie - Wirtschaftlichkeitsanalysen neuartiger Servicerobotik-Anwendungen und ihre Bedeutung für die Robotik-Entwicklung, 2011. http://www.ipa.fraunhofer.de/studien.

[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.

[LIRVCS13] Alex Lotz, Juan F. Inglés-Romero, Cristina Vicente-Chicote, and Christian Schlegel. Managing Run-Time Variability in Robotics Software by Modeling Functional and Non-functional Behavior. In S. Nurcan et. al., editor, *EMMSAD (Exploring Modelling Methods for Systems Analysis and Design)*, LNBIP 147, pages 441–455. Springer, 2013.

[OMG08a] OMG MARTE. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2, ptc/2008-06-08, June 2008.

[OMG08b] OMG RTC. Robotic Technology Component (RTC) Specification 1.0, 2008. `http://www.omg.org/spec/RTC`, May 15th 2011.

[QCG+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[RD10] RT-Describe. Iterative Design Process for Self-Describing Real Time Embedded Software Components, 2010. `http://www.esk.fraunhofer.de/en/projects/RT-Describe.html`, visited on May 15th 2011.

[SS11] A. Steck and C. Schlegel. Managing execution variants in task coordination by exploiting design-time models at run-time. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2064 –2069, sept. 2011.

[SSL12] 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*, pages 119–150. iConcept Press, 2012.