Journal of Software Engineering for Robotics

7(1), July 2016, 3-19
ISSN: 2035-3928

The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software

Dennis Stampfer* Alex Lotz Matthias Lutz Christian Schlegel
University of Applied Sciences Ulm, Department of Computer Science, Prittwitzstr. 10, 89075 Ulm, Germany.
{stampfer,lotz,lutz,schlegel}@hs-ulm.de.

Abstract—Service robots are complex software-intensive systems that need to fulfill a diversity of tasks in open-ended environments. In order to deal with their software complexity, one should make use of the latest software systems engineering principles and tools so that experts and stakeholders can smoothly interact and collaborate.
We argue that there is a lack of approaches addressing the overall integration challenge (i.e. systematic composition) in service robotics and efforts for integration are underestimated. To address this, one of the recent trends in service robotics is the use of Model-Driven Software Development (MDSD) approaches and the creation of dedicated Domain-Specific Languages (DSLs). We further argue that isolated DSLs need to be combined and integrated in order to realize a step change in robotics software development.
We propose the "SmartMDSD Toolchain v2" as an Integrated Development Environment (IDE) for robotics software development. The SmartMDSD Toolchain combines a set of DSLs and tools in one IDE that guides experts and stakeholders through a formalized development process. We report on our vision of a robotics business ecosystem and our basic principles to address this vision. We give a consistent view on the overall development process and its full support in the SmartMDSD Toolchain v2. Finally, we report on lessons learned during six years of experience in developing tools, methods and DSLs for software development for service robots and conclude with a user study to assess the benefits of the SmartMDSD Toolchain as reported by the users.

Index Terms—Service Robots, Domain Specific Languages (DSL), System Integration and Composition, Component-Based Software Engineering, Model Driven Software Development (MDSD)

1 Introduction

SERVICE robots are complex software-intensive systems that need to fulfill tasks in complex environments. In order to deal with their software complexity, it is necessary to split the overall problem into sub-problems that can be solved individually by experts from the particular domain. Component-based software engineering (CBSE) is widely accepted and becoming state-of-the-art for service robotics [1].

We use a service-oriented, component-based approach for our method and envision a robotics business ecosystem where various stakeholders with dedicated expertise network and collaborate in building robot software. To build this software, the stakeholders put together and reuse software-components and other building blocks.

Building software systems at such a level is dependent on the means of composition, the ability to separate roles and concerns, and the support by an according development methodology. We refer to composability as the ability to combine parts to a whole [2]; in our case combining software components to a system.

One of the recent trends in service robotics is the use of Model-Driven Software Development (MDSD) approaches to provide dedicated Domain-Specific Languages (DSLs) for isolated problems such as mobile manipulation, reasoning and planning, and dynamic task coordination [3]. Although we strongly support this trend, which has provided a valuable contribution for robotics, we argue that modeling approaches addressing the overall integration challenge (i.e. systematic composition) are underrepresented and that efforts are underestimated. The need for systematic software development

Regular paper – *Manuscript received August 03, 2015; revised February 05, 2016.*

- *The authors gratefully acknowledge research grants and funding provided by BMBF (robotics-related research: iserveU/01IM12008B, research related to the overall workflow and its application beyond robotics: FIONA/01IS13017C).*

mechanisms and software engineering approaches in robotics has been recognized by the Strategic Research Agenda for Robotics in Europe [4] and was identified as a "make or break" factor within the European SPARC Robotics initiative and the Multi-Annual Roadmap [5] for the development of robots.

In this paper, we address the challenge of integrating individual contributions, from various experts involved in the overall development workflow while, while maintaining system consistency. More precisely, we highlight the challenges involved in integrating and bridging several DSLs within one consistent modeling and development environment and report on our experience in solving these challenges.

Currently, robotics software development uses a variety of individual and dedicated tools: standard tools (editors, compilers) and tools tailored to the domain of robotics. However, isolated toolings and DSLs bring a benefit for an isolated problem, task or role, but cannot address the entire system engineering problem. Furthermore, such a collection of tools and specialized DSLs (each for a specific purpose) requires an *integrated* modeling approach that provides technical workflow-support and supports all involved stakeholders by guiding them through the formalized (overall) development process[1].

In other domains, such as e.g. automotive [6], the use of integrated modeling approaches and tools is widely accepted and have demonstrated their benefit. In robotics, however, such such modeling approaches and its related tools are not yet standard or common practice. Without an integrated approach and supportive toolchain, the overall workflow for robotics development may exist and be defined in documents, but this is not sufficient to properly apply and actually live these processes (i.e. the move from document-driven to model-driven software development). A toolchain reduces the effort of handing over and switching between multiple dedicated modeling tools and can, thanks to DSLs, assist in functions, views and information that are role-specific for stakeholders in each step and each sub-domain.

The presented work builds upon the service-oriented component-based approach SMARTSOFT. While the foundations of SMARTSOFT [7] are still in use today, SMARTSOFT nowadays is an umbrella term for abstract concepts (such as a systematic development methodology, best practices [8]) and implementations (reference implementations, a set of reusable components) to build robotics systems. We propose the "SmartMDSD Toolchain v2" as an Integrated Development Environment (IDE) for robotics software development (Fig. 1). The SmartMDSD Toolchain seamlessly integrates into the world of SMARTSOFT by realizing the concepts SMARTSOFT, thereby making them accessible to its users.

1. We think of "process" as the order and connection of steps on a conceptual level, and "workflow" as the order and connection as it is realized by a tool. However, as the workflow of the SmartMDSD Toolchain is very close to the conceptual process, this paper uses "process" and "workflow" as synonyms.

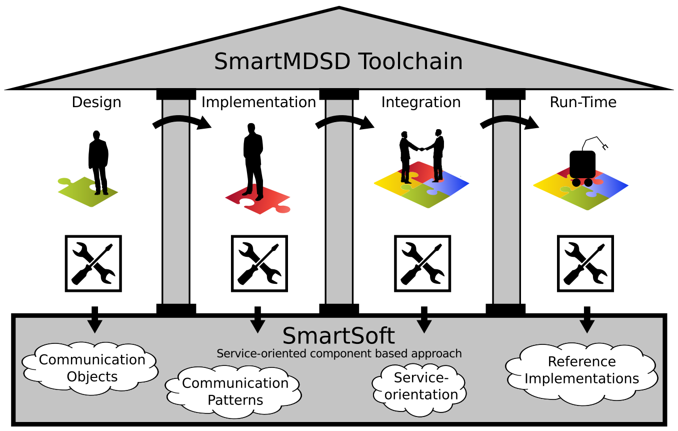


Fig. 1: The SmartMDSD Toolchain is an Integrated Development Environment (IDE) for robotics software development combining a set of dedicated DSLs in one integrated toolchain to guide all stakeholders through the robotics development workflow.

In this paper, we report on the "SmartMDSD Toolchain v2" in its third major generation. Considering the overall complexity of a complete robotics software development workflow and tooling, it is impossible to illustrate each relevant part with all its details in this paper. However, as we believe it is mandatory to have an integrated modeling environment that combines dedicated tools, it is equally important to present research about the relation and global setting between isolated solutions. As such, the focus of this paper is to contribute a consistent view on the overall development workflow, how it is supported by the "SmartMDSD Toolchain v2", and how the steps and outcomes of each step are related to each other. This paper further introduces new specific DSLs (some textual, some graphical) to address the system design step, system configuration and deployment. We describe how these new DSLs and existing DSLs (e.g. SmartTCL) are linked to each other within the toolchain. Where possible, we refer the reader to existing publications related to the first generation of the SmartMDSD Toolchain for more detailed information.

This paper is structured as follows: We first illustrate the basic principles behind the SMARTSOFT approach and workflow used to develop service robots and then present how the SmartMDSD Toolchain supports and guides users in applying these principles through the workflow. In each step, we show excerpts of the development of the collaborative butler scenario [9] where two service robots act as butlers, open cupboards and operate the coffee machine. Finally, the paper concludes with a user study and lessons learned during six years of experience with three generations of the SmartMDSD Toolchain.

2 Related Work

CBSE and MDSD paradigms are applied in various software intensive domains, such as automotive, cyber-physical and embedded-systems. The ever-increasing system and development complexity as well as the huge amount of configuration options drive the need for systematic tool support [6], [10].

Recently, several isolated DSLs for robotics (such as rFSM [11] for task coordination, deployment modeling [12] and architecture modeling [13]) have been presented, each solving one particular problem. However, in order to ensure overall system consistency, these DSLs and models need to be connected. In contrast, our approach is to integrate several dedicated DSLs into one consistent modeling and development environment which we consider crucial for transforming knowledge between and providing adequate representations for the involved roles.

The general lack of systematic integration mechanisms in robotics software development has been identified by the European SPARC Robotics initiative in the Strategic Research Agenda (SRA): “Investment in these technologies is critical to the timely development of products and services and a key enabling factor in the stimulation of a viable robot industry” [4]. The SmartMDSD Toolchain and the ideas behind it directly contribute to this line of research.

ROS [14] provides infrastructure, algorithms, libraries and dedicated isolated tools. To the best of our knowledge, ROS does not provide anything comparable to an IDE which could support the design, implementation and integration of ROS nodes. Instead, any preferred general purpose IDE (e.g. QtCreator) can be used [15]. Design and implementation agreements are solely the responsibility of the user and are not tool-supported. Some user-driven initiatives have attempted to address this, e.g. RIDE [16] and rxDeveloper [17]. RIDE aims to ”make the creation of ROS controllers from reusable components as easily as possible” [16]. Yet, the only provided functionality is to launch and to connect ROS nodes. Similar, rxDeveloper provides a GUI for modifying launch-file parameters of running ROS nodes. While this approach helps in executing the nodes, it does not help in either design or implementation.

The need for systematic tooling and separation of concerns is also applied in BRICS within the ”5Cs” [18] and the BRICS IDE (BRIDE) based on the BRICS Component Model (BCM) [18]. One of the core motivations behind BRIDE and BCM is to harmonize over (i.e. to find a common denominator for) different robotic frameworks. This, however, inevitably results in an oversimplified component model which discards important details from the relevant component models. For instance, the communication between components is defined on a too generic level, leaving too much freedom. Thus component developers can implement conflicting communication characteristics beyond the models. As a consequence, two component implementations that conform to the same BCM model can implement different communication mechanisms and are thus incompatible to other components. Incompatible components break reuse and integration via composition with black-boxes in new architectures. In contrast to BRIDE, the SmartMDSD Toolchain follows a *Freedom from Choice* philosophy [19], which provides sufficient details with respect to service definition, systematic integration and configuration of components in new systems and deployment to targets.

In alignment to Software Product Lines (SPLs), Hyperflex (a BRIDE extension) defines application specific reference architectures based on feature models [13]. In contrast to *application specific* reference architectures, the SmartSoft approach is *application independent*. This is due to its focus on composability (instead of just configurability) which guarantees smooth building block integration based on reusable service definitions.

Robot Technology (RT) Middleware [20] is based on the OMG Robot Technology Component (RTC) standard. The reference implementation OpenRTM-aist includes an Eclipse IDE. RT-Middleware was one of the leading modeling initiatives in robotics around a decade ago and had, at that time, a significant impact on robotics software development. Compared to recent approaches, RT-Middleware has a rather simplified data-flow-like communication model and the Eclipse-based tooling lacks state-of-the-art modeling techniques for structured system integration and deployment.

The PROTEUS project made a significant contribution with RobotML [21] towards addressing the separation of concerns on model level by grouping models in packages for communication, behavior, architecture and deployment. On an abstract model level, we agree with the ideas behind RobotML. In this paper, we additionally provide a toolchain comprising fully integrated model editors, code generators, build infrastructure, execution environment and other elements.

Although tools for collaborative work (e.g. revision control for models) exist, this paper focuses on a consistent model base to which these tools could be applied.

3 Towards a Business Ecosystem for Robotics

The main question behind our research is how to achieve a robotics business ecosystem [22] in which various stakeholders can network and collaborate. In our research, we use the definition of a “business ecosystem” as introduced by Moore [23] and Peltoniemi [24]. In our opinion, all stakeholders have dedicated experience and can contribute software building blocks using their individual expertise. These building blocks (open- or closed-source) need to be combined to new applications just by composing them “as-is” without the need for detailed inspection. Such an ecosystem might exist on a small-scale within an organization or company that builds a collection of standard components for reuse and composition in different applications. Or, on a larger and more open scale, this ecosystem might exist as a market of components where component suppliers provide components and alternatives for composition (selecting components that meet the demands of the application) with different functional implementations (diversity of performance).

The remainder of this section provides a set of basic principles that we consider as key requirements for developing the SmartMDSD Toolchain to advance towards the envisioned robotics business ecosystem.

The exchange of software components in the envisioned business ecosystem must occur at the proper level of abstraction for which the definition of reusable *services* (as e.g. in SmartSoft [8]) is a key element and thus is fundamental for system design. Services are stable architectural entities, used to describe different applications. Services define functional boundaries between building blocks and define their interaction: *how* (communication semantics) and *what* (data structure). Services act as a link between component supplier and system integrator when composing the application from components. Services allow the identification of functionality yet to be covered very early and are the most important building blocks of the system architecture. Services keep the architecture flexible (service-level abstraction) and are aggregated to components (component-level abstractions) that can either provide or require (use) services. As a result, services ensure ensure system level conformance and guarantee that the system can be integrated by composition using components as reusable building blocks.

When searching for patterns and structures that realize the envisioned ecosystem, the choice involves finding the *Sweet Spot between Freedom of Choice and Freedom from Choice* [19]. *Freedom of Choice* means not to enforce any decisions, but has a high price to pay since there is no guidance with respect to composability and system level conformance. Alternatively *Freedom from Choice* provides such guiding structures, for example at a component level, such that these components finally conform to system-level agreements. The sweet spot in between both of the choices relates to supporting as much freedom as possible while still ensuring guidance [8], [25].

Separation of roles [26] is necessary for a successful robotics business ecosystem where stakeholders have mutual benefits, are able to collaborate and to compete. Separation of roles reduces risks, efforts and costs since stakeholders no longer need to be an expert in every field of the application and every step of the workflow; instead, stakeholders can focus on their core contribution and role. A toolchain supporting the ecosystem has to manage the seamless handover from one role to the next. There has to be a way to explicate variation points on the level of a model to allow modifications (stepwise refinement) of building blocks to the needs of the application without, for example, investigating or even modifying source code.

Separation of Concerns is a very basic principle in software engineering [27] and mandatory for robotics. It identifies concerns and separates them by decoupling. SmartSoft and the SmartMDSD Toolchain, for example, achieve separation of concerns by gaining control over the component hull, providing a flexible API inside, and stable communication semantics outside, of the component [26].

Composability and composition are at the heart of an ecosystem. Composability [2] is the ability to assemble components, i.e. to combine parts/software-components to a whole/system. As such, composability is about finding the right abstraction and properties for building blocks so that they can be put together. Composition, as the process of putting together these building blocks, is equally important as means of developing an appropriate workflow and technical foundation.

4 Model Driven Software Development

We are convinced that MDSD based on DSLs is the most suitable technology to realize an integrated toolchain for robotics and that it is capable of steering robotics towards the envisioned business ecosystem for robotics software [22], [28]. It is necessary to provide each involved stakeholder with sufficient freedom. However, especially when several stakeholders work on separated problems, it is also necessary to restrict the design space in some matters in order to allow for successful handover between the roles so that finally the individual parts can be put together in the end. Meta-models explicate structure and guide at the same time (Freedom from Choice). Code-generators simplify the realization of separation of concerns to separate user-code from generated code. Non-expert users will thus gain advantage from expert knowledge inside generators. Domain Specific Languages (DSL) allow for tools, both graphical and textual, tailored to a specific task, view or role.

The SmartMDSD Toolchain is based on Eclipse Modeling Tools [29] and uses graphical and textual DSLs. It uses UML-profiles to implement the SmartSoft robotics meta-model SmartMARS [26] and uses PapyrusUML [30] for graphical modeling. Xtext [31] is used for textual modeling. Additional assistants, validators, checks and glue-logic create the necessary bridges between the DSLs to achieve the overall workflow. The SmartMDSD Toolchain uses graphical modeling, textual modeling or both by bridging between graphical and textual models or by embedding them.

In each step of the workflow (Fig. 2), DSLs are used to create models. The models are used, refined and annotated and handed over to the next step and/or role. Depending on the steps and actions, code is being generated that separates user-implementations from execution containers (component hulls) and data structures. During integration, the system is composed of building blocks (components, behaviors) and all collected artifacts are deployed to the robot for execution. Even during execution at run-time, the robot uses models or directly executes them (e.g. behavior).

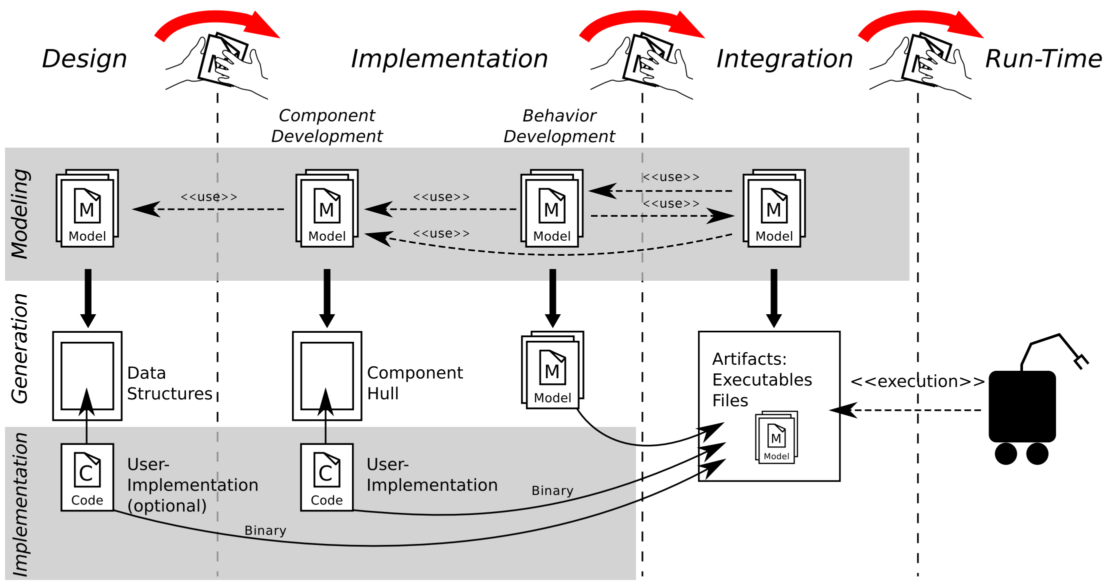


Fig. 2: The internal realization of the SmartMDSD Toolchain. The figure shows artifacts of the toolchain (models and code), their relationships and their handover (red arrows) through the workflow.

Xtend [32] is used for template-based code generation. The generation gap pattern [33] is used to realize a clear separation between generated code originating from the model and code that is implemented by the user. This is done by putting generated code and user code into separate files and classes and linking them by inheritance.

For example, the component developer (cf. Fig. 2) is responsible for (i) defining the component model and (ii) providing the internal business logic (i.e. the implementation of the component). The component model complies with the formal service definitions as elaborated on later in the paper. From the component model, two different code artifact-types are generated according to the generation gap pattern. The first artifact type represents the base classes related to model elements and is automatically regenerated each time the model is changed. The second artifact-type represents the skeleton classes which are generated only once and not re-generated. These skeletons are enriched with business logic by implementations by the component developer. In this way, models and code are always in sync and changes on code level do not compromise the component model.

Not to compromise the component model is by purpose, since changing the component model (e.g. changing services) should trigger new agreements with all involved roles in the related development steps. New agreements might involve a redesign of the according system parts whose overall system-level conformance can only be ensured on the model level, through related model-checks, and thus are separated from code.

5 The SmartMDSD Toolchain

This section describes how the SmartMDSD Toolchain supports and guides users in applying the principles behind SmartSoft. We first introduce an example use-case which is continuously refered to in the subsequent subsections. After introducing the general development workflow of the SmartMDSD Toolchain and the connection between the DSLs and steps, we describe each of them in the development workflow addressing the exemplary use-case. For a more detailed and practical example, we refer to our online video-tutorials [9] showing a complete walk-through through all stages of the development process and the major functionalities of the toolchain.

5.1 Exemplary Use-Case

To illustrate the use of the SmartMDSD Toolchain, we show an excerpt of the development of the collaborative butler scenario [9] where two service robots act as butlers by opening cupboards and operating the coffee machine. More specifically, we address the obstacle avoidance part used in both robots. Obstacle avoidance is realized using the Curvature Distance Lookup (CDL) algorithm [34], which is available as a SmartSoft component *SmartCdlServer* [35].

The CDL algorithm takes a laser scan and the next intermediate waypoint to a target location as input. It calculates the best combination of translational and rotational velocity to steer the robot to the waypoint considering the robots kinematics, dynamics and shape. We will concentrate on the output of the component, which is the velocity. The component provides these velocity values to the system using the communication

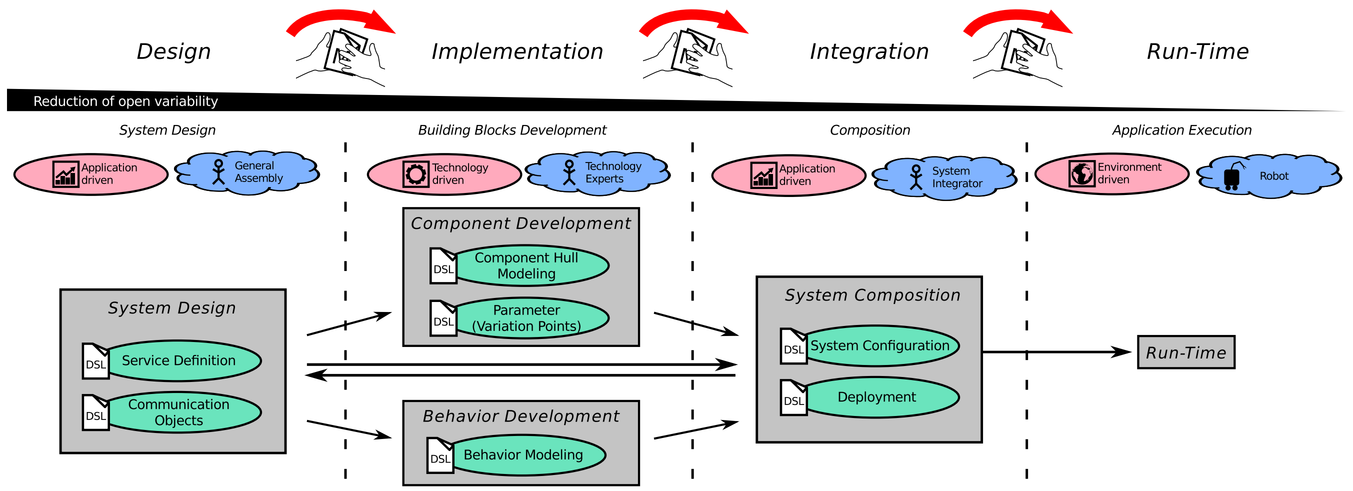


Fig. 3: The connection between development steps, roles and views and the corresponding DSLs that are utilized in each step within the overall workflow of the SmartMDSD Toolchain.

object *CommNavigationVelocity* and the SMARTSOFT send-pattern [36].

There are several parameters to the CDL algorithm and component. We will concentrate on how to limit the velocity to the needs of the application. We will show the development of the *SmartCdlServer* component and how it is finally integrated into the overall system via composition.

5.2 Development Workflow

The realization of the SmartMDSD Toolchain is guided by the superordinate objectives as described in Section 3. The overall development workflow (Fig. 3) starts with system design in which, for example, a general assembly of project representatives starts to define *services* by modeling service definitions. As services are defined with the overall application in mind, their functional boundaries shape the overall architecture. These design decisions (service definitions) are taken as input for component development, focused on technology implementation, and behavior development, focused on coordination to achieve tasks.

System integration is done by composing (putting together) software components. Components are selected such that the services they provide or require match the service definitions of the architecture from the system design step. Integration includes configuring components, adding behaviors and deploying them to the robot.

System design, implementation (component and behavior development) and integration (system composition) are clearly separated as motivated by the robotics business ecosystem. The major outcome of system design is the definition of services. Since these are the stable entities to build the robotics architecture, they are the foundation for both, component development and system integration (composition). As such, alternative components can be provided in the ecosystem or market of components. For example, the definition of a localization service can be used to provide alternatives; e.g. a component providing localization based on laser and another one providing localization based on GPS.

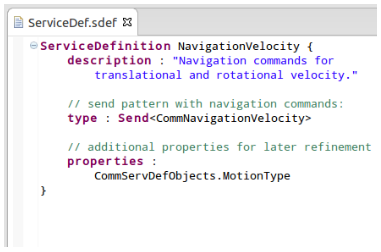
Changes in services have an impact on functional boundaries and thus also have direct influence on the resulting design. Examples for such changes are the need to change a communication pattern, a change in the communication object of a service or a change in the attributes of a communication. Typically, such problems are detected during integration. Changes in services cannot be made arbitrarily by the component developers since they would otherwise break the system architecture in an unauthorized way. Thus, such changes require the mutual agreement of all involved stakeholders (e.g. from the general assembly). These changes are then addressed in system design, going back from system composition to system design as illustrated in Fig. 3. This part of the workflow is ensured by links between models and by the code generation as described in Section 4.

Figure 3 illustrates the relation of development steps within the overall workflow and shows which DSLs are used within each step. However, variations of the overall workflow are possible and depend on the level of reuse; for example, systems might be developed from scratch with no re-use at all or systems might be developed reusing all components and behaviors from an in-house or global ecosystem.

5.3 System Design

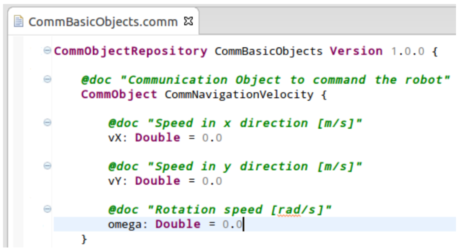
During system design, the general assembly (e.g. representatives of the domain) defines communication objects [8] and services with two according DSLs based on Xtext.

Service definitions are the foundation for later aggregation of services to components. Services become the central elements and building blocks of the application architecture



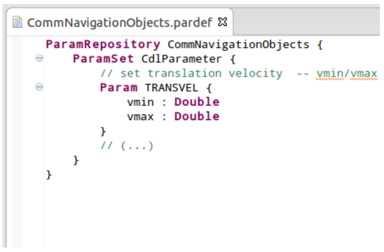
```
ServiceDefinition NavigationVelocity {
    description : "Navigation commands for
        translational and rotational velocity."

    // send pattern with navigation commands:
    type : Send<CommNavigationVelocity>

    // additional properties for later refinement
    properties :
        CommServDefObjects.MotionType
}
```

(a) Service definition model



```
CommObjectRepository CommBasicObjects Version 1.0.0 {

    @doc "Communication Object to command the robot"
    CommObject CommNavigationVelocity {

        @doc "Speed in x direction [m/s]"
        vX: Double = 0.0

        @doc "Speed in y direction [m/s]"
        vY: Double = 0.0

        @doc "Rotation speed [rad/s]"
        omega: Double = 0.0
    }
```

(b) Communication object model



```
ParamRepository CommNavigationObjects {
    ParamSet CdlParameter {
        // set translation velocity  -- vmin/vmax
        Param TRANSVEL {
            vmin : Double
            vmax : Double
        }
        // (...)
    }
}
```

(c) Parameter model

Fig. 4: Service definitions, communication objects and component parameters are modeled using a textual DSL. Human readable documentation is annotated in-line.

and they have an application-driven view and consider the requirements of the final application. For example, a service can be defined for navigation commands, for a laser ranger and for speech input and output. A set of service definitions allows for building a robot architecture based on these services. Depending on the specific needs of the application, components that provide or require these services can later be integrated by composition (system integration step). As such, the services (modeled as service definitions) are the stable architectural entities and can be used to describe different application architectures (architecture instances). Service definitions further narrow the design space and as such apply Freedom from Choice for improved composability.

The scope of design will determine whether services are being defined in a narrow and proprietary way,e.g. targeted for in-house design of a special-purpose robot, or in a more generic reusable way, e.g. towards a standard for navigation for reuse in a component market. This is solely the decision of the users applying this approach.

A *service definition model* (Fig. 4a) comprises a communication object [8] (data structure, Fig. 4b), communication pattern [36] (semantics to transfer data structure) and additional properties describing application-related information of a service (non-technical w.r.t. interaction of components, e.g. localization accuracy, image resolution, language of speech interaction, robot motion type) for later refinement by the component and use during system composition.

Figure 4a shows such a service definition for later use in the *SmartCdlServer* component. This service definition selects a communication pattern (SmartSoft send-pattern) and the communication object *CommNavigationVelocity*. The communication object defines the data structure to be transferred. In this example, the communication object contains values for translational velocities *vX* and *vY* and rotational velocity *omega*. The communication object can be reused in different service definitions. In turn, the service definition can be reused by different components, e.g. for implementing an alternative for the *SmartCdlServer*.

Users describe the model of the services independent of the implementation (separation of concerns), algorithms or internal structure of the component that will later provide or require that service. These models can then be (re)used (e.g. by a component developer thanks to separation of roles), although the target middleware and component implementation is not yet decided; that is, the model is implementable with different kinds of middleware and late-binding of the execution container is possible.

Service definitions are described once for consistent reuse in different applications. Service definitions are used to early identify white spots (services that are not yet provided or required by components) within the architecture. Service definitions decouple component development, behavior development and system integration in time and space. They also guarantee that reusable components fit together during integration, since one can rely on service definitions.

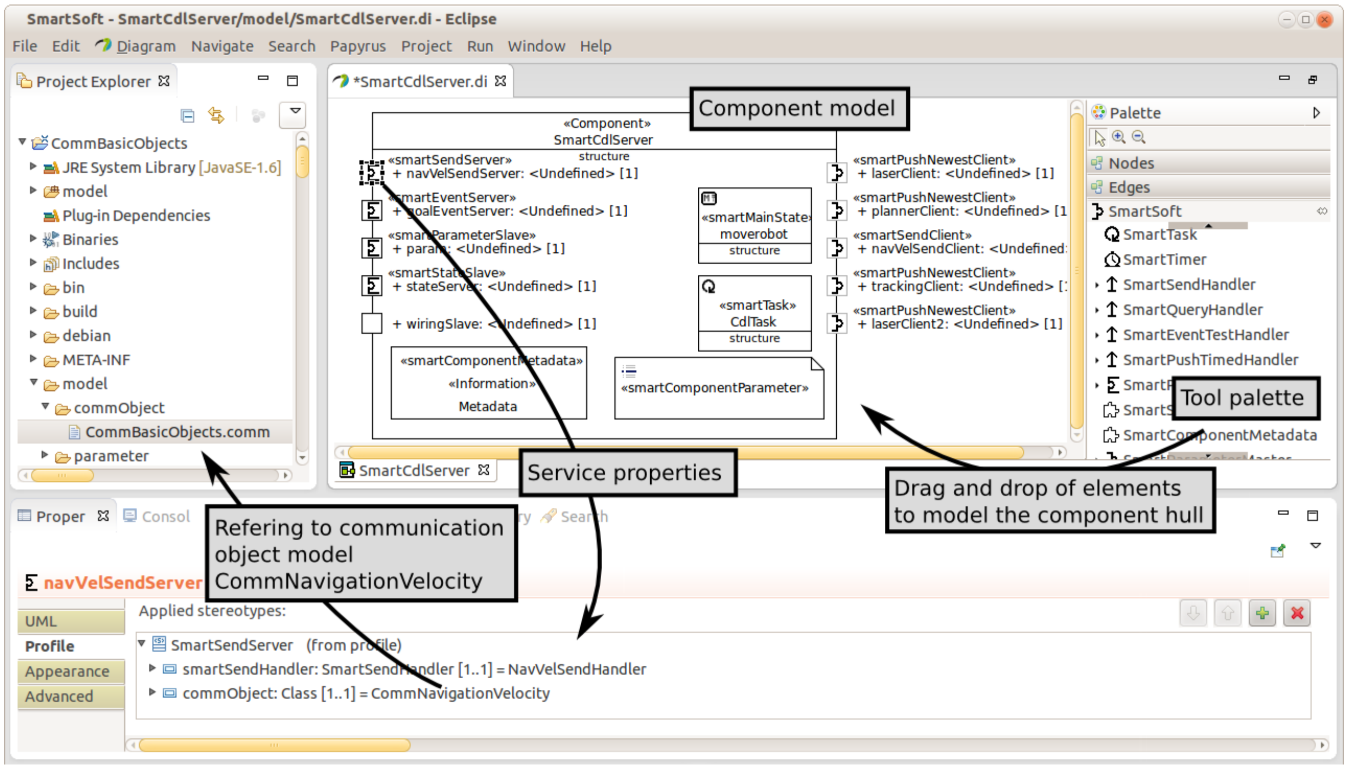
5.4 Component Development and Implementation

Component development provides a technology-driven view on a concrete sub-problem and uses a graphical DSL to model the component hull that meets one or more service definitions (Fig. 5). Component modeling is based on the SmartMARS meta-model which is implemented as a UML profile.

Modeled services refer to communication objects from service definitions. Variation points are modeled and purposefully left open for later refinement by configuration using a new parameter DSL [37] (Fig. 4c) which is implemented in Xtext. Simplified, the result is similar to explicating variables that are accessible from outside of the component.

The code generator generates structures and interfaces for the implementation of user-code, such as implementing algorithms and reusing libraries, as well as a middleware-independent interface and execution container (Fig. 2). Using other Eclipse plugins, e.g. CDT, the component developer can add business logic to the component and implement, for example, algorithms, glue code or reuse libraries.

Figure 4c, shows a variation point *CdlParameter.TRANSVEL* for the range of allowed translational velocity values *vmin* and *vmax*. This variation point definition (parameter) and the service definition *NavigationVelocity* with the communication object *CommNavigationVelocity* are reused for modeling the *SmartCdlServer* component (Fig. 5). Figure 6 shows


SmartSoft - SmartCdlServer/model/SmartCdlServer.di - Eclipse
File Edit Diagram Navigate Search Papyrus Project Run Window Help
Project Explorer
*SmartCdlServer.di
Component model
Palette
Nodes
Edges
SmartSoft
Tool palette
Service properties
Refering to communication object model CommNavigationVelocity
Drag and drop of elements to model the component hull
«Component» SmartCdlServer
navVelSendServer
UML
Profile
Appearance
Advanced

Applied stereotypes:
SmartSendServer (from profile)
smartSendHandler: SmartSendHandler [1..1] = NavVelSendHandler
commObject: Class [1..1] = CommNavigationVelocity

Fig. 5: View of a component developer. Services refer to, thereby reusing, communication objects. This screenshot shows the current public release of the SmartMDSD Toolchain. Here, the service refers to the communication object *CommNavigationVelocity* and communication pattern *SmartSend*. Our current internal release uses the properties to reference the service definition *NavigationVelocity* (see Fig. 4a).

the implementation of the *SmartCdlServer* component (more specifically, its *CdlTask* C++ class) and how to use a service and variation points from within the component implementation. In this example: limiting the velocity using *CdlParameter.TRANSVEL* and sending it via a send service and communication object *CommNavigationVelocity*.

A documentation DSL, implemented with Xtext, is provided within a documentation view (Fig. 8). The textual DSL can be used to annotate the graphical component model with documentation with the purpose of adding semantic documentation. An extensive human-readable specification and documentation is generated from the documentation model and the component model, which is thus consistent and ensures that all relevant information is up-to-date and available for later system integration.

A special diagnose service [38] allows embedding a monitoring infrastructure into the component for later access (at run-time) by specialized monitors in order to observe and to check internal states and configurations of that component.

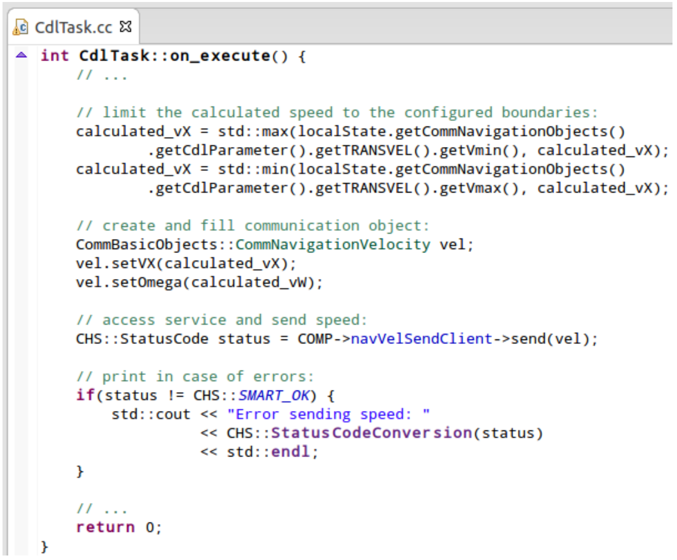
The output of component development is the component model and the component implementation. The component model is used for composition of components in the system configuration model. The component implementation can be in the form of source-code or binary.

Thanks to the separation of concerns, the component developer does not need to think about communication details; these are hidden within the execution container to be linked at a later time. Freedom from choice limits the access to defined services, but the inside view (implementation) is still flexible. Using the parameter DSL, the component developer can explicate variation points to be refined in the system configuration and via behavior execution at run-time. Both the system configuration and behavior see the components as black-boxes with explicated variation points thanks to separation of roles.

5.5 Behavior Development

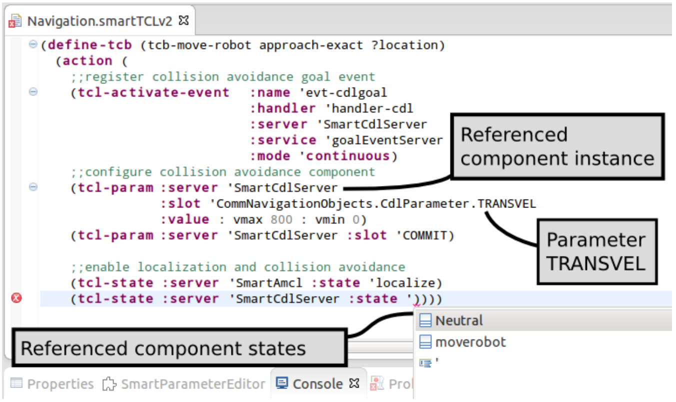
Software components in a robotic system have to be coordinated (configured, activated and deactivated) to make the robot perform complex tasks or behaviors [39]. In addition to an initial configuration, the software components also have to be configured at run-time to cope with different tasks and situations originating from the open ended environment in which the robot operates. Behavior models are used as formal representations of the desired robot behavior and describe how to achieve a certain task.

The user textually models behavior using the SmartTCL DSL and reuses already existing task blocks to model new



```
int CdlTask::on_execute() {
    // ...

    // limit the calculated speed to the configured boundaries:
    calculated_vX = std::max(localState.getCommNavigationObjects()
            .getCdlParameter().getTRANSVEL().getVmin(), calculated_vX);
    calculated_vX = std::min(localState.getCommNavigationObjects()
            .getCdlParameter().getTRANSVEL().getVmax(), calculated_vX);

    // create and fill communication object:
    CommBasicObjects::CommNavigationVelocity vel;
    vel.setVX(calculated_vX);
    vel.setOmega(calculated_vW);

    // access service and send speed:
    CHS::StatusCode status = COMP->navVelSendClient->send(vel);

    // print in case of errors:
    if(status != CHS::SMART_OK) {
        std::cout << "Error sending speed: "
                  << CHS::StatusCodeConversion(status)
                  << std::endl;
    }

    // ...
    return 0;
}
```

Fig. 6: The component developer implements and uses the components' services while considering the left open variation points.



```
(define-tcb (tcb-move-robot approach-exact ?location)
  (action (
    ;;register collision avoidance goal event
    (tcl-activate-event  :name 'evt-cdlgoal
                         :handler 'handler-cdl
                         :server 'SmartCdlServer
                         :service 'goalEventServer
                         :mode 'continuous)
    ;;configure collision avoidance component
    (tcl-param :server 'SmartCdlServer
               :slot 'CommNavigationObjects.CdlParameter.TRANSVEL
               :value : vmax 800 : vmin 0)
    (tcl-param :server 'SmartCdlServer :slot 'COMMIT)

    ;;enable localization and collision avoidance
    (tcl-state :server 'SmartAmcl :state 'localize)
    (tcl-state :server 'SmartCdlServer :state '))))
```

Fig. 7: Behavior development using the SmartTCL DSL. The editor assists with error checks and auto completion providing access to the variation points of the component model as explicated by the component developer using the parameter DSL.

behavior blocks (Fig. 7). The DSL extends SmartTCL [39] and is implemented using Xtext. At some point, the behavioral model maps into concrete configurations of the software components using the variation points as defined by the component developer. For example, the robots velocity-limits may be adapted according to the situation at run-time, e.g. to drive faster in long hallways. New variation points concerning *variability in operation* are introduced, which will be bound at run-time. The SmartMDSD Toolchain provides support via model checks, autocompletion and context-sensitive help thanks to the direct linking between behavior, component and system configuration models.

Figure 7 shows an example of a behavior model using the SmartTCL DSL editor within the toolchain. It uses the *CdlParameter.TRANSVEL* variation point to set the values for *vmin* and *vmax* for the *SmartCdlServer* at run-time.

In this step, the components can be seen as black-boxes, with left open variation points (separation of roles). The user is able to focus on task coordination and is relieved from dealing with low-level processing (separation of concerns). The behavior blocks are composable and their final expansion is performed at run-time given the current context and situation (composability and variability in operation). The reusable task coordination blocks describe the robot behavior at different levels of abstraction. Domain experts, who implement new robotic behaviors, use their knowledge about the application domain and reuse higher level behavior blocks to implement new robotic behaviors (separation of roles).

The result of this step are behavior models bound to a concrete system in the system configuration model that are later deployed to the robot for execution at run-time.

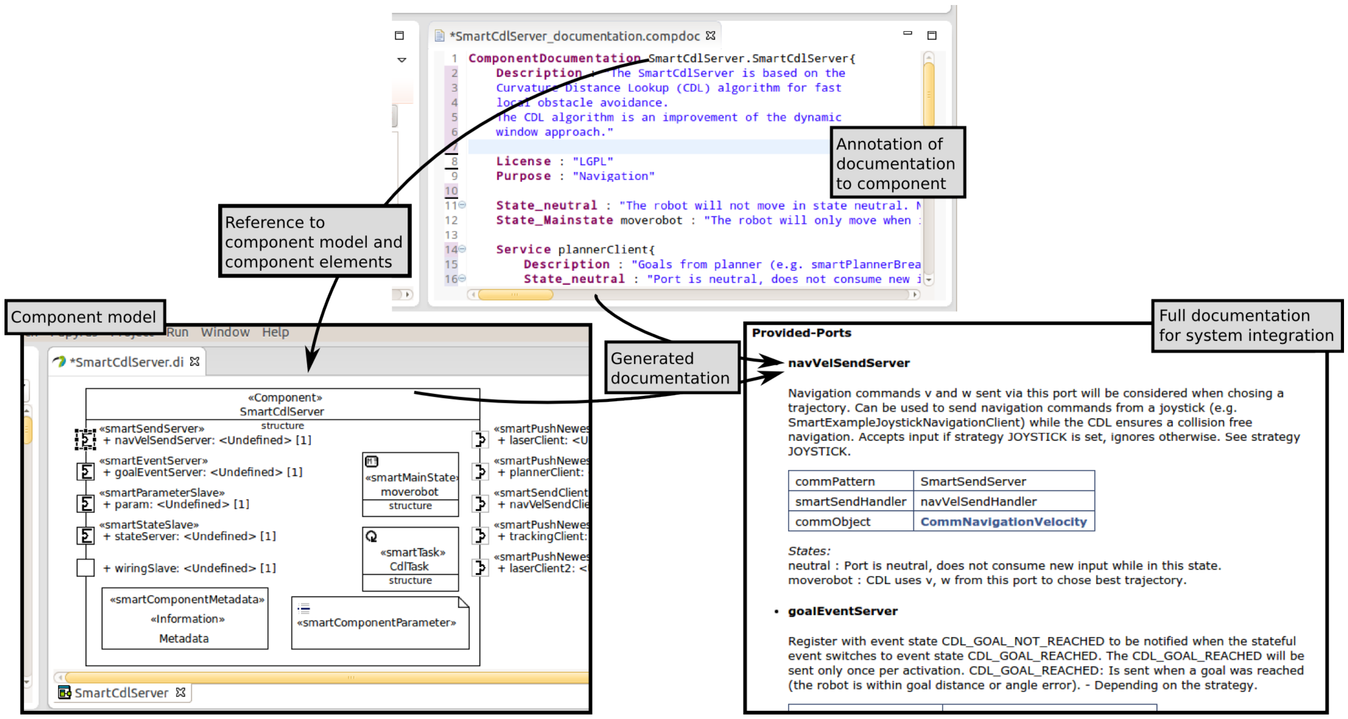
5.6 System Configuration

System configuration is an application-driven step that provides a software-view (Fig. 9) on the application to the user (system integrator). The main purpose of the system configuration is to compose the application by re-using and putting together software building blocks such as components and behavior models which were previously developed or come from a 3rd party or ecosystem. The result is the complete software system ready for deployment modeling and the actual deployment to the robot.

Components are seen as black-boxes with the internal structure and implementation hidden since these internal details are not of interest to the system integrator (separation of roles). Only the outer view on the hull (services) and the explicated configurations (variation points) are presented, due to separation of concerns, but cannot be extended; an example of applying Freedom from Choice in which only existing variation points are used.

The user graphically creates instances of components and initial wirings (Fig. 9) using a DSL based on a UML profile. Instances allow using the same component multiple times (e.g. for a front and back laser) with different configurations. Since components were not necessarily developed with the concrete application in mind, a component supplier does not need not to know in what application the component is used. Therefore, components can be configured by textually editing, thus refining their variation points using the parameter DSL from within the graphical model. However, some variation points are purposefully left open for run-time refinement. As part of instance configurations, additional files (e.g. map) and start/stop-scripts (e.g. for daemons) may be associated with the component instances.

Figure 9 shows how instances of the *SmartCdlServer* and other components are composed to the final application.



```
ComponentDocumentation SmartCdlServer.SmartCdlServer{
    Description : "The SmartCdlServer is based on the
    Curvature Distance Lookup (CDL) algorithm for fast
    local obstacle avoidance.
    The CDL algorithm is an improvement of the dynamic
    window approach."

    License : "LGPL"
    Purpose : "Navigation"

    State_neutral : "The robot will not move in state neutral. N
    State_Mainstate moverobot : "The robot will only move when :

    Service plannerClient{
        Description : "Goals from planner (e.g. smartPlannerBrea
        State_neutral : "Port is neutral, does not consume new i
```

Provided-Ports

navVelSendServer

Navigation commands v and w sent via this port will be considered when chosing a trajectory. Can be used to send navigation commands from a joystick (e.g. SmartExampleJoystickNavigationClient) while the CDL ensures a collision free navigation. Accepts input if strategy JOYSTICK is set, ignores otherwise. See strategy JOYSTICK.

commPattern	SmartSendServer
smartSendHandler	navVelSendHandler
commObject	**CommNavigationVelocity**

States:
neutral : Port is neutral, does not consume new input while in this state.
moverobot : CDL uses v, w from this port to chose best trajectory.

- **goalEventServer**

Register with event state CDL_GOAL_NOT_REACHED to be notified when the stateful event switches to event state CDL_GOAL_REACHED. The CDL_GOAL_REACHED will be sent only once per activation. CDL_GOAL_REACHED: Is sent when a goal was reached (the robot is within goal distance or angle error). - Depending on the strategy.

Fig. 8: The textual documentation DSL (top) as seen from component implementation. This DSL references elements of the component model (left) in order to annotate these elements with human-readable information about component use and semantics. Information from the documentation and component model is transformed to a complete documentation (right) for later system integration which assists the system integrator during composition (Fig. 9, lower left).

The variation point (parameter) *CdlParameter.TRANSVEL* of *SmartCdlServer* can be configured to initial values that match the application context. At run-time, these values can be further refined using SmartTCL (cf. Fig. 7).

The SmartMDSD Toolchain assists in choosing the right components, checking for valid configurations, valid wirings, compatible components and satisfied services according to service definitions before deploying and executing the scenario.

5.7 System Deployment

System deployment comprises mapping of software components onto target hardware, transferring files and starting the application. A deployment model (Fig. 10) provides the view for system integrators on hardware in terms of processing units that can run components. The deployment model models hardware and maps instances of software components onto it. Deployment is considered to be the handover from design-time to run-time.

In this example, the deployment model (Fig. 10) deploys the component instance of *SmartCdlServer* along with other instances, such as mapper, planner and laser to a single computer that runs the component instances.

The deployment model is realized with a graphical DSL (Fig. 10) using an UML profile and consists of devices representing computers (*SmartDevice*) and artifacts refering to component instances (*SmartArtifact*). Devices have several basic properties such as network configuration. The SmartMDSD Toolchain is able to deploy to multiple devices. The toolchain generates the execution container depending on the middleware, packs all artifacts (such as binaries, behavior models, additional files and start/stop hooks) and actually transfers (deploys) them to the robot. Finally, the toolchain optionally starts the application.

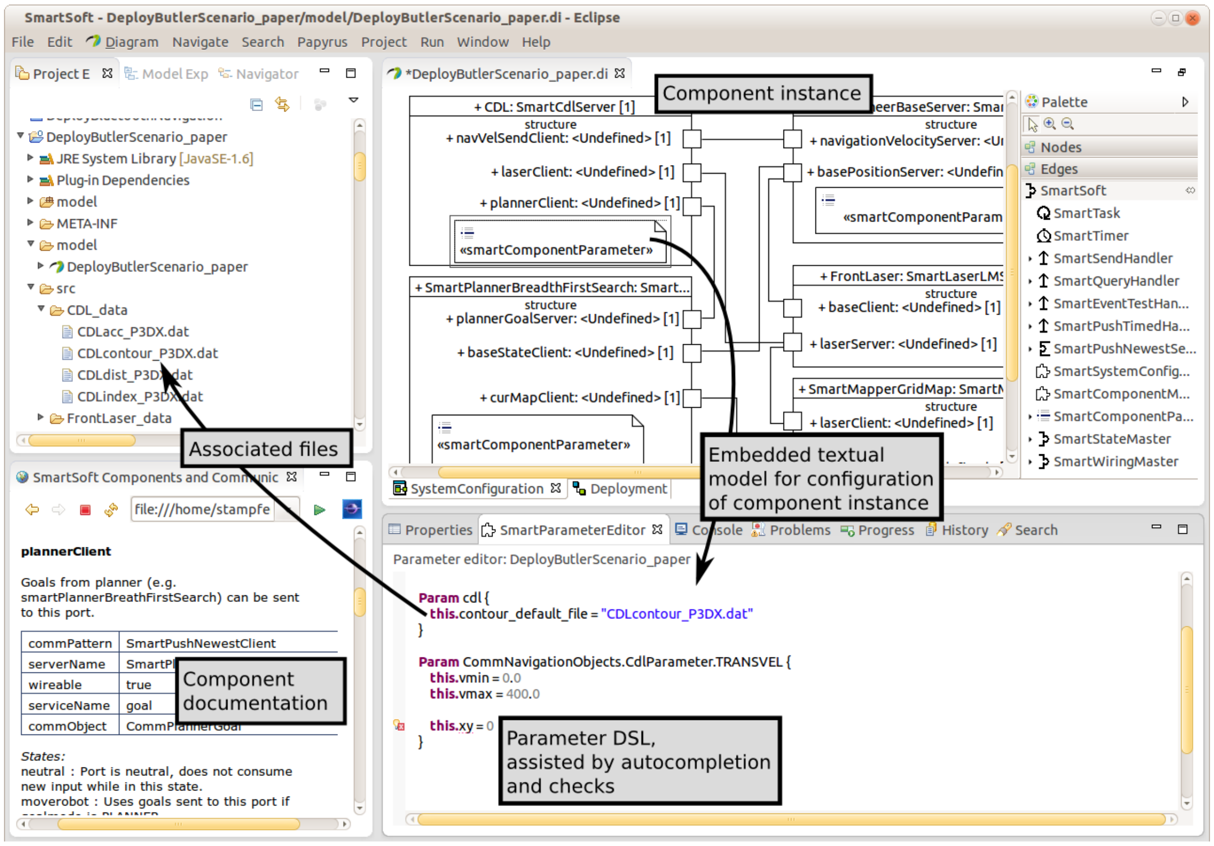
The collection of deployed artifacts includes everything required to run the application. There is no further dependency to the SmartMDSD Toolchain and the deployed scenario is able to run without the toolchain.

The system configuration and the deployment view are separated (roles and concerns), which allows focussing on the hardware-view in the deployment step. This step can only map but not modify the software architecture (Freedom from Choice).

5.8 Run-Time

At run-time the robot executes component instances and behavior models. The robot is considered an active role as it binds left-open variation points to deal with variants and contingencies of an open-ended environment.

Design-time models are used to adapt the behavior of the robot with respect to variability in operation and quality [28].


SmartSoft - DeployButlerScenario_paper/model/DeployButlerScenario_paper.di - Eclipse
File Edit Diagram Navigate Search Papyrus Project Run Window Help
Project E
Model Exp
Navigator
DeployButlerScenario_paper
JRE System Library [JavaSE-1.6]
Plug-in Dependencies
model
META-INF
model
DeployButlerScenario_paper
src
CDL_data
CDLacc_P3DX.dat
CDLcontour_P3DX.dat
CDLdist_P3DX.dat
CDLindex_P3DX.dat
FrontLaser_data
Associated files
*DeployButlerScenario_paper.di
Component instance
+ CDL: SmartCdlServer [1]
structure
+ navVelSendClient: <Undefined> [1]
+ laserClient: <Undefined> [1]
+ plannerClient: <Undefined> [1]
«smartComponentParameter»
+ SmartPlannerBreadthFirstSearch: Smart...
+ plannerGoalServer: <Undefined> [1]
+ baseStateClient: <Undefined> [1]
+ curMapClient: <Undefined> [1]
«smartComponentParameter»
+ navigationVelocityServer: <U
+ basePositionServer: <Undefin
«smartComponentParam
+ FrontLaser: SmartLaserLMS
+ baseClient: <Undefined> [1]
+ laserServer: <Undefined> [1]
+ SmartMapperGridMap: SmartM
+ laserClient: <Undefined> [1]
Embedded textual model for configuration of component instance
Palette
Nodes
Edges
SmartSoft
SmartTask
SmartTimer
SmartSendHandler
SmartQueryHandler
SmartEventTestHan...
SmartPushTimedHa...
SmartPushNewestSe...
SmartSystemConfig...
SmartComponentM...
SmartComponentPa...
SmartStateMaster
SmartWiringMaster
SystemConfiguration
Deployment
SmartSoft Components and Communic
file:///home/stampfe
Component documentation
Properties
SmartParameterEditor
Console
Problems
Progress
History
Search
Parameter editor: DeployButlerScenario_paper
Parameter DSL, assisted by autocompletion and checks

plannerClient

Goals from planner (e.g. smartPlannerBreathFirstSearch) can be sent to this port.

commPattern	SmartPushNewestClient
serverName	SmartPl
wireable	true
serviceName	goal
commObject	CommPlannerGoal

States:
neutral : Port is neutral, does not consume new input while in this state.
moverobot : Uses goals sent to this port if

```
Param cdl {
  this.contour_default_file = "CDLcontour_P3DX.dat"
}

Param CommNavigationObjects.CdlParameter.TRANSVEL {
  this.vmin = 0.0
  this.vmax = 400.0

  this.xy = 0
}
```

Fig. 9: Graphical and textual DSLs are used for system configuration where instances of reusable components can be created and configured to match the needs of the application. Additional documentation of the component is presented to the system integrator depending on the context of selection (clicking an element of the system configuration shows the relevant documentation).

For example, a robot should drive slowly while transporting coffee, but still fast enough to deliver it hot. The effort spent in object recognition is another example and depends on the required classification confidence; which can be low for juice flavour vs. high for medicine [40].

Monitoring is used to observe the internal states and configurations of component instances at run-time. Monitoring makes it easy to trace the cause of errors by analyzing the causal dependencies between components interacting in data-flow and functional chains.

6 Lessons Learned

The SmartMDSD Toolchain and MDSD approaches in general require meta-models. However, meta-models require stable structures and are dependent on precise semantics to be of value. As such, we agree with E. A. Lee that “the semantics of a modeling language is the foundation for the models. Weak foundations result in less useful models.” [19] For our approach, the underlying SmartSoft approach provides such structures and semantics. However, other approaches (such as ROS) need to specify these structures and semantics in order to make them accessible to useful MDSD approaches. For example, ROS misses clear semantics for topics with respect to synchronous/asynchronous communication, timings or buffers.

Developing the DSLs and the SmartMDSD Toolchain, applying MDSD and deciding on user-interface and presentation of toolings is always a trade-off between several factors. A set of superordinate objectives as described in Section 3 not only provides guidance in system design but also provides guidance for these trade-offs in designing the DSLs and the toolchain.

The SmartMDSD Toolchain provides a powerful IDE for users. By applying Freedom from Choice, we restrict the design space purposely to ensure the overall system level conformance. Since the SmartMDSD Toolchain builds upon the standard Eclipse world tooling, the developers can still make use of the tools provided by Eclipse including other plugins (e.g. CDT) and can use any library or programming paradigm. It is even possible to use the code generator without the toolchain or implement components using any other tool.

Based on our experience, MDSD and the use of DSLs have many times demonstrated potential as tools to enable the overall vision of a robotics business ecosystem. Whether

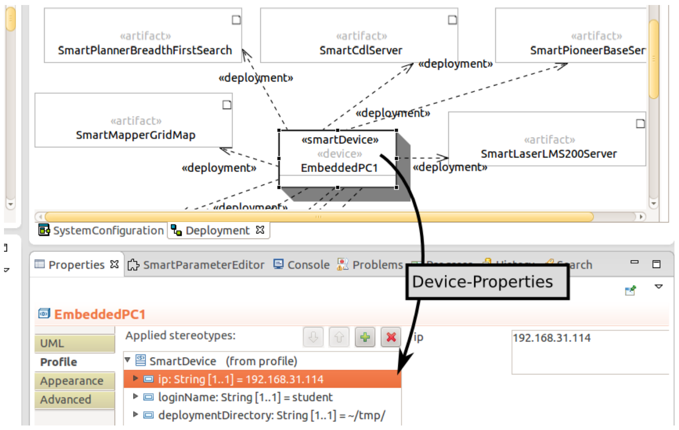

«artifact» SmartPlannerBreadthFirstSearch
«artifact» SmartCdlServer
«artifact» SmartPioneerBaseSer
«deployment»
«artifact» SmartMapperGridMap
«smartDevice» «device» EmbeddedPC1
«artifact» SmartLaserLMS200Server
SystemConfiguration
Deployment
Properties
SmartParameterEditor
Console
Problems
Device-Properties
EmbeddedPC1
UML
Profile
Appearance
Advanced
Applied stereotypes:
SmartDevice (from profile)
ip: String [1..1] = 192.168.31.114
loginName: String [1..1] = student
deploymentDirectory: String [1..1] = ~/tmp/
ip
192.168.31.114

Fig. 10: Deployment model. Artifacts represent instances of software components that can be deployed to one or many target devices. This model maps software components to hardware devices.

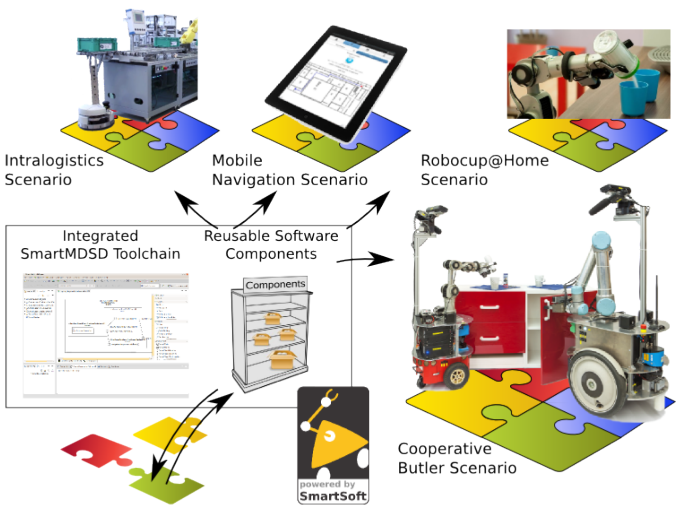

Intralogistics Scenario
Mobile Navigation Scenario
Robocup@Home Scenario
Integrated SmartMDSD Toolchain
Reusable Software Components
Components
powered by SmartSoft
Cooperative Butler Scenario

Fig. 11: The SmartMDSD Toolchain has been used by many partners and projects across several domains and on several robots.

a graphical or a textual DSL is used - whatever is more appropriate in the context and role should be preferred. DSLs should be user-focused, simple, compact and specific for a particular need or role to allow the user to focus on one specific task only. The SmartMDSD Toolchain integrates separated but specific DSLs instead of trying to merge everything into a single DSL. DSLs should be guided by the domain (sub-workflow and role) and its requirements. Doing so comprises e.g. early agreements on contracts between building blocks (components) and responsibilities (service definitions). Late modifications of models should trigger the need for mutual agreements, thus ensuring obligatory workflows through models and tooling. DSLs should enable documentation within models in order to provide up-to-date documentation; that is, use models as documentation or generated documentation instead of free form text as documentation.

Seamless transition between workflow steps and seamless access across models are important: access to textual modeling should be possible from within graphical models, not requiring to manually open a separate text document, and bidirectional seamless access within different DSLs to information shared in other models.

While isolated tools, methods or DSLs can be powerful for their individual purpose, the real step-change towards successful software development for robotics lies in the integration of DSLs or tools into a consistent overall workflow. Therefore, it is our opinion that robotics should not come up with yet another isolated tool or more isolated DSLs but rather address the challenge of fitting them seamlessly into the overall workflow and tooling.

However, when integrating DSLs or modeling tools, it is not only about the tooling and glue-code. The models and DSLs have to come up with the correct structures that allow for integrating them.

In the end, it will probably always be possible to break out of structures. However, the overall goal therefore must be to provide structures that do not require breaking out but still ensure system level conformance.

7 Results

Over the last six years, there was a total of 19 public releases [35] of the SmartMDSD Toolchain under an open source license. The technical realization of service definitions and behavior modeling are the most recent extensions of the toolchain and are not yet within the productive public release. However, all other parts of the toolchain as described in this paper are included in the current public releases and were actively used in our research collaborations. The concept of service definitions and the overall workflow was applied within projects as detailed subsequently. To date, 36 components are publicly available [35]. Including non-public components, we have about 64 components available for immediate composition.

We first summarize our observations on the benefits and experiences of using and applying the SmartMDSD Toolchain in three research projects, four research collaborations and other activities that resulted in robots delivering coffee, opening cupboards, playing connect-four, logistics applications and other scenarios [9]. We then show results of a user study in ongoing research projects which assessed how real users perceive the usability and benefit in using the presented work.

7.1 Applications of SmartSoft

Since the early days, the SmartMDSD Toolchain has been used in different domains for robotics systems engineering (Fig. 11) and other applications. The toolchain has been "demonstrated in operational environments" (technology readiness level 6 according to [5] as acknowledged in [41]).

Within the research projects ZAFH Servicerobotik [42], iserveU [43] and FIONA [44], there was a broad range of users with different levels of experience ranging from no robotics expertise to highly skilled experts with broad system knowledge. In all cases, the SMARTSOFT approach provided valuable guidelines to structure the work within the projects which was a key contribution towards successful system architectures, integrations and project demonstrators (videos at [9]). The SmartMDSD Toolchain provided easy access to the structures of SMARTSOFT.

Within the research project FIONA, SMARTSOFT and the toolchain structured the path towards a project-internal ecosystem with 22 software components provided by nine project partners distributed across Europe. These 22 components, that include several alternatives such as five alternative solutions for localization, allow the composition of over 60 variants of the final project demonstrator which was a smartphone navigator and a navigator for the visually impaired.

Within the research project iserveU, SMARTSOFT and the toolchain were used as a central integration and software architecture approach to realize the distributed development and integration of the outcomes of seven project partners in Germany. In this project, 14 components were successfully integrated into one final project demonstrator for hospital logistics.

SMARTSOFT and the SmartMDSD Toolchain were also applied in education for lectures and for seven generations of the Rococup@Home student project. In every generation of Robocup@Home, a team of five to twelve students worked towards participating in the German Open competition. Each team took over the software components and robot platform of the previous team. Each team added new components and abilities to conform to the evolving rules of the competition. This worked, although there was no overlap in team members and no transfer of knowledge in person. Since the student participants had no prior robotics knowledge and limited time besides their lectures, they had to rely on the black-box approach of components and were able to focus on their particular extension or task. The SMARTSOFT approach and the SmartMDSD Toolchain provided the necessary abstraction and tooling to reuse, compose, modify and continue the complex robotic system. Notably, one student team was able to reuse existing components to move to a different robot platform as detailed in [22].

SMARTSOFT, the toolchain and components are also officially available for the Robotino platform of Festo Didactic [45] for intralogistics applications.

Especially where complexity is managed by splitting the overall problem into sub-problems to be solved individually by experts, collaboration to ensure the smooth transition and handover between stakeholders can be supported with an IDE that guides through the development process. This is valid in activities ranging from laboratory prototypes to in-house product development to the robotics business ecosystem. Enhanced tool-support raises the software development from a document-driven to a model-driven approach, speeds up the process and leads to lower time to market. Furthermore, enhanced tool-support as described in this paper improves the development process by reducing and preventing errors, thereby improving the robustness of the robot itself.

The collaborative butler scenario, of which a part was used as an example in this paper, includes two service robots "Kate" and "Larry". Using the described approach, 18 of 22 components (82%) are the same, re-used "as-is", between Larry and Kate [22]. In lines of code using SLOCCount [46], this makes 42000 lines of user-code and 6700 lines of generated code identical (85% in total). Only the components for the specific hardware had to be exchanged (e.g. Pioneer vs Segway base, Katana vs UR5 manipulator and various types of laser scanners). Thanks to stable services, other components have been reused as-is and the overall service-architecture was not touched.

7.2 User Study

A user study was conducted to evaluate the experience in using the SmartMDSD Toolchain, in applying the proposed workflow and to evaluate the perceived benefits among active users. This section will describe the initial results of this user study.

We designed a questionnaire totaling 44 questions of which 30 questions related to the core topic, the other questions for personal background and skills. It took about 20-30 minutes to answer all questions. Results of the questionnaire were collected anonymously. For each question, the users were asked to rate a given statement based on their level of agreement. The questions were designed in five-level Likert [47] items with answers ranging from "strongly agree" to "neutral" to "strongly disagree" and similar (Fig. 12). The participants were asked to skip a question, giving no answer, if the question was not understood.

The user study was conducted with our partners in current research collaborations. Although there were few participants (18 responses), the user study can be considered representative since it is ensured that all participants had spent a good amount of time working with the proposed tools and methods and therefore can give a valuable evaluation.

The user study was organized in several sections, starting by collecting personal experience to set answers in a broader context. Then, follow-up questions specifically assessed the experiences and benefits perceived with the concepts in general and how the SmartMDSD Toolchain helped in applying and using these concepts. Most importantly, we asked about the experience during integration of the project demonstrator during integration meetings.

Participants: All 18 participants were using SMARTSOFT and the SmartMDSD Toolchain for the first time within their

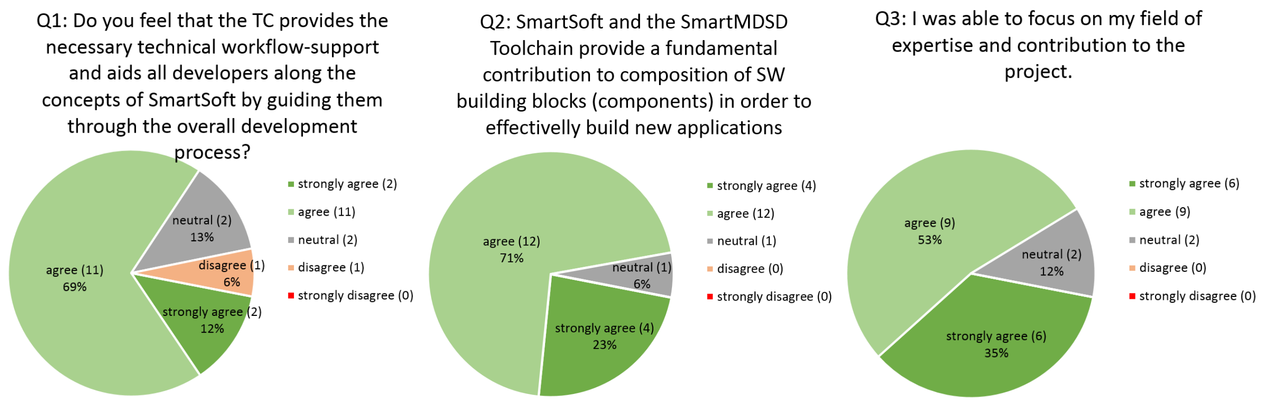


Fig. 12: Excerpts of the user study.

research project. About half of them were from industry (45%) or academia (55%). They work in domains such as embedded systems, automotive, robotics, artificial intelligence, sensors and control. Our users are quite experienced with the kind of system development as undertaken in these research projects: The majority is very experienced with software engineering (55%) but have little experience (60% with less than a year) in using software models or model driven methods. Almost all participants are familiar with Eclipse-based IDEs (94% with more than 1-2 years of experience).

When asked about the development methodology and tools that the participants used prior to the SmartMDSD Toolchain, we observed that there is a very heterogeneous set of separated development tools and methods of integration. The user study revealed that using an integrated IDE for development is not yet standard and that the integration methodology is focused on class-based integration. Reuse is made on the level of libraries, but very few component-based approaches are applied.

Usability: Most of the participants use the SmartMDSD Toolchain for developing components (94%). They use the functionality they expect from IDEs, such as auto-completion, syntax-checks and warnings on all levels, collected documentation, role-specific views, execution and debugging of the robotics application. As such, 82% state they "can work productively using the toolchain".

The seamless combination of otherwise separated tools and the seamless transition including handover of artifacts (models) between development steps and roles is easy for most participants (81%). Also, 81% of participants (see Q1 in Fig. 12) agreed that "the toolchain provides the necessary technical workflow-support and aids all developers along the concepts of SmartSoft by guiding them through the overall development process."

The adequateness of textual and graphical modeling is not easy to find and often a controversial question in MDSD. When asked about the adequateness of textual and graphical modeling for the workflow steps in our toolchain (system design, component modeling, configuration, composition and deployment modeling), all participants in general observe it as adequate, although, there is a slight shift towards "more graphical" modeling.

The participants especially liked the fast development of working applications. Participants noted: "being able to have a prototype working in short time" and "Fast project development, in all phases, is something [we are] looking for."

Integration and Composition: SmartSoft and the SmartMDSD Toolchain are a "fundamental contribution to the composition of software building blocks (components) in order to effectively build new applications" (94%, see Q2 in Fig. 12).

For most (67%) of the participants, it "was possible to compose software components to applications without further inspection or even reading source code. The information given through the models is determined to be sufficient for the interface descriptions". Only 12% disagreed with this statement.

All participants were able to compose new applications from existing components. 56% rated this benefit with "high benefit", 45% as "beneficial". Asked for the effort for composition, only two participants felt that the effort is too high.

Benefits: As with many software development approaches, it takes effort to get used to tools and methods. 56% of the participants needed several weeks while 33% needed several days to get used to it. However, the benefit as perceived by the participants is rewarding: 70% stated that the "initial effort pays off, especially with larger systems".

Almost all participants (88%) stated that they "were able to focus on [their individual] field of expertise and core contribution to the project", no one disagreed (Q3 in Fig. 12). This statement is supported by 88% saying that they "think that the needs, tasks and responsibilities of [their] specific role were clear and helpful" (one disagreed) and comments by the participants such as: "[the approach] helps to draw boundaries [between roles]".

The approach presented in this paper clearly "helped to structure project collaboration towards demonstrators" (93%)

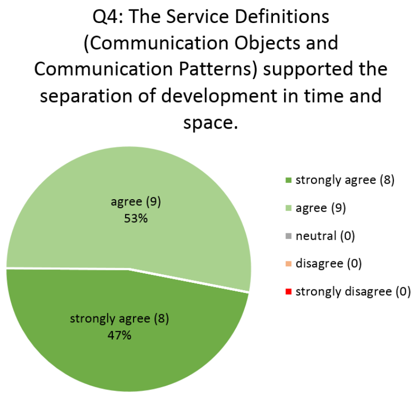


Fig. 13: Excerpts of the user study.

and provided guidance for almost all participants (81%). Participants especially made "benefit of clear definition of services in system design" (94%) which also successfully "supported the separation of development in space and time" (Fig. 13). One participant commented that "that developing prototypes is easier since you can easily avoid communication problems that normally arise at the beginning of a project."

Compared to other approaches, participants noted that they made less errors in system integration (69%) when using our toolchain. In case they encountered "errors or problems, it was easy to identify the specific architectural element that caused it. The problem or error was fixed within that element alone without further influencing others" (77%). To correct problems or errors, "it was easy to identify who (component or partner) is affected and needs to become active" (94%).

During integration activities, most of the actual problems that came up were algorithmic issues (41%). Since we know that 94% of the participants were also component suppliers who need to implement algorithms, we interpret this as a confirmation that the participants were able to focus on their specific role (algorithmic implementation of components). This is supported by a direct question, where almost all participants agreed that they were able to focus on their contribution to the projects (Q3 in Fig. 12). Most participants rated the effort for integration via composition of components as low (50%) while only 17% felt that the effort is high.

Weakness: Lower flexibility might be observed as a weakness when applying *Freedom from Choice*. However, participants stated that they "did not feel this as a limitation but rather a very helpful guidance". While most agreed with this statement (44%), some strongly agreed (25%) and only 19% disagreed.

When asked about weaknesses or ideas for improvements, many answers included tool-documentation in the sense of a user manual, enhancements for graphical tooling, suggestions for more intuitive use and hints on some minor bugs. Methods for structural debugging in the tooling and security capabilities of the approach were mentioned as possible next steps. Finally, one user reported about "not [being] friend of Eclipse, since it is too complex and overloaded".

Voices: Below are several direct comments from the participants:

... "A great approach when collaborating with different partners, from different countries, as it allows us to work in parallel."
... "I like the way it is organized in chunks (components). This is useful in simplifying a complex problem by providing a higher level of abstraction."
... "Easy modification and switching of compositions / deployments between different test-runs."
... "Traditional approach based in libraries/packages may be seen as more flexible, but implies a great effort in integration. The approach provided by SMARTSOFT / SmartMDSD is clearly more efficient, especially in collaboration projects."
... "There is a steep learning curve at the beginning where it is not much fun to use SMARTSOFT, but after investing some time it starts to be fun."
... "The SMARTSOFT ecosystem is a great solution to easily analyze new services and features, that can be easily deployed and evaluated."
... "[SMARTSOFT] can help project people in order to realize their tasks in communication intensive software systems in an easy way."

8 Conclusion

Frameworks and methods for robotics software development can be powerful, but when there is no tool-supported guidance, these mechanisms can remain unused, reducing development efficiency, wasting time and money, and in the end, reducing the capabilities of the robot. The SmartMDSD Toolchain makes concepts and methods of SMARTSOFT for robotics development accessible to users.

Based on our work, we are convinced that MDSD, DSLs and an integrated modeling approach and toolchain can make the step-change towards a successful business ecosystem for robotics software. The SmartMDSD Toolchain contributes to this step-change in general, but within our own activities, projects and partners, it has already performed this step-change and helps to actually experience this ecosystem at present with all its benefits. This statement is confirmed by the presented user study.

The SmartMDSD Toolchain and a set of reusable software components is available for download [35]. There have been 19 public releases so far and the toolchain has been "demonstrated in operational environments" (technology readiness level 6 according to [5] as acknowledged in [41]). Video-tutorials demonstrating the SmartMDSD Toolchain and videos of various scenarios that were developed using the toolchain are available online [9].

Acknowledgments

The authors gratefully acknowledge research grants and funding provided by BMBF (robotics-related research: iserveU/01IM12008B, research related to the overall workflow and its application beyond robotics: FIONA/01IS13017C).

The authors thank all participants of the user study who generously shared their time and provided valuable feedback.

References

[1] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part ii)," *IEEE Robotics Automation Magazine*, vol. 17, no. 1, pp. 100–112, March 2010. 1

[2] M. D. Petty and E. W. Weisel, "A Composability Lexicon," in *Proc. Spring 2003 Simulation Interoperability Workshop*, Orlando, USA, March 2003, 03S-SIW-023. 1, 3

[3] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, D. Brugali, J. Broenink, T. Kroeger, and B. MacDonald, Eds. Springer International Publishing, 2014, vol. 8810, pp. 195–206. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11900-7_17 1

[4] euRobotics aisbl, "Strategic Research Agenda," 2013–2014. 1, 2

[5] ——, "Robotics 2020 Multi-Annual Roadmap," Dec 2015. 1, 7.1, 8

[6] H. Heinecke, M. Rudorfer, P. Hoser, C. Ainhauser, and O. Scheickl, "Enabling of AUTOSAR system design using Eclipse-based tooling," in *International Conference: Embedded Real Time Software*, 2008. 1, 2

[7] C. Schlegel, "Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach," Ph.D. dissertation, University of Ulm, 2004. 1

[8] M. Lutz, D. Stampfer, A. Lotz, and C. Schlegel, "Service robot control architectures for flexible and robust real-world task execution: Best practices and patterns," in *Informatik 2014, Workshop Roboter-Kontrollarchitekturen*. Stuttgart, Germany, Springer LNI der GI, September 2014, ISBN 978-3-88579-626-8. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings232/article99.html 1, 3, 5.3

[9] Robotics@HS-Ulm, http://www.youtube.com/roboticsathsulm, visited: Jan. 25th 2016. 1, 5, 5.1, 7, 7.1, 8

[10] J. Thyssen, D. Ratiu, W. Schwitzer, E. Harhurin, M. Feilkas, and E. Thaden, "A System for Seamless Abstraction Layers for Model-based Development of Embedded Software," in *Proc. Envision 2020 Workshop*, 2010, pp. 137–148. 2

[11] M. Klotzbücher and H. Bruyninckx, "Coordinating Robotic Tasks and Systems with rFSM Statecharts," *Journal of Software Engineering for Robotics (JOSER)*, vol. 3, no. 1, 2012. 2

[12] N. Hochgeschwender, L. Gherardi, A. Shakhirmardanov, G. Kraetzschmar, D. Brugali, and H. Bruyninckx, "A model-based approach to software deployment in robotics," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Nov 2013, pp. 3907–3914. 2

[13] L. Gherardi and D. Brugali, "Modeling and reusing robotic software architectures: The hyperflex toolchain," in *IEEE International Conference on Robotics and Automation (ICRA)*, May 2014. 2

[14] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. 2

[15] IDEs in ROS, http://wiki.ros.org/IDEs, visited: Jan. 25th 2016. 2

[16] RIDE, https://code.google.com/p/brown-ros-pkg/wiki/RIDE, visited: Jan. 25th 2016. 2

[17] rxDeveloper, https://code.google.com/p/rxdeveloper-ros-pkg/, visited: Jan. 25th 2016. 2

[18] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS component model: a model-based development paradigm for complex robotics software systems," in *Proc. of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013. 2

[19] E. A. Lee, "Disciplined Heterogeneous Modeling," in *MODELS 2010*, Oslo, Norway, October 2010, invited Keynote Talk. 2, 3, 6

[20] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "RT-Component Object Model in RT-Middleware Distributed Component Middleware for RT (Robot Technology)," in *IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA '05)*, June 2005, pp. 457 –462. 2

[21] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *3rd International Conference on Simulation, Modeling and Programming for Autonomous Robots, SIMPAR*, ser. Lecture Notes in Computer Science, I. Noda, N. Ando, D. Brugali, and J. Kuffner, Eds. Springer Berlin Heidelberg, November 2012. 2

[22] C. Schlegel, A. Lotz, M. Lutz, D. Stampfer, J. F. Inglés-Romero, and C. Vicente-Chicote, "Model-Driven Software Systems Engineering in Robotics: Covering the Complete Life-Cycle of a Robot," in *Informatik 2013, Workshop Roboter-Kontrollarchitekturen*. Koblenz, Germany, Springer LNI der GI, September 2013. 3, 4, 7.1

[23] J. F. Moore, "Predators and Prey: A New Ecology of Competition," *Harward Business Review*, vol. 71, no. 3, pp. 75–83, 1993. 3

[24] M. Peltoniemi and E. Vuori, "Business Ecosystem as the New Approach to Complex Adaptive Business Environments," in *FeBR 2004 - Frontiers of e-Business Research 2004, proceedings of eBRF 2004*. Tampere University of Technology and University of Tampere, 2005, pp. 267–281. 3

[25] C. Schlegel and D. Stampfer, "The SmartMDSD Toolchain: Supporting dynamic reconfiguration by managing variability in robotics software development. Tutorial on Managing Software Variability in Robot Control Systems," in *Robotics: Science and Systems Conference (RSS 2014)*, Berkeley, CA, July 2014. 3

[26] C. Schlegel, A. Steck, and A. Lotz, "Robotic software systems: From code-driven to model-driven software development," in *Robotic Systems - Applications, Control and Programming*, A. Dutta, Ed. InTech, 2012, ISBN:978-953-307-941-7. 3, 4

[27] E. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall, 1976. 3

[28] A. Lotz, J. F. Inglés-Romero, D. Stampfer, M. Lutz, C. Vicente-Chicote, and C. Schlegel, "Towards a Stepwise Variability Management Process for Complex Systems: A Robotics Perspective," in *International Journal of Information System Modeling and Design (IJISMD)*, vol. 5, no. 3. IGI Global, 2014, pp. 55–74. 4, 5.8

[29] Eclipse Foundation, "Eclipse Modeling Project Website," http://eclipse.org/modeling, visited: Jan. 25th 2016. 4

[30] Papyrus UML, 2015, http://www.eclipse.org/papyrus/, visited: Jan. 25th 2016. 4

[31] Eclipse Foundation, "Xtext Website," http://www.eclipse.org/Xtext/, visited: Jan. 25th 2016. 4

[32] ——, "Xtend Website," http://www.eclipse.org/xtend, visited: Jan. 25th 2016. 4

[33] J. Vlissides, "Pattern Hatching: Design Patterns Applied," ser. The software patterns series. Addison-Wesley Professional, 1998, ISBN: 9780201432930. 4

[34] C. Schlegel, "Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot," in *Proc. Int. Conf. on Intelligent Robots and Systems (IROS)*, Victoria, Canada, 1998, pp. 594–599. 5.1

[35] SmartSoft Website, http://www.servicerobotik-ulm.de. 5.1, 7, 8

[36] C. Schlegel, A. Steck, and A. Lotz, "Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model," in *Introduction to Modern Robotics*, D. Chugo and S. Yokota, Eds. iConcept Press, 2012, ISBN:978-0980733068. 5.1, 5.3

[37] C. Schlegel, A. Lotz, M. Lutz, and D. Stampfer, "Supporting Separation of Roles in the SmartMDSD-Toolchain: Three Examples of Integrated DSLs," in *(invited talk) 5th Int. Workshop on Domain-specific Languages and Models for Robotic Systems (DSLRob) in conjunction with the Int. Conf. on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Bergamo, Italy, October 2014. 5.4

[38] A. Lotz, A. Steck, and C. Schlegel, "Runtime Monitoring of Robotics Software Components: Increasing Robustness of Service Robotic Systems," in *15th International Conference on Advanced Robotics (ICAR)*, Tallinn (Estonia), June 2011. 5.4

[39] A. Steck and C. Schlegel, "Managing execution variants in task coordination by exploiting design-time models at run-time," in *IEEE Int.*

Conf. on Intelligent Robots and Systems (IROS), San Francisco, CA, USA, 2011. 5.5

[40] D. Stampfer, M. Lutz, and C. Schlegel, "Information Driven Sensor Placement for Robust Active Object Recognition based on Multiple Views," in *In Proc. IEEE Int. Conf. on Technologies for Practical Robot Applications (TePRA)*, Woburn, Massachusetts, USA, 2012, pp. 133–138. 5.8

[41] "ITEA2 FIONA Project Review: Framework for Indoor and Outdoor Navigation Assistance," October 2015, (non-public document). 7.1, 8

[42] ZAFH Autonomous mobile Servicerobots, http://www.zafh-servicerobotik.de, visited: Jan. 25th 2016. 7.1

[43] Deutsches Zentrum für Luft- und Raumfahrt e.V., "Servicerobotik Projekte," http://www.pt-it.pt-dlr.de/de/3046.php, visited: Jan. 25th 2016. 7.1

[44] FIONA - Framework for Indoor and Outdoor Navigation Assistance, http://www.fiona-project.eu, visited: Jan. 25th 2016. 7.1

[45] FESTO Openrobotino Wiki, http://wiki.openrobotino.org/index.php?title=Smartsoft, visited: Jan. 25th 2016. 7.1

[46] D. A. Wheeler, "SLOCCount," http://www.dwheeler.com/sloccount/, visited: Jan. 25th 2016. 7.1

[47] R. Likert, "A technique for the measurement of attitudes." *Archives of Psychology*, vol. 22, no. 140, pp. 1–55, 1932. 7.2

Dennis Stampfer is a Research Associate at the University of Applied Sciences Ulm, Germany. He is a member of the service robotics research center (www.servicerobotik-ulm.de) and involved in the FIONA-Project (Framework for Indoor and Outdoor Navigation Assistance). His research interests include information driven object recognition and system composition for service robotics in a continuous development workflow using model-driven software development. He is a member of IEEE RAS TC-SOFT (Technical Committee on Software Engineering for Robotics and Automation). He has studied Computer Engineering and Information Systems in Ulm where he has received his Master's degree. He is working on a cooperative PhD with Technische Universität München (TUM).

Alex Lotz is a Research Associate at the University of Applied Sciences Ulm, Germany. He is a member of service robotics research center (www.servicerobotik-ulm.de) and is involved in a bilateral cooperation with Bosch. He was, together with Christian Schlegel, finalist for the 2012 euRobotics Technology Transfer Award ("Software concepts for service robots - Model-driven software development"). His research focus is on applying Model Driven Software Development methods to cope with the ever-increasing software complexity as a means towards a successful Robotics Software Business Ecosystem. He has studied Computer Engineering and Information Systems in Ulm where he has received his Master's degree. He is working on a cooperative PhD with Technische Universität München (TUM).

Matthias Lutz is a Research Associate at the University of Applied Sciences Ulm, Germany. He is a member of the service robotics research center (www.servicerobotik-ulm.de) and involved in the iserveU-Project (intelligent modular technologies for service robots in human environments like hospitals). His research interests are in the area of system integration for service robotics as one of the challenges from laboratory towards real world. He has studied Computer Engineering and Information Systems in Ulm where he has received his Master's degree. He is working on a cooperative PhD with Technische Universität München (TUM).

Christian Schlegel is professor in the Computer Science Department at the University of Applied Sciences Ulm, Germany. He is head of the real-time systems and autonomous mobile systems lab and head of the service robotics research center (www.servicerobotik-ulm.de). His research interests are in the area of algorithms and mechanisms for intelligent systems with a focus on service robotics. His mission is to further alleviate the gap between lab systems and robust everyday applications. His main research activity is in the field of model-driven software development for sensorimotor systems (www.servicerobotik-ulm.de/drupal/?q=node/20). He is associate editor of JOSER - Journal of Software Engineering for Robotics. Together with Alex Lotz, he has been finalist for the 2012 euRobotics Technology Transfer Award ("Software concepts for service robots - Model-driven software development").