Model-Driven Separation of Concerns for Service Robotics

Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann

Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de

Abstract

Robotics currently adopts model-driven engineering focusing software modeling languages. This forces domain experts to employ these languages instead of enabling application of more appropriate DSLs. This ultimately produces monolithic, hardly reusable applications. We present an infrastructure for the development of service robotics applications employing DSLs aimed at domain experts and tailored to domain challenges. It facilitates separation of concerns of participating robotics, domain, and software engineering experts and integrates their models via a component & connector reference architecture and a combined code generation framework. The infrastructure was successfully deployed and evaluated with robotics manufacturers, caregivers, and software engineers in a German hospital. We believe that model-driven engineering with languages tailored to the various stakeholders' needs can greatly facilitate robotic application engineering.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]: Domain-specific architectures

Keywords Separation of Concerns, Domain-Specific Languages, Code Generation, Service Robotics

1. Introduction

Engineering robotics applications is a complex endeavor that requires solutions from multiple domains as well as their successful integration. Domain experts (such as hospital IT staff) are rarely software engineering experts and due to the conceptual gap between domain challenges and their software engineering solutions (France and Rumpe 2007), integration of their solutions often results in monolithic software architectures that are hardly reusable in different contexts (Schlegel et al. 2011). This severely hampers service robotics adoption (Hägele et al. 2011). Model-driven engineering with domain-specific languages (DSLs) can facilitate robotics software engineering by abstracting from the complexity of general programming languages (GPLs) and by enabling domain experts to contribute solutions in better suitable languages – ultimately reducing the conceptual gap.

We propose to separate concerns of domain experts, robotics experts, and integration experts by providing appropriate DSLs as depicted in Fig. 1. These DSLs enable domain experts to describe tasks as sequences of goals a robot should achieve. Tasks operate in the context of a domain model, a robot model, and a world model. All are translated into integrated GPL implementations that employ a planner to solve tasks in a dynamic environment. The implementations are integrated with robot-specific GPL code provided by robotics experts. The task planner translates tasks into sequences of actions that are executed on loosely coupled robot platforms. The modeling infrastructure is integrated as a C&C architecture developed by integration experts.

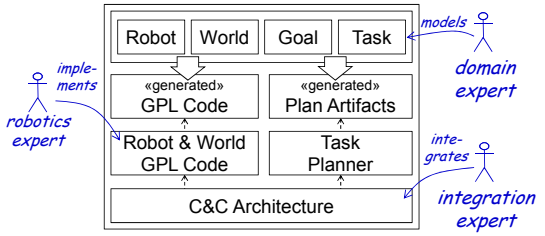

Figure 1. Separation of concerns is achieved by separation into roles with different expertises.

We have detailed the DSLs enabling such separation of concerns for service robotics applications in (Heim et al. 2015). This paper presents its (1) integrated code generation and model execution infrastructure and (2) an evaluation with praticioners in a German hospital.

In the following, Sect. 2 motivates separation of concerns and modeling in service robotics, before Sect. 3 sketches the DSLs. Afterwards, Sect. 4 presents the code generation infrastructure and Sect. 5 illustrates models execution. Sect. 6 describes the evaluation. Sect. 7 discusses observations and Sect. 8 highlights related work. Sect. 9 concludes.

DSM'16, October 30, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4894-2/16/10...$15.00
http://dx.doi.org/10.1145/3023147.3023151

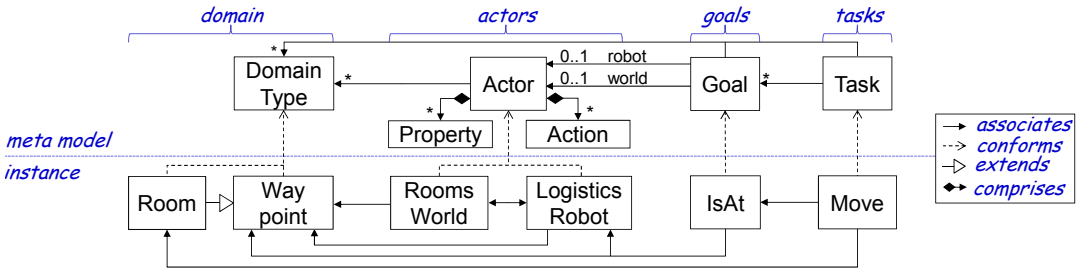

Figure 2. Top depicts the service robotics application meta model, bottom depicts conforming models.

2. Scenario

Consider engineering of a service robotics application to support hospital caregivers in logistics tasks. Successfully deploying it requires expertise in robotics, software engineering, and the application domain. Instead of forcing robotics experts and domain experts to use noisy (Wile 2001) and overly complex (France and Rumpe 2007) GPLs, they are provided with DSLs tailored to the challenges of service robotics logistics tasks. Proper integration of these DSLs and their automated transformation into GPL artifacts enables to describe solutions in a more abstract, reusable fashion without interfering with the other experts' domains.

An typical task in this scenario is the robotic delivery of supplies from a storage room to a patient's room. Successful deployment of a suitable robotic platform[1] requires that robotic experts provide the hardware and software required for a movable robot that can interact with caregivers. Domain experts contribute knowledge about the causalities of the hospital (for instance, which floors are accessible for the robot). Ultimately, software engineering experts integrate the solutions of robotics experts and domain experts.

With proper DSLs, domain experts define declarative tasks as sequences of reusable goals that abstract from the specific platform. Instead they describe requirements on robot and world in form of abstract actor models. For instance, a task may prescribe that the robot first visits the storage room, loads the required supplies, visits the patient's room, and unloads. The goals might entail requirements on the robot and world (such as having a position), but abstract from their implementations. From these models, the infrastructure generates interfaces that robot experts implement to realize functionalities on the platforms. Hence, they are liberated from dealing with domain causalities. The software engineers instantiate the infrastructure's reference architecture and extend it with application-specific components as necessary. The architecture uses the interfaces generated from task, domain, robot, and world models and takes care of task execution using the implementations provided by robotics experts. Deploying a similar application – for instance to another domain or with another platform – requires to adjust the implementations of robot models or the composition of tasks only. This facilitates successful deployment of model-driven service robotics applications.

[1] Platform comprises hardware and software to provide high-level skills.

3. Modeling Service Robotics Applications

Pervasive modeling of service robotics applications comprising of a domain, tasks, goals, robots, and worlds facilitates their platform-independent description (Heim et al. 2015). An application is modeled using heterogeneous, integrated modeling languages for different aspects (Haber et al. 2015). The meta model of applications is depicted at the top part of Fig. 2. The data types of a domain are modeled as UML/P class diagrams (CDs) (Rumpe 2016). They describe the static context of an application. Each application comprises two actors: The `world` describes properties and actions of the application environment, the `robot` describes properties and actions of the platform. Properties of an actor are dynamic domain qualities and actions are executable operations. An action can define parameters of domain data types and has two Boolean expressions: a precondition must hold for the action to be executable and a postcondition that is assumed to hold after the action has been executed. Statements of these expressions can be values of parameters or values of properties of another actor. In this, our modeling language for robots and worlds resembles classic planning languages (Fikes and Nilsson 1972). Action execution is delegated to actor implementations as employing robotics middlewares. Goal models may reference a robot, a world, and domain data types. The latter can be used to parametrize goals. Each goal model contains a Boolean expression that reflects a situation, which must be satisfied at an instant to consider the goal fulfilled. The value of the goals' parameters as well as imported properties of robot and world actors can be used as statements of the expression. Tasks are sequences of parametrized goal references. Similar to goal models, tasks can be parametrized with domain types. A task can provide its parameters to its referenced goals. For a task to be executed successfully, the referenced goals have to be satisfied sequentially. The existence of referenced model elements from different modeling languages is checked via inter-language well-formedness rules (Heim et al. 2015).

The bottom part of Fig. 2 describes an example application comprising instances conforming to the respective meta model elements. The example application describe as basic logistics scenario in which `Waypoint` instances describe

```
1 domain LogisticsDomain;                                          Actor
2
3 robot LogisticsRobot {
4   property WayPoint loc();
5   action move(WayPoint src, WayPoint dst) {
6     pre:  loc() == src && RoomsWorld.adj(src, dst);
7     post: loc() == dst;
8   }
9 }
```

Listing 1. The robot actor model `LogisticsRobot` has a property for the location of the robot and an action describing that the robot can move between waypoints.

the topology of the environment. Some waypoints are rooms that can be addressed with a name. The `RoomsWorld` actor knows waypoints and for each pair of waypoints, the `RoomsWorld` indicates whether they are adjacent or not. The application also features the `LogisticsRobot` actor which abstracts from the platform to be employed (cf. Lst. 1). It yields the property `loc` to indicate its current location in terms of waypoints (l. 4) and can move from one waypoint to an adjacent waypoint using the action `move` (ll. 5-8). The property in this example does not define parameters, and the action has two `WayPoints` parameters. The precondition (l. 6) of `move` ensures that the robot is at the `src` waypoint and that `src` and `dst` waypoints are adjacent, before `move` can be executed. For the postcondition (l. 7) to hold, the robot has to be at the waypoint `dst`. The robot can be instructed via `GoTo` tasks (cf. Lst. 2) to move from one room to another. However, the task allows only to move from and to `Room` instances. It contains two goal references that are to be satisfied sequentially. First, the goal `IsAt(src)` has to be satisfied, and then the goal `IsAt(dst)`. The referenced goal model `IsAt` (cf. Lst. 3) describes the situation in which the robot's current location equals the waypoint set as the goal's parameter (ll. 3-4).

During execution of a task, one goal at a time is planned and thereby translated into a list of actions. With this approach, a task describing the movement between rooms that are not adjacent in the world, can be executed by translating it into multiple actions that describe the movement between adjacent waypoints. An application model as described above only models a *logical* view on the application. Without a runtime environment including planning software and an employed action execution implementation, the models cannot be translated into executable GPL artifacts. The syntax of task models follows comprehensible rules (e.g., no loops or conditions) to cater potential non-programmers in a hospital's IT department.

4. From Application Models to Executable Artifacts

Domain experts develop models conforming to the actor, task, and goal DSLs to capture robotics platform-independent requirements related to the actions, properties, and tasks of

```
1 domain LogisticsDomain;                                          Task
2
3 task GoTo(Room src, Room dst) {
4   IsAt(src);
5   IsAt(dst);
6 }
```

Listing 2. The task model `GoTo` describes the robot moving from `Room src` to `Room dst` using two `IsAt` goal references.

```
1 domain LogisticsDomain;                                          Goal
2
3 goal IsAt(WayPoint wp) {
4   LogisticsRobot.loc() == wp
5 }
```

Listing 3. The goal model `IsAt` is satisfied if the robot has reached the location indicated by waypoint `wp`.

a robot and the world it operates in (cf. Fig. 1). A complete robotics system is modeled by a robot actor, a world actor, multiple tasks, multiple goals as well as an UML/P CD (Rumpe 2016) domain model.

To close the gap between these conceptual models and platform-specific realizations, we employ code generators producing classes containing information captured by models conforming to the different DSLs and component implementations for parts of a reference architecture modeled with the MontiArcAutomaton architecture description language (Ringert et al. 2015). The code generators produce a task class for each task model and a goal class for each goal model. Each task class is associated to the goal classes generated for the goals referenced by the task's model. Tasks and goals may be parametrized with parameters of types defined in their imported domain models. Tasks reference the parameters in their goals, while goals reference their parameters in their predicates (cf. Fig. 2). Further, the generators produce a class for each domain type modeled by the UML/P CD. Instances of the domain type classes encode runtime information. Task instances can be passed to the reference architecture to start their execution.

The reference architecture comprises components with domain-specific implementations generated from actor, domain type, and goal models as well as domain-independent components with predetermined implementations. Some of the generated component implementations are complete, whereas others have to be complemented with handcrafted code as illustrated in Fig. 3. The behavior of component `Controller` is predetermined. It processes incoming tasks and decomposes these into goals to determine which goal to accomplish next. The implementation of component `Planner` is completely derived from the actor and goal models as well as the UML/P CD describing the application's domain types. It transforms the actor, goal, and domain type models to appropriate PDDL (McDermott et al. 1998) problems and utilizes Metric-FF (Hoffmann and Nebel 2001) for solving these while using up-to-date world states that it requests from component `StateProvider`. Since the

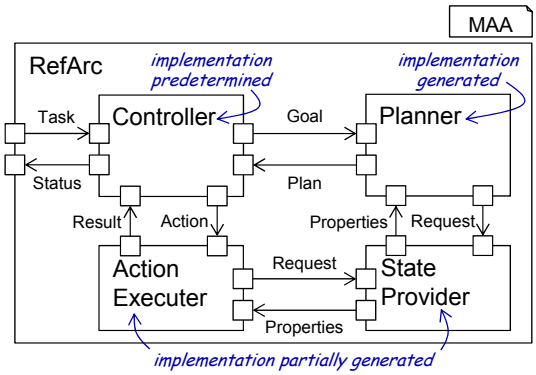

Figure 3. The reference software architecture featuring components to control task execution, using a planner and two components interacting with the platform.

`Planner` implementation is completely generated and it automatically calculates plans for the goals modeled by the domain experts, developers do not have to be planning experts. The implementations of components `ActionExecuter` and `StateProvider` are generated from the domain types and the actor models but have dependencies to code artifacts that have to be handcrafted by robotics experts (cf. Fig. 1). For this purpose the code generators produce the interface `IRobot` from the robot actor as well as the interface `IWorld` from the domain types and the world actor. Both interfaces consist of a method for each action and for each property of the corresponding actors, while the interface generated for the world actor additionally contains a method for each domain type. To complete the component implementations, robotics experts (cf. Fig. 1) provide implementations of these interfaces to interact with the specific platforms employed. This decouples logical task planning from its actual execution and enables to reuse tasks, goals, and the reference architecture with different robots and in different worlds. For instance, Fig. 4 depicts the interface generated for the robot actor illustrated in Lst. 1. The `StateProvider` component calls the methods generated for the properties and for the domain types to prompt the current system state at runtime, i.e., to calculate currently holding properties and to query currently existing objects relevant for planning. The `ActionExecuter` component invokes the methods generated for the actions to trigger their physical execution. Such a method might, for example, delegate action execution to an appropriate middleware.

The generated interfaces abstract from platform-specific implementation details. Thus, models and the generated reference architectures are reusable in different, conceptually similar applications. Only the platform-dependent implementations of the generated interfaces have to be adapted with respect to the platform-specific technologies. This facilitates separation of domain concerns captured by models conforming the DSLs presented in Sect. 3 and application specific software development concerns hidden by the generated interfaces. Abstracting from platform implementation details especially enables to reuse tasks and goals with platforms yielding different capabilities easily: whether the action `move` (cf. Lst. 1) is implemented using a humanoid robot or an unmanned aerial vehicle might be irrelevant to the conceptual domain challenges.

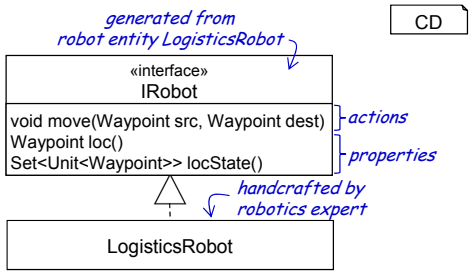

Figure 4. The interface `IRobot` is generated from the robot actor of Lst. 1 with its handcrafted implementation `LogisticsRobot`.

5. Executing Application Models

Integration experts can extend the `RefArc` software architecture with application-specific components with little effort by connecting its ports (cf. Fig. 1). The interface of `RefArc` consists of one port for receiving tasks and another for emitting status information. On task instantiation, values are substituted for a task's parameters and thereby also for the parameters of the goals the task consists of. After task instantiation, its execution can be initiated by using the interface of the reference architecture.

For each goal of each task, the `Planner` component needs to derive a plan as a sequence of actions (cf. Fig. 3, "Plan") that fulfills the goal given the current situation (cf. Fig. 3, "Properties"). The object space induced by the parameters of tasks and goals is potentially infinitely large. Furthermore, the domain types and properties of actors possibly model infinitely many environmental situations. Thus, it is practically not feasible to calculate all plans for all possible goal instances prior to runtime. Therefore, for each individual goal to be satisfied, the `Planner` component translates the goal, entity, and domain models as well as information about the current environmental situation, captured by objects encoding runtime information, into PDDL models at runtime. Then it uses the Metric-FF planner (Hoffmann and Nebel 2001) to solve the goal, and transforms the result back to lists of actions.

The planning process may succeed or fail, e.g., if a goal is unsatisfiable. If planning fails, the `Controller` initiates planning for the same goal up to three times, again, since the environmental situation might have changed. To avoid a deadlock, it aborts task execution and dismisses the current task in case planning fails four times in a row. In that case a request for remote operation is issued via its `Status` port. How this is treated depends on the application context and

the capabilities of the platform. We evaluated the reference architecture in a setting where a remote operator could access the robot's camera and proximity sensors to take control over the robot. If planning succeeds, the `Controller` tries to perform the actions of the plan received from component `Planner` in order of appearance. To this effect, it sends the next action to execute via the `Action` (cf. Fig. 3) port to component `ActionExecuter`, which tries to execute the action by calling the handcrafted methods of the actor implementations, e.g., `move` (cf. Fig. 4), and sends information indicating whether action execution was successful via the `Result` port (cf. Fig. 3). If the execution of an action fails, the system initiates a replanning process up to three times as described above. If an action is successfully executed and its corresponding plan contains more actions, the `Controller` tries to execute the next action. If a successfully executed action is the last action of a plan, the corresponding goal is satisfied. In this case, the system tries to satisfy the next goal, if there is any, or is ready for starting the execution of a new task, otherwise.

Executing the planned actions should lead to accomplishing the current goal. To ensure the assumptions made at planning time do not diverge from the actual situation at action execution (for instance, a door assumed to be open might have been closed in the meantime), the reference architecture's `ActionExecuter` evaluates each action precondition immediately prior to executing it.

6. The iserveU Robotics Application

We evaluated the presented infrastructure in the federal iserveU service robotics research project [2]. This project investigated pervasive model-driven software engineering for complex service robotics applications on example of hospital logistics applications. The 3-year project was conducted with three partners from academia and three industrial partners from different fields of robotics. In the project, we tested the overall system, including the DSLs, the reference architecture, its binding to the SmartSoft (Schlegel et al. 2011) robotics middleware, and novel robotics hardware for a week in the Katharinen Hospital in Stuttgart.

Domain experts provided 7 task models, 5 goal models, 2 actor models containing 7 properties and 6 actions, and 1 domain model with several classes. Integration experts developed 10 MontiArcAutomaton models for the reference architecture and robotics experts connected the system to a Robotino3 platform equipped with infrared sensors and laser scanners (cf. Fig. 5). For this evaluation, the reference architecture was extended with application-specific components. This enabled interaction with a remote operator as well as interaction with the hospital staff via a user interface running on a robot-mounted tablet PC. The system application faced daily hospital routines within a dynamic environment, i.e., it operated between patients and caregivers.

[2] `http://www.se-rwth.de/materials/iserveu/`

To evaluate the modeling languages, the three tasks were used: 1. Collect an item from a specific location and deliver it to a destination. 2. Guide a person from one location to another. 3. Follow a specific person. From these, collecting items was the task used most frequent. This might be a result of caregivers being more cautious in case patients interact with robots. Extension of functionality in the scenario merely increase the number of task and goal models. However, changing the employed platform or environment could urge an adaptation of the actors. This could lead to ripple effects within the task and goal models if an actor's interface changes, but the reference architecture remains robust.

Figure 5. iserveU evaluation using a Robotino3 platform connected to the reference architecture via the SmartSoft (Schlegel et al. 2011) robotics middleware.

7. Discussion

The presented DSLs describe distinct aspects of a service robotics application and an underlying execution system. They facilitate a separation of concerns into (a) domain experts, who model tasks, goals, and entities of the application; (b) robotics experts who contribute components and realize actions on the respective platforms, and (c) integration experts who compose the run-time system as a C&C architecture and ensure models are translated into compatible artifacts. This separation is independent of the employed platform or application domain, but more complex applications might require further decomposition of concerns. The DSLs to describe tasks and goals intentionally feature few elements only to enable non-programming domain experts to describe tasks and goals. For more experienced programmers, this might not be expressive enough and future work might show these prefer more expressive DSLs (cf. (Reckhaus et al. 2010; Diprose et al. 2012)).

The translation to PDDL models and planning at system runtime are complex and prohibit application of our framework in real-time critical contexts. For service robotics, real-time capabilities are usually less important, but for au-

tonomous systems, this approach might be unfeasible. Our reference architecture supports customization only in terms of adding additional components that can communicate with the reference architecture's predefined interfaces. Adding additional core functionality, for instance plan verification, might entail complex architecture reconfiguration.

8. Related Work

There are various modeling techniques for describing robot abilities and behavior (Nordmann et al. 2014), but these usually focus on programming experts instead of domain experts (Reckhaus et al. 2010; Diprose et al. 2012), target other domains (Thomas et al. 2013), lack extensible execution systems, or are tied to specific platforms. The heterogeneity of robotics rarely produces reference architectures (Lindström et al. 2000; Graf et al. 2009) for specific applications, but architectural styles (Quigley et al. 2009; Schlegel et al. 2011). These reference architectures also employ planners to solve tasks, but are limited to specific kinds of tasks (Graf et al. 2009) or are overly generic (Lindström et al. 2000).

9. Conclusion

We presented a modeling infrastructure for specification and execution of robotics tasks comprising a collection of declarative DSLs for their development. Their models are transformed into implementations for an extensible C&C software architecture that employs state-of-the-art planning to solve tasks into sequences of actions. Via small interfaces generated from actor models, the architecture interacts with the underlying robotics platform, which supports to reuse application domain expertise in form of tasks, goals, actors, and domain models with different robots. We successfully evaluated an implementation of the reference architecture with three tasks in a hospital environment and, hence, believe that software engineering of robotics applications benefits from model-driven separation of concerns.

References

J. P. Diprose, B. Plimmer, B. A. MacDonald, and J. G. Hosking. How People Naturally Describe Robot Behaviour. In *Proceedings of Australasian Conference on Robotics and Automation*, 2012.

R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial intelligence*, 1972.

R. France and B. Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*, 2007.

B. Graf, U. Reiser, M. Hägele, K. Mauz, and P. Klein. Robotic Home Assistant Care-O-bot® 3 - Product Vision and Innovation Platform. In *IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO)*, 2009.

A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Voelkel, and A. Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, 2015.

M. Hägele, N. Blümlein, and O. Kleine. Wirtschaftlichkeitsanalysen neuartiger Servicerobotik- Anwendungen und ihre Bedeutung für die Robotik-Entwicklung. Technical report, BMBF, 2011.

R. Heim, P. M. S. Nazari, J. O. Ringert, B. Rumpe, and A. Wortmann. Modeling Robot and World Interfaces for Reusable Tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.

J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 2001.

M. Lindström, A. Orebäck, and H. I. Christensen. Berra: A Research Architecture for Service Robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.

A. Nordmann, N. Hochgeschwender, and S. Wrede. A Survey on Domain-specific Languages in Robotics. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2014.

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

M. Reckhaus, N. Hochgeschwender, P. G. Ploeger, G. K. Kraetzschmar, and S. Augustin. A Platform-independent Programming Environment for Robot Control. In *Proceedings of the 1st International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)*, 2010.

J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 2015.

B. Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, 2016.

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*. iConcept Press, 2011.

U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *2013 ICRA IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

D. S. Wile. Supporting the DSL Spectrum. *Computing and Information Technology*, 2001.