Towards Easy Robot Programming
Using DSLs, Code Generators and Software Product Lines

Johannes Baumgartl[1], Thomas Buchmann[2], Dominik Henrich[1] and Bernhard Westfechtel[2]
[2]*Chair of Applied Computer Science I, University of Bayreuth, Universitaetsstrasse 30, 95440 Bayreuth, Germany*
[1]*Chair of Applied Computer Science III, University of Bayreuth, Universitaetsstrasse 30, 95440 Bayreuth, Germany*

Keywords: Model-Driven Development, DSL, Code Generation, Robot, Easy Programming, Software Product Lines.

Abstract: Programming robots is a complicated and time-consuming task. A robot is essentially a real-time, distributed embedded system. Often, control and communication paths within the system are tightly coupled to the actual physical configuration of the robot. Thus, programming a robot can hardly be achieved by experts of the domain in which the robot is used. In this paper we present an approach towards a domain specific language, which is intended to empower domain experts or even end users to specify robot programs with a dedicated background in programming techniques, but not with special knowledge in the robotics domain. Furthermore we introduce an idea to integrate a software product-line for a hardware and software transparent plug and play mechanism.

1 INTRODUCTION

A robot is essentially a real-time, distributed embedded system. Often, control and communication paths within the system are tightly coupled to the actual physical configuration of the robot. Robot systems consist of different hardware components and different sensors which results in a very complex and highly variable system architecture. As a consequence, these robots can only be assembled, configured, and programmed by experts. While this is the state of the art for industrial robots, it is evident that this approach is not feasible for personal robots (Wyrobek et al., 2008). As the name implies, personal robots are targeted towards individuals by providing a human interface and a special design. By definition, a personal robot enables individuals to automate the repetitive or menial part of their home and work lifes.

Model-driven software engineering (Frankel, 2003; Völter et al., 2006) puts strong emphasis on the development of high-level models rather than on the source code. Models are not considered as documentation or as informal guidelines how to program the actual system. In contrast, models have a well-defined syntax and semantics. Moreover, model-driven software engineering aims at the development of *executable* models. *Code generators* are used in model-driven software engineering, to transform the specification of higher-level models into source code. A *Domain-Specific Language (DSL)* is a programming or specification language which is dedicated to a particular problem domain.

Software Product Line Engineering (SPLE) (Clements and Northrop, 2001; Pohl et al., 2005; Weiss and Lai, 1999) deals with the systematic development of products belonging to a common system family. Rather than developing each instance of a product line from scratch, reusable software artifacts are created such that each product may be composed from a library of components. Furthermore, it provides means to capture and manage the variability of a particular application domain. In common approaches, *feature models* (Kang et al., 1990) are used for that purpose.

In this paper we present an approach towards a domain-specific language, which is intended to empower individuals to specify robot programs with a dedicated background in programming techniques but not in the robotics domain. Furthermore we sketch our product line approach which allows to manage the variability which is introduced in robotics by different hardware and different algorithms. Furthermore, the model-driven product line approach allows automatic configuration as well as automatic verification of existing DSL programs.

Baumgartl J., Buchmann T., Henrich D. and Westfechtel B..
Towards Easy Robot Programming - Using DSLs, Code Generators and Software Product Lines.
DOI: 10.5220/0004585305480554
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-PT-2013), pages 548-554
ISBN: 978-989-8565-68-6

2 APPROACH

2.1 Motivation

Before we start with a detailed description of our approach, let us briefly give a motivating example. In service robotics, *pick and place* applications are a standard usage scenario. The robot has to pick (possibly previously unknown) objects (Baumgartl and Henrich, 2012) from location A and place them somewhere in location B. This task, which sounds simple at first glance requires several complex subtasks:

1. **Object Modeling.** The object that has to be picked is either already known and needs to be identified in the database or it needs to be reconstructed by the robot's sensors. The captured scene has to be split into objects belonging to the environment and graspable objects.
2. **Grasp Planning.** Grasp planners should compute a high number of different stable grasps for the object to be manipulated in order to manage complex tasks in open environments.
3. **Path Planning.** Path planners are used to determine a short collision-free approach motion of the robot to the object.
4. **Grasp Object.** Select and execute one planning result from the previous two steps and physically grasp the object.
5. **Validation.** The validation step is performed to check the result of the previous task (Step 4). The following cases may occur: (1) The grasp totally failed, i.e. the object was not grasped. (2) After the grasp, the object slipped between the gripper fingers and the relative position of the gripper to the object differs from the planned result. (3) The grip was executed as planned.
6. **Placement Planning.** Placement planning differs with respect to robotic application that has to be developed. Standard robotic problems are for example: (1) bin packing, (2) peg in hole, (3) sorting, and (4) assembly.
7. **Path Planning.** Like described in step 3, another path planner is required to determine the motion of the robot to the target position.
8. **Execute.** Execute the results from the previous two steps and complete the task.
9. **Validation.** Analogously to Step 5, a validation is performed to check the result of the task execution.

Please note that the order of the different planning tasks my differ with respect to the application that has to be solved, i.e., some tasks might require placement planning before grasp planning and vice versa. Even with the support of a robotics framework which abstracts from the hardware and encapsulates algorithms for image detection, image processing and planning, C++ source code for our example pick-and-place scenario comprises several hundred lines of hand-written code.

2.2 Vision

2.2.1 Requirements

It is evident, that we can not expect end users to specify programs solving the scenario described in section 2.1. As a consequence domain-specific languages at a high-level of abstraction are required for that purpose. According to our example, we are able to define several core requirements, which have to be met by a DSL which is targeted towards end users:

Complex Operations. As demonstrated in section 2.1, grasping and placing objects requires several complex operations conducted in a given order. However, the end user should only need to deal with abstract concepts like grasp and place. Furthermore, the user does not have to deal with the order in which certain sub tasks, like grasp planning or placement planning are executed.

Abstraction. Personal robots need to be able to capture their open environment. Therefore they are equipped with different sensors. However, end users should not have to deal with accessing sensors and processing sensor data. Thus, the DSL should only provide means to specify and reference objects.

Variability Support. We observe variability in robotic applications on different levels: On hardware level, variability occurs for example within the sensors that are used to capture open environments, while on software level different implementations of planning algorithms exists which differ from each other in terms of execution speed or accuracy. Hence, our approach needs to provide support for this fact.

Plug and Play. Support for runtime reconfiguration is required, as hardware or software components might be changed or updated during the lifetime of a personal robot. Consequently, the DSL and the corresponding code generator need to be adopted to the new configuration accordingly by providing new or removing old language constructs for example.

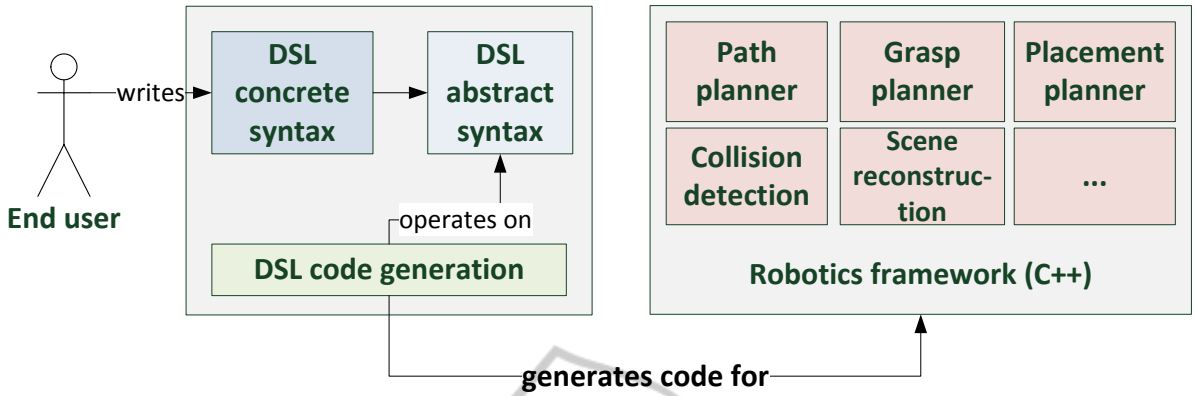

Figure 1: DSL and code generation on top of a C++ framework.

2.3 Solution Overview

Figure 1 depicts the first step towards the end user DSL for personal robots, which will be described in the following:

The DSL will be provided as a plug-in for the Eclipse environment. Thus, the abstract syntax is defined using the *Eclipse Modeling Framework (EMF)* (Steinberg et al., 2009). We chose to use a textual concrete syntax which will be implemented using *Xtext*[1]. Our approach enables end users to specify programs for personal robots at a high level of abstraction. As described in the previous section, sequences of complex operations are encapsulated by high-level operations like grasp, place or drop for example. After the robot program has been specified by the user, C++ code is generated which uses our robotics framework. We use *Acceleo*[2] as M2T engine, as it is the only M2T framework within the Eclipse context which is based on an official standard (MOF Model-to-text (OMG, 2008)). The robotics framework is composed of different building blocks, each of which is dedicated to a specific problem within a robotics application, e.g. path planning (Gecks and Henrich, 2009) including automatic sensor base adaption (Deiterding and Henrich, 2007), grasp planning, or placement planning. The resulting C++ code is then compiled and deployed to the target hardware (i.e. the robot). In section 3, we provide results from a case study in which we developed a DSL for pick-and-place operations. The only difference to the approach sketched in Figure 1 is that we generated specific code for a robot simulator rather than generating code which is executed on a real hardware.

As stated in the previous section, the domain of personal robots includes a large amount of variability. The high complexity is a result of the variability within hardware components as well as the variability within algorithms used in a robot's software. There is a wide range of different planning algorithms which differ with respect to execution speed or accuracy but also to required hardware. To manage the variability and the resulting complexity, we will apply a software product line engineering approach. Commonalities and variabilities of both hardware and software components within the robotic domain are captured in a feature model, as shown in Figure 2. The elements of the feature model are then mapped onto the corresponding implementation fragments. As both, the DSL concrete and abstract syntax as well as the Acceleo code generation are based upon Ecore, we plan to use *FAMILE* (Buchmann and Schwägerl, 2012) to map features on corresponding implementation fragments.

Please note, that the scope of the domain-specific language highly depends on the available hardware of the robot. Thus, it is important that the DSL and the corresponding code generation may be (automatically) customized to match a specific hardware configuration. E.g. there is a wide range of different grippers starting from parallel jaw grippers up to dexterous robot hands. The basic functionality of all is open and close the gripper. Commonly, additional features are choosing grasp speed, force, etc. Furthermore, there are grippers with special functions like, automatic sensor based repositioning of the grasped object within the gripper fingers. It is evident, that those advanced grippers need additional language constructs within the DSL to map their functionality.

After a valid feature configuration has been created either by a robot systems engineer or is automatically retrieved via a plug and play mechanism, it is applied to both the DSL as well as the framework. In particular the DSL is tailored towards the specific

[1]http://www.eclipse.org/Xtext
[2]http://www.eclipse.org/Acceleo

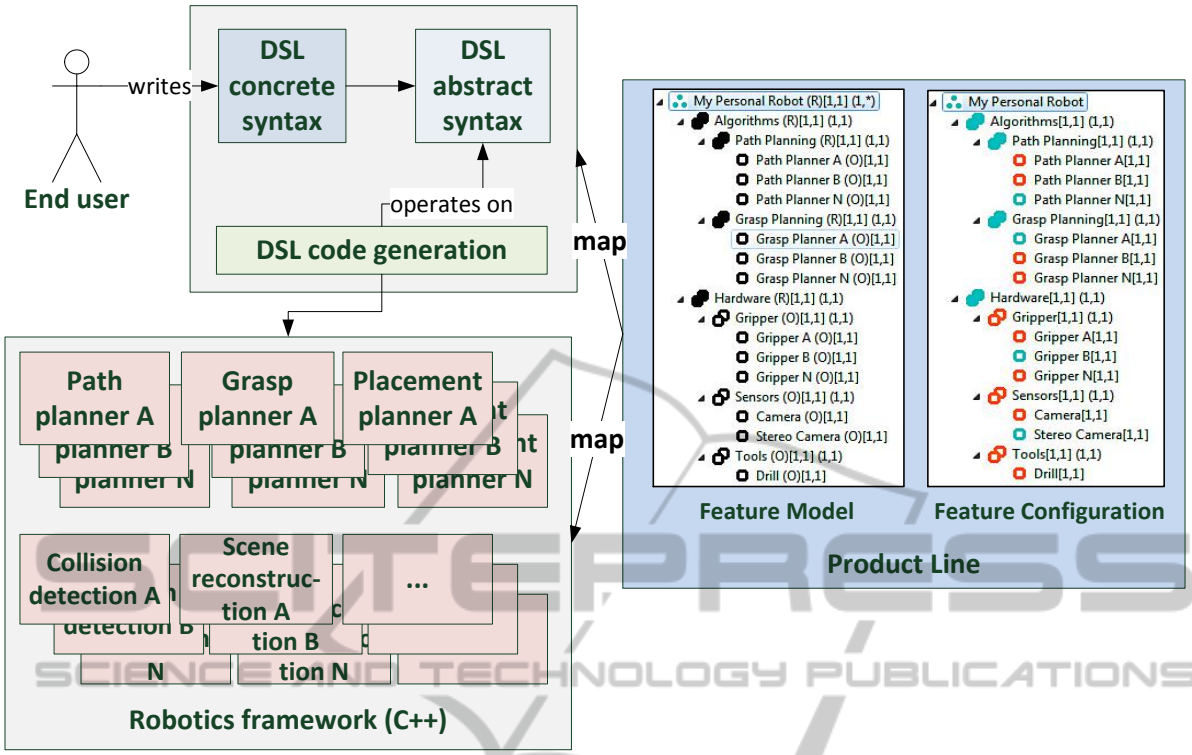

Figure 2: Integration with the product line approach, to realize variability.

needs, i.e. unused or unsupported language constructs are omitted. Consequently, also the code generator templates are adopted accordingly. Furthermore, the robotics framework is tailored to the specific needs in order to deploy only mandatory building blocks to the target hardware. As a result, a small and customized environment is generated in which the end user can now specify robot programs.

By applying our model-driven approach, we are able to verify the correctness of already existing robot programs, after a system reconfiguration (e.g. by changing / updating the robot's hardware) has been performed. Furthermore, model-to-model transformation may be applied to map obsolete language constructs to new ones.

3 CASE STUDY

To proof the validity and feasibility of our approach, we created a small case study located in the domain of pick and place applications. Pick and place operations are standard robot operations and therefore they serve as a good example for our DSL. To speed things up for our tests, we used *ROS (Robot Operating System)*[3], a robot simulation environment, as backend. ROS enables software developers to create robot applications by providing libraries for hardware abstraction, device drivers, visualizers, and many more.

In the following we describe our DSL and the correlated ROS code by means of a standard robotic problem: A given set of objects should be sorted according to specific criteria and one of the resulting subsets should be manipulated. In our example, the objects are located on a given table. First, the robot must detect the objects on this table. Note, in this example all objects are available in an object database. And second, the robot should throw all blue objects to the also given bin.

Basically, we implemented an object-centered language. Listing 1 shows the solution of the proposed problem. By means of this example we explain the basic ideas of our DSL.

The DSL program code has two sections, a define section for initialization, and a second section for the program logic. A program always starts with a define section where the mandatory descriptions for a proper execution are given, including the definition of objects, sets of objects, and topological arrangements.

Objects do have a configuration composed of a position and orientation. They also have different optional properties, like a material, mass or figure, offering the opportunity to apply operations to them.

The user has the possibility to pre-define topolog-

[3]http://www.ros.org/wiki

Listing 1: DSL program for sorting some reconstructed objects by color and throw the blue ones into the bin.

```
define {
  Object bin {
    figure bowl
    material Green
    position ( 0.65 0.00 0.74 )
    orientation ( 0.0 0.0 0.0 )
    mass 15.0
  }

  Object table {
    figure table
    material wood
    position ( 0.8 -0.34 0.74 )
    orientation ( 0.0 0.0 0.0 )
  }

  Topology  TOP { axis z direction pos }
}

Set s <- detectObjects( table, TOP )

s -> forAll(o.material == Blue) {
  grasp o
  drop(bin,TOP)
}
```

ical arrangements like TOP as shown in Listing 1 for further reuse. In our example TOP indicates that if object *A* is on top of object *B*, the corresponding z coordinate of property position of A has a greater value than the one of B.

Within the program logic section, the user may use the following language constructs to specify the robot behavior:

Grouping Objects. Using the Set keyword, arbitrary objects which could either be retrieved dynamically allocated by the robot's sensors or manually specified. For example by the detectObjects keyword. In our example provided in Listing 1 all objects on top of the table will be included in the set.

Selecting Objects. To manipulate sets of objects, different set operations are provided similar to the ones defined in OCL (OMG, 2012). E.g. forAll iterates over all elements of a set and selects the ones that satisfy the given constraint(s).

Manipulating Objects. There are two different ways how objects may be manipulated: (1) the position or (2) the orientation are changed. The DSL offers the keywords grasp, place and drop for this purpose. In contrast to a place operation where a stable configuration of the object after it was released is essential, a drop operation puts no post-conditions on the corresponding objects.

Between a grasp and a place or drop operation there is an implicit transfer movement. To provide a high-level of abstraction, we only take into account the configuration of an object rather its trajectory between source and target location. The advantage of this approach is that we need not take care about the movement, collision detection, safety, and many more aspects. Furthermore, the user does not need to do low-level sensor data processing, as for instance the detection of blue objects in the scene using a color camera.

However, the drawback is that tasks, which have trajectories as main aspect, can not be solved with the proposed DSL.

In case we wanted to specify the robot behavior as given in the example (Listing 1) instead by handwritten C++ code directly in the ROS environment then much more effort would be required. A prerequisite is to correctly setup the ROS environment by loading all required modules and starting corresponding services (approx. 50 LoC). After that the program logic might be defined. The first step is to detect all objects (30 LoC). Second, the detected objects are passed to the collision detection module (20 LoC). This module also returns a set of graspable objects. To grasp one of them, 45 LoC are needed and another 50 LoC are required to drop it. Please note, that we do not explain in detail the selection by color. In total our simple example written in ROS needs about 250 LoC C++ application code. Furthermore, the programmer needs to know the internals of every module and how they need to be combined.

In our DSL the order, dependencies, and error handling code is totally hidden from the programmer. We use an object-centered approach to describe the manipulation of objects, as it is closer to a human-like way describing tasks. If an ordinary programming language is used, all robot configurations and movements need to the specified in detail by the programmer. Thus, he must have the knowledge about the specific hardware, path, grasp and placement planning. If a robotic framework like ROS is used, the programmer has the opportunity to use predefined planners. What remains is that the programmer needs system knowledge. So in both cases the programmer needs to be an expert for robotic in an appropriate manner.

4 RELATED WORK

In (Inglés-Romero et al., 2012), the authors present an approach which uses a DSL to handle run-time variability in programs for service robots. The ap-

proach presented by Inglés-Romero et al. aims to support developers of robotic systems (e.g. experts in the robotics domain) while our approach has the goal to enable individuals without dedicated backgrgound in robotics to specify programs for robots. Furthermore, the DSL is only able to express variability information. It is not possible to specify the behavior of the robot.

Steck et al. present an approach (Steck et al., 2009) that is dedicated to a model-driven development process of robotic systems. They present an environment called *SmartSoft* (Steck and Schlegel, 2010) which provides a component based approach to develop robotics software. The SmartSoft environment is based on Eclipse and the Eclipse Modeling Project[4]. It uses Papyrus[5] for UML modeling. By using a model-driven approach, the authors focus on a strict separation of roles throughout the whole development life-cycle. Again, experts in the robotics domain are addressed with this approach while our approach doesn't required expert knowledge in robotics.

RobotML (Dhouib et al., 2012), a modeling language for robot programs also aims to provide model-driven engineering capabilities for the domain of robot programming. RobotML is an extension to the Eclipse-based UML modeling tool Papyrus. Papyrus puts strong emphasis on UML's profile mechanism, which allows domain-specific adaptations. RobotML provides code generators for different target platforms, like Orocos, RTMaps, Arrocam or Blender/-Morse. The approach presented by Dhouib et al. addresses developers of robot programs or algorithms, while our approach is targeted towards regular programmers or even end users instead.

Bubeck et al. present in (Bubeck et al., 2012) an overview about best practices for system integration and distributed software development in service robotics. Furthermore, the authors develop *BRIDE*[6], a graphical DSL for ROS developers. Using BRIDE, new ROS nodes or ROS-based systems can be specified in a graphical way and corresponding C++ or Python code may be generated. In addition, the required launch files for the ROS environment including the relevant parameters and dependencies are generated as well, similar to the approach which we used in our case study as described in section 3. Similar to the approaches discussed above, BRIDE also addresses robot experts while our approach is targeted towards end users or users that only required a background in programming rather than dedicated robotic skills.

[4]http://www.eclipse.org/modeling/

[5]http://www.eclipse.org/papyrus

[6]http://ros.org/wiki/bride

In (Schultz et al., 2007), Schultz et al. present an approach for a domain-specific language intended for programming self-configurable robots. The DSL is targeted towards the ATRON self-reconfigurable robot. Like all other approaches mentioned in this section, it aims to provide a higher-level of abstraction for robot experts.

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented our approach towards easy robot programming for personal robots. We demonstrated the feasibility of our approach by presenting the case study in which we implemented a domain-specific language for pick and place operations on top of the *Robot Operating System (ROS)*.

The next steps are: (1) Transfering the pick and place DSL from the Robot Simulation Framework ROS to real robots by using our own robotics framework. (2) Defining a DSL which also covers other robotic application domains and (3) apply our software product line approach to enable mass customization of the DSL in order to adopt to specific customer needs as well as specific hardware configurations.

We plan to evaluate our domain-specific language in projects with undergraduate students who do not have experienced programming skills. Furthermore, we additionally plan to use our approach in co-operative projects with high-schools.

REFERENCES

Baumgartl, J. and Henrich, D. (2012). Fast vision-based grasp and delivery planning for unknown objects. volume 7th German Conference on Robotics (ROBOTIK 2012), Munich, Germany, May 21 - 22, 2012.

Bubeck, A., Weisshardt, F., Sing, T., Reiser, U., Hagele, M., and Verl, A. (2012). Implementing best practices for systems integration and distributed software development in service robotics - the care-o-bot robot family. In *System Integration (SII), 2012 IEEE/SICE International Symposium on*, pages 609–614.

Buchmann, T. and Schwägerl, F. (2012). FAMILE: tool support for evolving model-driven product lines. In Störrle, H., Botterweck, G., Bourdells, M., Kolovos, D., Paige, R., Roubtsova, E., Rubin, J., and Tolvanen, J.-P., editors, *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications*, CEUR WS, pages 59–62, Building 321, DK-2800 Kongens Lyngby. Technical University of Denmark (DTU).

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA.

Deiterding, J. and Henrich, D. (2007). Automatic adaptation of sensor-based robots. IEEE International Conference on Intelligent Robots and Systems:1828–1833.

Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., and Ziane, M. (2012). Robotml, a domain-specific language to design, simulate and deploy robotic applications. In Noda, I., Ando, N., Brugali, D., and Kuffner, J., editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 7628 of *Lecture Notes in Computer Science*, pages 149–160. Springer Berlin Heidelberg.

Frankel, D. S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley Publishing, Indianapolis, IN.

Gecks, T. and Henrich, D. (2009). Sensor-based online planning of time-optimized paths in dynamic environments. GWR09 German Workshop on Robotics, Braunschweig, Germany.

Inglés-Romero, J. F., Lotz, A., Chicote, C. V., and Schlegel, C. (2012). Dealing with Run-Time Variability in Service Robotics: Towards a DSL for Non-Functional Properties. In Menegatti, E., editor, *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-12, co-located with SIMPAR 2012)*, Tsukuba, Japan.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute.

OMG (2008). *MOF Model to Text Transformation Language, Version 1.0*. OMG, Needham, MA, formal/2008-01 edition.

OMG (2012). *Object Constraint Language*. Object Management Group, Needham, MA, formal/2012-01-01 edition.

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Verlag, Berlin, Germany.

Schultz, U. P., Christensen, D. J., and Stoy, K. (2007). A Domain-Specific Language for Programming Self-Reconfigurable Robots. In *Workshop on Automatic Program Generation for Embedded Systems (APGES)*, pages 28–36.

Steck, A. and Schlegel, C. (2010). Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development. In *Proceedings of the First International Workshop on Domain-Specific Languages and models for ROBotic systems (IROS - DSLRob)*, Taipei, Taiwan.

Steck, A., Stampfer, D., and Schlegel, C. (2009). Modellgetriebene Softwareentwicklung für Robotiksysteme. In Dillmann, R., Beyerer, J., Stiller, C., Zöllner, J. M., and Gindele, T., editors, *AMS*, Informatik Aktuell, pages 241–248. Springer.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Boston, MA, 2nd edition.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Weiss, D. M. and Lai, C. T. R. (1999). *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Boston, MA.

Wyrobek, K. A., Berger, E. H., der Loos, H. F. M. V., and Salisbury, J. K. (2008). Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In *ICRA*, pages 2165–2170. IEEE.