A Model-based Approach to Software Deployment in Robotics

Nico Hochgeschwender, Luca Gherardi, Azamat Shakhirmardanov,
Gerhard K. Kraetzschmar, Davide Brugali and Herman Bruyninckx

Abstract— **Deploying a complex robot software architecture on real robot systems and getting it to run reliably is a challenging task. We argue that software deployment decisions should be separated as much as possible from the core development of software functionalities. This will make the developed software more independent of a particular hardware architecture (and thus more reusable) and allow it to be deployed more flexibly on a wider variety of robot platforms. This paper presents a domain-specific language (DSL) which supports this idea and demonstrates how the DSL is used in a model-driven engineering-based development process. A practical example of applying the DSL to the development of an application for the KUKA youBot platform is given.**

I. Introduction

Software development and integration in robotics is challenging. The complexity of the task increases significantly when having to cope with heterogeneous, networked hardware on integrated robots (e.g. Care-O-bot 3 [1] and PR2 [2]) and when having to integrate diverse computational approaches for solving functional problems (e.g. planning, perception and control). Although the community has developed several robot software frameworks (RSFs) that are rich of functionality (e.g. ROS [3], Orocos [4], OpenRTM [5], and SmartSoft [6]), some development activities are still cumbersome and prone to errors. One such activity is the deployment of a complete, integrated robot application.

Deployment is a crucial phase in the robot application development process and is mainly concerned with the aim of making the application ready to use. In doing so, during *design time*, system integrators have to cope with several issues, such as which components are realizing a certain functionality and which shall be deployed as processes or threads, and on which host? Generally speaking, deployment is an activity for which knowledge about different *architectural aspects* is fundamental. Often this knowledge needs to be adapted, sometimes repeatedly during the development process. For instance development platforms and their computational resources, such as the number of computers and the available amount of memory, very often differ significantly based on the robot systems on which they are finally deployed to run an application in real-world settings. Keeping track of such changes, in presence of developers and engineers with different areas of expertise, is challenging and can lead to erroneous and dysfunctional deployments.

To improve this situation, related and somewhat more mature domains, such as avionics and automotive domains, adopted the Model-Driven Engineering (MDE) approach [7]. In MDE the concept of *meta-models* and *domain-specific languages* are used to support code generation (or other artifacts) from abstract models that describe a domain. Thereby and through systematic separation of concerns software quality is enhanced. In the work presented here, MDE is adopted for the task of software deployment in robotics. More precisely, we provide a hierarchy of architectural concepts and means to model deployment aspects in an explicit and framework-independent manner. To support this, we introduced a Domain-specific Language (DSL) for the declarative description of architectures and deployment activities, which facilitates the deployment task at design time and enables modeling *for* and *by* reuse.

II. Motivation: Software Deployment in Robotics

Deploying a complete, integrated robot application on service robots is a cumbersome exercise which is prone to errors. The following anecdote exemplifies this: during RoboCup@Home[1] competition 2010 the first author deployed accidentally the speech recognition component of a service robot to another host while forgetting to maintain the software connection to the microphone. Hence, during the competition the robot was not able to understand speech commands anymore. After a troublesome assessment we identified the error in the deployment description where no software connection checking was performed.

In robotics, software deployment is usually achieved through some kind of deployment infrastructure provided by the underlying robot software frameworks. For instance, the tool `roslaunch`[2] of the popular ROS framework takes a description of the architecture as an input and initiates the deployment according to it, i.e., nodes are started, parameters are set and so on. The framework-specific deployment description is created by the system integrator and contains two main parts: a component description and a system description. Depending on the underlying component model of the framework the component description contains information about the provided and required input and output in terms of services, topics or data ports whereas the system description encodes which components interact with each other.

Nico Hochgeschwender and Gerhard K. Kraetzschmar are with the Department of Computer Science, Bonn-Rhine-Sieg University of Applied Sciences, Sankt Augustin, Germany `firstname.lastname at h-brs.de`

Azamat Shakhirmardanov and Herman Bruyninckx are with the Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium `firstname.lastname at mech.kuleuven.be`

Luca Gherardi and Davide Brugali are with the Department of Engineering, University of Bergamo, Italy `firstname.lastname at unibg.it`

[1] `http://www.robocupathome.org/`

[2] `http://www.ros.org/wiki/roslaunch`

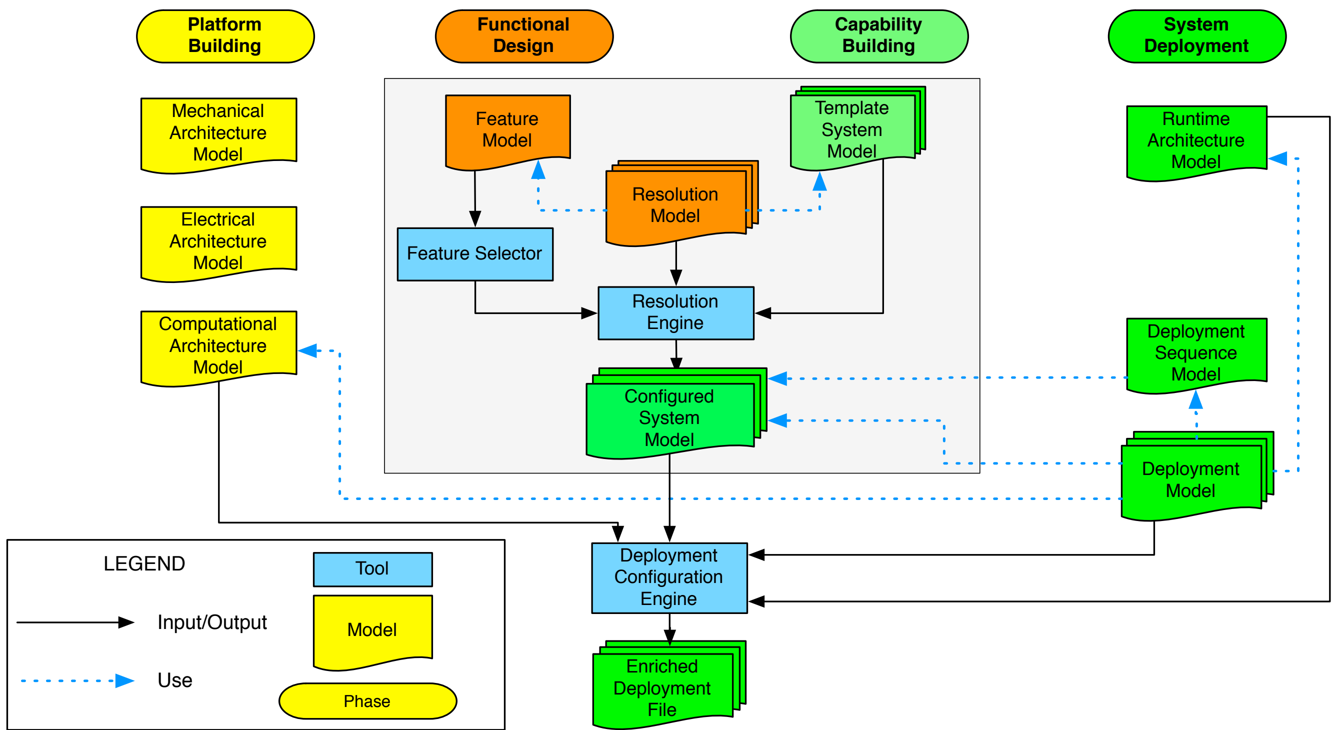

Fig. 1. The model-based software deployment approach, which includes the four *phases* of the development process shown on top. In each phase models are created. A color coding is used to show which models are created in which phase. The models are either used (referenced) by other models or serve as input/output for a tool. To support the editing of the models we provide several graphical and textual editors which are not shown in the Figure.

As the software infrastructure for real-world applications is easily composed of 20 to 300 components, this low-level and framework-specific deployment description becomes hard to maintain. It is current practice in robotics to implicitly encode knowledge about other architectural aspects (e.g., computational resources) and design decisions in framework and platform-specific tools. Very often heterogeneous representations are used during design and development and means for documentation are missing. Thus, the identification and dissemination of design principles and best practices is limited. As service robots are built by several engineers and developers working on different architectural aspects such information is crucial and should be captured as a part of a deployment description.

In summary, software deployment is cumbersome and it remains challenging

- to model different architectural aspects in a framework-independent and explicit manner,
- to reuse and deploy a configured component-architecture on another platform, and
- to ensure properties and check constraints prior to the actual deployment on the robot.

III. Model-based Software Deployment

To improve software deployment in robotics we introduce a *model-based software deployment approach* (see Figure 1), which is described in the following. The main objective of the approach is to allow systematic deployment taking into account both *architectural aspects* and *constraints* introduced by different *phases* of the development process.

We decompose a robot application into software and hardware aspects which are further decomposed in sub-aspects (see Figure 2). To encode information about these aspects we propagate model-based techniques [7], because the concrete models of the aspects are changing rapidly whereas their representation (meta-models) remains rather static. We argue that different architectural aspects are modeled by developers and engineers within different phases of a development process who are not necessarily performing the deployment themselves. Figure 1 depicts the four phases of the BRICS robot application development process (RAP) [8] that are relevant for the deployment task. RAP is a holistic process model for developing robotics applications in both academic and industrial settings, which has been defined in the EU-funded project BRICS (*Best Practices in Robotics*)[3]. RAP foresees eight different phases, each of which requires several steps to complete the task. The four phase relevant for this paper, together with the models that they use as input and produce as output, are described in the following sections.

A. The BRICS Research Camp Use Case

To exemplify our approach we make use of a real-world application developed within the scope of the 4*th* BRICS Research Camp on Robot Software Architectures[4]. The use case consists of a robot moving through an artificial arena

[3] `www.best-of-robotics.org`
[4] `www.best-of-robotics.org/4th_researchcamp`

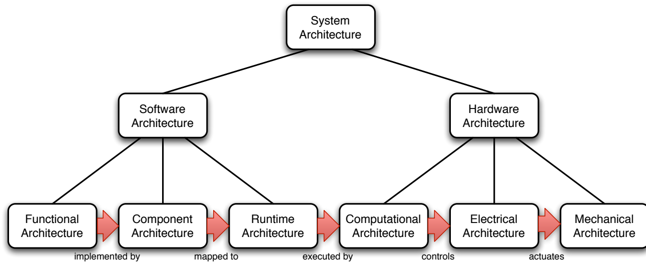

Fig. 2. A decomposition of architectural aspects relevant for deployment. The software architecture is decomposed into the functional architecture (FA), component architecture (CA), and runtime architecture (RA). The FA aspect focuses on functionality rather than software issues. The CA aspect concerns all aspects of the actual software implementation of the FA, especially software modules and their interaction, the interfaces of the software modules and the relevant data structures. The RA aspect maps the software architecture onto a particular computational architecture, mainly by mapping software components onto processes and threads, and by mapping processes and threads onto the computational devices available.

avoiding obstacles. Optionally the robot is instructed to pick-up an object at one location (inside the arena) and delivers this object to another location (see also Figure 3). The robot navigation can be performed according to two navigation strategies: map-based (a geometrical map of the environment is provided) or marker-based (the robot simply follows visual markers placed on the floor). This scenario allows the development of different applications that are presented in subsection III-C. These applications require several functionalities, such as inverse kinematics, bounding box estimation, navigation, localization and mapping. Even though, the application is simple from a functional point of view, the deployment structure is rather complex as several variation points have to be resolved.

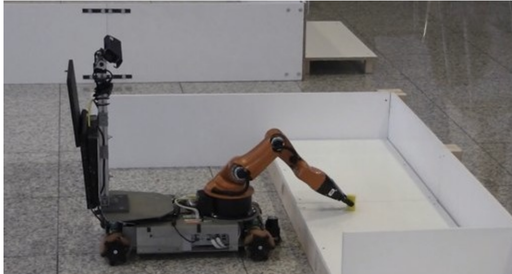

Fig. 3. The fetch and carry application used for evaluation.

B. Platform Building Phase

According to the RAP process, the platform building phase is performed once a good understanding exists of which tasks the target robot application needs to solve and how the environment and task execution context looks. Therefore, in the platform building phase the robot will be configured in terms of required sensors and actuators as well as their mounting and placement on the robot (see *Mechanical and Electrical Architecture Model* in Figure 1). Other configuration choices may pertain to the computational hardware to be integrated (see the *Computational Architecture Model (CAM)* in Figure 1, whose meta-model is depicted in Figure 4). In robotics, however, modeling the *CAM* is either neglected or performed in an ad-hoc manner. We propose to adapt existing and industrial-proven *CAM* modeling approaches such as AADL [9]. In the pick and place use case a KUKA youBot is used. The robot is equipped with two computational hosts: an internal computer and an external computer. The internal host is connected to the EtherCat bus which provides access to the joint controllers and the external computer, which is more powerful and is mounted on the sensor tower of the youBot. To model the *CAM* we provide a set of primitives shown in Figure 4. The *CAM* meta-model is inspired by the AADL platform meta-model and allows us to model the basic structure of computational hosts (e.g., CPU, memory, and bus connections) and their properties relevant for deployment, such as hostname, IP address, and amount of memory.

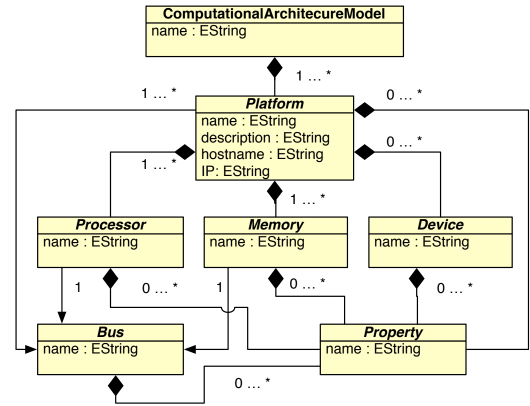

Fig. 4. The computational architecture meta-model.

C. Functional Design and Capability Building Phases

This subsection describes the Functional Design and the Capability Building phases, which are contained in the grey box depicted in Figure 1. These phases and the associated models and tools are the result of previous works which are presented in ([10], [11], [12]) and are available on GitHub.[5] This paper provides a brief summary of the overall approach.

During these phases the engineers design a set of models that describe the architectural and the functional variability of a software product line, which is a family of similar applications that share the same architecture and are built reusing a set of software components [13]. The ultimate goal of these phases is to automatically generate a model that describes the architecture of a specific application of the product line (the *Configured System Model (CSM)*) starting from a selection of functionalities. This *CSM* will be then later deployed on the robot.

The architecture of the software product line is represented in a *Template System Model (TSM)*, which specifies: (a) the set of components that can be used for building all the possible applications of the family (some are mandatory, some others optional) and (b) a set of connections among

[5] http://robotics-unibg.github.com/VARP

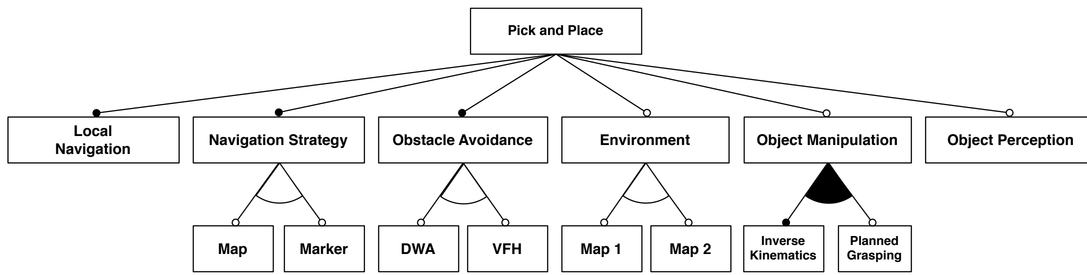

Fig. 5. The feature model that describes the functional variability of the case study. Features with the black circle on the top are *mandatory* and represent functionalities that have to be present in every application. Features with the white circle represent instead *optional* functionalities. White arcs represent *alternative* containments (only one sub-feature can be selected) while black arcs depicts *or* containments (at least one sub-feature has to be selected).

components (some are stable, some others variable). The architectural variability of the product line can be resolved by selecting optional components, their specific implementation, the values of their configuration properties, and the variable connections. In this way starting from the *TSM* it is possible to generate the *CSM*. In other words the *TSM* is a skeleton that can be customized for defining the architecture of a specific application. The CSM can be designed manually or can be generated by means of a set of automatic transformations as described below.

The *TSM* of the Research Camp use case contains around 25 components that provides several functionalities for moving the base and avoiding obstacles, for implementing the different navigation strategies and for manipulating and perceiving objects. According to the desired application different set of functionalities should be used.

While the *TSM* describes the architectural variability of the product line, a second model is required for orthogonally representing the variability in terms of functionalities. This model is the result of a domain analysis that investigates the stable and variable concepts of the functional domain. In our previous works we proposed to model the functional variability by means of the *Feature Models* formalism [14], A feature model is a hierarchical composition of features. A *feature* defines a software property and represents an increment in program functionality. It is good practice to organize the features according to the application requirements and assign them names that are standard de facto in the specific domain. The action of composing features, or in other words selecting a subset of functionalities from all the features contained in a feature model, corresponds to defining the configuration of a software that belongs to the product line and provides the required functionalities. The orthogonality between the *TSM* and the *Feature Model* is important because they are typically defined by engineers with different competencies. Indeed the design of a good architecture (*TSM*) requires advanced software engineering techniques, while the functional variability modeling requires a deep understanding of the application domain, which is part of the robotic experts' knowledge.

The functional variability of the Research Camp use case is described in the feature model depicted in Figure 5. They are described in the following list which also highlights how the selection of a functionality influences the definition of the application architecture.

- **Local Navigation**: this mandatory feature means that every possible application has to provide basic navigation functionalities such as trajectory generation, obstacle avoidance, mobile base control and localization.
- **Navigation Strategy**: the alternative containment of this feature represents the possibility of choosing between the map-based and the marker-based navigation. Selecting one of the two strategies implies using different components that provide different functionalities (these components are selected from the set of components defined in the *TSM*). As a consequence of this choice the connections between the components change. The functionalities required for the navigation have been described in [11] and [12].
- **Obstacle Avoidance**: the alternative containment of this feature represents a variation point regarding the algorithm used for avoiding obstacles. According to the selected feature we have to use different implementations for the component that provides this functionality.
- **Environment**: when the robot performs map-based navigation, it can operate in different environments that are represented by different maps. The alternative containment of this feature allows the specification of the map, which will be used by the components that provide the path planner and the localizer functionalities. These components have a property that allows us to define the path of the map description file.
- **Object Manipulation**: this optional feature represents the capability of manipulating objects. When the engineer want to deploy this functionality the component that provides the inverse kinematics is mandatory while the component for the *Planned Grasped* is optional.
- **Object Perception**: this optional feature represents the last variation point of the product line. This functionality can be used for detecting objects that have to be grasped but also for detecting objects that represent visual markers that the robot has to follow.

The feature models also allow the expression of constraints among the selection of features. In our use case there is a single constraint: when the feature *Object Manipulation* is selected to be part of the application, the *Object Perception*

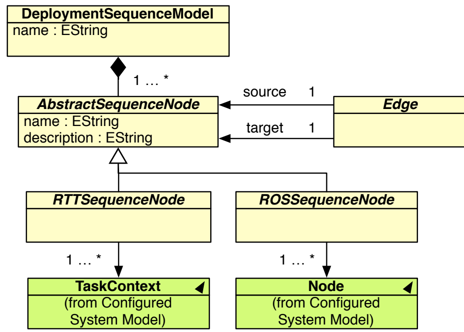

Fig. 6. The deployment sequence meta-model.

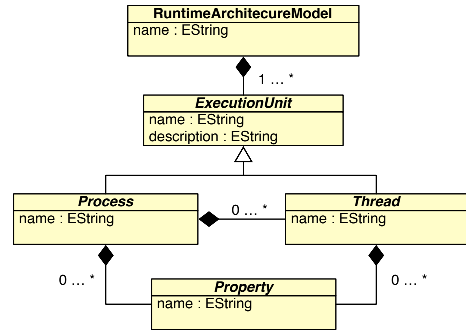

Fig. 7. The runtime architecture meta-model.

feature has to be present.

Once the architecture of the product line has been defined and its functional variability modeled, the engineer has to define how the *CSM* can be automatically generated starting from the *TSM* and a selection of functionalities (i.e. features). This information is encoded in the *Resolution Model*, which specifies how the architectural variability of the *TSM* has to be resolved according to the feature selected in the *Feature Model*. The resolution model associates each feature with a set of transformations, which have to be executed on the *TSM* when the corresponding feature is selected. These transformations allow us to automatically (a) select which components have to be used and set their implementation, (b) configure the values of their properties, and (c) create and remove connections between them. For example the *Resolution Model* of our use case specifies that when the feature *Marker Based Navigation* is selected the components that provide the marker-based functionalities (marker detection, marker path planning, ...) have to be used and connected to the components that provide the local navigation functionalities.

The last step concerns the automatic generation of the *CSM* that describes the architecture of a specific application. The engineer uses the *Feature Selector* to select the set of features that reflect the functional requirements of the desired application. The *Resolution Engine* receives as input the feature selection and by using the three models described above automatically generates the *CSM*. Basically the *Resolution Engine* (a) checks that all the constraints imposed on the Feature Model are satisfied by the feature selection; (b) creates a copy of the *TSM*; (c) modifies this copy by applying only the transformations (described in the *Resolution Model*) that are associated with the selected features.

In our use case the engineer can deploy several applications that go from a simple map-based navigation (the selected features are *Local Navigation*, *Marker* and one of the *Obstacle Avoidance* sub-features) to the pick and place of objects with marker-based navigation.

Despite the product lines support the separation of stable and variable concepts, new requirements can appear over the time, which imply the evolution of the product line and the definition of new functionalities and new components. The SPL literature documents several approaches for the evolution of the architectures [15] and the feature models [16].

D. System Deployment Phase

During the *System Deployment* phase the previous separated architectural aspects and models are composed into a coherent *Deployment Model (DM)* (meta-model in Figure 6). This is done by mapping the component architecture defined in the *CSM* to executable units such as processes and threads modeled in the *Runtime Architecture Model (RAM)* (see meta-model in Figure 7) and by mapping the executable units on hosts defined in the *CAM*. In addition, the *Deployment Sequence Model (DSM)* (meta-model in Figure 8) defines the order in which the components are deployed. The optional, sequential deployment of components might be required for testing purposes or to cope with the complexity imposed by large-scale component networks (e.g., step-wise and layered deployment). We model the sequence as a directed acyclic graph (DAG).

In the Research Camp use case this enables us to encode that the arm components must be deployed before any other components since it performs a homing procedure. Similarly to the *CAM*, the *RAM* is inspired by the AADL meta-model on execution units. For the pick and place use case, depending on the selected functionalities, the components are deployed as processes or threads.

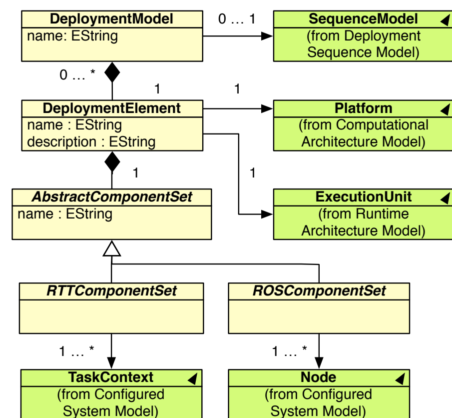

Fig. 8. The deployment meta-model.

IV. A Domain-specific Language for Deployment

To support the model-based deployment approach introduced in Section III we developed a Domain-specific Language (DSL), which enables system integrators to create concrete domain models and to generate (enriched) framework-specific deployment descriptions. We defined the following language requirements (see also Kolovos *et al.* [17]).

- **Orthogonality.** The DSL primitives and abstractions shall correspond to the domain concepts introduced in Section III. More precisely, each language construct should represent one distinct concept in the domain in order to keep them orthogonal and to ease the development of code generators.
- **Supportability.** To increase acceptance, a DSL user demands an infrastructure which supports typical activities such as model creation, editing, deleting, and transforming.
- **Quality.** The DSL should support the development of faultless deployment descriptions via the language itself or through tooling.

A. Implementation

We developed a language infrastructure which allows model editing, validation and generation (see *Resolution* and *Deployment Configuration Engines* in Figure 1). The tools related to feature models and software architecture are graphical and based on the Eclipse Modeling Framework (EMF)[6], while the rest of the infrastructure is based on the Eclipse framework Xtext[7]. For the deployment DSL itself the straightforward integration of Xtext in Eclipse helped us to develop the language in a rapid manner. For instance, a DSL language editor with syntax highlighting and code completion can be automatically generated from the meta-models. This built-in support helped us to focus on the development of the language itself. Even though, for some targeted DSL user the use of Eclipse as a language framework might be arguable, the syntax specification is independent of Eclipse and hence, could be used for other frameworks as well. In addition, the deployment description generation facilities have been realized with Xtend,[8] a special-purpose programming language which eases the writing of code generators through powerful template expressions.

B. Syntax and Example

In Figure 9 an excerpt of the deployment DSL is shown. Here a component set is defined, which imports a set of components from the *Configured System Model* that models the architecture of the perception components and components related to arm control. Additionally, two different computational hosts are imported which are modeled in two different *Computational Architecture Models*. In the example, a deployment sequence is defined (the arm control should be deployed before the perception) and later used in the deployment model. To ease modeling we equipped the DSL with some syntactic sugar in the form of high-level keywords such as *deploy* and *on*.

Deployment DSL

```
ComponentSet perception {
  import bounding_box from CSM.perception
  //...
}

ComponentSet arm {
 import armctrl from CSM.manipulation
 //...
}

import Platform internal_pc from CAM.embeddedPC
import Platform external_pc from CAM.laptop

DeploymentSequence {
 deployment_sequence s {
  arm.armctrl before perception.bounding_box
 }
}

DeploymentModel dm {
 deploy arm on internal_pc
 deploy perception on external_pc
 apply_sequence s
}
```

Fig. 9. An excerpt of the deployment DSL.

C. Constraint Checking

Once domain concepts are represented as meta-models we can also define constraints on concrete domain models which conform to these meta-models. In the work presented here, we applied the OCL (Object Constraint Language)[9] formalism to model constraints. We distinguish between two types of constraints:

- **Atomic constraints** which are valid for domain models conforming to one single meta-model (e.g., constraints for the *CSM)*.
- **Composition constraints** which appear when we compose the models in the deployment model.

For instance, every *CSM* is either a ROS-based or Orocos-based component network. Therefore, we modeled several framework-specific constraints such as those shown in Figure 10. Here, we ensure that the types of an input and output port match and that a connection between them is well-configured according to the underlying Orocos RTT component model. Another example of atomic constraints is

context ConnectionPolicy
inv: **self.***inputPort.type* = **self.***outputPort.type*

context ConnectionPolicy
inv: **self.***name.size()* <> 0 **and self.***bufferSize* <> 0

Fig. 10. An excerpt of the Orocos RTT-specifc OCL constraints.

[6] http://www.eclipse.org/modeling/
[7] http://www.eclipse.org/Xtext/
[8] http://www.eclipse.org/xtend/
[9] http://www.omg.org/spec/OCL/2.0/

shown in Figure 11 where we ensure that every platform has a non-empty hostname. In addition, composition constraints are defined which are more complex. Due to space limitation we present only some of them in natural language:

- Each component of a *CSM* needs to be deployed on one host defined in the *CAM*.
- Each component of a *CSM* may not be deployed several times on a host defined in the *CAM*.
- Each connection between components in the *CSM* implies that the components are deployed on the same host defined in the *CAM*, or that the different hosts are connected with each other.
- Each component of a *CSM* which are deployed as thread and belongs to one process defined in the *RAM* must run on the same host defined in the *CAM*.

context Platform **inv**: **self.***hostname.size() <> 0*

Fig. 11. An excerpt of the platform OCL constraints.

D. Model Generation for Robot Software Frameworks

To make the DSLs usable in a working environment and to show the feasibility of the overall approach we developed a proof of concept model generator for ROS. As the presented DSL is framework-independent, the DSL concepts and abstractions have to be transformed to framework-specific concepts. In this work we generate roslaunch configuration files. The roslaunch tool is the major deployment mechanism in ROS, which reads configuration files (specified in a XML format) and launches nodes on specified hosts. We used the top-level elements of the XML specification to transform the corresponding DSL concepts. For each component set specified in the DSL a dedicated ROS launch file is generated. In addition, each component specified in the *CSM* and included in the *DM* is transformed to a ROS node (`<node>` tag) in the launch file. The host information for each component is used for the `<machine>` tag in the XML specification.

Even though we are able to generate valid roslaunch files, it is not possible to transform concepts such as the deployment sequence, which is not considered in the roslaunch tool. When the user initiates the generation of a roslaunch file, based on a deployment model where a deployment sequence is specified, a warning will appear. Generation for other frameworks such as Orocos RTT will follow the same approach, namely language elements will be transformed to the primitives available in the framework-specific deployment infrastructure. For example, in Orocos RTT we would generate Lua deployment scripts. Here, the application of a deployment sequence is possible as the Orocos RTT component model foresees a component lifecycle which can be used to start and deploy components in a specific order.

V. Discussion and Related Work

The BRICS Research Camp use case exemplifies the huge variability which characterizes modern robot applications. To cope with this variability and the effects on deployment we keep the architectural aspects separated and compose them as late as possible. The DSL snippet shown in Figure 9 exemplifies this. To modify the deployment setting (e.g., to deploy components on another host) simply involves the change of a couple of lines. In addition, the feature-oriented approach allows a quick generation of the architecture of a specific application by simply selecting its functionalities and reusing existing solutions.

A. Requirements Revisited

- **Orthogonality.** In general, it is possible to transfer the architectural aspects to concrete language abstractions. Surprisingly a handful high-level keywords such as the `before` statement are sufficient to encode rather complex situations. In addition, the models are orthogonal as possible. In fact the sequence and the deployment model are the only models that are affected by changes in other models.
- **Supportability.** The language infrastructure is Eclipse-based and therefore graphical. However, we observed that developers hesitate to use this environment because they are used to work with command-line tools. Nevertheless we argue that the faster development time that comes with using the tools, they will consider adopting the graphical approach.
- **Quality.** Thanks to the constraint checking we can identify incomplete and erroneous information already during modeling. Furthermore, the powerful composition constraints are aimed for the situations described in Section II, where a change in one model affects another model. This automatic checking is very advantageous when compared to the manual inspection of framework-specific configuration and deployment scripts.

B. Lessons Learned

We can report the following lessons with respect to the overhead introduced by the DSL.

- From DSL users perspective an external DSL needs to be well integrated in the overall development workflow. Otherwise it is unlikely that the DSL will be used in a daily working environment. In our future work we will therefore integrate the deployment DSL with BRIDE[10] which is a model-driven engineering IDE for robotics based on Eclipse. The *TSM* can be already designed with BRIDE and the feature and resolution models are already integrated within BRIDE. Furthermore, the resulting *CSM* can be visualized with BRIDE.
- From DSL developers perspective there is clearly the overhead to develop the language itself, corresponding code generation facilities and tooling. For those activities good domain knowledge is demanded. However, modern DSL frameworks such as those applied in this work help to master those activities and to focus on the language design itself.

[10]`http://www.best-of-robotics.org/bride/`

C. Related Work

Quite recently, model-based software development approaches have become popular in robotics. In [18] the RobotML DSL is presented. The DSL includes a domain model which contains four packages, namely, architecture, behavior, communication and deployment. The deployment package specifies a set of constructs that can be used to define the assignment of a robotic system to a target platform (middleware or simulator). Hence, deployment in RobotML refers to platform-specific code generation. In contrast to our approach the link to the development process is missing and constraints are not checked that elaborately. In [5] the deployment infrastructure for OpenRTM is presented. Here, deployment is considered as a part of component and system lifecycle management. The authors present a pragmatic approach to deployment, focusing on the implementation level details (e.g., how manager services interact and how components are instantiated). To realize the deployment process so-called profile descriptions (component and system profiles) are required. One can view component and system profile descriptions as DSLs. However, in contrast to our approach the focus is on elaborated tooling and not on the composition of different architectural aspects. Similarly, in Schlegel *et al.* [6] a platform description model is defined which serves as a deployment target. However, the explicit separation of architectural aspects has not been adressed.

VI. Conclusion and Future Work

In this paper we introduced a model-based approach to software deployment in robotics. We separated different architectural aspects, which are linked to a software development process in robotics, and introduced a domain-specific language. The DSL and the feature-oriented approach allows us to model these aspects in a declarative and framework-independent manner. Therefore, we can reuse models and can easily make changes in the deployment setting. To ensure certain properties already at design time we make use of constraints which are checked before the actual deployment on the robot. To avoid ambiguities in a model-based approach one has to agree on the available concepts, their terminology and their meaning. In particular in the presence of engineers and developers with different areas of expertise. However, so far we have not experienced any ambiguities in the applications of our model-based approach as the concepts are widely accepted. In the future we want to make the models also available at runtime in order to support deployment changes, e.g., in the presence of hardware failures. Furthermore, we aim to support several robotic software frameworks with our model generation such as Orocos.

Acknowledgement

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics). In addition, Nico Hochgeschwender received a PhD scholarship from the Graduate Institute of the Bonn-Rhein-Sieg University of Applied Sciences which is gratefully acknowledged. Furthermore, the ongoing support of the Bonn-Aachen Institute of Technology is gratefully acknowledged.

References

[1] B. Graf, U. Reiser, M. Hagele, K. Mauz, and P. Klein, "Robotic home assistant care-o-bot 3 - product vision and innovation platform," in *Proceedings of the IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO)*, 2009.

[2] W. Meeussen, M. Wise, S. Glaser, S. Chitta, C. McGann, P. Mihelich, E. Marder-Eppstein, M. Muja, V. Eruhimov, T. Foote, J. Hsu, R. B. Rusu, B. Marthi, G. Bradski, K. Konolige, B. P. Gerkey, and E. Berger, "Autonomous door opening and plugging in with a personal robot," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2010.

[3] 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 *Proceedings of the Workshop on Open Source Software held at the International Conference on Robotics and Automation (ICRA).*, 2009.

[4] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 2003.

[5] N. Ando, S. Kurihara, G. Biggs, T. Sakamoto, H. Nakamoto, and T. Kotoku, "Software deployment infrastructure for component based rt-systems," *Journal of Robotics and Mechatronics*, vol. 23, no. 3, pp. 350–359, 2011.

[6] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design abstraction and processes in robotics: From code-driven to model-driven engineering," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2010.

[7] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[8] G. K. Kraetzschmar, A. Shakhimardanov, J. Paulus, N. Hochgeschwender, and M. Reckhaus, "Best practice in robotics (brics) deliverable d-2.2: Specifications of architectures, modules, modularity, and interfaces for the brocre software platform and robot control architecture workbench," 2010.

[9] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2006-TN-011, 2006.

[10] L. Gherardi and D. Brugali, "An eclipse-based feature models toolchain," in *6th Italian Workshop on Eclipse Technologies (EclipseIT 2011)*, Milano, Italy, September 22-23 2011.

[11] D. Brugali, L. Gherardi, A. Luzzana, and A. Zakharov, "A reuse-oriented development process for component-based robotic systems," in *3rd International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2012)*, Tsukuba, Japan, November 5-8 2012.

[12] L. Gherardi, "Variability modeling and resolution in component-based robotics systems," Ph.D. dissertation, Università degli Studi di Bergamo, 2013.

[13] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[14] K. Kang, "Feature-oriented domain analysis (FODA) feasibility study," DTIC Document, Tech. Rep., 1990.

[15] M. Svahnberg and J. Bosch, "Evolution in software product lines: Two cases," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 6, pp. 391–422, 1999.

[16] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski, "Model-driven support for product line evolution on feature level," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2261–2274, 2012.

[17] D. S. Kolovos, R. F. Paige, T. Kelly, and F. A. C. Polack, "Requirements for Domain-Specific languages," in *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD 2006)*, 2006.

[18] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications." in *SIMPAR*, ser. Lecture Notes in Computer Science, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, Eds., vol. 7628. Springer, 2012, pp. 149–160.