BRIDE - A toolchain for framework-independent development of industrial service robot applications

Alexander Bubeck, Fraunhofer IPA, alexander.bubeck@ipa.fraunhofer.de, Germany
Florian Weisshardt, Fraunhofer IPA, florian.weisshardt@ipa.fraunhofer.de, Germany
Alexander Verl, Fraunhofer IPA, alexander.verl@ipa.fraunhofer.de, Germany

Abstract

Software integration is still a challenging and time consuming task and therefore a major part of the development of industrial and domestic service robot applications. The presented toolchain BRIDE is able to streamline this process by the separation of user roles and the separation of developer concerns of software components to ensure a frame-work independent implementation. The impact of the BRIDE toolchain in the development process is demonstrated in a case study on the SyncMM mobile manipulation control framework.

1 Introduction

Today's autonomous service robots are complex, software-intensive and highly integrated systems with impressive capabilities concerning mobile manipulation. Although many capabilities are developed open source, software quality, platform dependencies and the missing encapsulation of system complexity reduce the chance of reusing software and slowing down development processes towards an application on these complex robot systems. During the last decade several initiatives have been created to support software development in robotics through the promotion of functionality-rich robot software frameworks such as Orocos, OpenRTM, Player and ROS. These frameworks extend the general ideas of component-based software design and development by reusable software components with well-defined interfaces as shown in [1]. But as Brugali *et al.* [2] exemplified, the reuse of complete component-based applications and architectures remains challenging. In particular, the reuse concerning changing platform, task and environment conditions is still an open issue.

In order to tackle these problems related domains adopted the Model Driven Engineering (MDE) approach. Primarily, the goal of MDE is to improve the process of generating code from abstract models that describe a domain and thereby increase development speed, enhance software quality, achieve systematic separation of concerns and foster software development by the reuse through models [3]. In the work presented here, model driven engineering is transfered to the domain of robotics in order to carry the concept of the systematic separation of different concerns and the separation of user roles in the development process, as described in section 2. In contrast to other applications of MDE in the robotic domains, as in Smartsoft [4] or Proteus [5], this approach is not mapping UML profiles to robotic software but creating a domain specific language directly for the existing robot framework ROS. Because of this approach, the current development workflows of the robot domain experts can be directly integrated in the MDE workflow and thus the fast adaption of the tool chain can be supported. Furthermore, the usage of proprietary components of ROS, which are not based on the MDE tool chain, is possible and transparent to the end user. The model driven approach is implemented in an Eclipse based tool chain mainly targeting the commonly used robot software framework ROS. Additionally a targeting for the realtime-capable middleware OROCOS was realized, allowing the usage of both middleware in parallel to implement software architectures with encapsulated realtime capabilities. This implementation is explained in section 3 and the application on the mobile manipulation control system SyncMM analyzed in section 4.

2 A model driven approach to ROS

The Robot Operating System (ROS) is one of the most commonly used software frameworks in robotics research today. The large user and developer community is a benefit but also a challenge for the usage of ROS on complex robot systems. The software concepts for creating a ROS component are clearly defined in a documentation but are implicitly coupled with the code after development, creating a large variety of quality, style and behaviors of ROS components in the community. Therefore, the selection of and the integration into a specific platform or application remains a challenging task. The aim of model driven engineering is to encapsulate complexity and to enforce interfaces, architectures and user roles for a specific software domain. Therefore, it provides the opportunity to increase the quality of a software produced, as many applications in computer science have shown.

The Object Management Group, which is a world wide organization for model driven software approaches, defines model driven engineering in a multiple layer architecture

[1] http://www.omg.org/mda/

ISBN 978-3-8007-3601-0

that form a MDE tool chain[1]. The lowest layer (M0) is the actual running code that conforms to a specific implementation model (M1). The mechanisms of describing this implementation model are defined in the meta-model (M2) layer. Usually an additional superordinate layer provides the toolchain with the mechanism of describing meta-models, the meta-meta-model (M3). This model driven engineering concept is also visualized in the overview figure 1.
The application of this model driven engineering architecture to the ROS component framework allows to model different concerns, discussed below, in a meta-model description. Based on this, the different end-users can implement specific models for the concerns of their development phase. MDE then gives the opportunity to auto-generate implementation code based on the models specified in the M1 layer. As the framework dependent aspects of the component are explicitly modeled in the M1 layer the code generation can separate the code into framework-dependent and framework-independent parts. In contrast to mapping high-level meta-models, such as UML or SysML, to the domain of robotics, a direct modeling of the ROS framework was chosen. This reduces the number of non-formalized assumptions in the code generation from the M1 to M0 layer and allows to reuse components that were not created by the MDE approach because there framework mechanisms can be modeled directly afterwards.

2.1 Separation of Concerns

A software component has to implement a number of different aspects in order to be functional in a component framework. These aspects usually can be associated with a specific concern of the usage of this component. Most of the times, the reuse of software means modification or exchange of one or many specific concerns. For example moving a specific control component from one system to another could only require a change in the configuration of the control component. Therefore, explicit separation of these software aspects from each other in the implementation will improve the reuse of the components. In software engineering one can distinguish the configuration, the communication, the coordination and the actual computational part of a component [6]. Additionally, it makes sense to further distinguish the mechanisms of composition in case of systems with multiple components.

2.2 Separation of User Roles

The development of a robot application takes place in different development phases, which implicate different activities and require different knowledge of the developer. This work focuses on the explicit separation of two phases of the development process the capability building and the system deployment. The capability building usually requires specific and detailed knowledge of the domain of the capability and therefore is generally realized by domain experts. These developers usually lack knowledge or capacity in order to also handle the second phase, the system deployment. On the other side, software engineers that can handle the complexity and structure of large component systems, as found on complex robot systems like Care-O-bot® 3 and rob@work 3, do not dispose the detailed knowledge of all domains of robotics.

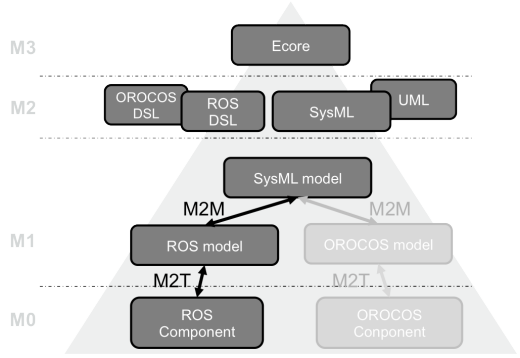

Figure 1: Layers of domain specific languages and models implemented in the BRIDE tool chain.

3 Implementation of BRIDE

The Eclipse Modeling Framework (EMF) and associated projects provide a toolbox to develop model driven tool chains. More precisely, we used the EMF language Ecore (M3 in figure 1) to define a domain specific language (DSL) for ROS (M2 in figure 1). Based on this DSL, an infrastructure including a graphical editor and code generator was developed. This infrastructure, called BRICS IDE (BRIDE), allows to model capabilities and systems in an explicit manner with respect to the corresponding phase in the development process. The code generator creates runnable ROS code and structures the software into platform independent and platform dependent parts. The platform dependent parts are fully autogenerated.
The general programming abstractions and concepts available in ROS were formalized and used to develop the meta-model in BRIDE. This ROS DSL is the origin of our visual domain specific language and the code generation facilities. The concepts, which are modeled in the meta-model, are available as primitives in our tool chain for later use on the M1 layer. In order to ease the model iteration and tooling the model is structured as a tree with the architecture primitive as root node. Further, the package contains topics, services, nodes and actions as core primitives (see [5]). The meta-model is separated into five concerns, namely Computation, Coordination, Configuration, Communication and Composition. This separation enables the systematic formalization of constraints for each concern. For instance, a Service demands at least one ServiceServer (Communication) and every Parameter demands a name, type and value (Configuration). These

ISBN 978-3-8007-3601-0

and more constraints are checked in the code generation facilities.

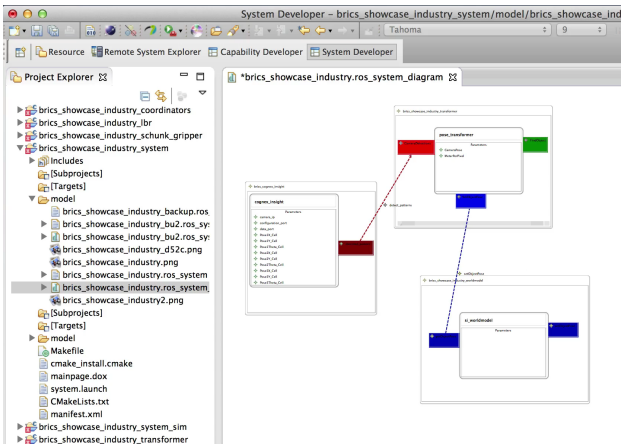

Figure 2: The graphical representation of a subsystem model for a rob@work 3 application in the BRIDE editor.

Graphical and textual model editors based on the ROS DSL for the creation of component models (M1 in figure 1), which e.g. represent the different aspects of a component, can be derived. The generation of multiple model editors based on one domain specific language was chosen to map the different aspects formulated in the DSL to the user roles in the development process. Since the editors are developed using a model driven process in Eclipse, they are directly linked to the ROS Ecore model. Changes in the Ecore model are therefore directly reflected in the graphical editors.
Once the capability and system models are created, they are automatically transformed to another representation, usually source code (M0 in figure 1). This so-called model-to-code generation was realized by implementing templates for the Epsilon generator, which is part of the Eclipse tool chain. This approach allows to enforce code quality and standards and to transparently support different target languages for the implementation. In case of the capability model templates for the generation of C++ ROS components and Python ROS components were realized. The code generation takes care of creating a node skeleton as well as the necessary tool chain files such as manifest.xml, the dynamic reconfigure files, the CMakeLists.txt etc., finishing with a ready-to-compile C++ or Python ROS component. In the prepared user code part of the package the capability builder now can implement the specific capability independent of the component around it. The information described in the system deployment model is transferred to a ROS launch file starting the nodes, renaming the publisher and subscriber topics to the configured one and setting the parameters to the configured values. As the auto-generated code and the implementation of the user can be distinguished, the model-to-code generation can be executed multiple times when there are changes in the model resulting in only minor changes by the user if there are changes in the interfaces. Due to this fact, the templates for the different targets can be enhanced iteratively and the improved code can be easily disseminated to many implementations.
In case of the capability models, the code generation is enforcing a standardized computational model in the ROS components. This is an extension of the ROS component model, that is necessary to allow a clear separation of ROS dependent and independent code. Only with this strict separation the transfer to other robotic software frameworks can be implemented. The computational model enforces the usage of a configure function, that is triggered during start of a component to allow configuration of the necessary objects and interfaces. During runtime of the component, the capabilities have to be triggered from within an update function that is either called in a sequential way or called by data received on defined input data ports (in case of ROS: subscriber). With this approach, code generation of the capabilities into an OROCOS component is possible in BRIDE in addition to the creation of ROS components. The generation is realized in a way that allows the usage of the OROCOS component integrated in the ROS application using the rtt_ros_integration project[2].

[2]http://wiki.ros.org/rtt_ros_integration

4 The BRIDE tool chain in use

As a demonstration of the application of the model driven engineering tool chain to a real world problem, the MDE-based refactoring of the SyncMM controller framework [7] for Care-O-bot® 3 will be presented in the following section. The code of the original implementation, with a "hand-written" integration in ROS, will be compared to the refactored version for experimental validation of our approach.

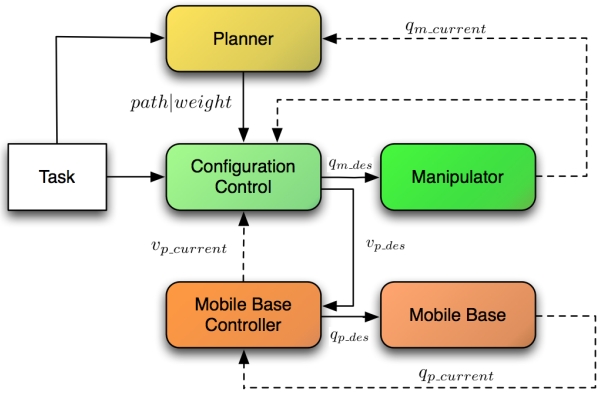

Figure 3: Functional overview of the SyncMM framework with $current$ and desired joint positions (q) and Cartesian velocities (v) of $p$lattform and $m$anipulator

ISBN 978-3-8007-3601-0

4.1 The SyncMM Mobile Manipulation Controller Framework

The synchronous mobile manipulation (SyncMM) controller framework of Care-O-bot® 3 consists of a component based structure for the subcomponents of the robot; the non-holonomic over-actuated mobile base and the redundant manipulator and the overall control of the kinematic chain with redundancy handling. A functional overview of the different components is shown in Figure 3. While the manipulator controller is implemented using a standard impedance controller inside the KUKA LBR controller, the mobile base controller cob_undercarriage_control [8] transfers the Cartesian velocities in a spherical space in order to directly control the instantaneous center of rotation (ICR) of the base while fulfilling the kinematic constraints. This allows singularity avoidance, resolution of redundancies in the mobile base and smooth trajectory tracking.

The cob_configuration_control component takes care of the redundancy resolution of the combined kinematic chain of manipulator and base giving the controller the ability to control augmented tasks in addition to the end effector tracking task. Joint angle avoidance, manipulability maximation and path following are examples of such augmented tasks. Finally, another component is implementing a high level controller for trajectory following, allowing the Care-O-bot® 3 to execute tasks that require mobile manipulation movements such as opening of doors or cupboards.

In the original implementation, the different functional blocks were implemented in ROS nodes "by hand" and connected by a mixture of direct configuration of topics and services in the source code and configuration in multiple launch files.

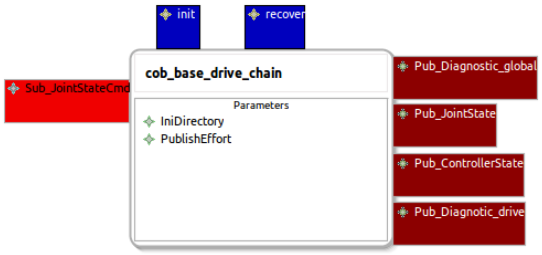

Figure 4: Graphical representation of the cob_base_drive_chain component

4.2 Implementation of capabilities

For the model driven refactoring of SyncMM for each of the different controllers, a component model was created using the graphical capability modeling editor in BRIDE. The communication mechanisms and parameters of the controller were modeled graphically, as can be seen for the example of the cob_base_drive_chain in Figure 4. The textual representation of the component used for the auto-generation of the component skeleton is listed in Listing 1. The auto-generation can be done for the different target languages C++ and Python in the same way. In this example, a C++ implementation was chosen. After the auto-generation the existing controller code was ported to the provided classes for platform independent user code. The implementation details of ROS techniques such as topics or dynamic_reconfigure are encapsulated from the component developer.

Listing 1: XML representation of capability model for cob_base_drive_chain

```
<?xml version="1.0" encoding="UTF-8"?>
<ros:Package xmi:version="2.0" xmlns:xmi="http://www.
      omg.org/XMI" xmlns:ros="http://ros/1.0" name="
      cob_base_drive_chain" author="Alexander Bubeck"
      description="Driver for mobile base" license="LGPL"
      >
 <node name="cob_base_drive_chain">
  <publisher name="Pub_JointState" msg="sensor_msgs::
     JointState"/>
  <publisher name="Pub_ControllerState" msg="
     pr2_controllers_msgs::JointTrajectoryControllerState"
     />
  <publisher name="Pub_Diagnostic_global" msg="
     diagnostic_msgs::DiagnosticArray"/>
  <publisher name="Pub_Diagnotic_drive" msg="
     diagnostic_msgs::DiagnosticStatus"/>
  <subscriber name="Sub_JointStateCmd" msg="
     pr2_controllers_msgs::JointTrajectoryControllerState"
     />
  <serviceServer name="init" msg="cob_srvs::Trigger"/>
  <serviceServer name="recover" msg="cob_srvs::Trigger"
     />
  <parameter name="IniDirectory" value="Platform/
     IniFiles/" type="string"/>
  <parameter name="PublishEffort" value="false" type="
     bool"/>
 </node>
 <depend>diagnostic_msgs</depend>
 <depend>sensor_msgs</depend>
 <depend>cob_srvs</depend>
 <depend>pr2_controllers_msgs</depend>
 <depend>cob_canopen_motor</depend>
</ros:Package>
```

4.3 Implementation of System Architecture

After all controllers had been ported, the system configuration was realized by using the system deployment editor in BRIDE. The component code of the capabilities is generated in a way that all topics, services and actions are disconnected by default, meaning that all connections have to be configured explicitly. Since this configuration can be realized graphically, it provides the system engineer with the full overview of the communication and dependencies of the components in the system. Additionally, all parameters defined by the capability builder are accessible to the

ISBN 978-3-8007-3601-0

system engineer, making direct deployment time configuration (e.g. for CAN devices) possible. A part of the SyncMM system configuration in BRIDE can be seen in the screenshot in Figure 1. Since the launch file for the system is generated automatically the system deployer can completely configure the SyncMM framework graphically. The configuration is implemented in a single launch file in contrast to the original implementation.

4.4 Evaluation

After the refactoring with the MDE tool chain the performance of the system was compared to the original SyncMM implementation using a similar use case of opening a cupboard as was demonstrated with the original version (see Figure 5). No difference in performance was noticed. This was expected since no changes in the algorithms of the controllers were made.

Figure 5: Execution of the SyncMM controller

As can be seen in Table 1, for each component the amount of code that had to be written by the user was reduced, since the ROS framework code was auto-generated from the model. On components with small computational parts, as the cob_config_controller, the ratio of auto-generation is very high. This not only accelerates the development, it also reduces the possibility of software bugs and guarantees a certain code quality. The only code coming from the ROS system, in the part, the capability developer accesses, are the ros_msgs definitions, which are structural data classes that are independent of the ROS framework. The capability developer did not have to dispose of any knowledge of the ROS framework mechanisms.

Table 1: Statistics of auto-generated lines of code

Component	manual-coded	auto-generated	%
cob_undercarriage_control	1397	260	18.61
cob_base_drive_chain	2292	256	11.1
cob_config_controller	594	289	48.6
cob_cartesian_trajectories	781	296	37.9

Table 2: Comparison of ROS independent code in number of lines

Component	SyncMM	Model driven SyncMM
cob_undercarriage_control	965	1397
cob_base_drive_chain	1998	2292
cob_config_controller	357	594
cob_cartesian_trajectories	0	781

Table 2 shows that that the code that is independent of ROS is increased. This is a major improvement for the reusability of the algorithms in different frameworks, especially for the algorithms that were tightly integrated with ROS, as the cob_cartesian_trajectories component. For demonstration, one component, the cob_base_drive_chain, was additionally ported to OROCOS using BRIDE. The ROS independent part of the component, created with the MDE tool chain, could completely be reused. The autogenerated ROS part of the component was regenerated by using the OROCOS code generation templates, without any manual input. In this example the amount of code generated was 86 lines.
Since the original SyncMM implementation used three different launch files and had implicit connections between components in the source code, it was impossible to configure the system deployment without knowledge of the component internals. In contrast, this information was completely transparent to the system deployer in the case of the MDE implementation. Additionally, the configuration of every deployment aspect in one launch files accelerated the testing process afterwards.

5 Conclusion

Adopting the concepts of model driven engineering to robotics has high impact on reusability of existing software and the performance of robot application, system and capability development as was shown in this work for the use case of mobile manipulator control. Furthermore, the separation of user roles similar to product development processes in industry can lower the entry barrier towards complex mobile manipulation system. The tool chain shown here is promoted inside the ROS community for the development of applications and components on additional platforms, in e.g. the ROS industrial initiative. An integration with an additional different robot software framework, OROCOS, has been demonstrated. As the number of users of BRIDE rises more robot frameworks will be supported.

References

[1] D. Brugali and A. Shakhimardanov, “Component-based Robotic Engineering Part II: Systems and Models,” *IEEE Robotics and Automation Magazine*, vol. 17, no. 1, pp. 100–112, 2010.

[2] D. Brugali, L. Gherardi, A. Luzzana, and A. Zakharov, “A Reuse-Oriented Development Process for

ISBN 978-3-8007-3601-0

Component-based Robotic Systems," in *International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2012)*, 2012.

[3] T. Stahl and M. Völter, *Model-Driven Software Development*. Wiley & Sons, 2006.

[4] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering," in *Simulation, Modeling, and Programming for Autonomous Robots - Second International Conference, SIMPAR 2010, Darmstadt, Germany, November 15-18, 2010. Proceedings*, vol. 6472. Springer, 2010, pp. 324–335.

[5] G. Lortal, S. Dhouib, and S. Gérard, "Integrating ontological domain knowledge into a robotic DSL," in *Proceedings of the 2010 international conference on Models in software engineering*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 401–414.

[6] M. Radestock and S. Eisenbach, "Coordination in Evolving Systems," in *International Workshop on Trends in Distributed Systems*, 1996.

[7] A. Bubeck, C. Connette, M. Haegele, and A. Verl, "SyncMM - A Reactive Path Planning and Control Framework for the Mobile Manipulator Care-O-bot 3," in *Proceedings of the International Symposium on Robotics (ISR)*, 2012.

[8] C. Connette, A. Pott, M. Hägele, and A. Verl, "Addressing input saturation and kinematic constraints of overactuated undercarriages by predictive potential fields," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2010*, pp. 4775–4781.

Acknowledgments

The work leading to these results has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement no 609206.

ISBN 978-3-8007-3601-0