Journal of Software Engineering for Robotics 6(1), December 2015, 33-57
ISSN: 2035-3928

Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems

Jan Oliver RINGERT[1,2] Alexander ROTH[1] Bernhard RUMPE[1] Andreas WORTMANN[1]

[1] Software Engineering, RWTH Aachen University, Germany
[2] School of Computer Science, Tel Aviv University, Israel

Abstract—Engineering software for robotics applications requires multi-domain solutions. Model-driven development (MDD) promises efficient means for developing domain-specific and reusable models of robotics software. Code generators transform these models into executable code for specific robotic platforms. Robotics is heterogeneous and robotics applications pose various challenges to leveraging the potential of MDD. Combinations of modeling languages and platforms are often problem-specific. Generative software development for multi-domain applications requires both the effective integration of modeling languages and composition of code generators. We present the extensible MontiArcAutomaton architecture modeling framework for the generative development of robotics applications with a strong focus on language integration and generator composition. Its core modeling language is a component & connector architecture description language that can be extended with problem-specific component behavior modeling languages. We present how MontiArcAutomaton supports syntactic and semantic behavior modeling language integration to describe component behavior in most suitable modeling languages and code generator composition to synthesize code from integrated models. We sketch a process for model-driven development of robotics applications using MontiArcAutomaton.

Index Terms—Model-Driven Development, Software Architectures, Software Language Composition, Code Generation.

1 Introduction

Successful development of robotics applications requires solutions to challenges from different domains. Non-trivial robotics applications require to integrate solutions from domains such as navigation, communication, trajectory planning, and software engineering. To achieve this, robotics software is usually developed by teams of domain experts providing solutions for specific problems on specific platforms. This leads to monolithic robotics software and seriously challenges reusability for different projects or platforms [1], [2].

To enable the reuse of functionality and subsystems, the structuring and composition mechanisms of component-based software engineering (CBSE) have been applied to robotics software engineering [3], [4], [5], [6]. These approaches are mainly based on the exchange of binary or source code components and thus tied to specific platforms and general-purpose programming languages (GPLs). Software integration thus requires in-depth knowledge of these GPLs and software engineering principles. Model-driven development (MDD) reduces this "conceptual gap" [7] between the problem domains and software engineering. Models enable the representation of software systems at a higher level of abstraction than GPLs and thus allow for greater reuse. In addition, domain-specific software descriptions using models specific to the problem domain reflect the heterogeneity of the developed system and its concerns [8]. In combination with platform-specific code generators and analyses, models can serve as primary development artifacts which increases the software's comprehensibility and reuse on different platforms [2], [9]–[11]. These potential benefits of MDD do not come for free and robotics applications pose various challenges to successful application of MDD. For example, the reuse of models for different platforms is possible in theory but in practice it requires

Regular paper – *Manuscript received September 4, 2014; revised December 14, 2015.*

- *This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IM12008C. The responsibility for the content of this publication is with the authors. J. O. Ringert acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research.*

[RRRW15b] J. O. Ringert, A. Roth, B. Rumpe, A. Wortmann:
Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems.
In: Journal of Software Engineering for Robotics (JOSER), 6(1):33–57, 2015.
www.se-rwth.de/publications

efficient means to integrate or develop platform-specific code generators. As another example, the heterogeneous challenges of engineering robotics applications require adequate means to extend and modify modeling languages to adapt to emerging requirements. Finally, the complex nature of robotics hardware and software demands integration of models and generated code with legacy components and handcrafted code.

To address these challenges of MDD for robotics applications we have developed the MontiArcAutomaton architecture modeling framework consisting of modeling languages, code generators, and powerful extension mechanisms. The goals of MontiArcAutomaton are to provide

- an extensible and **compositional modeling language** for the structure and behavior of robotics applications,
- an extensible and compositional **code generator framework** allowing reuse across different platforms and different modeling language aggregates,
- **integration mechanisms of legacy code** and platform-specific components, and
- a **methodology** for MDD of robotics applications.

With MontiArcAutomaton, robotics applications are modeled as component & connector (C&C) software architectures. While the logical architecture of applications is prescribed in C&C models, behavior of components is defined from component composition or embedded behavior modeling languages. The separation of software into logical components enables independent development and integration via well-defined software interfaces. This allows for encapsulation of platform-specific code or legacy code and forms the basis for syntactic and semantic behavior modeling language integration.

First, extension mechanisms of the modeling language allow embedding domain-specific and problem-specific behavior descriptions into component definitions. Language engineers define the syntax of the embedded languages and use existing facilities for defining well-formedness checks (also called *static semantics*) for models of the combined languages. The well-formedness of models is defined on an intra-language level, i.e., between elements of models of the new language, and on an inter-language level, i.e., between elements of models of the new language and those of existing languages.

Second, MontiArcAutomaton comprises powerful compositional code generation facilities for the transformation of models into executable GPL code for various robotics target platforms. The integration mechanisms for modeling languages are mirrored by corresponding integration mechanisms of code generators for these languages. They allow to exploit the benefits of generative software development and to avoid monolithic and hardly reusable code generator aggregates.

Challenges are the coordination of multiple code generators each responsible for specific models or parts of models. This includes the selection of code generators supporting a common target platform, to handle language restrictions a code generator might impose, and to propagate necessary information between generators, such that integrating code generators does not require their modification.

In this article we extend and summarize our previous work in the context of MontiArcAutomaton [12]–[20]. The main contribution of this article is a consolidated and concise description of the framework and its main concepts. In addition, it contains technical contributions and improvements over some of our previous work. Mainly, a running example for behavior language integration summarizing all steps of previous technical contributions, the configuration of MontiArcAutomaton with a new domain-specific embedded language, and an overview of case studies applying MontiArcAutomaton. We discuss differences to previous publications in Section 8.5.

First, we illustrate MontiArcAutomaton by example (Section 2) before we provide the technical background on MontiArcAutomaton (Section 3). We illustrate developer roles in a MontiArcAutomaton MDD process (Section 4). Then we explain the modeling language integration mechanisms which enable modeling component behavior with the most appropriate modeling languages (Section 5). Based on these, we describe our concepts of code generator composition (Section 6) and their realization in MontiArcAutomaton. Afterwards, we list applications of the MontiArcAutomaton process and framework in case studies (Section 7). Then we review related work (Section 8). Finally, we discuss the proposed process, language integration and generator composition (Section 9) and conclude this contribution (Section 10).

2 Problem Statement and Example

The development of non-trivial robotics applications involves a multitude of challenges from various domains. Model-driven development provides means for representing, analyzing, and evolving the context and solutions to problems of these domains in a domain-specific way by abstracting from implementation details. This abstraction further enables reuse of solutions in MDD. However, enjoying these benefits for the engineering of robotics applications poses various challenges.

First, the domains of concern for robotics applications vary from project to project and are often not completely known initially. This variation reflects in the requirement that an MDD solution needs to be configurable and extensible with domain-specific modeling languages.

Second, the robotics target platforms for code generation from models exhibit similar heterogeneity to problem domains. Different versions of operating systems, application programming interfaces (API), and target GPLs are employed and have to be supported by MDD solutions. To benefit from the raised level of abstraction in domain-specific modeling languages these need to be supported by more than one code generator according to their use. It is thus required to separate languages and code generators and to provide code generator composition mechanisms that match the ones of the language integration.

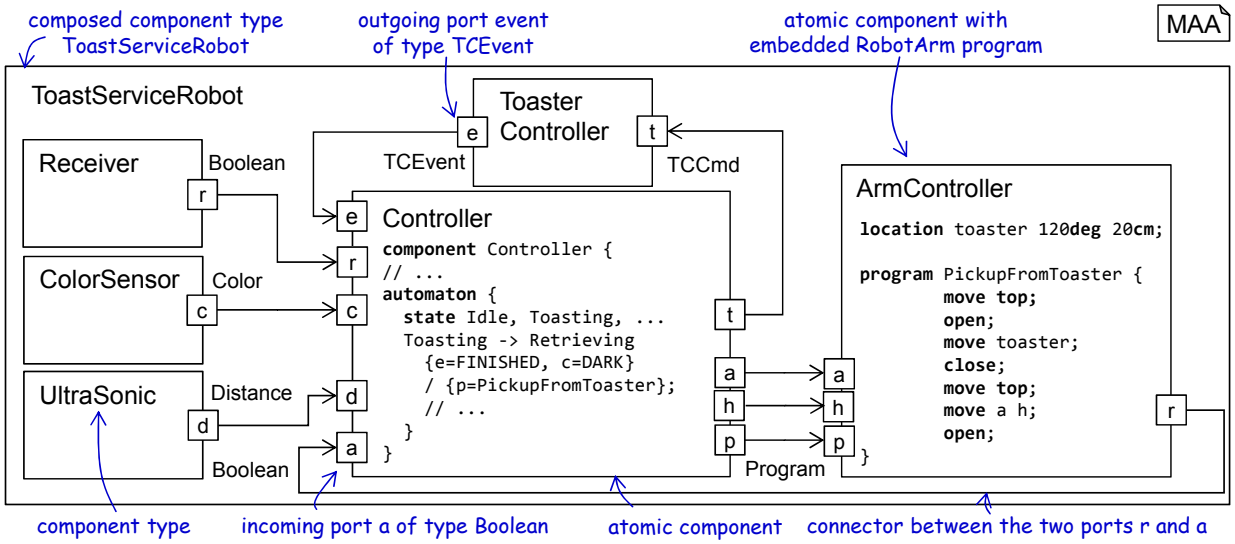

Fig. 1: Architecture of the `ToastServiceRobot` with embedded RobotArm programs.

In the remainder of this paper, we address these two challenges of domain-specific language composition and code generator composition. The context of our work are C&C architectures (ADL) [21], where software architectures are modeled as hierarchically composed components. Specifically, the MontiArcAutomaton ADL is a C&C ADL that enables embedding of problem-specific component behavior modeling languages into components. While the ability to use specific behavior modeling languages allows to develop specific applications, the encapsulation of models in components with well-defined and stable interfaces allows to modify component internals easily, e.g., to modify a specific behavior model, while retaining a stable architecture [22].

Throughout the article we use the following terms in their (sometimes specific) MontiArcAutomaton interpretation:

- *Application*: A model of the C&C architecture and behavior of the robot including configurations of languages and selection of code generators for a platform.
- *Platform*: The software and hardware the generated code of an application runs on.
- *Modeling language*: Description of syntax and well-formedness rules to express models including symbols for referencing and translating between models of languages.
- *Model*: Abstracted description of an aspect of a software system conforming to a modeling language.
- *Structure*: Structure of the C&C architecture of an application in terms of logical components, connectors, and composition.
- *Behavior*: Definition of interaction via connectors between components, e.g., to control actuators of the robot.

2.1 Example

A software engineer is responsible for the development of the software for a robotic arm. The robot assists a physically disabled person in a kitchen environment to operate a toaster, i.e., place bread in a toaster, operate the toaster, and deliver the toast to a nearby plate. Figure 2 shows part of the system as implemented with Lego NXT robots and its environment.[1]

Assuming that the software engineer has no expertise in programming manipulators, she models the logical architecture of the robot as a composition of components, e.g., the component `ColorSensor` to check the toast color and the component `ArmController` to prescribe movements of the robotic arm. She decides that the behavior of the main component `Controller` is best modeled as an automaton and the behavior of `ArmController` is best described using a domain specific language for robotic arms from an earlier project. The engineer configures MontiArcAutomaton to embed an automaton language following the I/O$^{\omega}$ automata paradigm [23], [24] and a language for RobotArm (RA) programs. The latter describes motion of the arm in terms of defined locations and gripper commands.

The model of the software architecture with two embedded behavior languages is depicted in Figure 1. The component `Controller` receives distances and toast colors from attached sensors. The automaton modeling the behavior of `Controller` translates these inputs into commands for the `ToasterController`, which starts and stops the toaster, and commands for the component `ArmController`, which actuates the robotic arm to pick up and deliver toast. The behavior of component `ArmController` is modeled as a set of RA programs.

1. A video of the system is available from http://www.monticore.de/robotics

Fig. 2: Lego NXT hardware and environment of the `Toast-ServiceRobot` application.

To generate executable code from the model, the software engineer needs to select code generators for the embedded automata and for the embedded RA programs that translate both into code for the NXT platform. She reuses existing generators by composing them according to the used language composition and the Lego NXT platform.

After evaluation of the Lego NXT prototype the project of the engineer is promoted to be turned into a commercial product. The robotic application will employ more reliable hardware and the Robot Operating System (ROS). The engineer configures her application to use code generators for the new target platform ROS. Code generators for the MontiArcAutomaton ADL and embedded automata are already available, however, a code generator for the RA language is missing. The missing code generator is subsequently developed by a team of developers with expertise in translating the high-level descriptions of robot arm movements into code for the ROS platform.

This example shows how language composition mechanisms of MontiArcAutomaton enable the use of the most suitable modeling language to describe component behavior and how the decoupling of code generators from languages as well as code generator composition enable efficient transitions between target platforms.

3 MontiArcAutomaton and MontiCore

MontiArcAutomaton is a framework for the pervasive model-driven engineering of robotics applications as component & connector software architectures with exchangeable component behavior modeling languages. At the core of MontiArcAutomaton is the extensible MontiArcAutomaton ADL [25], which is built on the formal concepts of the FOCUS [26] framework and supports black-box embedding of domain-specific component behavior languages with little effort. The language integration features of MontiArcAutomaton rely on the language workbench MontiCore. The framework facilitates translation of such language aggregates into GPL artifacts with a compositional code generator framework.

3.1 The MontiArcAutomaton ADL

The concept of encapsulation from C&C ADLs [21], [27] allows not only a logically distributed development and a physically distributed computation model, but also the composition of component behaviors independent of their internal behavior description. The MontiArcAutomaton ADL exploits the C&C encapsulation mechanism and allows the embedding of modeling languages into components for providing the most suitable behavior modeling language per component. This enables developing structure and behavior of the architecture in integrated models, reduces traceability efforts, and eases model co-evolution.

With the MontiArcAutomaton ADL, application modelers describe the architecture of robotics applications as the hierarchical composition of components and connectors. Components encapsulate functionality and provide it via well-defined, stable interfaces comprising sets of typed directed ports. Components are either atomic or composed: atomic components describe their input-output behavior either via embedded behavior models or via a handcrafted behavior implementation in a GPL. The behavior of composed components emerges from their subcomponents and their interaction.

Components interact by sending and receiving messages via directed connectors connected to their input and output ports. The types of the interface's ports determine the possible messages a component may receive or send and thus determine its potential connections to other components. Types of ports are defined in UML/P class diagrams (CD) [28], which are integrated with the MontiArcAutomaton ADL to ensure compatibility of port and variable types.

At its foundation MontiArcAutomaton ADL imposes stable component interfaces identified necessary for the separation of concerns into component (behavior) implementation and system integration [2], [22], [29]. Similar component behavior language integration is supported by the Architecture Analysis & Design Language (AADL) [30], [31] and xADL [32]. Both consider syntactic integration only, without taking inter-language well-formedness or code generation into account. MontiArcAutomaton takes care of both.

3.2 The MontiCore Language Workbench

The MontiArcAutomaton ADL is implemented with the MontiCore language workbench [33]. Its concrete and abstract syntax are defined in an extended context-free grammar (CFG) format [25]. From these grammars, MontiCore generates infrastructure to parse models of the language into abstract syntax trees (ASTs). MontiCore languages are textual, which allows software developers to retain the tools used to process

and manage GPL artifacts, while graphical representations of the models can be developed with some additional effort. An integration with the Eclipse Modeling Framework allows also the development of graphical editors for editing MontiArcAutomaton models.[2]

Checks of the well-formedness (static semantics) of models, called *context conditions*, are implemented in Java [34] and enable to perform static analyses not expressible with CFGs (such as whether the message types of connected ports are compatible). MontiCore distinguishes intra-language well-formedness rules and inter-language well-formedness rules [35]. The former characterize properties of well-formed models of a single modeling language and the latter consider integrated modeling languages.

MontiCore provides comprehensive language composition mechanisms [14] supported by its *symbol table* framework [34]:

1) Language *embedding* combines languages to use (parts of) multiple languages in a single model at well-defined extension points, e.g., component behavior languages are embedded into components.
2) Language *aggregation* combines multiple independent modeling languages into a *language family*, which enables to interpret their models together. For instance, within MontiArcAutomaton, its ADL and class diagrams are part of the framework's language family.
3) Language *extension* enables to reuse the grammar of the parent language and to add and overwrite its productions, e.g., the MontiArcAutomaton ADL extends the ADL MontiArc [36].

Details on all of these mechanisms are available [14], [17].

Symbols are data structures describing the essence of model parts free from the technical necessities of theirs ASTs. As such, every MontiCore language amenable for integration does provide the technical infrastructure to create symbols from its models' ASTs and to resolve names (e.g., of components) to corresponding symbols. The resulting symbols are decoupled from changes to the underlying AST and hence provide a stable model interface, which facilitates integration via adaptation. Ultimately, this supports checking context conditions across embedded and imported models.

MontiCore also facilitates development of code generators using the FreeMarker[3] template engine to process ASTs and code templates written in a target language [12], [35].

3.3 The MontiArcAutomaton Framework

The MontiArcAutomaton framework employs the MontiCore language integration mechanisms to enable embedding of component behavior modeling languages. It further comprises modeling languages to describe applications and compositional code generators. Application configuration models are specific to a software architecture model and describe how it is processed. Thus, for a single software architecture, application configurations can select multiple sets of code generators to translate it into artifacts of different GPLs. This, for example, allows to select a first group of code generators to translate the architecture to Java and a second group of code generators to translate the architecture to ROS nodes implemented in Python. Changing the code generators to employ amounts to changing this selection. Code generator models describe execution requirements of the represented code generator and properties of the generated code relevant to composition.

Figure 3 illustrates the MontiArcAutomaton framework and how its parts interact with each other. The top illustrates the framework with its modeling languages, modules, and extension points (left) as well as the artifacts it produces with their dependencies (right). The bottom part shows how the framework is configured and used: first the behavior modeling languages are integrated (left), then an architecture using these languages is modeled (middle), and the generators to apply are selected (right). The framework contains the MontiArcAutomaton ADL, which interfaces the UML/P CD language for data types. The ADL has an extension point for component behavior languages. The framework furthermore comprises language integration components for the integration of syntax and well-formedness rules. Resulting language aggregates are the basis for model-to-model (M2M) and model-to-text (M2T) transformations ultimately translating the application into multiple GPL implementations. The M2M transformations replace platform-independent components with platform-specific components according of the application configuration (cf. [20]). The configuration also defines the generators to apply in the subsequent M2T transformations. Code generator developers provide generator models to MontiArcAutomaton, which are processed for generator composition and execution.

Code generator composition ensures that artifacts generated by a set of compatible code generators conform to a run-time environment (RTE). A RTE is a set of target GPL artifacts that provide common functionality, such as base classes for components or interfaces for handcrafted behavior implementations. Conformance to a RTE means agreement on the interfaces employed for components and component behavior implementation. Generated component structure artifacts, the RTE, and the handcrafted component behavior artifacts can be specific to desired platforms.

4 Developing Robotics Applications with MontiArcAutomaton

As software systems under development become more complex it is important to follow a process that provides activities and roles responsible for enacting these. We now give

2. Video of an editor for synchronous graphical and textual editing of MontiArcAutomaton models: http://www.monticore.de/robotics/

3. Website of the FreeMarker Java template engine: http://freemarker.org/

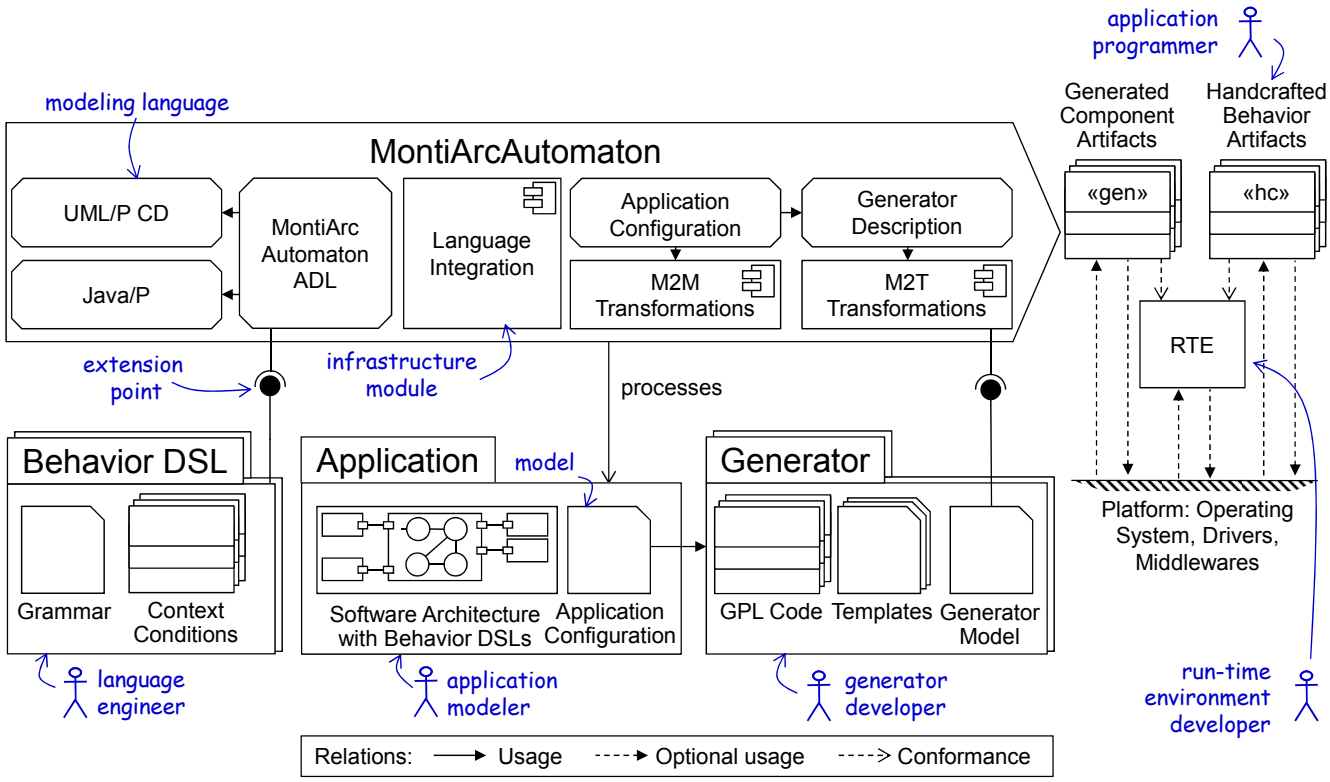

Fig. 3: The MontiArcAutomaton framework comprises several modeling languages to describe and process C&C software architectures and provides extension points for component behavior modeling languages and for code generators.

an overview over the roles participating in the MontiArcAutomaton development process shown in Figure 3, their responsibilities and activities.

In a typical project setup, the application modeler creates models for the structure and behavior of the robotics application as well as the application configuration model. The latter references generator models that describe the interfaces and requirements of the generators to apply. These are provided by the generator developer. The generated platform-specific code is executed by the RTE developed by a run-time developer. For fixed component behavior modeling languages and platform, the RTE and code generators typically remain stable. Thus, in a minimal setup, application modeler is the only role enacted. In case a component modeled by the application modeler requires manual implementation, e.g., to interface with application-specific legacy code, the application modeler models the component for manual implementation by the application programmer (see Section 4.1). For recurring tasks and behavior descriptions not conveniently expressible with existing languages of MontiArcAutomaton, a language engineer may extend the framework for with additional behavior DSLs. A generator developer then has to implement a code generator for the new component behavior modeling language to produce executable component implementations. Therefore, she provides a generator model and an implementation that generates code conforming to the designated run-time environments' properties. Due to the modularization of the framework and the division of responsibilities to roles, MontiArcAutomaton addresses the different skill sets of engineers. For example, a domain expert can enact the role application modeler and is not required to be an expert in interfacing with the low-level software and hardware of the platform. This expertise is provided by an application programmer whose work can be reused for multiple applications running on the same platform.

Using MontiArcAutomaton entails tailoring it to the particular component behavior language needs, before a software architecture can be modeled. Afterwards, composing code generators allows to produce GPL code from the architecture. The overall process follows the three stages depicted in Figure 4. The first stage focuses on customization of the modeling language and checks whether the MontiArcAutomaton language family has to be extended. In the second stage, the software architecture and the application configuration model, which compose the robotics application, need to be developed. In this stage, the application developer needs to decide whether a platform-independent or platform-specific model should be developed. Eventually, in the last stage, one or multiple code generators need to be selected or composed. This allows to select code generators for different target platforms, and, hence, to easily translate the same architecture to different GPLs.

Composition is only required if no monolithic code generator using the desired of ADL features, behavior languages, and platform exists.

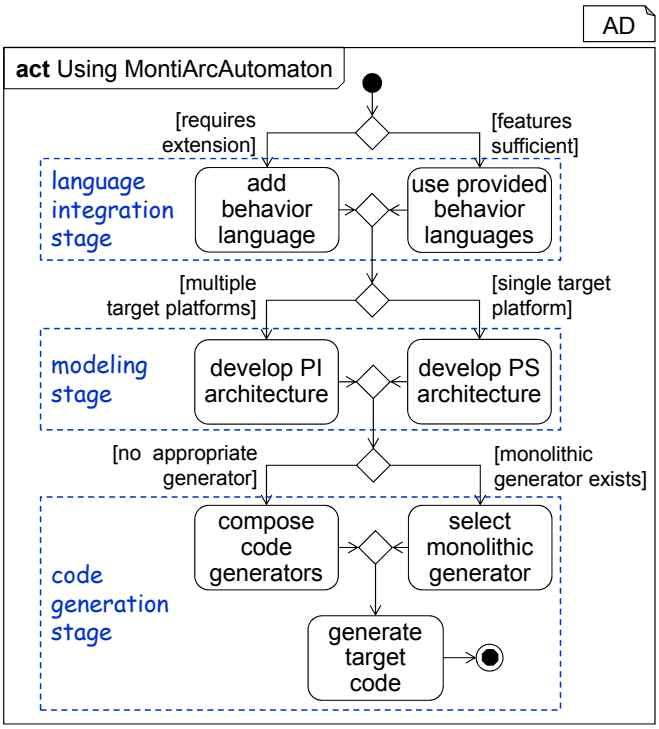

Fig. 4: Using MontiArcAutomaton entails three stages from behavior language integration to modeling to code generation (as introduced in [19]).

In this contribution, we present the language integration stage and the code generation stage. The modeling stage and the M2M transformation it employs are presented in [20].

4.1 Component Behavior Language Integration

Prior to modeling or code generation, the required component behavior modeling languages must be integrated. To this effect, the application modeler examines the MontiArcAutomaton ADL and identifies the requirements for new behavior languages. These requirements are passed to the language engineer, who provides these languages. The MontiArcAutomaton framework provides well-defined extension points for integration of MontiCore component behavior languages into components. Embedding languages into MontiArcAutomaton ADL components is declarative, i.e., it does not require changes to the existing languages' grammars [34]. Thus, the language engineer of the new behavior languages requires no expertise of the existing MontiArcAutomaton languages. It is important to note that the MontiArcAutomaton ADL is only extended with component behavior modeling languages. The concept of components, ports, and communication via connectors underlies the framework and serves for encapsulation.

Embedding a behavior modeling language requires registration of its grammar and little integration code. The latter includes context conditions of the integrated languages and symbol table infrastructure (cf. Section 3.2). With this infrastructure, the new component behavior modeling language can be integrated into the MontiArcAutomaton language family using the MontiCore language integration mechanisms. This enables to reference and reason about models from other languages of the family. Section 5 describes the details of language integration with MontiArcAutomaton.

4.2 Software Architecture Modeling

After component behavior language integration, the application modeler can develop the software architecture of the system under development with models of the integrated languages. To this end she decomposes the system into components and decides for each atomic component whether its behavior can be modeled or requires a handcrafted GPL implementation. For the latter, the application programmer develops proper implementations. Then she configures a M2M transformation to replace components according to the platform as presented in [20]. The resulting, platform-specific software architectures can be translated into GPL code then.

4.3 Code Generation

Translating a software architecture with embedded component behavior modeling languages requires code generators capable of translating the embedded behavior models. To this end, the code generation framework of MontiArcAutomaton enables composition of code generators for component structure, component behavior, and data types. However, MontiArcAutomaton also supports monolithic code generators supporting architectures models with a fixed set of embedded behavior languages. In both cases, the application modeler selects the participating generator(s) via the application configuration model, which may include invoking generator developers to produce proper generators. To be amenable for selection, the code generator developers must provide the code generators with generator models. These models describe the code generators' responsibilities, requirements, and interfaces. MontiArcAutomaton checks, composes, and executes the selected code generators based on these models. Development of such code generators is detailed in [19]. In this contribution, Section 6 presents our notion of code generator types and its implementation in MontiArcAutomaton.

5 Language Integration

Modeling language integration either requires to compose languages a priori, or to design them independently and to apply non-invasive composition mechanisms. Composing languages for a specific integration yields monolithic language families that are hardly reusable between projects with different language requirements.

RobotArm

```
1  robotarm ToastService {
2
3    input int angle;
4    input int height;
5
6    // name, angle, gripper height
7    location toaster 120deg 20cm;
8
9    program PickupAndDeploy {
10     move top;
11     open;
12     move toaster;
13     close;
14     move top;
15     move angle height;
16     open;
17   }
18
19 }
```

Listing 1: An excerpt from the concrete syntax of a stand-alone RA model `ToastService` with two inputs, a location, and a program.

MontiArcAutomaton embeds component behavior modeling languages into the MontiArcAutomaton ADL and forms a language family of all participating modeling languages, which allows to check inter-language context conditions between languages of the family. This section introduces language integration in MontiArcAutomaton.

5.1 Syntactic Behavior Language Embedding

Embedding is a purely syntactic language integration mechanism in which the host language's grammar provides an extension point in form of a designated *external* production for component behavior. To facilitate reuse, not the languages' grammars are integrated, but the parsers MontiCore generates from these. This allows use of the host language without embedding, facilitates exchange of embedded languages, and does not require modification of the participating languages' grammars.

The MontiArcAutomaton ADL grammar provides such an extension point that is conditionally mapped by the MontiArcAutomaton language configuration model to different component behavior language grammars. The generated parser for MontiArcAutomaton ADL models delegates parsing of external productions to the parser responsible for the embedded productions of the respective behavior language. We illustrate the embedding of behavior languages on the example of the RobotArm language RA.

The RA language describes the behavior of a robot arm in terms of sequential movements to physical positions and gripper actions. RA models consist of a name, inputs, constants, and programs contained in `robotarm` blocks. Inputs are data sources the programs may react on. Constants define robot-specific measurements (`top`, `bottom`) or locations in the joint space of the robot arm (keyword `location`). Programs are sequences of commands to move the arm (`move`) and to use the gripper (`open`, `close`). Movement is either to a robot-specific measurement (`top`, `bottom`), a defined location, or to a joint space position based on input. Listing 1 shows an excerpt of a stand-alone RA model comprising the two inputs `angle` and `height` (ll. 3-4), one location definition `toaster` (ll. 7), and the program `PickupAndDeploy` (ll. 9-17).

RobotArm models neither prescribe how they are executed, nor have knowledge about the robot arm they are used with. These technical details depend on the capabilities and APIs of the system the software is deployed on and are in the responsibility of the generator and generated code. In our implementation, the code generated for RA models tracks the current absolute positions of the joints and actuates the motors of the arm according to these.

MontiArcAutomaton enables the use of RA models in the context of larger robotic applications by embedding parts of the RobotArm language into component definitions. It does not embed the RA inputs as embedded RA models receive input from MontiArcAutomaton ADL ports instead. This allows to define component models including RA models in a single integrated artifact. Listing 2 shows the textual representation of the component `ArmController` as depicted in Figure 1. It begins with the keyword `component`, followed by the component's type name and a body (l. 1). The body contains an interface of typed ports (ll. 3-7) followed by an embedded RA model (ll. 9-21) beginning with its keyword `robotarm`. The models contains a location definition (l. 10) and a single program (ll. 12-20).

Please note, that the program `PickupAndDeploy` still receives input from sources named `angle` and `height` (l. 18). In the context of the stand-alone RA program `PickupAndDeploy`, the symbols `angle` and `height` represent RA program inputs, but in the context of the MontiArcAutomaton ADL component `ArmController`, these symbols represent ports. To enable proper integration, such as inter-language well-formedness checking, this change of meaning must be explicated. For instance, the MontiArcAutomaton framework must ensure that `angle` and `height` are of data types compatible to RA `move` commands. Thus, syntactic language embedding alone does not suffice, but requires *symbolic* integration as well.

5.2 Symbolic Language Integration

Names referencing other parts of models are considered symbols with a certain meaning. When integrating a component behavior modeling language into the MontiArcAutomaton ADL, the meaning of names may change. To reflect this, MontiArcAutomaton and its ADL rely on the language aggregation mechanisms of MontiCore, which require that the

MontiArcAutomaton

```
1  component ArmController {
2
3    port
4      in int angle,
5      in int height,
6      in ArmControllerProgram cmd,
7      out ArmControllerResult result;
8
9    robotarm {
10     location toaster 120deg 20cm;
11
12     program PickupAndDeploy {
13       move top;
14       open;
15       move toaster;
16       close;
17       move top;
18       move angle height;
19       open;
20     }
21   }
22
23   // ...
24 }
```

Listing 2: An excerpt from the concrete syntax of MontiArcAutomaton component `ArmController` with embedded RA model.

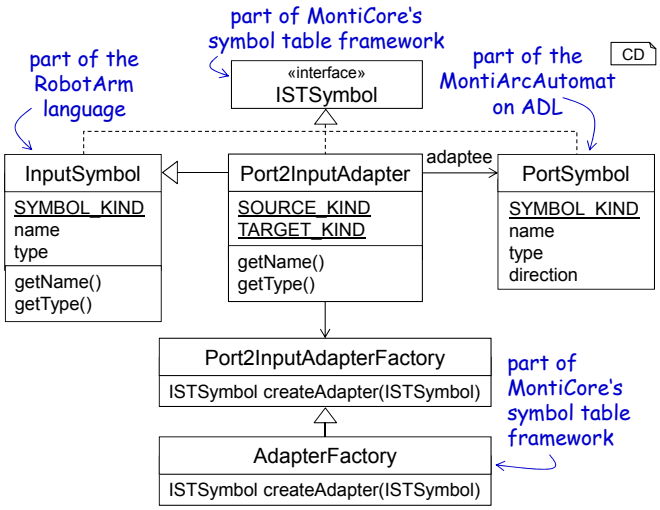

Fig. 5: Adaptation between a `InputSymbol` of the RA language and a `PortSymbol` of the MontiArcAutomaton ADL.

symbols of model parts are made explicit. This allows adaptation between symbols and, hence, their correct interpretation as well as well-formedness checking after integration. To this end, embedding component behavior modeling languages requires proper adaptation between the participating languages' symbols, as well as integration of new inter-language well-formedness rules that cannot be checked on the level of a single language only. For instance, integration of RA programs into the MontiArcAutomaton ADL requires to ensure that the names of inputs are interpreted as names of ports (adaptation) and that the referenced ports are incoming ports (an inter-language well-formedness rule not checkable within RA alone). Furthermore, the `input` elements of RA (cf. Listing 1, ll. 3-4) should be prohibited to avoid underspecification (another well-formedness rule specific to integration).

Integration of the RobotArm language with its symbol table infrastructure into the MontiArcAutomaton ADL language family allows joint interpretation as well as creation of symbols of the respective model parts and execution of well-formedness checks from both languages. However, the RA symbol table treats names after `move` as references to `input` elements. These are not part of embedded RobotArm models. Thus, to complete integration, the language engineer configuring MontiArcAutomaton with RA provides an adapter between the `input` symbol of RA and the `port` symbol of the MontiArcAutomaton ADL as depicted in Figure 5.

The adapter `Port2InputAdapter` extends from `InputSymbol`, overrides its methods `getName()` and `getType()`, and yields static class fields `SOURCE_KIND` and `TARGET_KIND`. The former equals the `SYMBOL_KIND` of `InputSymbol` and the latter the `SYMBOL_KIND` of `PortSymbol`. Hence, whenever the RA language family looks up an `InputSymbol`, the MontiArcAutomaton ADL language family might return a `Port2InputAdapter` that adapts a `PortSymbol`. To ensure these adapters are created properly, the `Port2InputAdapter` provides its own factory to the language family. This factory is used to produce `Port2InputAdapter` instances whenever symbols of its `TARGET_KIND` (i.e., port symbols) are produced by MontiArcAutomaton. The method of `Port2InputAdapterFactory` therefore receives a `PortSymbol` and - if the port symbol's `direction` is incoming - copies its properties to its own fields. In consequence, resolving the name `angle` in Listing 2 returns an instance of `Port2InputAdapter` that adapts the corresponding port symbol and can be used with inter-language checks of RA (such as validating that the type of `angle` actually is numeric). Please note that this also ensures that ports references by `move` commands are incoming ports, otherwise resolving the corresponding `InputSymbol` will fail as neither the `InputSymbol`, nor the `Port2InputAdapter` exist. Hence, a dedicated well-formedness rule for this is not necessary.

Adding inter-language well-formedness rules amounts to adding a single Java class to the language family that extends MontiCore's `ContextCondition` class and implements specific interfaces of the MontiArcAutomaton ADL or RobotArm depending on the language element to check. These interfaces prescribe `check()` methods that receive

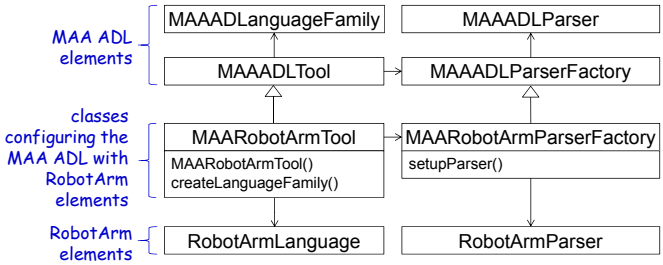

Fig. 6: The class `MAARobotArmTool` integrates the RobotArm language into the MontiArcAutomaton ADL to enable processing and checking of integrated models.

Groovy Integration DSEL

```
1 name      "RobotArm"
2 tool      new RobotArmTool()
3 language  new RobotArmLanguage()
4 behavior  "RobotArm.Content"
5 checks    new NoInputsContextCondition()
6 adapters  new Port2InputAdapter()
```

Listing 3: Tool configuration for embedding the `RobotArm` language.

the symbols to check, perform checks, and can report issues. Based on these interfaces, the MontiArcAutomaton ADL well-formedness checking executes such context conditions.

Adding, for instance, a rule prohibiting the existence of `input` elements for integrated RA models requires to implement and register a class extending from `ContextCondition` that implements the interface to check `input` elements. The implementation of the corresponding `check()` method reports violations MontiArcAutomaton ADL after finding an `input` element in the model.

5.3 Language Integration Infrastructure

The MontiArcAutomaton ADL language family can be configured with new component behavior modeling languages via handcrafting language processing tools or configuration files.

For a language processing tool, the language engineer provides two classes: one extends from `MAAADLTool`, which is the language processing tool of the MontiArcAutomaton ADL. The other extends from `MAAADLParserFactory`, which combines the parsers of participating languages. Figure 6 displays both classes for the integration of RA into the MontiArcAutomaton ADL. The `MAARobotArmTool` integrates the RobotArm language into the MontiArcAutomaton ADL language family, which includes adding adapters and registering context conditions. The `MAARobotArmParserFactory` registers the RA parser at the behavior language extension point of the MontiArcAutomaton ADL.

Using the `MAARobotArmTool` allows processing integrated models, correct interpretation of their inter-language references, and checking their well-formedness. However, configuration of the language family with adapters and context conditions would require some understanding of MontiCore and its symbol table framework. To liberate MontiArcAutomaton users from this, component behavior modeling languages may be added via an domain-specific embedded language (DSEL) implemented in the GPL Groovy [37]. Groovy is a language for the Java virtual machine that allows to omit syntactic sugar and hence lends itself to define embedded DSLs [38]. Models of this language serve as configuration and processing these models exploits the modeling language infrastructure of MontiCore to collect relevant integration properties. From these models, MontiArcAutomaton instantiates the `GroovyTool` and the `GroovyParserFactory` (both also descendants of `MAAADLTool` and `MAAADLParserFactory` depicted in Figure 6, respectively) and configures both with the information from the Groovy models.

Each model describes one behavior language to embed. Multiple models can be combined. For one, the integration framework expects that each language candidate provides a language processing tool that extends from MontiCore's class `ETSTool` (cf. [34]), which holds information of the language's well-formedness rules and symbol table infrastructure. Furthermore, it expects that each language candidate provides a language definition extending from MontiCore's class `Language` (also [34]), which contains additional configuration. Both requirements are standard for MontiCore languages, hence, they do not restrict embedding.

Listing 3 shows a configuration model describing five properties. First, following the keyword `name`, each model specifies a unique name for this embedding (l. 1). Afterwards, the tool class and the `Language` class to be used are defined (ll. 2-3). The property `behavior` describes which production of the `RobotArm` grammar should be embedded (l. 4). Afterwards, `checks` (l. 5) declares a list of well-formedness rules that are merged with the MontiArcAutomaton ADL rules and the RA rules retrieved from the language class. Finally, `adapters` lists adapters for inter-language adaptation (l. 6).

While the expressiveness of this DSEL is restricted to the most common use case, it typically allows to configure the MontiArcAutomaton ADL with new component behavior modeling languages with minimal effort. In case the language to be embedded has special requirements, handcrafting the integration allows to harness the full power of MontiCore. After integration, parsing and analyzing components with embedded behavior models is possible. Providing code generation capabilities for integrated languages is part of the code generator composition framework presented in the next section.

6 Code Generator Composition and Execution

A MontiArcAutomaton robotics application is a C&C model with embedded behavior models of multiple languages. Enabling code generation for such aggregates requires code generators capable of translating composed models into target GPL code. Usually, code generators are monolithic and capable of translating a fixed combination of languages. As MontiArcAutomaton supports problem-specific, flexible, language embedding, code generators must mirror this flexibility. MontiArcAutomaton provides a code generator composition framework that supports integration of component structure generators with component behavior generators as required by the language combination. Hence, code generator composition in MontiArcAutomaton amounts to

1) identifying platform requirements (including target GPL),
2) selecting proper generators for the languages to be processed and the intended platform,
3) configuring MontiArcAutomaton with the selected generators,
4) invoking a descendant of `MAAADLTool` (depending on the chosen language integration mechanism), which parses the ASTs of components with embedded behavior models, checks their well-formedness, and ultimately composes the selected code generators to produce platform-specific GPL artifacts.

To support this process, MontiArcAutomaton comprises not only modeling languages for C&C structures and component behavior, but also for code generator description, and application configuration (including generator selection).

The code generator composition mechanism of MontiArcAutomaton exploit the C&C nature of its ADL and relies on *generator types* explicating the different aspects of this nature. The types govern the information required by conforming participating generators and are realized by *code generator interfaces*. These interfaces describe the properties MontiArcAutomaton code generators require from and contribute to composition.

In the following, Section 6.1 introduces code generator types. Afterwards, Section 6.2 presents the code generator types employed with MontiArcAutomaton and their usage during code generator development. Finally, Section 6.3 explains how to select code generators for a software architecture and their composition by MontiArcAutomaton.

6.1 Generator Properties and Types

Code generation in MontiArcAutomaton amounts to code generation for the structure of components, the different embedded behavior languages, and data types. This separation leads to three types of generators: *component generators*, *behavior generators*, and *data type generators*. Component generators translate C&C modeling elements, while behavior generators

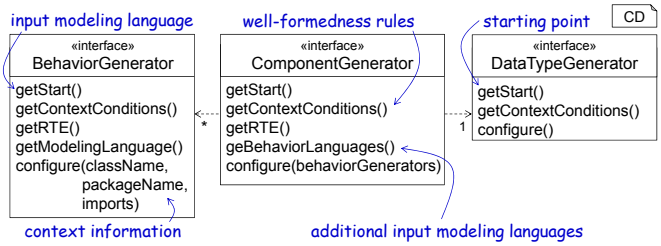

Fig. 7: Java interfaces for the different code generator types.

translate component behavior models free from C&C details. Hence, the behavior generator developers can be liberated from C&C expertise. Each type governs the properties conforming code generators must exhibit, which is a subset of the following:

1) Modeling languages: One or more modeling languages the code generator can translate.
2) Generation well-formedness rules: These constraints restrict the set of valid input models for the generator. For example, a code generator for embedded automata might reject models with nondeterminism.
3) Output representation: A description of produced output to ensure compatible GPL artifacts. For example, a component generator might generate Java artifacts while a behavior it should be composed with produces Python artifacts only. This may lead to incompatibilities.
4) Execution information: Information on how the code generator can be invoked.
5) Generation context information: Information the generator requires at run-time. This can be information about the model or about the generation process of participating generators. Rather than relying on implicit assumptions about the generation process and generated artifacts, we make these assumptions explicit with the generation context information.

The three generator types of MontiArcAutomaton restrict these properties differently: While all generator types prescribe the declaration of execution information and allow to declare well-formedness rules, only behavior generators and component generators may declare output properties. Behavior generators also may explicate the supported modeling language and component generators may declare a set of additionally supported behavior modeling languages.

Component generators must process MontiArcAutomaton ADL components, but may process behavior languages as well. They may add well-formedness rules and exhibit properties regarding their output. Data type generators must translate UML/P classdiagram models. Thus, specifying additional supported modeling language is prohibited. All code generators in MontiArcAutomaton conform to exactly one code generator type. Thus, each participating generators provides the information characterized by its type and is processed accordingly.

6.2 Code Generator Types of MontiArcAutomaton

Each MontiArcAutomaton code generator must conform to a single generator type. This conformance is expressed by implementing an interface describing the properties of each type. For MontiArcAutomaton, which is implemented in Java, the code generator interfaces are as depicted in Figure 7. These interfaces realize the information that generators of the individual types may provide as presented in Section 6.1 via the provided methods. For instance, a method `getModelingLanguage()` returns the MontiCore `Language` class of the modeling language that generators of this type can process. Furthermore, they describe the required generation context information as parameters of the `configure()` method. Please note that the interface for component generators expects conforming generators to receive and execute behavior generators. In MontiArcAutomaton component generators process the complete AST of composed models and delegate parts to behavior generators. With these interfaces, composition may treat participating code generators as black-boxes and does not rely on code generator internals as proposed in [35].

Implementations of the generator interfaces can be handcrafted by generator developers or defined in terms of *generator models*. In the first case, the generator developer provides a Java class implementing the interface of the generator type under development following a naming convention. In the second case, she provides a generator model that conforms to the generator description modeling language depicted in the overview in Figure 3 and employs MontiArcAutomaton to produce an implementation from it.

These models contain keywords that correspond to properties of generator interfaces and provide corresponding values. Listing 4 displays a generator model for the `RobotArmPython` generator. The models begins with the keyword `generator`, followed by its name, the keyword `conforms` and the name of its generator type's interface (ll. 1-2). The model for `RobotArmPython`, for instance, declares that is is a behavior generator (l. 2). The model's body contains keyword-value pairs to describe it starting point (executing information, l. 4), its modeling language (l. 5), its run-time environment (output representation, l. 6), and its context conditions (well-formedness rules, ll. 8-10). This generator can be started using its FreeMarker template `robotarmpy.Main` and is capable to translate models of the language `robotarmpy.RobotArm.Content`. Furthermore, it produces artifacts compatible to the run-time environment `runtimes.pythontimesync` and declares a single well-formedness rule in form of the context condition `robotarmpy.NoGenericDataTypes` (which prohibits to use it with RA models that rely on generic data types for their inputs). The name `robotarmpy.RobotArm.Content` identifies the production `Content` of MontiCore grammar `RobotArm` residing in package `robotarmpy`. This granularity supports code generation for language aggregates where only parts of behavior modeling languages are embedded.

GeneratorConfiguration

```
1  generator RobotArmPython
2    conforms generators.BehaviorGenerator {
3
4    start robotarmpy.Main;
5    language robotarmpy.RobotArm.Content;
6    rte runtimes.pythontimesync;
7
8    contextconditions {
9      robotarmpy.NoGenericDataTypes
10   }
11 }
```

Listing 4: The generator configuration for the RobotArm generator describes that it implements the interface `BehaviorGenerator` and provides further information.

MontiArcAutomaton supports automated translation of such models into Java classes implementing the declared generator type interfaces. These classes implement all methods of the generator type's interface and return the generator information from the model where applicable (e.g., `getContextConditions()` returns a set of context condition instances as declared in the `contextcondition` block). MontiArcAutomaton requires such implementations for all participating generators and composes these according to their types.

6.3 Composing and Executing Code Generators

Translating a C&C architecture model into GPL artifacts requires a set of code generators that contains one component generator, one behavior generator for each embedded modeling language not supported by the component generator, and a data type generator. If all generators produce artifacts conforming to the same GPL and the component generator and the behavior generators produce artifacts of the same RTE, we denote such a collection as a *generator family*.

The application modeler defines which generators to apply in order to translate a software architecture. Adding this information to the component models would tie models to code generators. Instead, such is defined in *application configuration models*. These models conform to the application configuration language depicted in Figure 3 and reference a single software architecture as well as multiple code generator families to apply as shown in Listing 5.

Each model begins with the keyword `application`, followed by a name, the keyword `for`, and the name of the software architecture it references (l. 1). Afterwards, a body delimited by curly brackets follows, which contains `platform` blocks (ll. 2-6 and ll. 7-11). Each block begins with the keyword `platform`, followed by a name, and a body. This body contains a set of code generator names that references the code generators of a generator family. Specifying the individual roles of the participating generators is unnecessary

ApplicationConfiguration

```
application TR for ToastServiceRobot {
  platform rospython {
    componentgenerators.ComponentsTSPython;
    behaviorgenerators.RobotArmPython;
    behaviorgenerators.IOAutomatonPython;
  }
  platform nxtjava {
    nxtjava.Components;
    nxtjava.RobotArmJava;
    nxtjava.AutomataJava;
  }
}
```

Listing 5: Application configuration model for the toaster robot application using the `RobotArmPython` and the `RobotArmJava` generators for components with embedded RA models.

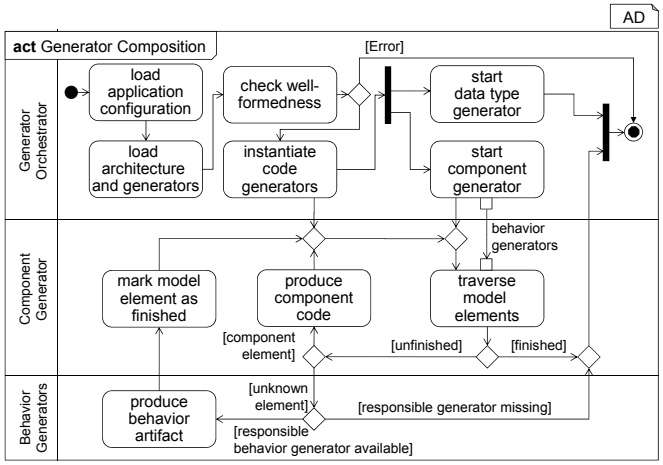

Fig. 8: Overview of code generator composition with a generator orchestrator and generator interfaces.

as MontiArcAutomaton resolves the corresponding generator models.

Given a MontiArcAutomaton ADL software architecture with embedded behavior modeling languages, families of code generators for specific platforms, and an application configuration selecting these families, the process of translating the architectures is as depicted in Figure 8. First, the generator orchestrator of MontiArcAutomaton loads the application configuration model. Afterwards, its loads the referenced software architecture model and the generator models. Based on these, it performs intra-language well-formedness checks and inter-language well-formedness checks to validate the models. In case of successful validation, it instantiates the participating code generators via their interfaces and starts code generation. The generation of data types is decoupled from that of components with embedded behavior and can be performed in any order. The generator orchestrator starts the component generator by passing the participating behavior generators to it. The component generator starts processing the component model elements. For each component element visited, the generator produces code. For each unknown element, it tries to identify the responsible behavior generator (via its modeling language property). In case a generator is found, the component generator passes generation context information as required by its `configure()` method to it and executes it. Per agreement on a RTE, the behavior generator produces compatible GPL artifacts. After it finished, control is passed back to the component generator, the processed element is marked finished, and AST traversal continues. The code generation process finishes when all AST elements have been traversed.

For instance, translating MontiArcAutomaton architecture models with embedded RobotArm programs component behavior modeling languages to ROS [39] in its C++ implementation requires three code generators:

- A component generator that translates components, ports, and connectors to C++ ROS nodes. This code generator must provide a generator model identifying it as a component generator and explicate the properties required by generators of this type. The generator is independent of any behavior modeling language and hence can be reused for arbitrary behavior language combinations to be used with ROS.
- A behavior generator that translates RobotArm programs into C++ motor commands and provides a proper generator model. Translation of RA primitives into arm commands maybe independent of ROS or ROS-specific. In the former case, the generator can be reused for any translation of RobotArm programs to C++; otherwise, this is specific to ROS.
- A data type generator to translate UML/P port class diagram data types to ROS messages.

A component generator for ROS must flatten the hierarchy of the architecture prior to code generation as ROS does not support hierarchical nodes. Afterwards, the resulting architecture can be translated into a graph of nodes connected by topics. The latter correspond to the connectors between ports of component models. Topics resemble typed message buses that support n:m communication, whereas connectors express 1:n communication in the underlying FOCUS [26] semantics. Thus, the component generator should ensure that each generated topic has a single publisher only. Otherwise, the semantics of model and generated code may diverge. Furthermore, ROS messages do not support generic data types yet. Consequently, translation from UML/P class diagrams to ROS messages must take care of eliminating generic data types. This can be achieved by calculating the values of the generic parameters for each usage and produce specific data types from the result.

Given these three code generators, specifying their usage

for a MontiArcAutomaton architecture model is as simple as presented in Listing 5 and MontiArcAutomaton takes care of proper composition and code generation as depicted in Figure 8. Hence, with this framework, migration of the system under development to new platform versions amounts to providing compatible generators only. The models can remain unchanged.

7 Platforms and Case Studies

To evaluate MontiArcAutomaton on different platforms, we have developed four code generators to different target languages [12]. Some applications require additional component models to access platform-specific software and hardware. MontiArcAutomaton organizes these models and their platform-specific implementations

- for Robotino robots using ROS [39] with Python,
- for Robotino robots using SmartSoft [2] with Java,
- for Lego NXT robots using leJOS[4],
- for arbitrary ROS-compatible robots using rosjava[5], and
- for simulators including ROS turtlebot and Simbad[6].

Besides these robot-specific collections of models, code generators and RTE, we have also developed GPL-specific models and RTE that provide GPL functionalities to MontiArcAutomaton (e.g., file I/O). The collections contain between 4 (ROS turtlesim) and 21 (leJOS NXT) components and can be easily imported by MontiArcAutomaton applications to deploy these to different platforms.

We have evaluated MontiArcAutomaton in professional and educational settings with different platforms to test our concepts, modeling languages, and code generators.

7.1 Case Study NXT

With MontiArcAutomaton, a behavior language based on automata, a Java code generator family, and a Java RTE, we evaluated the MontiArcAutomaton framework during a university lab course. Evaluation took place in the winter term 2012/13 with eight master level students [40]. The students had previous experience with Java from courses throughout their studies. The students developed a distributed robotic coffee service consisting of the three robots illustrated in Figure 9. The run-time environment and system component implementations were given. During the lab course the students participated in a survey and interviews regarding the effort and benefits of applying MontiArcAutomaton to a distributed robotics application. The results are discussed in [40].

4. leJOS website: http://lejos.sourceforge.net/
5. rosjava website: https://code.google.com/p/rosjava/
6. Simbad website: http://simbad.sourceforge.net/

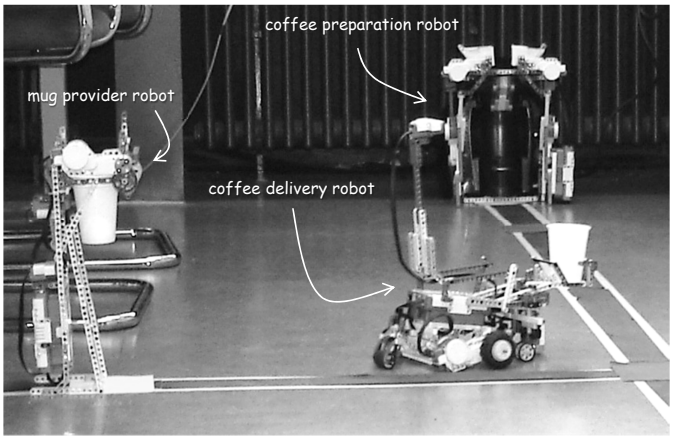

Fig. 9: The distributed robotic coffee service as implemented with Lego NXT robots running leJOS as platform.

7.2 Case Study ROS

We also evaluated MontiArcAutomaton with another automaton language, the ROS-Python code generator family, and the ROS-Python Robotino modules in another lab course in winter term 2013/14. The group of nine master level students acted as application modelers and application programmers to develop a logistics application using the Robotino platform (Figure 11). The students had little to no previous experience with ROS [39] or Python, which allowed for a better evaluation than in the previous lab course. The software architecture of this application consists of 31 components, of which 9 contain automata and 17 have platform specific implementations. Figure 10 shows the top-level architecture with the components `Navigation`, `MapProvider`, and `TaskManager` being composed from other components. During this course, the students acted as application modelers and applications programmers (as, for instance, the behavior of the ROS-specific components wrapping the user interaction had to be implemented manually). In this application, the robot receives tasks such as “deliver item X to room Y” from a web site. Automata embedded in components translate these into motion and interaction commands. These commands are passed to the components `Navigation` and `UserInterface`, which translate these into platform-specific GPL primitives.

To interface ROS, the students have developed components with Python behavior implementations that acted as publishers or subscribers. Components sending message to ROS receive these messages via ports, wrap these into ROS messages, and send these to the connected nodes. Components reading messages from ROS yield a behavior implementation that subscribes to topics and emits the received, unwrapped, messages via their ports. Buffering and translation of ROS messages is subject to the individual components and was optimized depending on the interfaced topics.

The students participated in two surveys on the efforts

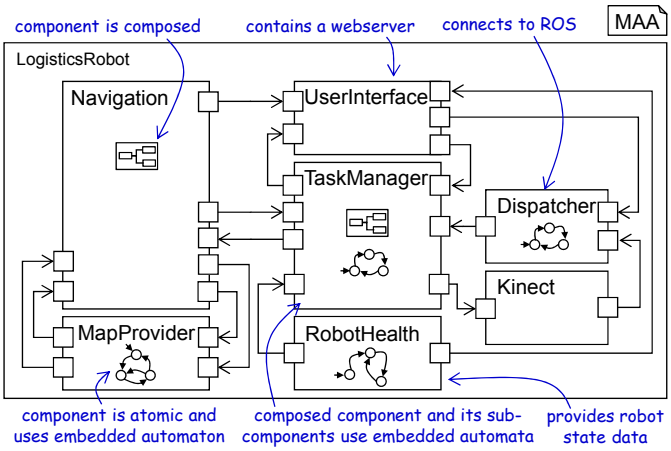

Fig. 10: The top-level architecture of the logistics application implemented during winter term 2013/14 for a Robotino running ROS and Python as platform.

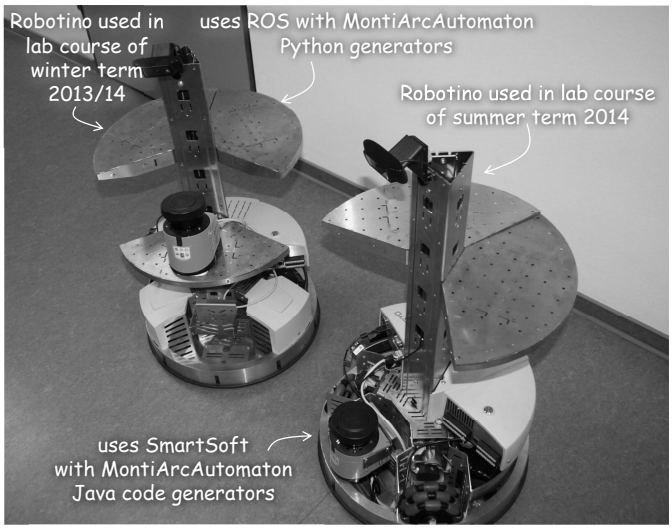

Fig. 11: The Robotino platforms used in the lab courses.

and benefits of modeling with MontiArcAutomaton. The first survey of 12 questions was conducted after a few weeks, the second survey of 18 questions at the end of the course. In the beginning, the students did require less effort understanding the MontiArcAutomaton and the I/O$^{\omega}$ automata language than understanding ROS. The students were more confident in the Python artifacts produced by themselves (6.8 out of 10) and their team mates (6.7 out of 10), than in the behavior models they had developed (6.5 out of 10). Nonetheless, the students spend ten times as much time developing conceptually wrong artifacts with Python (34% of their time) than with the behavior modeling language (3% of their time). We consider this an indicator for the benefits of modeling abstractness over accidental complexities of a GPL.

In the second survey, the students estimated they spent 44% of their time learning ROS, which had doubled from the first survey. The times actually spend learning the MontiArcAutomaton ADL and the I/O$^{\omega}$ automata languages were considered reduced by ca. 33%. Interestingly, the time spend doing conceptually wrong things with the MontiArcAutomaton ADL, I/O$^{\omega}$ automata, and Python converged during the lab course. In the second survey, the students have reduced conceptually wrong Python programming to 8% of their time, while conceptually wrong behavior modeling increased to 9%. In the same time, the estimated efforts to fix bugs in Python artifacts and ROS nodes doubled, while the estimated time to fix bugs in MontiArcAutomaton ADL components and I/O$^{\omega}$ automata remained constantly low. This also points to the benefits accredited to modeling (such as their abstraction being beneficial to understanding and maintenance).

Due to the task's complexity and the small group size, a separation of a control group was not feasible. This raises threats to both the evaluation's internal and external validity. First, there is a selection bias as the students were well-trained in developing object-oriented systems. Also, instrumenting questionnaires suffers from biased answers due to the students' self-perception as well as from biases regarding avoidance of the provided scales upper and lower ends. The evaluation's external validity is threatened as the course was held under the topic of model-driven development and the students received grades for their participation. Hence, there may have been an implicit bias to assume MDD is beneficial.

7.3 Case Study SmartSoft

In a lab course of summer term 2014 we again assigned the task to develop a robotics logistics application. In this course, we used Robotino robots with the SmartSoft [2] middleware to control the robot. The students modeled the application logic with MontiArcAutomaton and an I/O$^{\omega}$ automata behavior modeling language. Handcrafted behavior was implemented in Java. The students acted as application modelers and applications programmers again.

The produced architecture is depicted in Figure 12. Of the depicted subcomponents, five are composed components and two are atomic. The students of this lab course answered similar questionnaires (again, one in the beginning and one in the end) as the group of 2013/14. In the beginning, the students considered learning the middleware (estimated 40% of their time) more complex than learning the modeling languages (15% for MontiArcAutomaton ADL and 13% for I/O$^{\omega}$ automata). The students spend much more time doing conceptually wrong things with SmartSoft (35% of their time) and Java (26%) than with MontiArcAutomaton ADL and I/O$^{\omega}$ automata (13% and 9%). They also estimated to have recreated or revised the Java implementations (8 times) and SmartSoft models (5 times) more often than structural MontiArcAutomaton ADL elements (3 times) and I/O$^{\omega}$ automata (1 times).

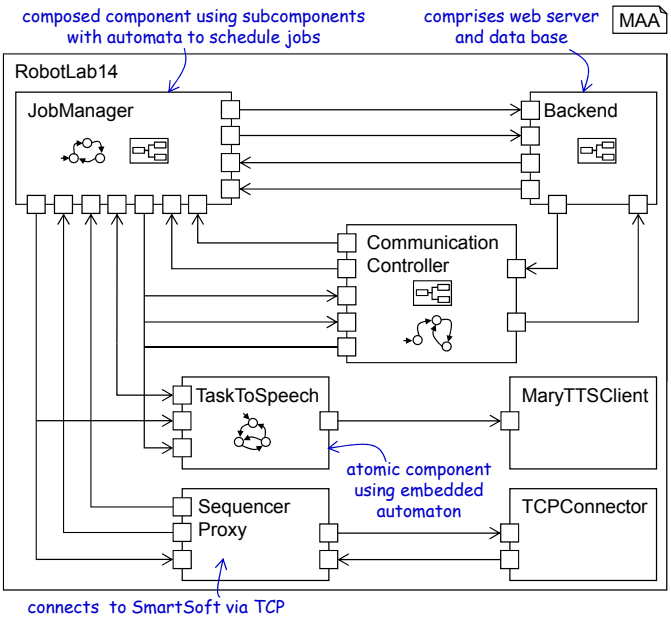

Fig. 12: Top-level architecture of the logistics application implemented in summer term 2014 for a Robotino running SmartSoft and Java. Five of the displayed components types are composed.

In the the second survey, the time estimated learning the modeling languages reduced to 5% each, while the time to understand SmartSoft increased to 47%. Interestingly, the time spend doing conceptually wrong things with SmartSoft (17% of their time) and Java (7%) and MontiArcAutomaton ADL (8%) decreased significantly, while the time spend to conceptually wrong I/O$^{\omega}$ automata modeling remained stable (8%). In the same time, the students' confidence in I/O$^{\omega}$ automata models decreased (from 8.3 to 5.2 out of 10) and the confidence in SmartSoft models increased (from 4.6 to 7.4 out of 10). This contradicts the previous lab course, where the students' confidence in models remained stable over time.

Again, the group was too small to separate into control group and experimental group. Furthermore, the students were not only well-trained in object-oriented programming, but in Java programming. Hence, trying to solve tasks in Java instead of modeling techniques provided a strong bias. Besides the students experience in Java, the same threats to external and internal validity as above hold for this case study.

7.4 iserveU Service Robotics Project

We are applying and evaluating MontiArcAutomaton in the ongoing 3-year iserveU[7] project with industrial partners from automotive and robotics. The project is funded by the German Federal Ministry for Education and Research to investigate the pervasive model-driven engineering of robotics applications. In this project, the MontiArcAutomaton ADL is used to model the architecture and parts of the behavior for a hospital logistics service [41]. The MontiArcAutomaton ADL serves as ADL for a high-level controller that interfaces SmartSoft [2]. To this end, the architecture model describes the structure of that controller and most components are modeled using a state-based component behavior language.

7. See http://www.pt-it.pt-dlr.de/de/3046.php

8 Related Work

We presented the MontiArcAutomaton architecture modeling framework with its MontiArcAutomaton ADL and extensions for component behavior language embedding and code generator composition. MontiArcAutomaton aims to increase abstraction and reuse in engineering robotics applications. However, there are other approaches to robotics software development that employ models as primary development artifacts as well. Section 8.1 discusses these. Developing software as C&C architectures is not specific to robotics and there is a multitude of related C&C modeling languages as well. Thus, Section 8.2 presents and discusses these. As the language integration features of MontiArcAutomaton rely on the language workbench MontiCore, Section 8.3 examines related language workbenches. Finally, Section 8.4 reviews other approaches to code generator composition.

8.1 Model-Driven Engineering in Robotics

Research in robotics produced many approaches to employ component-based software engineering [3], [4], [6], [39], [42], [43]. Lately, robotics has turned to model-driven engineering as well. This has brought forth languages for imperative and event-driven robot programming [44]–[48], geometric relations and kinematics [49]–[51], assembly task modeling [52], [53], perception [54], [55], and software architectures [2], [56]–[59]. Many popular robotics architecture modeling toolchains [11], such as SmartSoft [2], SafeRobots [60], RobotML [57], or BRICS [59] facilitate robotics software engineering and provide solutions to domain-specific issues. These issues, such as advanced communication patterns, deployment, or planning, are not tackled by MontiArcAutomaton. Although many of these toolchains employ state of the art language workbenches, they neither consider exchangeable component behavior languages, nor composition of code generators.

Most robotics ADLs employ notions of components and connectors to describe the structure of the system under development and require development of component behavior with GPLs [2], [56], [59]. The RobotML [57] modeling language employs finite state machines to model component behavior. In this, it is similar to the MontiArcAutomaton version presented in [12]. In contrast to MontiArcAutomaton, RobotML does support embedding of other component behavior modeling languages. The AMARSi language family [58] supports modeling of "Motor Skill Architectures" that employ software

components with behavior descriptions as well. Here, the behavior of components is modeled via differential equations and AMARSi does not support behavior language embedding. The BRICS [59], [61] meta-model of C&C software architectures focuses on the separation of concerns between the development responsibilities. Therefore, it explicates features like scheduling, monitoring, and component configuration. BRICS does not support behavior modeling yet and thus relies on GPL component behavior implementations as well. DiasSpec [62] is a modeling language for development of Sense/Compute/Control [63] architectures that contains structural modeling elements as well. With DiaSpec, component behavior has to be defined via GPL artifacts. The OpenRTM [64], [65] architecture modeling language provides a component model where components may contain state machines that use GPL expressions to enable and fire transitions. OpenRTM does not support exchanging component behavior languages. Orocos [43], [66] is a modeling language for robotic software architectures that supports component behavior description in C++, Python, and Lua. Embedding other DSLs is not supported. Generally, we employ models over GPL component behavior descriptions, as these can be translated into different target platform-specific GPLs more easily. Furthermore, this approach allows to embed problem-specific component behavior languages, such as the RobotArm language. In contrast, component implementation with DSELs, such as SMACH [67] or other state-based formalisms, widens the conceptual gap [7] again.

8.2 Component & Connector Modeling Languages

The Architecture Analysis & Design Language (AADL) [31] is a similar modeling language for C&C systems comprising software and hardware components. AADL does neither provide a semantic foundation similar to FOCUS [26], nor mechanisms for code generator composition as presented here.

The AutoFocus [68] C&C ADL and modeling toolchain for the engineering of distributed system is based on FOCUS [26] as well. In AutoFocus, component behavior is modeled with state transition diagrams which are similar to I/O$^{\omega}$ automata. In contrast to MontiArcAutomaton, AutoFocus lacks a distinction between component types and instances which restricts component reuse.

Ptolemy II [69] is a modeling and simulation tool for actor (=component) systems. Ptolemy II allows the composition of models with heterogeneous models of computation (semantics domains and scheduling) [70]. A code generation module of Ptolemy II supports Java and C [71] for some of Ptolemy II's supported models of computation. Code for different target platforms may be generated by manually selecting templates from different packages.[8]

GenoM3 is an approach towards middleware independence of robotics software components to achieve a clear separation of concerns [72]. The separation addresses reusable algorithmic parts and integration frameworks. Any possible language can be used to describe the applications. GenoM3 employs template-based code generation and integrating additional templates is possible. However code generation and reuse take place on template level rather than on code generation level. Thus it requires a white-box insight into the templates to reuse and their integration lacks stable interfaces. Additionally, neither inter-language constraints, that have to be addressed between code generation templates, are addressed, nor the separation of code generators into different types.

MathWorks Simulink [73] comprises a block diagram language to model components & connectors. With Stateflow[9], blocks can be extended with state transition diagrams. Similar to AADL, Simulink lacks the compositionality in terms of modeling language integration and generator composition as well.

SysML [74], [75] is a modeling language family based on a subset of extended UML [76]. It comprises a language internal block diagrams which resembles MontiArcAutomaton. As component behavior can be modeled as state machine diagrams, SysML enables to express architectures similar to MontiArcAutomaton. Modeling with SysML focuses on the requirements phase [77] and thus does not take code generator composition into account.

The xADL [78], [79] is based on a meta-model for XML-based ADLs and focuses on architecture extensibility as well. It shares many features with the MontiArcAutomaton ADL but focuses on syntactic language integration. It does neither support black-box language integration, nor integration of model processing infrastructure [32]. Also, to the best of our knowledge, xADL does not support code generator composition. Instead, its "architecture instantiation schemas" [78] tie software architecture models to specific implementations.

Other ADLs, such as ArchJava [80] or Plastik [81], are implemented as domain-specific embedded languages [82], [83]. While this allows to reuse arbitrary concepts of the host language, including loops and conditional expressions, it limits their application to platforms supporting the host GPL, yields the "notational noise" [84] DSLs aim to avoid, and gives rise to "accidental complexities" [7]. The majority of ADLs however, does not consider integration of component behavior modeling languages [85]–[94].

8.3 Language Workbenches

MontiArcAutomaton is no full-fledged language workbench, but a framework to configure its ADLs with component behavior modeling languages. Its language integration capabilities rely on the language workbench MontiCore. The authors of [95] have conducted a review of language workbenches

8. See developer documentation at http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII10.0/ptII10.0.1/ptolemy/cg/README.html

9. Website of Stateflow: http://www.mathworks.de/products/stateflow/.

related to MontiCore. They reviewed the workbenches Ensō [96], Más [97], MetaEdit+ [98], MPS [99], Onion [95], Rascal [100], Spoofax [101], SugarJ [102], Whole Platform [103], and XText [104]. The review examined syntax, validation, semantics, and editor services. All language workbenches support syntactical composition, but compositional validation of integrated models, for instance for naming and type checking, similar to MontiCore is supported only by MPS and XText. None of the workbenches supports black-box code generator composition.

The Meta Programming System (MPS) [105] follows a parser-less, projectional, approach to language integration. Language designers directly model the AST and editors for specific AST nodes. Lacking an abstraction from the AST language integration breaks whenever the AST changes sufficiently.

An interesting application of MPS is mbeddr [106], a domain-specific IDE for embedded systems development. mbeddr is an implementation of C in MPS that can be extended with domain-specific languages. Embedded languages are then translated to the base language C of mbeddr. In contrast, MontiArcAutomaton keeps embedded languages on the model level to support later transition to different target GPLs and target platforms.

Xtext [104] also generates parsers from context-free grammars to produce ASTs. It supports language inheritance and language aggregation similar to MontiCore but does not provide support for language embedding.

Further discussions of related language workbenches have been presented earlier and detailed arguments for the development of modeling languages [107], their integration [34], and development of code generators [35] with MontiCore have been raised accordingly.

8.4 Generator Composition

The presented approach for code generator composition is based on explicit generator types, explicit generator interfaces and models, and run-time composition. Although it relies on the C&C nature of MontiArcAutomaton and restricts participating generators to three types, it is closely related to modular code generator design.

GenVoca [108], [109] is an approach to build software systems generators based on composing object-oriented layers. Different layers can use control blocks to exchange information. A layer is not seen as a code generator but the composition of all layers is a code generator. In contrast to this approach, we do not focus on a layered architecture of a code generator but an infrastructure for code generators composition. Our approach can be expressed in GenVoca terminology where each layer is a code generator with an explicit interface and relations to other layers. Each code generator (GenVoca layer) has to be executed to generated artifacts. In contrast in GenVoca all layers are composed and then executed.

The application building center [110] is a multi-purpose modular framework for modeling software systems. Genesys [111] is an extension that allows to develop service-oriented code generators. Each code generator represents a service that can be composed with other services. Information exchange is managed by using shared memory communication. Our approach is similar if we consider code generators to be services with interfaces. However, our approach introduces a broader generator interface to regard input language, output representation, input language constraints, execution information, artifact dependencies, and generation context information. This information is used to manage the execution and composition of the code generators.

Code generator composition using aspect-orientation at the artifact level has been described in [112]. The authors assume that a code generator produces operationally complete code fragments that are merged by a code fragment weaver. Additionally, in feature-oriented model-driven development (FOMDD) [113], multiple code generators are used to produce a software product line. Composition of code is achieved after code generation by manually writing glue code. In contrast, we do not consider manual artifact composition but focus on an infrastructure to compose code generators. We nevertheless consider composing generated artifacts relevant for reusing code generators and will address this topic in future work.

8.5 Previous MontiArcAutomaton Publications

This article is based on our previous works in the context of MontiArcAutomaton [12]–[20], [25]. In the MontiArcAutomaton report [25], we presented an earlier version of the MontiArcAutomaton ADL, where a state-based behavior modeling language was fixed part of the MontiArcAutomaton ADL grammar. In [12] we have presented a code generation framework for MontiArcAutomaton for different target languages. We extended this framework with code generator composition in [15]. In the current article we present the notion of *code generator types* that are realized as code generator interfaces, which abstracts from the MontiArcAutomaton implementation presented in [15]. Our previous publications on language integration [14], [17] presented a general overview of the language composition mechanisms of MontiCore on the example of MontiArcAutomaton. The current article focuses on syntactic and symbol language behavior language embedding in MontiArcAutomaton and contributes a technical description as well as a concrete language configuration mechanism in Section 5. The short paper [13] and the tool demonstration paper [18] introduced the ideas and goals of the MontiArcAutomaton framework and the capabilities of our implementation. This article can be seen as an extension of [13], which provides technical details of the implementation presented in [18]. The first lab course of the evaluation (Section 7) is discussed in [40]. We presented concepts for supporting multiple platforms for code generation in [16] and supporting transformations in [20]. We describe in Section 4 how these integrate into

the MontiArcAutomaton framework. Finally, [19] presented tailoring activities for MontiArcAutomaton, which are part of the development process description in Section 4 as well.

9 Discussion

This section presents and discusses some of our observations during development and application of MontiArcAutomaton.

9.1 Development Process

We have introduced the MontiArcAutomaton MDD process based on software language integration and code generator composition in Section 4. The process relies on participant roles adapted from [114]. In the case studies reported in Section 7, we have instantiated the process with different sets of roles and restricted to subsets of activities. We have observed that the separation into the reported roles is feasible in practice and helps to tailor MontiArcAutomaton to the given skills available and tasks necessary in a development project. As an example, for the first case study described in Section 7.1, the students enacted the role of application modelers and were able to produce a running system of three robots in six weeks (for more details see also [40]).

9.2 Repetition in Semantic Language Integration

Utilizing the most appropriate component behavior modeling languages requires to integrate these properly into the MontiArcAutomaton host ADL. While, with the language workbench MontiCore, the syntactic integration is naturally provided, the semantic integration still requires the language developer to implement infrastructure for symbolic reasoning about (inter-)language elements.

During language development and extension, we have observed that key concepts of the host language are adapted and translated for every embedded language. As an example, language elements representing ports or variables have to be present in most embedded component behavior modeling languages. This repetition of similar tasks happens because all behavior languages are embedded at the same point in the host language, namely inside atomic components.

We believe that this special case of language embedding has a potential for further automation on the framework level to further assist language engineers. This assistance could produce adapters automatically from symbol descriptions, check whether the languages to be embedded are actually compatible, or propose required code generators to the developers.

9.3 Generalization of Code Generator Composition

We presented a conceptual approach for composition of code generators based on the notion of generator types in Section 6. The ideas are implemented within the MontiArcAutomaton framework to enable post hoc embedding and use of new component behavior modeling languages. To broaden its applicability this approach requires future work on syntax, methods, and technical solutions.

Composition of arbitrary code generators without assumptions on their actual integration is hardly realizable because, in general, generator composition demands a more expressive composition configuration than the application configuration presented above. For instance, the composition of the code generation process may require a code generator to be executed multiple times for every input model or to fill extension points provided by another generator. Moreover, execution of a code generator may not be triggered by a model type but by selecting a code generator for a particular set of input models. A generic model to configure an application has to express such process information and constraints. Thus, future research will look into modeling these aspects. In addition, for general composition of code generators, a clear understanding of code generator composition on the different levels (input model, generator, and generated output) is required.

The generator composition illustrated above assumes that the composition of generators reflects the language embedding for component behavior. Other language integration mechanisms, such as language aggregation or language inheritance [14] will require a more complex composition. The generator for an inheriting language might, for example, require the generator for the inherited language to be executed first, such that the latter only generates additional artifacts for the model elements introduced by the inheriting language. Even the code generator interface might be adapted to support language inheritance. Future work will therefore examine the notion of generator extension points as well.

Finally, modeling language composition mechanisms have led to language reuse and language libraries. We hope to gain similar libraries and advantages from facilitating code generator composition.

10 Conclusion

We have motivated two main challenges for successful application of MDD to robotics applications. These challenges are first the need for adequate representations of multi-domain solutions in modeling languages and second the support of heterogeneous target platforms for code generation. We address these challenges in the context of C&C software architectures in MontiArcAutomaton by providing powerful extension and composition mechanisms for languages and code generators.

MontiArcAutomaton is a framework for model-driven generative development of robotics applications as C&C software architectures. At its core, it comprises a C&C ADL that allows extension with component behavior modeling languages. It further comprises modeling languages to describe framework parts and configuration. Code generators systematically transform integrated architecture models to GPL code.

With the MontiArcAutomaton ADL, component behavior can be modeled with domain-specific languages provided

by language developers and used by domain experts. This reduces the conceptual gap between domain challenges and software engineering solutions. Sophisticated language integration mechanisms based on MontiCore's capabilities support extensibility of MontiArcAutomaton with component behavior modeling languages.

This extensibility requires efficient generator composition to avoid development of multiple similar code generators for slightly different language combinations. We have detailed our concept for code generator composition based on explicit code generator types and generator models. The types are generic to C&C language integration and are realized by generator interfaces specific to MontiArcAutomaton. They enable code generators to explicate information required for composition. MontiArcAutomaton composes and executes the code generators based on models describing properties of the represented generators. Although our implementation of code generator composition is based on the language workbench MontiCore and the C&C nature of MontiArcAutomaton, we belief that these concepts generalize well into a broader scope.

Finally, we have presented case studies of different platforms MontiArcAutomaton has successfully been used with. These case studies employed different robots, middlewares, and simulators, including ROS, SmartSoft, and leJOS.

Acknowledgments

We thank the anonymous reviewers of JOSER who helped to improve the focus and presentation of this article.

Appendix

Grammars

This section presents simplified MontiCore grammars of the MontiArcAutomaton ADL and the RobotArm language. These grammars present the essence of both languages with respect to integration, but omit technical details (e.g., annotations to control AST generation) for better comprehension.

MontiArcAutomaton ADL Grammar Excerpt with Extension Point

An excerpt of the MontiArcAutomaton ADL grammar is shown in Listing 6. The grammar uses the extended EBNF form of MontiCore [33], which introduces keywords and rules to control the generation of AST classes. The grammar begins with the keyword `grammar`, followed by its name, the grammar keyword `extend` and the name of the MontiArc [36] grammar (ll. 1-2). Thus, the MontiArcAutomaton ADL grammar inherits all productions of the MontiArc grammar listed in the appendix of [36]. Afterwards, it declares a component (ll. 4-7) to begin with the model keyword `component`, followed by a name, a `ComponentHead`, and a body delimited by curly brackets. The AST part produced by `ComponentHead` will be stored in the member `head`

MontiCore Grammar

```
1  grammar MontiArcAutomatonADL
2      extends MontiArc {
3
4    Component = "component" Name
5      head:ComponentHead "{"
6        ArcElement*
7    "}";
8
9    ComponentHead =
10     generics:TypeParameters?
11     ("[" Parameter ("," Parameter)* "]")?
12     ("extends" ReferenceType)?;
13
14   interface ArcElement;
15
16   Ports implements ArcElement =
17     "port" Port ("," Port)* ";";
18
19   Port = (["in"] | ["out"]) Type Name;
20
21   Connector implements ArcElement =
22     source:Name "->" targets:Name
23                      ("," targets:Name)*;
24
25   ComponentBehavior implements ArcElement =
26     "behavior" Name? "{"
27       BehaviorModel
28     "}";
29
30   external BehaviorModel;
31
32   // ...
33 }
```

Listing 6: Excerpt of the MontiCore grammar of MontiArcAutomaton. The language inherits many productions from the MontiArc grammar [36].

of the AST node for `Component` (l. 5). The body (l. 6) is characterized by a set of `ArcElement` productions and may be empty (as denoted by the star operator "*").

The production `ComponentHead` (ll. 9-12) captures generic type parameters, configuration parameters, and extension declaration of components (cf. [25]). Generic type (l. 10) parameters are optional (as denoted by the question mark operator "?") and stored in the member `generics` of the `ComponentHead` AST.

Component parameters (l. 11) are optional and may be specified as a comma-separated list between square brackets. They are stored in a member of name `parameter` of the `ComponentHead` AST, which is a list of `Parameter` elements. The member's name is derived from its type. Each component may extend from another component as denoted by the model keyword `extends` (l. 12). The productions `TypeParameters`, `Parameter`, and `ReferenceType` are inherited from MontiArc.

`ArcElement` (l. 14) is an interface production, which

MontiCore Grammar

```
1  grammar RobotArm {
2    Programs = "robotarm" Name "{"
3      Context Content
4    "}";
5
6    Context  = ("input" Type Name ";")*;
7
8    Content  = Location* Program+;
9
10   Location = "location" Name
11                 degrees:Int "deg"
12                 height:Int "cm" ";";
13
14   Program  = "program" Name "{"
15                 (Command ";")+
16               "}";
17
18   interface Command;
19
20   Grasp implements Command =
21     (["open"] | ["close"]);
22
23   Move implements Command =
24     "move" (["top"] | ["bottom"] |
25             Name | Name Name );
26 }
```

Listing 7: MontiCore grammar of the RobotArm language

resembles interfaces from object-oriented languages, i.e., these productions are abstract (cannot be parsed) and the language must provide proper interface implementations. Everything usable in a component body must hence implement this interface production.

The component interface consists of a set of typed ports covered by the production `Ports` (ll. 16-17), which implements the `ArcElement` interface. The production begins with the model keyword `port` followed by a non-empty list of ports. Each port is captured by the `Port` production, which begins either with model keyword `in` or `out` to designate the port's direction. Afterwards, the port's type and name follow.

The component body also contains a set of connectors as instances of the production `Connector` (ll. 21-23), which also implements the interface `ArcElement`. Each connector begins with a name indicating its source, followed by the token `->` and a non-empty, comma-separated list of names indicating its targets. Most important, the MontiArcAutomaton ADL grammar introduces the production `ComponentBehavior` (ll. 25-28), which implements the `ArcElement` interface and begins with the model keyword `behavior`. After an optional name, a body enclosed in curly brackets follows. The body consists of the production `BehaviorModel`, which is an `external` production (l. 30) and denotes that a production of another language must be embedded here. This is the extension point for additional behavior description languages discussed in Section 5.

RobotArm Grammar

The RobotArm grammar is shown in Listing 7 and comprises eight productions. Its main production `Programs` (ll. 2-4) describes a set of RobotArm programs. Such a set begins with the model keyword `robotarm`, followed by a name and a body delimited by curly braces. The body (l. 3) contains a `Context` (l. 6), which describes a set of inputs, followed by a `Content` (l. 8). The latter consists of a set of locations followed by a set of programs. While the set of locations may be empty, the set of programs may not (as indicated by the plus operator "+").

Locations begin with the model keyword `location` followed by a name, a number, the model keyword `deg`, another number, and the model keyword `cm` (ll. 10-12).

A program (ll. 14-16) begins with the model keyword `Program`, followed by its name, and curly brackets. Each program contains a non-empty list of commands that are terminated by semicolons (l. 15). `Command` (l. 18) is an interface production that behaves similar to interfaces in object-oriented languages. It does not prescribe syntax, but can be implemented by other productions. The RobotArm grammar provides two implementations of `Command`: one is `Grasp` (ll. 20-21), the other is `Move` (ll. 23-25). `Grasp` commands either open or close the gripper and thus correspond to the model keyword `open` or `close`. `Move` commands begin with the model keyword `move` followed by one of the keywords `top` or `bottom`, a single name (interpreted as a location name), or two names (interpreted as names of inputs).

REFERENCES

[1] M. Hägele, N. Blümlein, and O. Kleine, "Wirtschaftlichkeitsanalysen neuartiger Servicerobotik- Anwendungen und ihre Bedeutung für die Robotik-Entwicklung," BMBF, Tech. Rep., 2011. 1

[2] C. Schlegel, A. Steck, and A. Lotz, "Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model," in *Introduction to Modern Robotics*, D. Chugo and S. Yokota, Eds. iConcept Press, 2011. 1, 3.1, 7, 7.3, 7.4, 8.1

[3] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Towards Component-Based Robotics," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2005, pp. 163–168. 1, 8.1

[4] D. Brugali, A. Brooks, A. Cowley, C. Côté, A. Domínguez-Brito, D. Létourneau, F. Michaud, and C. Schlegel, "Trends in Component-Based Robotics," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Springer Berlin Heidelberg, 2007, vol. 30, ch. 8, pp. 135–142. 1, 8.1

[5] D. Brugali and E. Prassler, "Software Engineering for Robotics," *IEEE Robotics and Automation Magazine*, vol. 16, no. 1, pp. 9–15, 2009. 1

[6] T. Niemueller, A. Ferrein, D. Beck, and G. Lakemeyer, *Design Principles of the Component-Based Robot Software Framework Fawkes*, ser. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, 2010, vol. 6472, pp. 300–311. 1, 8.1

[7] R. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering 2007 at ICSE.*, 2007, pp. 37–54. 1, 8.1, 8.2

[8] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. 1

[9] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld *et al.*, "BRICS – Best practice in robotics," in *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*. VDE, 2010, pp. 1–8. 1

[10] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 195–206. 1

[11] A. Ramaswamy, B. Monsuez, and A. Tapus, "Model-driven Software Development Approaches in Robotics Research," in *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, 2014. 1, 8.1

[12] J. O. Ringert, B. Rumpe, and A. Wortmann, "From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems," in *Software Engineering 2013 Workshopband*, ser. LNI, Stefan Wagner and Horst Lichter, Ed., vol. 215. GI, Köllen Druck+Verlag GmbH, Bonn, 2013, pp. 155–170. 1, 3.2, 7, 8.1, 8.5

[13] ——, "MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems," in *Workshops and Tutorials Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, May 6-10 2013. 1, 8.5

[14] M. Look, A. Navarro Perez, J. O. Ringert, B. Rumpe, and A. Wortmann, "Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems," in *Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC)*, Miami, Florida, USA, 2013. 1, 3.2, 3.2, 8.5, 9.3

[15] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann, "Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems ," in *1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014)*, ser. CEUR Workshop Proceedings, vol. 1319, York, Great Britain, July 2014, pp. 66 – 77. 1, 8.5

[16] J. O. Ringert, B. Rumpe, and A. Wortmann, "Multi-Platform Generative Development of Component & Connector Systems using Model and Code Libraries," in *1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp 2014)*, ser. CEUR Workshop Proceedings, vol. 1281, Valencia, Spain, September 2014, pp. 26 – 35. 1, 8.5

[17] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Voelkel, and A. Wortmann, "Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components," in *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Angers, France: Scitepress, 2015. 1, 3.2, 8.5

[18] J. O. Ringert, B. Rumpe, and A. Wortmann, "Composing Code Generators for C&C ADLs with Application-Specific Behavior Languages (Tool Demonstration)," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2015. New York, NY, USA: ACM, 2015, pp. 113–116. 1, 8.5

[19] ——, "Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development," in *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, U. Aßmann, C. Atkinson, E. Burger, T. Goldschmidt, and R. Reussner, Eds., ACM New York, NY, USA. L'Aquila, Italy: ACM New York, NY, USA, July 2015, pp. 41–47. 1, 4, 4.3, 8.5

[20] ——, "Transforming Platform-Independent to Platform-Specific Component and Connector Software Architecture Models," in *2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp) 2015*, ser. CEUR Workshop Proceedings, vol. 1463, Ottawa, Canada, September 2015, pp. 30 – 35. 1, 3.3, 4, 4.2, 8.5

[21] N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, 2000. 2, 3.1

[22] D. Brugali and P. Salvaneschi, "Stable Aspects In Robot Software Development," *International Journal of Advanced Robotic Systems*, vol. 3, 2006. 2, 3.1

[23] B. Rumpe, "Formale Methodik des Entwurfs verteilter objektorientierter Systeme," Doktorarbeit, Technische Universität München, 1996. 2.1

[24] J. O. Ringert and B. Rumpe, "A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing," *International Journal of Software and Informatics*, vol. 5, no. 1-2, pp. 29–53, July 2011. 2.1

[25] J. O. Ringert, B. Rumpe, and A. Wortmann, *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*, ser. Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2014, no. 20. 3, 3.2, 8.5, A

[26] M. Broy and K. Stølen, *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001. 3, 6.3, 8.2

[27] N. Medvidovic, E. M. Dashofy, and R. N. Taylor, "Moving architectural description from under the technology lamppost," *Information and Software Technology*, vol. 49, no. 1, pp. 12–31, 2007. 3.1

[28] B. Rumpe, *Modellierung mit UML*, 2nd ed., ser. Xpert.press. Springer Berlin, September 2011. 3.1

[29] C. Schlegel, T. Hassler, A. Lotz, and A. Steck, "Robotic software systems: From code-driven to model-driven designs," in *Advanced Robotics, 2009. ICAR 2009. International Conference on*, 2009, pp. 1–8. 3.1

[30] R. Bedin Franca, J.-P. Bodeveix, M. Filali, and J.-F. Rolland, "The AADL behavior annex-experiments and roadmap," in *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems. Washington, DC: IEEE Computer Society*, 2007, pp. 377–382. 3.1

[31] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012. 3.1, 8.2

[32] L. Naslavsky, L. Xu, H. Ziv, and D. J. Richardson, "Extending xADL with Statechart Behavioral Specification," in *Third Workshop on Architecting Dependable Systems (WADS), Edinburgh, Scotland*. IET, 2004, pp. 22–26. 3.1, 8.2

[33] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: a framework for compositional development of domain specific languages," in *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, 2010, pp. 353 – 372. 3.2, A

[34] S. Völkel, *Kompositionale Entwicklung domänenspezifischer Sprachen*, ser. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011. 3.2, 4.1, 5.3, 8.3

[35] M. Schindler, *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*, ser. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012. 3.2, 3.2, 6.2, 8.3

[36] A. Haber, J. O. Ringert, and B. Rumpe, "MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems," RWTH Aachen, Tech. Rep. AIB-2012-03, february 2012. 3, A, 34

[37] D. König, P. King, G. Laforge, H. D'Arcy, C. Champeau, E. Pragt, and J. Skeet, *Groovy in Action*. Manning Publications, 2015. 5.3

[38] D. Ghosh, *DSLs in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010. 5.3

[39] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. 6.3, 7, 7.2, 8.1

[40] J. O. Ringert, B. Rumpe, and A. Wortmann, "A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata," in *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, H. Giese, M. Huhn, J. Philipps, and B. Schätz, Eds., 2013, pp. 30–43. 7.1, 8.5, 9.1

[41] R. Heim, P. M. S. Nazari, J. O. Ringert, B. Rumpe, and A. Wortmann, "Modeling Robot and World Interfaces for Reusable Tasks," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 1793–1798. 7.4

[42] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty Architecture for Robotic Autonomy," in *Aerospace Conference, 2001, IEEE Proceedings.*, vol. 1, 2001. 8.1

[43] H. Bruyninckx, "Open Robot Control Software: the OROCOS project," in *2001 ICRA IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3. IEEE, 2001, pp. 2523–2528. 8.1

[44] H. Mühe, A. Angerer, A. Hoffmann, and W. Reif, "On reverse-engineering the KUKA Robot Language," in *First International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, 2010. 8.1

[45] J.-C. Baillie, A. Demaille, Q. Hocquet, and M. Nottale, "Events! (Reactivity in urbiscript)," in *First International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, Oct. 2010. 8.1

[46] I. Lütkebohle and S. Wachsmuth, "Requirements and a Case-Study for SLE from Robotics: Event-oriented Incremental Component Construction," *Workshop on Software-Language-Engineering for Cyber-Physical Systems*, 2011. 8.1

[47] A. Angerer, R. Smirra, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "A Graphical Language for Real-Time Critical Robot Commands," in *Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012)*, 2012. 8.1

[48] J. Baumgartl, T. Buchmann, and D. Henrich, "Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines," *8th International Conference on Software Paradigm Trends (ICSOFT-PT'13)*, 2013. 8.1

[49] U. P. Schultz, D. J. Christensen, and K. Stoy, "A Domain-Specific Language for Programming Self-Reconfigurable Robots," in *Workshop on Automatic Program Generation for Embedded Systems*, 2007, pp. 28–36. 8.1

[50] M. Frigerio, J. Buchli, and D. G. Caldwell, "A Domain Specific Language for kinematic models and fast implementations of robot dynamics algorithms," in *Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2011)*, 2011. 8.1

[51] T. D. Laet, W. Schaekers, J. D. Greef, and H. Bruyninckx, "Domain Specific Language for Geometric Relations between Rigid Bodies targeted to robotic applications," in *Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012)*, 2012. 8.1

[52] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A New Skill Based Robot Programming Language Using UML/P Statecharts," in *2013 ICRA IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, 2013. 8.1

[53] Vanthienen, Dominick and Klotzbuecher, Markus and De Laet, Tinne and De Schutter, Joris and Bruyninckx, Herman, "Rapid application development of constrained-based task modelling and execution using Domain Specific Languages," in *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Tokyo, Japan: IROS2013, 2013, pp. 1860–1866. 8.1

[54] Blumenthal, Sebastian and Bruyninckx, Herman, "Towards a Domain Specific Language for a Scene Graph based Robotic World Model," in *Fourth International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, November 2013. 8.1

[55] Hochgeschwender, Nico and Schneider, Sven and Voos, Holger, and Kraetzschmar, Gerhard K., "Towards a Robot Perception Specification Language," in *Fourth International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, November 2013. 8.1

[56] P. Trojanek, "Model-driven engineering approach to design and implementation of robot control system," in *Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2011)*, 2011. 8.1

[57] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, I. Noda, N. Ando, D. Brugali, and J. Kuffner, Eds. Springer Berlin Heidelberg, 2012, vol. 7628, pp. 149–160. 8.1

[58] A. Nordmann and S. Wrede, "A Domain-Specific Language for Rich Motor Skill Architectures," in *Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012)*, 2012. 8.1

[59] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1758–1764. 8.1

[60] A. Ramaswamy, B. Monsuez, and A. Tapus, "SafeRobots: A Model-Driven Framework for Developing Robotic Systems," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 1517–1524. 8.1

[61] D. Vanthienen, M. Klotzbuecher, and H. Bruyninckx, "The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming," *JOSER: Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 17–35, 2014. 8.1

[62] D. Cassou, P. Koch, and S. Stinckwich, "Using the DiaSpec design language and compiler to develop robotics systems," in *Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2011)*, 2011. 8.1

[63] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, 1st ed. John Wiley and Sons, Inc., 2009. 8.1

[64] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "RT-Middleware: Distributed Component Middleware for RT (Robot Technology)," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 1, 2005, pp. 3933–3938. 8.1

[65] N. Ando, T. Suehiro, and T. Kotoku, "A Software Platform for Component Based RT-System Development: OpenRTM-Aist," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2008, pp. 87–98. 8.1

[66] M. Klotzbücher, P. Soetens, and H. Bruyninckx, "Orocos RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages," in *International Workshop on Dynamic languages for RObotic and Sensors*, 2010. 8.1

[67] J. Bohren and S. Cousins, "The smach high-level executive [ros news]," *IEEE Robotics & Automation Magazine*, vol. 4, no. 17, pp. 18–20, 2010. 8.1

[68] F. Hölzl and M. Feilkas, "AutoFocus 3-A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems," in *Model-Based Engineering of Embedded Real-Time Systems*. Springer, 2011, pp. 317–322. 8.2

[69] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014, accessed 10/13, cited like this on request by the authors. [Online]. Available: http://ptolemy.org/books/Systems 8.2

[70] E. A. Lee, "Disciplined heterogeneous modeling - invited paper," in *MoDELS (2)*, 2010, pp. 273–287. 8.2

[71] G. Zhou, M. Leung, and E. A. Lee, "A code generation framework for actor-oriented models with partial evaluation," in *Embedded Software and Systems, [Third] International Conference, ICESS 2007, Daegu, Korea, May 14-16, 2007, Proceedings*, 2007, pp. 193–206. 8.2

[72] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "GenoM3: Building middleware-independent robotic components," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, May 2010, pp. 4627–4632. 8.2

[73] A. K. Tyagi, *MATLAB and SIMULINK for Engineers*. Oxford University Press, 2012. 8.2

[74] T. Weilkiens, *Systems Engineering mit SysML/UML*. UML. dpunkt. verlag, 2006. 8.2

[75] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*, ser. The MK/OMG Press. Elsevier Science, 2011. 8.2

[76] Object Management Group, "OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05)," May 2010, http://www.omg.org/spec/UML/2.3/Superstructure/PDF/ [Online; accessed 2015-12-17]. 8.2

[77] H. Giese and S. Henkler, "A survey of approaches for the visual model-driven development of next generation software-intensive systems," *Journal of Visual Languages & Computing*, vol. 17, no. 6, pp. 528–550, 2006. 8.2

[78] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language," in *Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2001, pp. 103–112. 8.2

[79] ——, "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages," in *24rd International Conference on Software Engineering (ICSE 2002)*. IEEE, 2002, pp. 266–276. 8.2

[80] J. Aldrich, C. Chambers, and D. Notkin, "Connecting Software Architecture to Implementation," in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*. IEEE, 2002, pp. 187–197. 8.2

[81] A. Joolia, T. Batista, G. Coulson, and A. T. Gomes, "Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform," in *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*. IEEE, 2005, pp. 131–140. 8.2

[82] P. Hudak, "Building Domain-Specific Embedded Languages," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, p. 196, 1996. 8.2

[83] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000. 8.2

[84] D. S. Wile, "Supporting the DSL Spectrum," *Computing and Information Technology*, vol. 4, pp. 263–287, 2001. 8.2

[85] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, "A language and environment for architecture-based software development and evolution," in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. IEEE, 1999, pp. 44–53. 8.2

[86] R. Van Ommering, F. Van Der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000. 8.2

[87] M. Bernardo, L. Donatiello, and P. Ciancarini, "Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language," in *Performance Evaluation of Complex Systems: Techniques and Tools*. Springer, 2002, pp. 236–260. 8.2

[88] S. Faucou, A.-M. Déplanche, and Y. Trinquet, "AN ADL CENTRIC APPROACH FOR THE FORMAL DESIGN OF REAL-TIME SYSTEMS," in *Architecture Description Languages*. Springer, 2005, pp. 67–82. 8.2

[89] A. Smeda, M. Oussalah, and T. Khammaci, "MADL: Meta Architecture Description Language," in *Software Engineering Research, Management and Applications, 2005. Third ACIS International Conference on*. IEEE, 2005, pp. 152–159. 8.2

[90] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The FRACTAL component model and its support in Java," *Software, Practice, and Experiance*, vol. 36, no. 11-12, pp. 1257–1284, 2006. 8.2

[91] P. Cuenot, D. Chen, S. Gerard, H. Lönn, M.-O. Reiser, D. Servat, C.-J. Sjöstedt, R. T. Kolagari, M. Törngren, and M. Weber, "Managing Complexity of Automotive Electronics Using the EAST-ADL," in *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*. IEEE, 2007, pp. 353–358. 8.2

[92] A. Amirat and M. Oussalah, "C3: A Metamodel for Architecture Description Language Based on First-Order Connector Types," in *11th International Conference on Enterprise Information Systems (ICEIS 2009)*, 2009, pp. 76–81. 8.2

[93] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009. 8.2

[94] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: A contract place where architectural design and code meet together," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 75–84. 8.2

[95] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning, "The State of the Art in Language Workbenches," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, M. Erwig, R. Paige, and E. Van Wyk, Eds. Springer International Publishing, 2013, vol. 8225, pp. 197–217. 8.3

[96] T. van der Storm, W. R. Cook, and A. Loh, "The design and implementation of Object Grammars," *Science of Computer Programming*, 2014. 8.3

[97] Más website http://www.mas-wb.com, [Online; accessed 2014-10-12]. 8.3

[98] S. Kelly, K. Lyytinen, and M. Rossi, "Metaedit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment," in *Advanced Information Systems Engineering*, 1996. 8.3

[99] S. Dmitriev, "Language Oriented Programming: The Next Programming Paradigm," *JetBrains onBoard*, vol. 1, no. 2, 2004. 8.3

[100] P. Klint, T. van der Storm, and J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*, 2009. 8.3

[101] L. C. L. Kats and E. Visser, "The Spoofax language workbench. Rules for declarative specification of languages and IDEs," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, M. Rinard, Ed., 2010, pp. 444–463. 8.3

[102] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, "SugarJ: Library-based Syntactic Language Extensibility," in *ACM SIGPLAN Notices*, 2011. 8.3

[103] R. Solmi, "Whole platform," Ph.D. dissertation, University of Bologna, 2005. 8.3

[104] M. Eysholdt and H. Behrens, "Xtext - Implement your Language Faster than the Quick and Dirty way," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 307–309. 8.3

[105] M. Voelter and K. Solomatov, "Language and IDE Modularization, Extension and Composition with MPS," *Software Language Engineering (SLE'10)*, p. 16, 2010. 8.3

[106] M. Voelter, D. Ratiu, B. Kolb, and B. Schätz, "mbeddr: instantiating a language workbench in the embedded software domain," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 339–390, 2013. 8.3

[107] H. Krahn, *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*, ser. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, 2012. 8.3

[108] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 4, pp. 355–398, Oct. 1992. 8.4

[109] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The genvoca model of software-system generators," *IEEE Softw.*, vol. 11, no. 5, pp. 89–94, Sep. 1994. 8.4

[110] B. Steffen, T. Margaria, R. Nagel, S. Jörges, and C. Kubczak, "Model-driven development with the jabc," in *Hardware and Software, Verification and Testing*, ser. Lecture Notes in Computer Science, E. Bin, A. Ziv, and S. Ur, Eds. Springer Berlin Heidelberg, 2007, vol. 4383, pp. 92–108. 8.4

[111] S. Jörges, *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*, ser. LNCS sublibrary: Programming and software engineering. Springer Berlin Heidelberg, 2013. 8.4

[112] S. Zschaler and A. Rashid, "Towards modular code generators using symmetric language-aware aspects," in *Proceedings of the 1st International Workshop on Free Composition*, ser. FREECO '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:5. 8.4

[113] S. Trujillo, D. Batory, and O. Diaz, "Feature oriented model driven development: A case study for portlets," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 44–53. 8.4

[114] H. Krahn, B. Rumpe, and S. Völkel, "Roles in Software Development using Domain Specific Modelling Languages," in *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006*. Finland: University of Jyväskylä, 2006, pp. 150–158. 9.1

Jan Oliver Ringert received his computer science diploma from Technical University Braunschweig, in 2009, and the Ph. D. degree in computer science from RWTH Aachen University, in 2014. Currently, he has a post doc position at Tel Aviv University. His research interest covers software engineering, model-driven software development, and robotics. J. O. Ringert received scholarships from the German National Academic Foundation, the German Research Foundation, the Minerva Foundation, and the ACM and Springer Best Paper Awards of the MoDELS conference in 2011 and 2015. He is a member of ACM and IEEE.

Alexander Roth received his B. Sc. and M. Sc. degrees in computer science from RWTH Aachen University in 2010 and 2012. Currently, he is a research assistant and Ph.D. candidate at Software Engineering at RWTH Aachen University. His research interest covers software engineering, design and development of code generator, and code generator product lines.

Bernhard Rumpe is chair of the Department for Software Engineering at the RWTH Aachen University, Germany. His main interests are software development methods and techniques that benefit form both rigorous and practical approaches. This includes the impact of new technologies such as model-engineering based on UML-like notations and domain-specific languages and evolutionary, test-based methods, software architecture as well as the methodical and technical implications of their use in industry. He has furthermore contributed to the communities of formal methods and UML. Since 2009 he started combining modeling techniques and cloud computing. He is author and editor of eight books and editor-in-chief of the Springer International Journal on Software and Systems Modeling. See http://www.se-rwth.de/topics/ for more.

Andreas Wortmann received his computer science and business informatics diplomas from RWTH Aachen University in 2010 and 2011. Currently, he is a research assistant and Ph.D. candidate at Software Engineering at RWTH Aachen University. His research interest covers software engineering, model-driven software development, robotics, and software language engineering. He is a member of IEEE and the Technical Committtee on Software Engineering for Robotics and Automation.