Composing Code Generators for C&C ADLs with Application-Specific Behavior Languages (Tool Demonstration)

Jan Oliver Ringert  
School of Computer Science, Tel Aviv University, Israel  
http://cs.tau.ac.il/

Bernhard Rumpe and Andreas Wortmann  
Software Engineering, RWTH Aachen University, Germany  
http://www.se-rwth.de/

Abstract

Modeling software systems as component & connector architectures with application-specific behavior modeling languages enables domain experts to describe each component behavior with the most appropriate language. Generating executable systems for such language aggregates requires composing appropriate code generators for the participating languages. Previous work on code generator composition either focuses on white-box integration based on code generator internals or requires extensive handwriting of integration code. We demonstrate an approach to black-box generator composition for architecture description languages that relies on explicit interfaces and exploits the encapsulation of components. This approach is implemented for the architecture modeling framework MontiArcAutomaton and has been evaluated in various contexts. Ultimately, black-box code generator composition facilitates development of code generators for architecture description languages with embedded behavior languages and increases code generator reuse.

Categories and Subject Descriptors D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.13 [Software Engineering]: Reusable Software

Keywords Model-Driven Engineering, Component & Connector Architectures, Code Generation, Code Generator Composition

1. Introduction

Modeling software systems as component & connector (C&C) architecture description languages (ADLs) and application-specific component behavior languages (CBLs) facilitates the separation of concerns between domain experts and system integrators. The former provide component models with stable interfaces and behavior models formulated in the most appropriate component behavior modeling language. The latter integrate these components into an overall architecture.

Generating executable systems from ADLs with embedded CBLs requires language integration mechanisms and either re-permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

GPCE'15, October 26–27, 2015, Pittsburgh, PA, USA  
ACM, 978-1-4503-3687-1/15/10...
http://dx.doi.org/10.1145/2814204.2814224

Composing Code Generators for C&C ADLs with Application-Specific Behavior Languages (Tool Demonstration).  
http://www.se-rwth.de/publications

This contribution, we demonstrate the code generation of MontiArcAutomaton for an example software architecture. To this effect, Sect. 2 introduces preliminaries, before Sect. 3 introduces the MontiArcAutomaton toolchain. Afterwards, Sect. 4 illustrates the example and Sect. 5 discusses related work. Finally, Sect. 6 concludes.

2. Preliminaries

The modeling language integration mechanisms of MontiArcAutomaton are implemented on top of the language workbench MontiCore that is implemented with the architecture modeling framework MontiArcAutomaton. MontiCore provides a modeling language to describe the syntax of DSLs and generates language processing infrastructure. MontiCore languages are defined as context-free grammars with well-formedness rules (“context conditions”) to model properties not expressible with context-free grammars. Based on a DSL’s grammar, MontiCore generates infrastructure to translate textual models into an abstract syntax tree (AST) representation, check the DSL’s context conditions, integrate it with other languages, and translate its models into general-purpose programming language (GPL) artifacts. For each non-terminal rule of the DSL’s grammar, MontiCore generates an AST class to represent the rule’s content. Terminal rules are mapped to primitives. The syntax-oriented, black-box language integration mechanisms of MontiCore support language aggregation, language embedding, and language inheritance. MontiArcAutomaton uses language embedding to integrate elements of CBLs into the ADL. To this effect, the model processing infrastructure for each language can be generated individually and without considering future embedding. MontiCore combines these infrastructures to process integrated models and generates DSL-specific tools that encapsulate everything required to parse and check models of the DSL. The template-based code generation framework of MontiCore employs the FreeMarker template engine to translate models into GPL artifacts. However, it does not provide means to compose code generators in a black-box fashion. Detailed descriptions of MontiCore and its infrastructure are available.
3. MontiArcAutomaton

The MontiArcAutomaton modeling framework\(^2\) for C&C software architectures integrates an extensible ADL with application-specific CBLs and a comprehensive code generation framework. The MontiArcAutomaton ADL models logically distributed C&C software architectures where components perform computations and connectors control communication. Components are black-boxes with stable interfaces that consist of typed and directed ports’ configuration parameters, and type parameters to support definition of generic component types. Components are either atomic or composed: atomic components yield a component behavior description – either a CBL model or a reference to a GPL artifact. Composed components instead contain a hierarchy of components and their behavior emerges from their interaction. The MontiArcAutomaton ADL supports component inheritance and distinguishes between component types and their instantiation. It also enables embedding arbitrary MontiCore languages into an extension point for non-terminals of CBLs and has been used with different languages, such as I/O\(^a\) automata [5], IO Tables [4], or programs for a simple robot arm [2]. Embedding allows describing components and their behavior in integrated artifacts and facilitates maintenance and evolution of the architecture. All data types are modeled as UML/LP [11] class diagrams (CDs). Figure 1 depicts the software architecture ToasterBot that models a robotic toast service system. The robot consists of seven components of which ToasterBot is composed from multiple subcomponents. The robot receives input from its sensors Receiver, ColorSensor, and UltraSonic which emit signals over their outgoing ports (cf. port d of data type Float). Based on its inputs, the component Controller calculates the next action and instructs the component ArmController to perform a certain program. The behavior of component Controller is modeled with an I/O\(^a\) automaton CBL and the behavior of is modeled ArmController is modeled using the RobotArm CBL.

3.1 MontiArcAutomaton Toolchain

Architecture models are parsed by MontiArcAutomaton’s model processing infrastructure before being checked for well-formedness, and finally transformed into executable systems using template-based code generators [1]. Figure 3 illustrates the toolchain and its constituents. After integrating the behavior languages required for parsing component models with integrated CBLs, the application’s component models are processed. This entails parsing the models into an AST and checking their well-formedness rules. Afterwards, the code generators selected in the application configuration model are instantiated and configured. Ultimately, the composed code generator transforms the components into GPL artifacts that comply to the same run-time system as the artifacts of handcrafted component implementations. Building MontiArcAutomaton applications requires not only invoking its code generation toolchain but also dependency management, code compilation, and execution of tests. To this effect, MontiArcAutomaton employs the Maven\(^3\) build automation tool with a special plugin that configures and executes parsing, checking, and code generation. Maven takes care of build process order, dependency resolution, uses the DSL’s tools generated by MontiCore to process models, compiles the resulting artifacts, and executes tests. Furthermore, Maven enables integration into the Eclipse IDE as well as with command line clients.

3.2 Code Generator Composition

MontiArcAutomaton combines modeling languages for three separate concerns: C&C structure, data types, and component behavior. These languages are developed independently of each other by different domain experts. The development of code generators follows the same separation, e.g., code generator developers for the structural C&C part do not require expertise about code generators for behavior. Code generator interaction is governed by MontiArcAutomaton. As the three concerns are well-separated and follow the language integration mechanisms, code generator composition in MontiArcAutomaton and similar C&C ADLs can be described in terms of the participating code generators’ types. The code generator composition infrastructure of MontiArcAutomaton [2] builds upon three types of code generators: (1) component structure generators translate component interfaces, ports, connectors, and other structural ADL elements to GPL artifacts; (2) behavior generators translate component behavior models into GPL artifacts; (3) data type generators translate CDs into GPL artifacts.

Component structure generators process complete architecture models and thus require invoking component behavior generators whenever they process a component with embedded behavior model. Structure generators invoke registered behavior generators based on the language elements (AST nodes) these are responsible for. Following the separation of concerns between architecture experts and behavior experts, component structure generators and component behavior generators produce individual GPL artifacts. Integration patterns for generated code are formalized in interfaces, which component structure generators and behavior generators explicate. All participating code generators define how they may be invoked and may restrict the models they are capable to process by additional context conditions. For each generator type, an interface exists, that formalizes these requirements and each participating code generator is expected to implement exactly one of these interfaces. Extracts of two such interfaces and their relations to other composition artifacts are depicted in Figure 3, which briefly recapitulates [2]. Here, IComponentGenerator represents the interface of a component generator and IBehaviorGenerator represents a behavior generator. Both interfaces describe properties of the respective code generator, such as the processable modeling language

\(^2\) Available via: http://monticore.de/robotics/montiarcautomaton

\(^3\) Apache Maven website: https://maven.apache.org/
component implementations generated from CBL models and handcrafted behavior implementations for atomic components without embedded behavior models. The packages ComponentCode, IOAutomatonCode, and RobotArmCode are generated by the respective generators. Package ComponentCode contains the implementations for all components of the generated system. For atomic components, these rely on implementations of interface IComputable to provide behavior as explicated per getRuntimeSystem() and entailed by the convention that each RTS contains a single behavior interface of this name. Each behavior generator produces artifacts that implement this interface. As components are black-boxes, the configuration of behavior models can only be passed through the configuration parameters of the interface. Furthermore, the component generator controls the class name of the resulting behavior artifact (Fig. 3). Hence, it may directly reference to the results of the behavior generators and their arguments.

4. Example

We have developed a robotic system to toast bread

as depicted in Fig. 5. The system consists of multiple Lego NXT controllers, sensors, and actuators. It is modeled as the MontiArcAutomaton software architecture depicted in Fig. 1 and uses two embedded CBLs: I/O\textsuperscript{10} automata and RobotArm programs. The MontiArcAutomaton toolchain was extended with the RobotArm language and a behavior generator was developed as illustrated in Fig. 3. The atomic components Receiver, ColorSensor, UltraSonic, ToasterController contain no behavior model and hence, component behavior implementations were handcrafted. Fig. 5 displays the outline of the ToasterBot’s Eclipse project, which depicts its models, handcrafted implementations, and generated Java artifacts in the project explorer on the left. The right depicts the corresponding application configuration model that selects the code generators and part of code generated for ArmController.

5. Related Work

Previous work on modeling C&C software architectures focuses on modeling and analysis aspects [14] [15] [16] instead of language integration or code generation issues. Where language integration of CBLs is considered [17] [16] it is either restricted to the meta model level and does neither support black-box language integration, nor integration of language processing infrastructure [13]. C&C ADLs that consider code generation are specialized to code generators for specific language aggregates [19] [20]. Approaches towards code generator composition for arbitrary modeling languages instead either focus white-box integration based on code generator internals [21] or extensive handcrafting [22] of integration code. De-
Figure 6: The ToasterBot application contains the robot’s software architecture models, handcrafted component implementations for atomic components without behavior model, an application configuration that selects code generators, and the code generated by executing their composition. Detailed discussions of related work for different aspects are available in the respective publication.

6. Summary

We have presented the MontiArcAutomaton modeling framework and its toolchain for modeling C&C software architectures with application-specific component behavior languages. This framework enables modeling software architectures with the most-appropriate behavior languages and code generation from integrated models to GPL artifacts using compositional generators. These generators exploit a separation of concerns of ADLs with embedded behavior languages and rely on interfaces representing these concerns. The MontiArcAutomaton toolchain composes such generators based on simple generator configuration models and thus facilitates code generator reuse. MontiArcAutomaton has been applied in various contexts and is subject to ongoing research.

References