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 handcrafting 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 with component & connector (C&C) architecture description languages (ADLs) [1] 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 requires specific, hardly reusable, code generators for each combination of languages or means to integrate generators for the participating languages. We have presented an interface-based approach to black-box code generator composition for ADLs with embedded languages [2] that is implemented with the architecture modeling framework MontiArcAutomaton [3, 4, 5].

In 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 [6, 7]. 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 [8] 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 [9, 10], and translate its models into general-purpose programming language (GPL) artifacts [11]. For each nonterminal 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 [10]. 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 [11] employs the FreeMarker template engine[1] 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 [12, 8, 11].

[1] FreeMarker website: http://freemarker.org/.

GPCE'15, October 26–27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3687-1/15/10...$15.00
http://dx.doi.org/10.1145/2814204.2814224

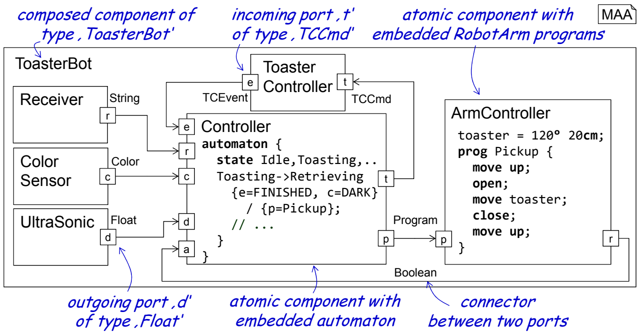

Figure 1: MontiArcAutomaton software architecture ToasterBot.

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$^{\omega}$ 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/P [11] class diagrams (CDs). Fig. 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$^{\omega}$ 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 [3]. Fig. 2 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.

[2] Available via http://monticore.de/robotics/montiarcautomaton

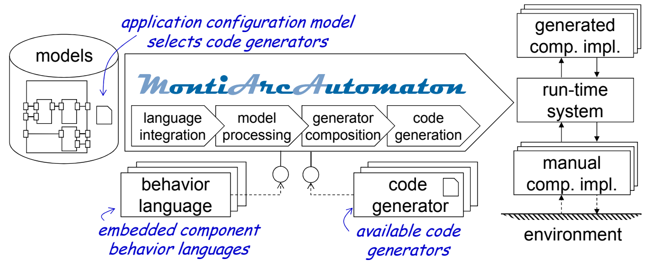

Figure 2: MontiArcAutomaton processes models with integrated CBLs and combines code generators to produce executable systems that comply to a run-time system.

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 Fig. 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

[3] Apache Maven website: https://maven.apache.org/

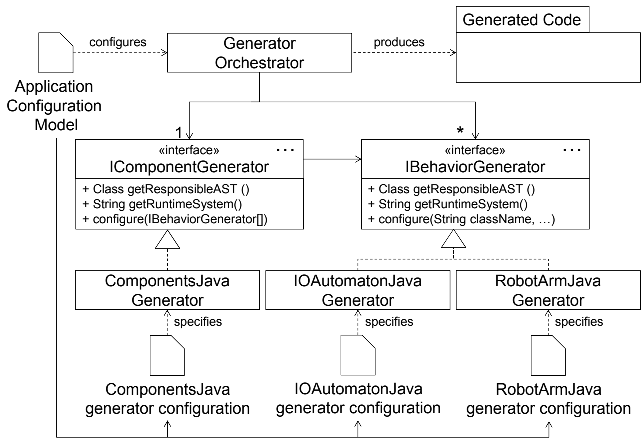

Figure 3: MontiArcAutomaton code generators belong to one of three generator types [2] and implement the corresponding interfaces. An application configuration model selects the code generators and the `GeneratorOrchestrator` combines the generators accordingly.

element (in terms of the related AST node via method `getResponsibleAST()`) and provides a method to `configure()` the code generator at run-time. To reduce code generator development effort, the actual implementations of these interfaces are generated from *generator configuration models* [2] (although manual implementation of these interfaces for special requirements is possible). An *application configuration* model selects the software architecture and the participating code generators. With this, the `GeneratorOrchestrator` is configured, which checks whether exactly one generator for each modeling language is present. Afterwards, it instantiates the implementation of `IComponentGenerator` and configures it with the available implementations of `IBehaviorGenerator`. The component generator traverses the AST and whenever it visits a behavior model, the responsible behavior generator is configured with information from the currently processed AST and invoked. Application configuration models and generator models are introduced in [2].

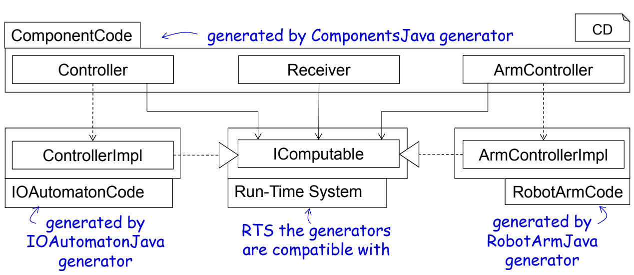

Figure 4: Compatibility of artifacts produced by different generators is ensured by compliance to run-time systems.

Each code generator in MontiArcAutomaton produces independent artifacts that are integrated via compatibility to specific interfaces of explicit *run-time systems* (RTS). This separates behavior implementations from component implementation internals and reduces sources of errors from misuse of component structure code by component behavior code. It also reduces comprehension effort required by component behavior generator developers. Instead of accessing ports, component configuration parameters, and other component internals directly, the information relevant to component implementations (such as the current inputs) is passed to behavior implementations in simplified form.

The integration of behavior implementations utilizes delegators [13] as depicted in Fig. 4. This pattern unifies the integration of

Figure 5: A robotic toast that employs multiple CBLs and for which multiple code generators are composed.

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[4] 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$^{\omega}$ 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. 6 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 [18]. 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-

[4] The toast system in action: `https://youtu.be/5EggJHtTg0c`.

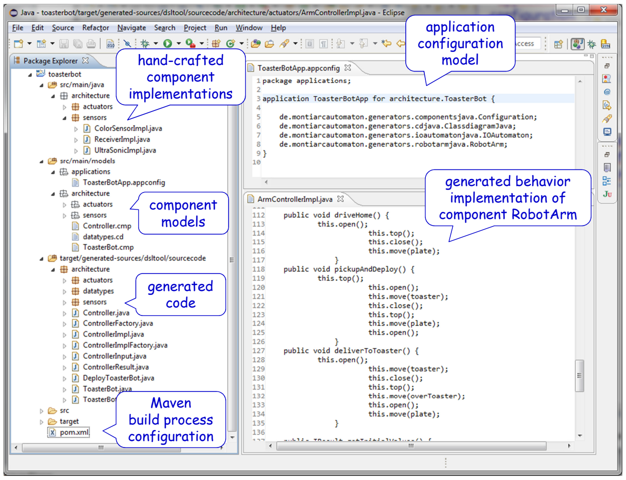

Figure 6: The `ToasterBot` application contains the robot's software architecture models, handcrafted component implementations for atomic components without behavior model, an application configuration that selects code generators, and the code generated by executing their composition.

tailed 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

[1] Nenad Medvidovic and Richard N. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages". In: *IEEE Transactions on Software Engineering* (2000).

[2] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas 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)*. 2014.

[3] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. "From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems". In: *Software Engineering 2013 Workshopband*. 2013.

[4] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. "MontiArcAutomaton : Modeling Architecture and Behavior of Robotic Systems". In: *Workshops and Tutorials Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2013.

[5] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag, 2014.

[6] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "MontiCore: Modular Development of Textual Domain Specific Languages". In: *Proceedings of Tools Europe*. 2008.

[7] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "MontiCore: a Framework for Compositional Development of Domain Specific Languages". In: *International Journal on Software Tools for Technology Transfer (STTT)*. 2010.

[8] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Shaker Verlag, 2011.

[9] Markus Look, Antonio Navarro Perez, Jan Oliver Ringert, Bernhard Rumpe, and Andreas 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)*. 2013.

[10] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas 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*. 2015.

[11] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Shaker Verlag, 2012.

[12] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Shaker Verlag, 2012.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[14] Manfred Broy, Franz Huber, and Bernhard Schaetz. "AutoFOCUS – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme". In: *Informatik-Forschung und Entwicklung* (1999).

[15] Frank Höwing. "Effiziente Entwicklung von AUTOSAR-Komponenten mit domänenspezifischen Programmiersprachen". In: *Proceedings of Workshop Automotive Software Engineering*. 2007.

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

[17] Eric M Dashofy, André Van der Hoek, and Richard N Taylor. "A Highly-Extensible, XML-Based Architecture Description Language". In: *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*. IEEE. 2001.

[18] Leila Naslavsky, Hadar Ziv Dias, H Ziv, and D Richardson. "Extending xADL with Statechart Behavioral Specification". In: *Third Workshop on Architecting Dependable Systems (WADS), Edinburgh, Scotland*. IET. 2004.

[19] Christian Schlegel, Andreas Steck, and Alex Lotz. "Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model". In: *Introduction to Modern Robotics*. 2011.

[20] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications". In: *Simulation, Modeling, and Programming for Autonomous Robots*. 2012.

[21] Markus Voelter and Iris Groher. "Handling Variability in Model Transformations and Generators". In: *7th OOPSLA Workshop on Domain-Specific Modeling*. 2007.

[22] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.