[BKRW17a] A. Butting, O. Kautz, B. Rumpe, A. Wortmann:
Architectural Programming with MontiArcAutomaton.
In: International Conference on Software Engineering Advances (ICSEA'17), pages 213-218, Athens, Greece, 2017.
www.se-rwth.de/publications/

Architectural Programming with MontiArcAutomaton

Arvid Butting, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann

Software Engineering
RWTH Aachen University
Aachen, Germany
Email: {lastname}@se-rwth.de

***Abstract*—Modeling software architectures usually requires programming the behavior of components interfacing general programming language (GPL) libraries. This raises a gap between modeling activities and programming activities that entails switching between both activities, which requires considerable effort. Current research on architecture description languages (ADLs) focuses on employing state-based component behavior modeling techniques or integrating handcrafted GPL artifacts into skeletons generated from architecture models. The former is rarely feasible to interface with GPL libraries, the latter opens the aforementioned gap. We integrate GPLs, reified as modeling languages, into the MontiArcAutomaton ADL to enable defining component behavior on model level without considering the idiosyncrasies of generated artifacts. To this effect, we apply results from software language engineering to enable a configurable embedding of GPLs as behavior languages into ADLs. This ultimately enables architecture modelers to focus on modeling activities only and, hence, reduces the effort of switching between modeling and programming activities.**

Keywords–Model-Driven Engineering, Architecture Description Languages, Architectural Programming.

I. Introduction

Component-based software engineering pursues the vision of constructing software from reusable, off-the-shelf building blocks that hide their implementation details behind stable interfaces to facilitate their composition. The behavior of such software components requires implementation with general-purpose programming languages (GPLs), which creates a conceptual gap between the problem domains and the solution domains of discourse and ultimately gives raise to the accidental complexities of programming [1]. Model-driven engineering aims at reducing this gap. To this end, it lifts models to primary development artifacts. These models are better suited to analysis, communication, documentation, and transformation. Consequently, the notion of software components has been lifted to component models that conform to architecture description languages (ADLs) of which research and industry have produced over 120 [2]. However, most of these languages focus on structural architecture aspects only. Where component behavior is considered, it is either in form of state-based modeling techniques or requires integration of handcrafted GPL artifacts. The former usually is insufficient to describe the behavior of components interfacing GPL libraries or frameworks. The latter requires architecture modelers to switch between modeling and programming activities, which require different mindsets and different tooling. Both approaches ultimately complicate development, especially considering the fact that developers switch between various activities 47 times per hour on average already [3].

We present a notion of *architectural programming* that lifts programming to modeling by reifying GPL (parts) as behavior languages and embedding these into ADL components. This enables reusing libraries and frameworks from within component models. Ultimately, this prevents architecture modelers from facing the idiosyncrasies of generated GPL artifacts and from the elaborate patterns [4] to integrate handcrafted code. Achieving this requires:

- **R1** The GPL of choice is integrated into the ADL such that a single model contains ADL parts and GPL parts.
- **R2** The relevant modeling elements of the ADL are accessible from the GPL.
- **R3** The integration of GPL elements is configurable to prevent introducing unnecessary accidental complexities into the ADL.
- **R4** The integrated artifacts consisting of ADL and GPL parts are translatable into various target languages.

In the following, Section II presents preliminaries, before Section III motivates our approach by example. Section IV presents the language embedding mechanism and its application. Afterwards, Section V presents a case study with the MontiArcAutomaton ADL. Section VI debates related work and Section VII discusses our approach. Section VIII concludes.

II. Preliminaries

To prevent architecture modelers from switching between modeling and programming, we apply the language composition mechanisms of the MontiCore [5] language workbench to the MontiArcAutomaton ADL [6] and embed the Java/P [7] modeling language into its components. This section introduces all three.

A. MontiCore

MontiCore [5] is a workbench for compositional modeling languages. It supports the integrated definition of concrete syntax (words) and abstract syntax (structure) via extended context-free grammars. From these, it generates Java parsers and abstract syntax classes for each production as depicted in Figure 1. Each generated class yields members to capture the production's right-hand sides. Underspecified *interface productions* are translated to Java interfaces. MontiCore uses the generated parsers and abstract syntax classes to translate

 ISBN: 978-1-61208-590-6

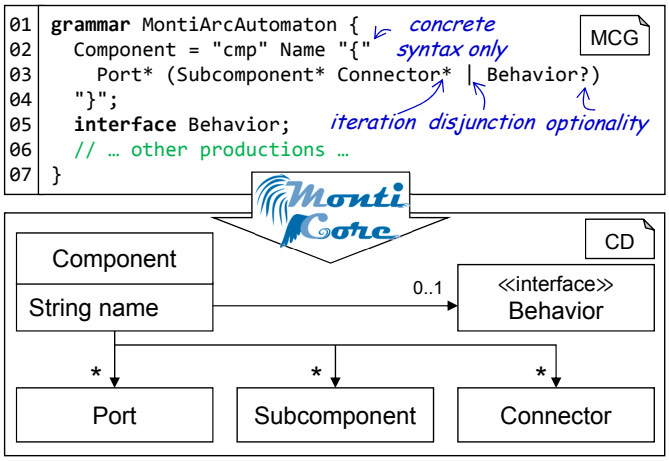

Figure 1. MontiArcAutomaton grammar describing hierarchical components with ports and connectors and the resulting abstract syntax classes.

textual models into abstract syntax trees (ASTs) on which model analyses and transformations are performed. Its visitor framework enables registering Java rules that process the ASTs to ensure the models' static semantics (well-formedness). Template-based code generators realize the languages' dynamic semantics (behavior) by translating ASTs into target language artifacts. MontiCore supports language inheritance, language embedding, and language aggregation to integrate modeling languages [8]. Inheriting languages can arbitrarily reuse productions of their parent languages (*e.g.,* to create specialized language variants). Embedding languages integrates parts of embedded languages into their underspecified *interface productions* (*e.g.,* Java embedding SQL to realize database queries). Language aggregations loosely couple languages via references (*e.g.,* state machines referencing members of the class diagram types they operate in). The language integration mechanisms rest on grammar combination (inheritance, embedding) and symbolic integration (all). For the symbolic integration of behavior languages with MontiArcAutomaton, *e.g.,* to check the validity of assignments to ports, the AST types of relevant language elements are adapted to their MontiArcAutomaton counterparts (*e.g.,* variables to ports).

B. MontiArcAutomaton

MontiArcAutomaton [6] is an architecture modeling infrastructure that comprises an extensible component & connector (C&C) ADL, various model transformations, and a modular, template-based code generation framework. Its ADL is realized with MontiCore and the infrastructure employs MontiCore's language integration mechanism to enable plug-and-play embedding of modeling languages to describe component behavior, including concrete syntax, abstract syntax, static semantics, and dynamic semantics. Core concepts of its ADL are sketched in Figure 1. It has been configured with various state-based behavior modeling languages and successfully deployed to service robotics applications [9].

C. Java/P

Java/P [7] is a MontiCore language resembling Java 1.7. It is used as action language in the UML/P [7] language family and supports the complete concrete and abstract syntax of Java 1.7 as well as its well-formedness rules. Figure 2 displays

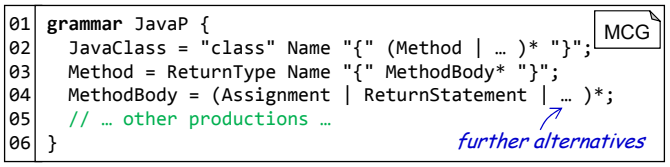

Figure 2. Simplified excerpt of the Java/P grammar.

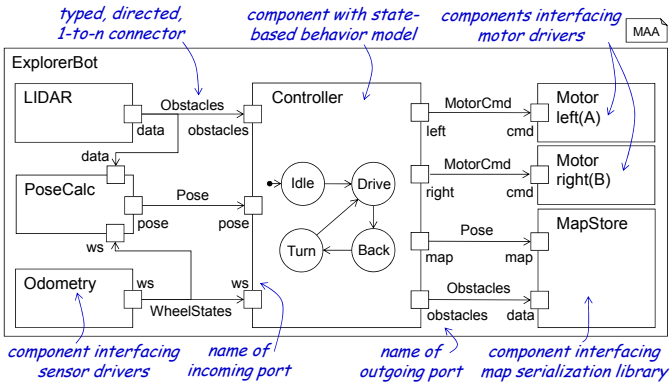

Figure 3. Software architecture for a mobile exploration robot.

an excerpt of its grammar. Its models are translated into Java artifacts using MontiCore's code generation framework. Reifying a GPL as a modeling language enables to easily extend it with new constructs (*e.g.,* notions of components as in ArchJava [10]), well-formedness rules (such as preventing assignment of `null` values), and shortcuts (for instance, automatically generating getters and setters for members). Reifying Java with MontiCore furthermore yields a parser capable of processing Java 1.7 classes, which allows to *model* Java programs interacting with libraries and frameworks.

III. Example

Consider modeling the software architecture of a mobile service robot that should explore and map uncharted territories. Such a robot must be able to perceive its environment, calculate actions based on these perceptions, and perform these actions to manipulate the environment. Perception and manipulation require interfacing sensors and actuators, respectively. Usually, these are accessed using GPL libraries providing high-level functionalities based on their mechatronic realization and software drivers. To interface these libraries, model-driven approaches must either lift GPL-like imperative programming mechanisms to model level or postpone integration to the level of generated code. For the latter, various patterns have been developed [4], all of which require by nature comprehending the idiosyncrasies of the GPL code generated from the models. The C&C software architecture for such a system is depicted in Figure 3. It perceives the environment via components interfacing a LIDAR scanner and the robot's odometry, decides on the next action via its fully modeled controller, and passes the results to its two motors and the map storage. The sensors and motors interface GPL code ultimately controlling the respective drivers. The components `PoseCalc` and `MapStore` interface GPL libraries for mathematical operations and serialization respectively.

Without lifting sufficient expressive GPL-like programming to model level, the architecture modeler must describe the

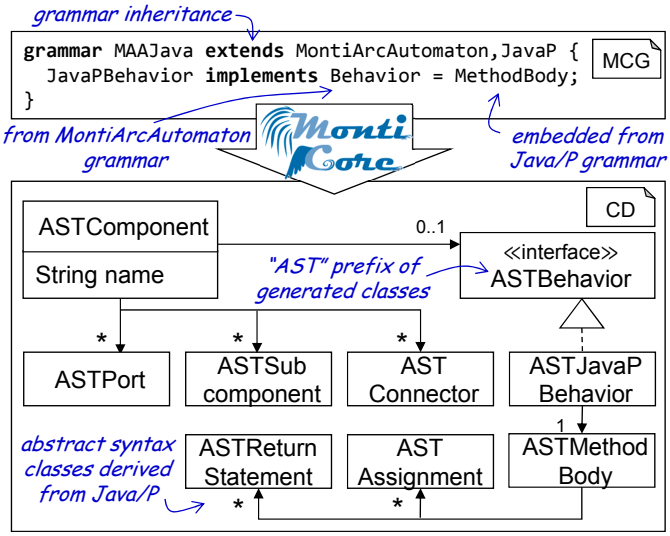

Figure 4. Grammar integrating Java/P into MontiArcAutomaton.

behavior of each of these components in the same GPL than code generated from the architecture elements. This might entail coping with various accidental complexities not directly related to computing behavior based on incoming port values, such as networking, exception handling, type casting, *etc.* Most of this can be abstracted away on model level. Additionally, the pattern selected for integrating handcrafted component behavior with generated code might give rise to further accidental complexities also not arising on model level. Selecting, for instance, the popular generation gap [4] pattern exposes the architecture modeler to all implementation details of the technical component concerns via subclassing. This exposure is another source of errors. Delegation, partial classes, protected regions, *etc.*, all yield similar complexities due to operating on GPL level. Reifying and integrating (parts of) GPLs, such as Java/P can reduce these. Finally, generation to multiple target GPLs requires re-implementing the behavior of each component type for each GPL, whereas using behavior models requires a corresponding code generator only.

IV. Embedding Java/P into MontiArcAutomaton

Embedding a modeling language into another requires combining their concrete syntaxes, abstract syntaxes, static semantics, and dynamic semantics [11]. Embedding Java/P into MontiArcAutomaton consequently rests on combining the related MontiCore artifacts accordingly. For concrete syntax and abstract syntax, this entails binding the production `MethodBody` of the Java/P grammar (Figure 2) to the interface production `Behavior` of the MontiArcAutomaton grammar (Figure 1). With MontiCore, this is achieved by leveraging its language inheritance mechanism as depicted in Figure 4. Through language inheritance, integrating the related concrete syntax and abstract syntax into MontiArcAutomaton requires to provide a corresponding implementation of its `Behavior` interface only. This enables parsing integrated artifacts into combined ASTs (**R1**).

To ensure the integrated models are well-formed with respect to both MontiArcAutomaton and Java/P, we provide an extensible well-formedness checking visitor that applies the MontiArcAutomaton well-formedness rules by default and can be extended with additional rules. As the well-formedness rules of MontiCore languages are Java classes responsible for processing a specific abstract syntax class, they can be reused without modification through registration with the MontiArcAutomaton visitor.

The integration of behavior languages into components aims at describing their input-output behavior. Hence, models of these languages must be related to the inputs and outputs of components (*i.e.,* their ports). To this effect, the names used in embedded Java/P assignments must be interpreted as symbolic references to the surrounding component's ports. For Java/P, these names usually reference to variables, hence, with MontiCore, this requires registering a corresponding adapter acting as a variable of Java/P that delegates to a port of MontiArcAutomaton [8]. In contrast to Java/P variables, ports of MontiArcAutomaton cannot be used bidirectionally: incoming ports are read-only and outgoing ports are write-only. As the Java/P well-formedness rules must be unaware of such a restriction, embedding Java/P requires integrating new well-formedness rules ensuring this. Figure 5 (top) illustrates the adaptation between MontiArcAutomaton's `Port` and Java/P's `Variable`, which takes place between the *symbols* created for each named relevant AST element. These symbols act as intermediate layer for language integration and support resolving and caching named entities from the same and other models. This adaptation enables reusing all Java/P well-formedness rules in the context of ports (**R2**). Details on symbols, their creation, and processing are available [8]. Figure 5 (bottom) also shows a new well-formedness rule to ensure incoming ports cannot be assigned values. To this effect, its `check()` method (ll. 4-12) is called by MontiArcAutomaton's visitor framework for each assignment found in an integrated model. It resolves the symbol of the assignment's target (l. 5) and, in case this actually adapts to MontiArcAutomaton's port (l. 6), checks that the ports have the correct direction (l. 8) or reports an error (l. 9). Ultimately, symbolic adaptation facilitates integration of static semantics and enables integration of new well-formedness rules easily. Eliminating and adding new well-formedness rules can also be used to tailor the language to application-specific requirements. To this effect, MontiArcAutomaton features a well-formedness rule to check for prohibited AST classes. The rule can be parametrized with a set of abstract syntax classes and iterates over all embedded abstract syntax instances to ensure none of the passed classes is used. This restriction mechanism enables excluding language elements from integration (such as `ASTReturnStatement`, which does not make sense in our integration context) as well as tailoring the embedded language to application-specific requirements (**R3**).

As MontiCore languages typically realize their dynamic semantics via code generation, integrating the dynamics of Java/P into MontiArcAutomaton requires composing their code generators. To this end, the code generator integrated for Java/P must agree on the same run-time system (RTS) as the MontiArcAutomaton code generator. A RTS is a collection of interfaces and classes enabling execution of generated models and entails, for instance, the interfaces implemented by generated component behavior realizations [6]. The interface for behavior implementation artifacts imposed by the MontiArcAutomaton RTS entails input-output semantics, *i.e.,* generated behavior realizations must provide a specific method receiving and

 ISBN: 978-1-61208-590-6

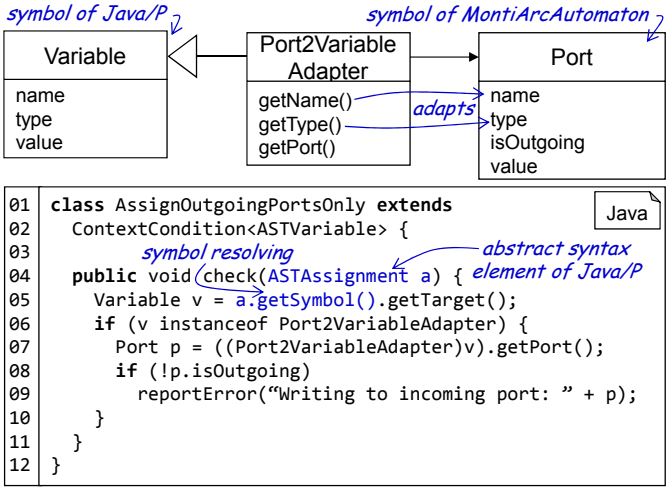

Figure 5. Symbolic integration between ports and variables and a well-formedness rule using it.

returning a set of named values. The artifacts generated from components delegate behavior computation to this method by passing the current values on incoming ports and assigning the returned name-value pairs to outgoing ports accordingly. Details on this generator composition mechanism are available [6]. The integration of Java/P into MontiArcAutomaton consequently requires a code generator agreeing on the same RTS that produces classes receiving input values, performing computations according to the method body's statements, and returning the results. Creating a generator wrapping the code generated for instances of `ASTMethodBody` (*cf.* Figure 2) requires little effort.

For MontiArcAutomaton and Java/P, code generation is template-based using the FreeMarker [12] template engine. With this, generator integration requires the single template depicted in Figure 6. This constructs a method called `compute` (l. 1) with a signature expecting all incoming ports of the component type it is generated for (ll. 2-4). Afterwards, the method creates local variables from each outgoing port (ll. 6-8) and calls the Java/P generator template responsible for translating method bodies (l. 9). Code generated from the latter operates on method parameters and local variables created for the ports and, hence, is correct by construction. This is ensured by restricting references in embedded method bodies to ports or local Java/P variables. Finally, the resulting values are collected and returned (ll. 10-14). The code generated for structural component aspects receives the resulting map and assigns the values to corresponding ports. Through enforcing type-compatibility on model level already, this assignment is straightforward.

All of this is configured by a specific internal domain-specific language (DSL) for language embedding into MontiArcAutomaton. Its models can be configured with the behavior language's grammar production to be embedded, its well-formedness rules to be reused, and its code generator to be integrated [13]. Figure 7 presents a model of this DSL responsible for embedding Java/P. Internal DSLs usually are realized via fluent GPL APIs [14] in which the methods yield names corresponding to language keywords. This enables interacting with libraries of the host GPL easily, but restricts

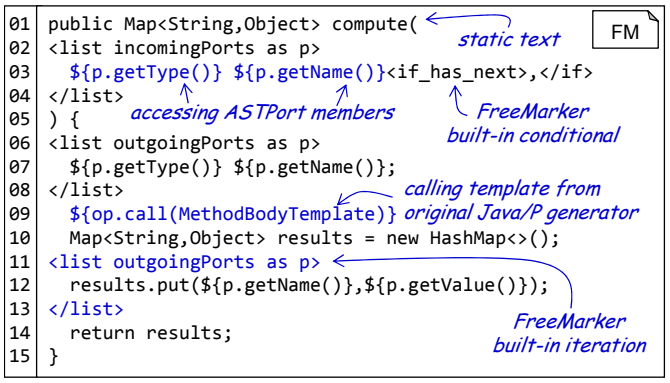

Figure 6. FreeMarker template wrapping code generator parts of Java/P for usage with MontiArcAutomaton.

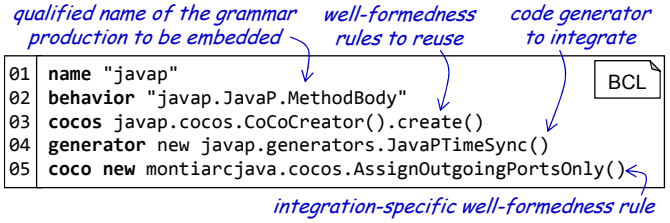

Figure 7. Model integrating Java/P into MontiArcAutomaton.

language extension. Various modern GPLs enable omitting syntactic elements if their context is unambiguous, such as the parentheses of method calls with single arguments. This further enables designing fluent APIs to resemble DSLs. The behavior configuration language model depicted in Figure 7 is realized on top of a Groovy fluent API. It consists of five concatenated method calls that configure the integration of Java/P into MontiArcAutomaton: The first line defines the unique name of the embedded behavior language. Afterwards, it references the grammar production to be embedded (l. 2) from which the integration framework synthesizes a grammar similar to `MontiArcJava` depicted in Figure 2. Subsequently, it collects the well-formedness rules of the embedded language (l. 3) and the code generator to be integrated (l. 4). Finally, it also adds the new well-formedness rule `AssignOutgoingPortsOnly` (*cf.* Figure 5) to the resulting language composition (l. 5).

V. Case Study

Reconsider the exploration robot's software architecture (Figure 3). Without lifting reifying and integrating a GPL into the ADL, implementing the behavior of its `Motor` component requires switching between modeling and programming activities, comprehending patterns for integrating handcrafted with generated implementations, and exposes the architecture modeler to the accidental complexities of the generated code.

Figure 8 describes the integrated model for component `Motor`: after declaring its type and parameters (l. 1), it yields an incoming port `cmd` of data type `MotorCommand` (l. 2) and contains an embedded behavior description (ll. 4-12). Everything between the opening bracket (l. 4) and the closing bracket (l. 12) is an embedded Java/P model. The embedded `MethodBody` (*cf.* Figure 2, l. 4) instance declares a local integer variable `speed` (l. 5), checks the value of port `cmd` and

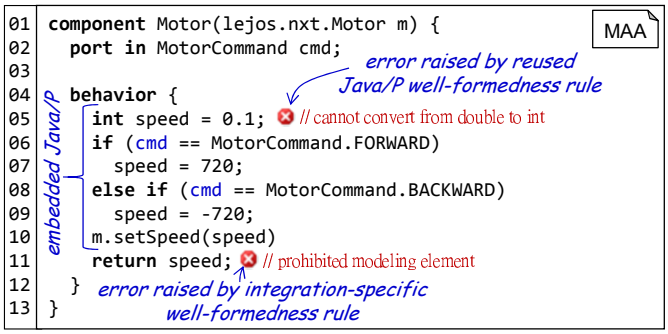

Figure 8. Textual `Motor` component with embedded Java/P behavior (ll. 5-10).

sets the value of `speed` accordingly (ll. 6-9), sets the speed of the leJOS API [15] motor instance `m` passed to the component (*cf.* l. 1), and returns the value of `speed`. Assigning a float value to an integer variable is prohibited by one of the reused Java/P well-formedness rules and consequently an error is raised (l. 5). Additionally, a prohibited instance of Java/P's return statement is found and reported also (l. 11). Overall, this syntactic and semantic integration of Java/P prevents the architecture modeler from facing generated code, supports developing with artifacts integration component structure and behavior as depicted in Figure 8. It also reduces the effort of continuously evolving and aligning two separate artifacts.

VI. Related Work

Most ADLs focus on structural concerns. Where describing component behavior is considered, many languages support state-based behavior modeling only. More complex behavior usually requires linking component models to GPL artifacts.

DiaSpec [16] is an ADL for pervasive computing systems that supports integrating behavior via the generation gap pattern [4]. Koala [17] is an ADL for the development of embedded software. Its code generator produces `C` interfaces (header files) from components and developers provide component behavior via handcrafted interface implementations. ArchFace [18] uses program points and their implementations to separate structure and behavior. Program points in component interfaces are coordinated with implementations in different GPLs. Component behavior in DAOP-ADL [19] is divided into an interface definition in form of a pointcut and its implementation potentially defined in a different artifact and language. Specifying component behavior in C2SADEL [20] requires integrating handcrafted code also.

Other ADLs feature embedded state-based behavior descriptions. In AADL [21], state-based behavior can be modeled following its behavior annex. Palladio [22] enables modeling behavior via service effect specifications, a variant of activity diagrams that describe the quality-of-service properties of components, that describe component behavior. The xADL [23] supports mapping components, connectors, and types to Java classes [23]. An extension on modeling component behavior with state machines [24] enables integrated development of structural and behavioral aspects.

Only few ADLs support other behavior descriptions. For example, in Æmilia [25], behavior is structured in blocks containing EMPA_{gr} terms [26], which are then translated into state machines. ArchJava [10] embeds ADL concepts into Java, hence it can use Java to describe component behavior. However, it does not support tailoring the language to specific requirements and is bound to Java. The π-ADL [27] also supports producing GPL code: Its behavior description mechanism supports dynamically composing, decomposing, and replicating (parts of) architectures, conditional statements, and directives for sending or receiving values via component interfaces. PiLar [28] is a reflective ADL for evolving software system structures. It provides various constructs for integrated behavior and structural modeling, such as directives for dynamically modifying architectures, conditionals, loops, and commands for exchanging message. Neither π-ADL, nor PiLar support tailoring the behavior description mechanism.

VII. Discussion

Our approach to architectural programming requires reifying the GPL of choice as a modeling language. While being of similar complexity than developing and integrating state-based behavior languages (*cf.* [29]), the lower effort in code generator development (essentially pretty printing the GPL model parts) for contexts where only a single target GPL is relevant compensates for this. Where multiple target languages are required, transforming the concepts of embedded GPL parts into another GPL is feasible as long as these concepts are generally supported or can be worked around. However, the inclusion of libraries is feasible only, if they are available for both target GPLs. Moreover, including libraries might require considering complexities that may be excluded from the embedded GPL parts (*e.g.,* exception handling is used in a library, but not included in the embedded part of the GPL). Another challenge arises from the integration of code generators through wrapping and reuse, which requires that relevant parts of the embedded GPL's code generator are accessible for wrapping. Where this is not fulfilled, generator reuse might not be possible.

VIII. Conclusion

We presented a method for pervasive architecture modeling by reifying programming languages as action languages and embedding these into the C&C ADL MontiArcAutomaton. Embedding relies on grammar interfaces, symbol adaptation, and code generator composition, all of which focus on minimizing the integration effort. The integration method supports tailoring the embedded action languages in the process by embedding what is required only and prohibiting undesired action language concepts. Modeling component behavior with integrated action languages reduces the need for switching between modeling and programming activities and, hence, ultimately reduces the effort in modeling architectures.

References

[1] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," Future of Software Engineering (FOSE '07), 2007.

[2] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What Industry Needs from Architectural Languages: A Survey," Software Engineering, IEEE Transactions on, 2013.

[3] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014.

 ISBN: 978-1-61208-590-6

[4] T. Greifenberg, K. Hoelldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Mueller, A. Navarro Perez, D. Plotnikov, D. Reiss, A. Roth, B. Rumpe, M. Schindler, and A. Wortmann, "A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages," in Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, 2015.

[5] H. Krahn, B. Rumpe, and S. Völkel, "MontiCore: a Framework for Compositional Development of Domain Specific Languages," International Journal on Software Tools for Technology Transfer (STTT), 2010.

[6] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann, "Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems," Journal of Software Engineering for Robotics, 2015.

[7] B. Rumpe, Modeling with UML: Language, Concepts, Methods. Springer International, 2016.

[8] A. Haber, M. Look, and P. e. a. Mir Seyed Nazari, "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.

[9] 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.

[10] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation." in International Conference on Software Engineering (ICSE) 2002, 2002.

[11] T. Clark, M. v. d. Brand, B. Combemale, and B. Rumpe, "Conceptual Model of the Globalization for Domain-Specific Languages," in Globalizing Domain-Specific Languages, 2015.

[12] L. A. Tedd, J. Radjenovic, B. Milosavljevic, and D. Surla, "Modelling and implementation of catalogue cards using FreeMarker," Program, vol. 43, no. 1, 2009, pp. 62–76.

[13] A. Butting, , B. Rumpe, and A. Wortmann, "Modeling Embedding of Component Behavior DSLs into the MontiArcAutomaton ADL," in Proceedings of the 4th Workshop on the Globalization of Modeling Languages (GEMOC), 2016.

[14] M. Fowler, Domain-Specific Languages. Addison-Wesley Professional, 2010.

[15] "leJOS API Website," (accessed: 2017-02-21). [Online]. Available: http://www.lejos.org/

[16] D. Cassou, J. Bruneau, C. Consel, and E. Balland, "Toward a Tool-Based Development Methodology for Pervasive Computing Applications," IEEE Transactions on Software Engineering, 2012.

[17] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," Computer, 2000.

[18] 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, 2010.

[19] M. Pinto, L. Fuentes, and J. M. Troya, "A dynamic component and aspect-oriented platform," The Computer Journal, 2005.

[20] 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, 1999.

[21] R. B. Franca, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas, "The AADL behaviour annex–experiments and roadmap," in Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems. Washington, DC: IEEE Computer Society, 2007.

[22] R. Reussner, S. Becker, and E. e. a. Burger, The Palladio Component Model. Karlsruhe, 2011.

[23] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages," in Proceedings of the 24th International Conference on Software Engineering, 2002.

[24] L. Naslavsky, L. Xu, M. Dias, H. Ziv, and D. J. Richardson, "Extending xADL with Statechart Behavioral Specification," in In Third Workshop on Architecting Dependable Systems (WADS'04), Edinburgh, Scotland, 2004.

[25] S. Balsamo, M. Bernardo, and M. Simeoni, "Combining stochastic process algebras and queueing networks for software architecture analysis," in Proceedings of the 3rd International Workshop on Software and Performance, 2002.

[26] M. Bravetti and M. Bernardo, "Compositional asymmetric cooperations for process algebras with probabilities, priorities, and time," Electronic Notes in Theoretical Computer Science, 2000.

[27] F. Oquendo, "π-ADL: An Architecture Description Language Based on the Higher-order Typed π-calculus for Specifying Dynamic and Mobile Software Architectures," SIGSOFT Softwware Engineering Notes, 2004.

[28] C. E. Cuesta, P. de la Fuente, M. Barrio-Solórzano, and M. E. G. Beato, "An "abstract process" approach to algebraic dynamic architecture description," The Journal of Logic and Algebraic Programming, 2005.

[29] L. Naslavsky, H. Z. Dias, H. Ziv, and D. Richardson, "Extending xADL with Statechart Behavioral Specification," in Third Workshop on Architecting Dependable Systems (WADS), Edinburgh, Scotland, 2004.

 ISBN: 978-1-61208-590-6