Embedding Component Behavior DSLs into the MontiArcAutomaton ADL

Arvid Butting, Bernhard Rumpe, Andreas Wortmann

Software Engineering
RWTH Aachen University
www.se-rwth.de

Abstract. Component & connector architecture description languages often need to capture application-specific or company-specific requirements. Therefore, it is a crucial prerequisite for their successful application to adapt the ADLs by customizing the languages themselves. Pervasive modeling with tailored ADLs can benefit from integration of DSLs to model-specific forms of component behavior. This requires expertise of the underlying language integration mechanisms. Current research in integrating heterogeneous component behavior DSLs into an ADL focuses on integration of specific kinds of DSLs or is restricted to syntactic integration. However, language integrators can be liberated from requiring in-depth language integration expertise using appropriate abstractions. To this effect, we present a compact DSL for the integration of behavior DSLs into a component & connector ADL that guides and facilitates this form of language integration. Modeling the embedding of behavior DSLs into ADLs facilitates their composition and ultimately the pervasive modeling of complex architectures.

1 Introduction

For a good acceptance of component & connector (C&C) architecture description languages (ADLs) [17], adapting to application-specific or company-specific requirements is a crucial prerequisite [16]. Where domain experts contribute component behavior, adaptation might include integrating appropriate component behavior DSLs into the ADL of choice [8,18]. Model-driven development and meta modeling can support such adaptation [15], but require expertise of the underlying language workbenches [6], particularly their definition and integration mechanisms [3].

Language workbenches usually provide means to define a subset of abstract syntax, concrete syntax, static semantics, and dynamic semantics for DSLs as well as generic integration mechanisms, such as importing, inheritance, or embedding, on top of these definitions (such as Eco [5], Kermeta [12], or Spoofax [25]). While providing powerful means to compose new DSLs from existing ones, these integration mechanisms are usually unaware of their operation context and hence impose that language customization requires extensive language workbench implementation knowledge. However, where the context of DSL integration is restricted, such as inheritance from a fixed parent DSL or embedding a DSL into

a well-defined extension point of an ADL, the degrees of integration freedom can be greatly reduced to facilitate language integration.

We present a DSL for the adaptation of C&C ADLs in terms of embedding application-specific component behavior DSLs. This DSL enables language integrators to easily embed DSLs as required by contributing domain experts. To this effect it exploits features of the architecture modeling infrastructure MontiArcAutomaton [20] and its underlying language workbench MontiCore [11] to alleviate reuse of the embedded DSL's syntax and semantics. This ultimately facilitates architecture modeling with domain experts.

In the following, Sect. 2 motivates the benefits of easy, post-hoc DSL embedding before Sect. 3 highlights MontiCore and MontiArcAutomaton. Sect. 4 presents the embedding DSL and Sect. 5 highlights observations. Afterwards, Sect. 6 discusses related work and Sect. 7 concludes.

2 Example

Consider the development of an assembly line robot that sorts parts according to their quality. The robot features a camera to detect parts and means to determine their quality based on various inputs. According to their quality, the parts are sorted in different bins. The architecture of the system is as depicted in Fig. 1: it consists of a composed component `PartSorter` that yields subcomponents for sensors (`Camera` and `QualityChecker`), control, and manipulation. Components exchange messages via connectors between their typed, directed ports only and are either composed or atomic, i.e., yield a behavior specification in form of a DSL model or general programming language (GPL) artifact. The behavior of components `Controller` and `Arm` is provided by two respective domain experts: one being a professional in part quality and sorting that prefers to use Statecharts. The other is an expert in robot arm movement (including trajectory planning and grasping) and prefers to use the LightRocks [24] DSL for robotic manipulation. Hence, each domain expert should be enabled to use the most appropriate DSL as depicted without requiring the system integrator to become a language engineering expert.

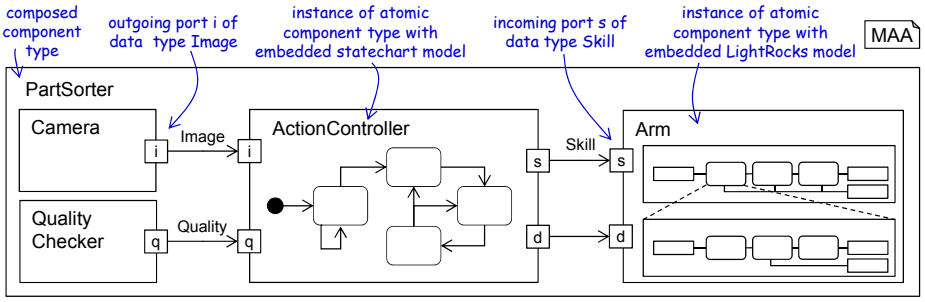

Fig. 1. Architecture of a robot that sorts parts using two embedded DSLs.

As the effect of behavior DSL embedding is restricted to ADL concepts 'visible' in atomic components, the system integrator can leverage model-driven development to describe integration of the component behavior DSLs using a specific integration DSL. The compact models of this DSL require minimal knowledge of the underlying language engineering principles and their representations in language workbenches. The DSL is tailored to component behavior DSL embedding into the specific ADL and exploits this to reduce the freedom (and complexity) of arbitrary DSL composition. In this restricted context, the integrator can thus easily embed new component behavior DSLs by adding small configuration models per DSL to be embedded. The next sections present this.

3 Preliminaries

MontiCore [14,11] is a language workbench for the engineering of compositional DSLs. It generates DSL processing infrastructure from context-free grammars (CFGs), which define abstract syntax and concrete syntax of a DSL in an integrated fashion. MontiCore DSLs yield Java-based well-formedness rules, called context conditions, for ensured static semantics, and code generators to realize the dynamic semantics of a DSL. The generated DSL processing infrastructure comprises abstract syntax tree (AST) classes, modular parsers for each production of the CFG, a symbol table acting as language interface, and context condition infrastructure. MontiCore supports three kinds of DSL composition: embedding, inheritance, and aggregation. Embedding is purely syntactic and combines the CFGs of participating DSLs, such that their instances can be processed as integrated models. This, for instance, is useful, for embedding SQL into Java or similar. Inheritance enables one DSL to extend or refine another via inheriting or overriding the parent CFG's productions. A typical use case is the introduction of new constructs to an existing DSL, such as the ADL ArchJava [1], which extends Java with C&C modeling elements. Aggregation enables the joint processing of models of independent DSLs, such as class diagrams describing the data types of a C&C ADL's ports.

For embedding, MontiCore supports extension points in form of *external productions* in the host DSL's CFG. The productions of embedded DSLs are conditionally mapped to these extension points and MontiCore combines the parsers for the corresponding productions accordingly (i.e., whenever an instance of the external production should be parsed, the corresponding embedded production is parsed instead). For aggregation, the meta models of the participating DSLs are related (for instance, to check that two ports with data types defined in class diagrams can be connected). Instead of coupling the participating DSLs' ASTs, their *symbol tables* are related. Symbol tables represent DSL interfaces for specific integration purposes and, as such, may abstract from the technical necessities of the AST classes. With aggregation, models of the participating DSL may remain in separate artifacts. For embedding with joint interpretation we apply aggregation although the models reside in integrated artifacts, hence in the following, embedding also includes joint interpretation. Instances of MontiCore's `DSLTool` class combine the generated parsers with context conditions and

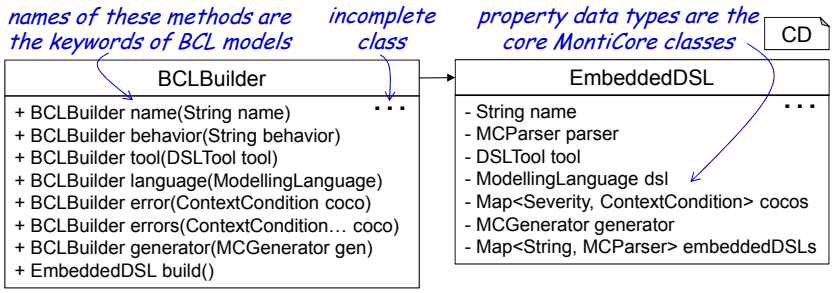

Fig. 2. Class `BCLBuilder` describes the syntax of BCL models and ultimately produces instances of `EmbeddedDSL` for each component behavior DSL to be embedded.

symbol table infrastructure to ease programmatic model processing. `DSLTool` instances further can be extended with workflows that add additional model processing capabilities (such as pre-processing model-to-model transformation or automated extraction of documentation).

MontiArcAutomaton [20] is an infrastructure for model-driven development of C&C architectures that is built with MontiCore. At its core is the MontiArcAutomaton ADL [21] that enables to model architectures as hierarchies of connected components. Components are black boxes with interfaces of directed, typed ports for communication. Port types are modeled as class diagrams. Component models in MontiArcAutomaton are either composed, if their behavior is defined by a subcomponent configuration, or atomic. The behavior of atomic components is defined via GPL artifacts or embedded DSL models. To this effect, MontiArcAutomaton comprises an extensible ADL that supports integration of DSLs to describe component behavior. To this extent, its CFG yields an external production used by the production for components. It also supports compositional code generation enabling transformation of such integrated models into GPL artifacts. This relies on the identification of three code generator kinds with specific interfaces describing the provided and required information for composition. For the specific context of behavior DSL embedding, these three kinds suffice to enable black-box code generator composition.

4 A DSL for Behavior Language Embedding

The Behavior Configuration Language (BCL), which is responsible for configuring the embedding of component behavior DSLs into the MontiArcAutomaton ADL, is realized for usage with the language workbench MontiCore and as an internal DSL implemented in Groovy. The former entails that it uses concrete syntax referencing important concepts and artifacts of MontiCore DSLs. As MontiCore is implemented in Java and Groovy is compatible with Java, it also follows that models of the integration DSL can directly interface MontiCore artifacts.

The BCL is realized as a fluent interface [9] using the builder pattern [10] to enable DSL-like usage, while actually being a Groovy GPL API. Basically, methods of the builder class become keywords of BCL models and ultimately the builder produces an instance of an embedding configuration that is used by the

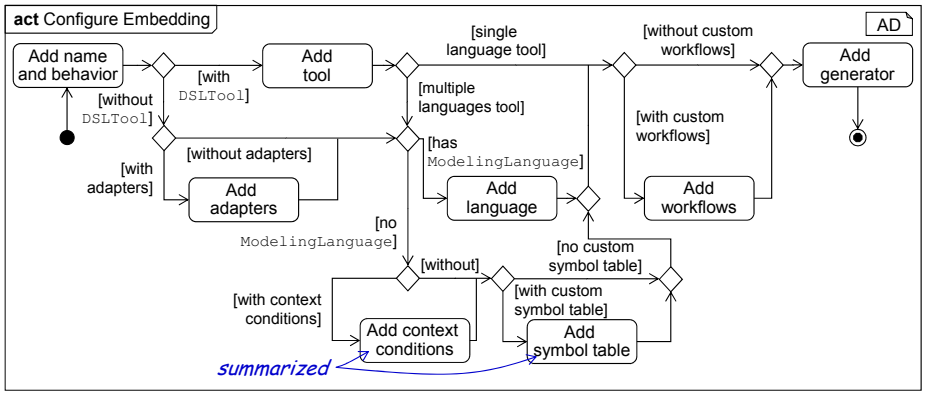

Fig. 3. Specification of name, behavior production, tool, and generator suffices to embed a DSL. More complex embedding scenarios are supported as well.

MontiArcAutomaton infrastructure. To this effect, it uses the MontiCore classes `ContextCondition`, `ModelingLanguage`, `MCGenerator`, and `MCParser`. The class diagram depicted in Fig. 2 illustrates this. The methods of class `BCLBuilder` contribute the concrete syntax and abstract syntax to describe BCL models and successively create and fill an instance of `EmbeddedDSL`. The BCL well-formedness rules are encoded in the `build()` method of `BCLBuilder` and, for instance, ensure that referenced grammar productions exist and minimal configuration is available. With Groovy, much of the notational noise [26] of a GPL can be omitted, hence, BCL models actually are chained calls of `BCLBuilder` methods with brackets and dots omitted. Omitting relations to `BCLBuilder` is possible as Groovy enables to process *scripts* in the context of so-called *base classes*. The Groovy parser is configured for `BCLBuilder` as base class and interprets method calls in its context.

BCL models vary in complexity, which depends on the behavior DSL's 'completeness' and the possibilities to derive DSL infrastructure elements from others. The activity diagram depicted in Fig. 3 describes the configuration flexibility of the BCL: providing `name`, `behavior`, and `generator` are the only mandatory properties of each BCL model. The `name` specifies which production of the behavior DSLs grammar is mapped to the unique extension point in MontiArcAutomaton's grammar (cf. Sect. 3), hence each model must at least identify the grammar production (`behavior`) to be embedded. With MontiCore defining concrete syntax and abstract syntax in integrated grammars, this specification covers embedding of both. Also, supporting usage of arbitrary productions of the behavior DSLs enables reusing required DSL parts only. As a `tool` in MontiCore may process multiple DSLs, it may comprise symbol table infrastructure, workflows, and context conditions for all processable DSLs. If not all DSLs of a `DSLTool` are to be reused, the required constituents can be specified individually in terms of their respective MontiCore `ModelingLanguage` instance [11]. If it only processes a single DSL, this DSL, together with its symbol table, workflows, and context conditions is derived from the `DSLTool`. Where no

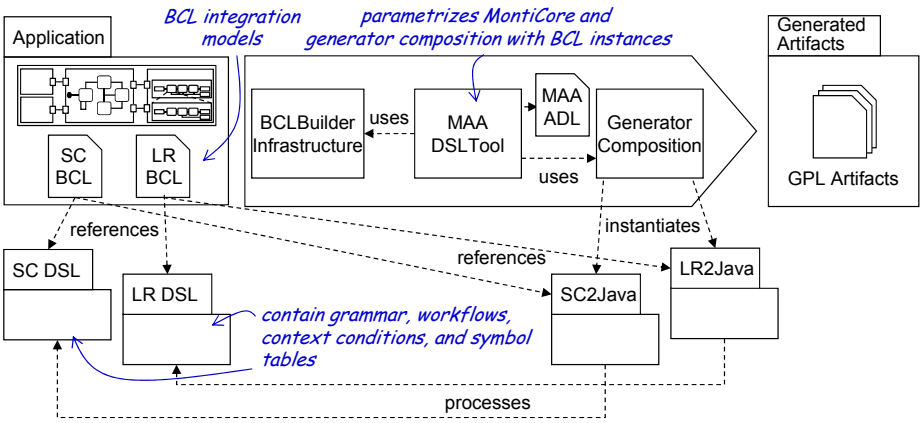

Fig. 4. Principal artifacts in processing architecture models with embedded behavior DSL models using the BCL and MontiArcAutomaton.

`ModelingLanguage` is available, their constituents must be provided individually. A BCL model also may add new `workflows` and a code generator compatible to the behavior DSL and the code generator of the MontiArcAutomaton ADL. Where additional information is necessary (such as inter-language well-formedness rules [23]), additional modeling elements support their specification.

Interpretation of Groovy scripts does not require generating any infrastructure artifacts specific to the language combination (such as combined parsers of symbol table classes), saving the expense of a generation process for each application-specific language combination. We thus extended the infrastructure of MontiArcAutomaton (i.e., its `DSLTool` subclass) to initially parse all Groovy files in a MontiArcAutomaton project in the context of the `BCLBuilder` base class and store the `EmbeddedDSL` instances before processing architecture models with embedded behavior models. Overall, the MontiArcAutomaton `DSLTool` processes architecture models and BCL models as depicted in Fig. 4 using the `BCLBuilder` infrastructure. From the models, it combines the parsers generated by MontiCore for the referenced behavior DSL grammars with the MontiArcAutomaton parser and aggregates context conditions, symbol table infrastructures, and workflows. It then parses the architecture with integrated behavior models, creates symbol table instances, and applies the aggregated context conditions and workflows. Afterwards, it loads the referenced code generators and composes their execution to produce corresponding GPL artifacts.

The most compact case of DSL integration is illustrated in Lst. 1, where UML/P Statecharts [22] are integrated into the MontiArcAutomaton ADL. The integration infrastructure addresses properties of this embedding with the name "statechart" (keyword `name`, l. 1). It embeds the production `SCBody` of the Statechart DSL (keyword `behavior`, l. 2) and extracts context conditions, symbol table, and workflows from the DSL's `SCTool` (keyword `tool`, l. 3). Furthermore, it will use the `SC2Java` code generator (keyword `generator`, l. 4) to produce combined artifacts. The latter is a reference to a generator description artifact of

```
1 name "statechart"
2 behavior "umlp.sc.SC.SCBody"
3 tool new umlp.sc.SCTool()
4 generator new SC2Java()
```
BCL

Lst. 1. BCL model that configures the integration of the Statechart DSL into MontiArcAutomaton

```
1 // abstract + concrete syntax
2 name "lightrocks"
3 behavior "lr.LightRocks.SkillDefinition"
4 tool new LightRocksTool()
5 language new SkillLanguage()
6 // inter-language well-formedness rules
7 error new ValidMotorRotation()
8 error new RespectPortDirection()
9 warning new TooManyActions()
10 // dynamic semantics
11 generator new Skill2Java()
```
BCL

Lst. 2. BCL model that configures the integration of the LightRocks Skill DSL into MontiArcAutomaton

MontiArcAutomaton that describes provided and required information for code generator composition (cf. [20]).

The BCL model for the integration of the LightRocks DSL is depicted in Lst. 2. The listing begins with `name`, `behavior`, and `tool` (ll. 1-3). The LightRocks `DSLTool` supports processing of multiple DSLs for the specification of robotic manipulation processes, tasks, skills, and actions. As only skills are being reused within MontiArcAutomaton, the corresponding MontiCore infrastructure must be selected manually. Hence, here the `DSLTool` contributes workflows only, and the `SkillLanguage` (l. 5) contributes context conditions and symbol table infrastructure related to robotic manipulation skills. Afterwards, the model adds three inter-language context conditions (ll. 7-9) that are specific to the integration of skills into MontiArcAutomaton components and either raise errors or warnings. Ultimately, it also adds a code generator (l. 11). Following this approach enables to process architecture models with embedded behavior models as depicted in Lst. 3, which shows the component `ActionController` of Fig. 1. After importing the `ImageProcessor` data type, it defines the `ActionController` component type (ll. 3-15), which comprises an interface of three ports (ll. 4-7) and a body using an embedded Statechart model (ll. 9-14). The keyword `behavior` (l. 9) indicates that an embedded behavior model follows. The subsequent concrete syntax is part of the Statechart language and the tool chain configured with the BCL model illustrated in Lst. 1 will select the `SCBody` parser generated by MontiCore to process it.

5 Related Work

Integrating component behavior DSLs is supported by AADL [8] and xADL [13] as well: AADL supports integration of state-based DSLs via its behavior annex [2], but does not consider integration of static semantics or dynamic semantics. The same holds for xADL, where embedding of a component behavior

```
1 import ImageProcessor;                                                    MAA
2
3 component ActionController {
4   port
5     in Image i,
6     in Quality q,
7     out Skill s;
8
9   behavior statechart {
10    state Waiting, Grasping, Moving, Deploying;
11    Waiting [ImageProcessor.findPart()]
12      -> Grasping / s = new Grasp(ImageProcessor.partPose());
13    // ...
14  }
15 }
```

Lst. 3. MontiArcAutomaton component model using an embedded Statechart model to describe its behavior.

DSLs also enforces to introduce a new corresponding component type [18]. Neither supports language integration in a specialized, compact DSL.

The byADL infrastructure [4] resembles a workbench for ADLs. It supports various language integration operations that enable great flexibility. However, they are generic to a broad class of application scenarios and lack the structured guidance of integration DSLs for specific purposes.

The π-ADL [19] enables to describe structure and behavior of a software architecture based on the π-calculus. In this, it generally supports to add arbitrary behavior modeling capabilities on layers on top of its ADL. This, however, ultimately yields a monolithic language aggregate where the individual components can be hardly exchanged. Furthermore, it does not support black-box composition of code generators for the individual behavior DSLs.

The Kermeta language workbench enables to mashup [12] meta-languages, static semantics, and dynamic semantics to compose DSLs. This is a more general form of composition and yields the same challenges than other general composition mechanisms in other language workbenches.

6 Discussion

On the one hand, the presented BCL is specific to its implementation with MontiArcAutomaton and MontiCore, which is reflected in its concrete syntax. On the other hand, where the integration context is sufficiently elaborated, applying integration DSLs translates to other language workbenches as well. A benefit of combining MontiCore with an internal integration DSL is in its agility: AST classes and parsers generated for ADL and behavior DSLs can be reused without requiring intermediate code generation to produce integrated AST classes and parsers (cf. [11]). Also, no artifacts are produced from the integration model, which is validated in the context of related MontiCore artifacts at design time. Our notion of DSL embedding differs from the self-extension identified in [7] and discussed in [11]. Where we advocate embedding of independent, external DSLs, self-extension usually refers to internal DSLs that can easily extend their concrete syntax with new APIs. However, with the embedding of sufficient complex DSLs (such as the Java DSL presented in [23]), self-extension can be realized

with embedding. The BCL relieves language engineers from detailed knowledge about the MontiCore language composition mechanisms. The embedding of a new behavior DSL takes place in a single BCL model, instead of registering DSL infrastructure across scattered artifacts. Embedding is decoupled from the DSLs, enabling to reuse behavior DSLs in different contexts.

7 Conclusion

We have presented a small DSL for the agile adaptation of C&C ADLs via embedding of component behavior DSLs. It supports ADL developers in fitting their ADL to the participating domain experts' DSL requirements and supports agile, post-hoc customization exploiting properties of the underlying MontiCore language workbench as well as its internal DSL nature. In the future, similarly elaborated language integration purposes might produce further specific integration DSLs (for instance, to easily integrate and exchange the data type description language an ADL operates on) for different purposes and on top of different language workbenches. We believe that such specialized integration operations can advance the application of model-driven development, where adaptation of existing DSLs is crucial.

References

1. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation. In: International Conference on Software Engineering (ICSE) 2002 (2002) 3
2. Bedin Franca, R., Bodeveix, J.P., Filali, M., Rolland, J.F.: The AADL Behavior Annex-Experiments and Roadmap. In: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (2007) 5
3. Clark, T., den Brand, M., Combemale, B., Rumpe, B.: Conceptual Model of the Globalization for Domain-Specific Languages. In: Globalizing Domain-Specific Languages. Springer (2015) 1
4. Di Ruscio, D., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: Developing Next Generation ADLs Through MDE Techniques. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (2010) 5
5. Diekmann, L., Tratt, L.: Eco: A Language Composition Editor. In: International Conference on Software Language Engineering (2014) 1
6. Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: The State of the Art in Language Workbenches. In: Software Language Engineering. Springer International Publishing (2013) 1
7. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. Computer Languages, Systems & Structures (2015) 6
8. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL:An Introduction to the SAE Architecture Analysis& Design Language. Addison-Wesley (2012) 1, 5

9. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010) 4
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995) 4
11. Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Voelkel, S., Wortmann, A.: 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) 1, 3, 4, 6
12. Jézéquel, J.M., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. Software and Systems Modeling (2013) 1, 5
13. Khare, R., Guntersdorfer, M., Oreizy, P., Medvidovic, N., Taylor, R.N.: xADL: Enabling Architecture-Centric Tool Integration with XML. In: System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference (2001) 5
14. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: Proceedings of Tools Europe (2008) 3
15. Lago, P., Malavolta, I., Muccini, H., Pelliccione, P., Tang, A.: The Road Ahead for Architectural Languages. Software, IEEE (2015) 1
16. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What Industry Needs from Architectural Languages: A Survey. IEEE Transactions on Software Engineering (2013) 1
17. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000) 1
18. Naslavsky, L., Dias, H.Z., Ziv, H., Richardson, D.: Extending xADL with Statechart Behavioral Specification. In: Third Workshop on Architecting Dependable Systems (WADS), Edinburgh, Scotland (2004) 1, 5
19. Oquendo, F.: Pi-ADL: an Architecture Description Language based on the Higher-order Typed pi-calculus for Specifying Dynamic and Mobile Software Architectures. SIGSOFT Software Engineering Notes (2004) 5
20. Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. Journal of Software Engineering for Robotics (JOSER) (2015) 1, 3, 4
21. Ringert, J.O., Rumpe, B., Wortmann, A.: Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. Aachener Informatik-Berichte, Software Engineering, Shaker Verlag (2014) 3
22. Rumpe, B.: Modeling with UML. Language, Concepts, Methods. Springer International (2016) 4
23. Schindler, M.: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Software Engineering, Band 11, Shaker Verlag (2012) 4, 6
24. Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., Wortmann, A.: 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) 2
25. Wachsmuth, G.H., Konat, G.D.P., Visser, E.: Language Design with the Spoofax Language Workbench. IEEE Software (2014) 1
26. Wile, D.S.: Supporting the DSL Spectrum. Computing and Information Technology 4 (2001) 4