Extending Architecture Description Languages With Exchangeable Component Behavior Languages

Robert Heim[1], Bernhard Rumpe[1,2], and Andreas Wortmann[1]

[1] Software Engineering, RWTH Aachen University, Germany, www.se-rwth.de

[2] Fraunhofer FIT, Aachen, Germany, www.fit.fraunhofer.de

Abstract

Architecture description languages (ADLs) encapsulate domain concerns in components. Most ADLs enforce domain experts to use general purpose programming languages (GPLs) or an especially designed, fixed component behavior language. Domain-specific languages (DSLs), on the other hand, aim to reduce the conceptual gap between problem domain challenges and GPL solutions. Current ADLs focus on software engineering and disregard integration of domain-specific component behavior languages. We combine results from DSL-based software language engineering with component & connector ADLs to present a concept for the non-invasive and exchangeable integration of both. The concept is realized with the MontiArc-Automaton component & connector ADL. This liberates domain experts from using GPLs and facilitates their contribution.

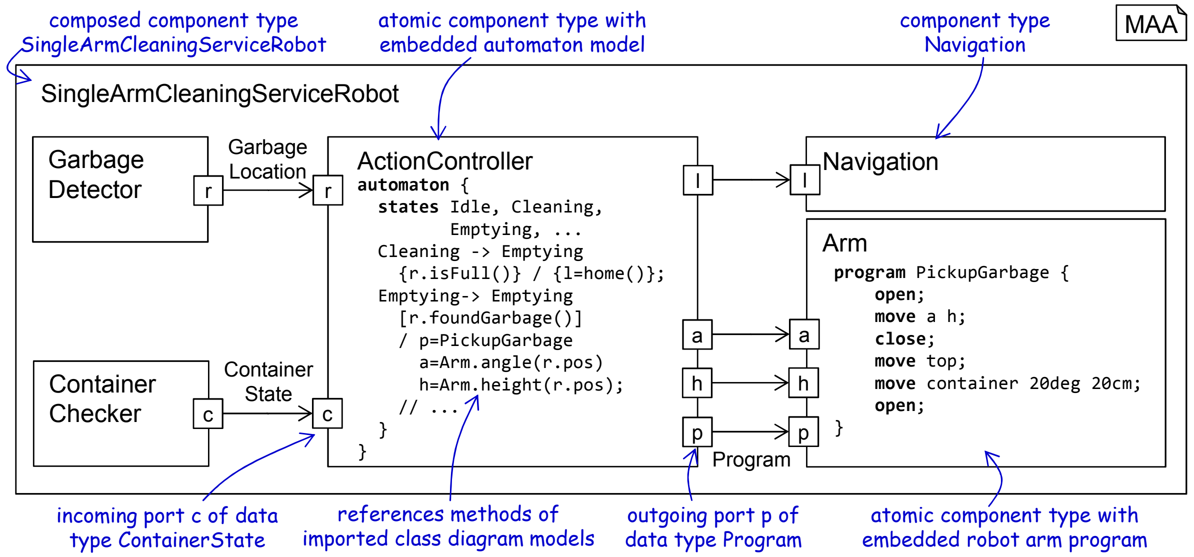
1 Introduction

Engineering non-trivial software systems requires abstraction, domain expertise, and separation of concerns. Domain experts are rarely software experts. Enforcing their contribution via general-purpose programming languages (GPLs) introduces notational noise [1] and raises accidental complexities [2]. Instead, experts should be enabled to use the most appropriate domain-specific languages (DSLs). Their integration requires to separate domain concerns from integration concerns, while supporting to reuse participating DSLs in different contexts. Component & connector (C&C) architecture description languages (ADLs) [3] enable composition of software architectures from component models. Encapsulation of components empowers separation of concerns. Nonetheless, most ADLs require domain experts to contribute using GPLs or models of apriori fixed DSLs. The former gives rise to accidental complexities, the latter demands that domain experts use DSLs foreign to their domain. We present a concept for engineering C&C software architectures with exchangeable component behavior DSLs that facilities contribution of domain experts. It relies on results from software language engineering. The contribution of this paper is a concept for language integration into C&C ADLs and its realization in MontiArcAutomaton [4] based on the MontiCore language workbench [5]. Sec. 2 motivates behavior language integration by example. Sec. 3 describes preliminaries. Afterwards, Sec. 4 presents the integration concept and Sec. 5 describes its realization. Sec. 6 discusses observations and Sec. 7 presents related work. Sec. 8 concludes.

2 Example

Consider a robotics company producing cleaning service robots. For better comprehension, reuse, separation of domain concerns, and translation into GPLs for multiple target platforms, the software architectures of the robots are modeled with a C&C ADL. The architecture for cleaning robots with a single arm to pick up garbage is depicted in Fig. 1. It relies on a garbage detector to locate garbage in its vicinity and uses a container checker to estimate whether it must be emptied. Based on their results, a central action controller derives the next action. The action is emitted via the controller's ports to a navigation component that realizes movement and to an arm component that operates the robot's arm. The behaviors of `GarbageDetector`, `ContainerChecker`, and `Navigation` are implemented in a GPL to interface robotics middleware (such as ROS [6]). The behavior of `ActionController` and `Arm` is modeled using DSLs: `ActionController` uses embedded automata (see Sec. 5), `Arm` a language to describe movement of a robot arm in terms of joint space locations [4].

DOI reference number: 10.18293/SEKE2016-13



```
automaton {
  states Idle, Cleaning,
         Emptying, ...
  Cleaning -> Emptying
    {r.isFull()} / {l=home()};
  Emptying-> Emptying
    [r.foundGarbage()]
    / p=PickupGarbage
      a=Arm.angle(r.pos)
      h=Arm.height(r.pos);
    // ...
  }
}
```

```
program PickupGarbage {
    open;
    move a h;
    close;
    move top;
    move container 20deg 20cm;
    open;
}
```

Figure 1. The software architecture for cleaning service robots with single arms comprises six components. The components ActionController and Arm encapsulate behavior in form of domain-specific language models.

The company's robot behavior expert and its robot arm control expert can contribute models of these languages instead of dealing with the accidental complexities of programming. The embedding of the languages into components furthermore enables domain experts to contribute solutions without considering cross-cutting integration concerns. As they also are independent of the target platform's GPL, the components can be transformed into artifacts for different GPLs to be reused with different platforms.

This illustrates how behavior language integration into a C&C ADL enables domain experts to use the most suitable languages to contribute component behavior and increases reuse of components with different platforms.

3 Preliminaries

MontiArcAutomaton [4] is an architecture modeling infrastructure for the model-driven engineering of C&C software architectures centered around the extensible MontiArcAutomaton ADL [7]. It supports translation of integrated components into GPL artifacts via compositional code generators [8].

With MontiArcAutomaton, modelers describe software systems as hierarchically composed components that interact via static connectors. Components encapsulate functionality behind interfaces of sets of typed, directed ports. Types of ports are modeled as class diagrams. Components either are atomic or composed: atomic components define input-output behavior via embedded behavior models or GPL behavior implementation artifacts. The input-output behavior of composed components is defined by the interaction of their subcomponents.

MontiArcAutomaton relies on the MontiCore [9, 5] language workbench. MontiCore languages are extended context-free grammars (CFG) with well-formedness rules implemented in Java. From a language's grammar, MontiCore generates infrastructure to parse its models into abstract syntax trees (ASTs). It features comprehensive language composition mechanisms: *Language embedding* syntactically integrates languages to combine (parts of) different languages within one model at well-defined extension points. *Language aggregation* combines modeling languages to enable joint interpretation of their models, which remain in separate artifacts. *Language inheritance* enables to reuse the complete CFG of the parent language to add or extend its productions. Composition relies on symbol tables, which are data structures describing the essence of model elements free from the technical coercions of their ASTs. Every MontiCore language provides the infrastructure to create and to resolve symbols. This supports correct interpretation of names and well-formedness rule checking across integrated models. MontiCore processes models with *DSLTools* that reference the language they can process and its infrastructure.

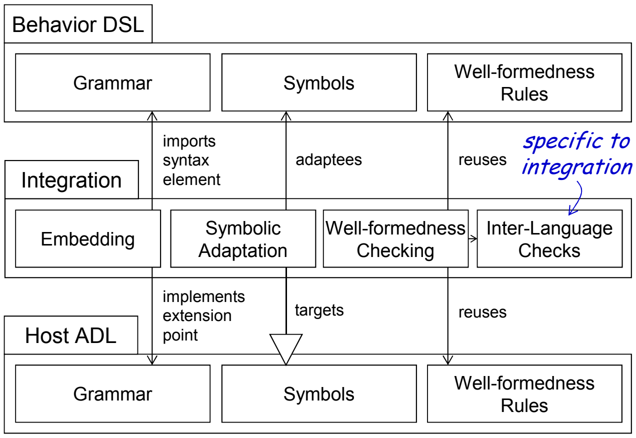
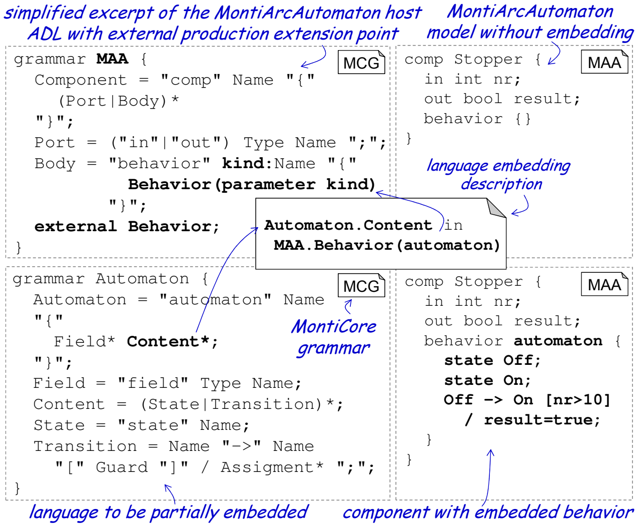


Figure 2. Behavior language integration relies on integration of syntax and static semantics.


simplified excerpt of the MontiArcAutomaton host ADL with external production extension point
MontiArcAutomaton model without embedding
MCG
MAA
language embedding description
MCG
MAA
MontiCore grammar
language to be partially embedded
component with embedded behavior

```
grammar MAA {
  Component = "comp" Name "{"
    (Port|Body)*
  "}";
  Port = ("in"|"out") Type Name ";";
  Body = "behavior" kind:Name "{"
           Behavior(parameter kind)
         "}";
  external Behavior;
}
```

```
comp Stopper {
  in int nr;
  out bool result;
  behavior {}
}
```

```
Automaton.Content in
 MAA.Behavior(automaton)
```

```
grammar Automaton {
  Automaton = "automaton" Name
  "{"
    Field* Content*;
  "}";
  Field = "field" Type Name;
  Content = (State|Transition)*;
  State = "state" Name;
  Transition = Name "->" Name
    "[" Guard "]" / Assigment* ";";
}
```

```
comp Stopper {
  in int nr;
  out bool result;
  behavior automaton {
    state Off;
    state On;
    Off -> On [nr>10]
      / result=true;
  }
}
```

Figure 3. Embedding integrates parts of a behavior language into an extension point.

4 A Concept for the Integration of Behavior Languages

For integration of behavior languages into MontiArcAutomaton, syntax and static semantics of both languages must be integrated. The latter include to reuse well-formedness rules of the embedded language and to capture changes in the meanings of references. Fig. 1 illustrates both: to parse the component `ActionController`, MontiArcAutomaton must be able to process the embedded automaton syntax as well. Furthermore, the meaning of references on transitions change: where automata models expect to operate on inputs and outputs of the automaton language, its integration should operate on ports of the surrounding component. Embedded models also are subject to (most) well-formedness rules of their stand-alone language. Integration also may entail new well-formedness rules: for embedded automata models, one might prohibit to define their own inputs and outputs in favor of ports.

Our approach to behavior language integration relies on MontiCore's language embedding, symbol adaptation, and well-formedness check reuse as depicted in Fig. 2. Embedding conditionally integrates parts of behavior languages into components, symbolic adaptation changes the interpretation of references, well-formedness checking reuses existing rules and integrates new rules. For syntactic embedding, language engineers must specify a mapping from behavior language elements to ADL language elements. Adaptation of the symbols expects that embedded languages operate on dedicated inputs and outputs. Otherwise, integration into components will hardly produce input-output behavior.

Overall, the goals is a black-box language integration in which model elements and well-formedness rules of stand-alone behavior modeling languages can be reused within components to describe their input-output behavior. Thus, integration of behavior languages into MontiArcAutomaton entails the following requirements: **R1** The (partial) syntax of stand-alone behavior languages can be conditionally integrated into component models. **R2** The meaning of references used in integrated models can be changed. **R3** Selected well-formedness rules of the embedded language can be reused. **R4** Adding integration-specific well-formedness rules is possible. The latter should be as little as possible. Ideally, it is sufficient to map symbols between the languages and extra effort for inter-language rules is not necessary.

5 Integration into MontiArcAutomaton

Successful integration of behavior languages into components must combine the language's syntaxes, ensure a joint interpretation of references, and allow to reuse their well-formedness rules. The next sections present mechanisms for these aspects.

5.1 Syntactic Integration

The grammar of MontiArcAutomaton contains an *external* production [5] that acts as extension point for component behavior. A production of the behavior language is registered for this extension point with a specific keyword (e.g., `automaton` or `program`). Hence, whenever MontiArcAutomaton parses an integrated model meeting such keyword, infrastructure for processing the component behavior language's production is invoked. With this, integration acts

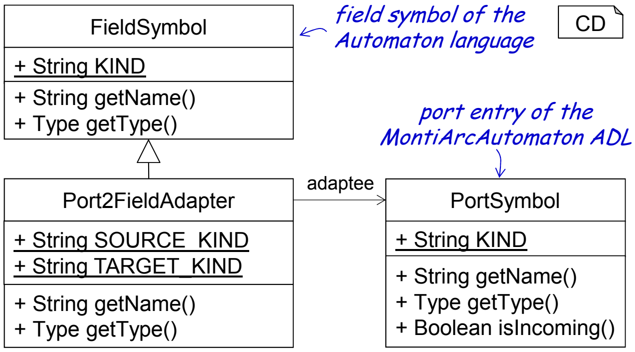


Figure 4. Adaptation between symbols allows to interpret references to behavior language inputs and outputs as references to component ports.

as a single grammar and produces a single combined AST. Fig. 3 illustrates language embedding and resulting models with an excerpt of the MontiArcAutomaton ADL (top left) and a simplified behavior language to define automata (bottom left). The MontiArcAutomaton ADL grammar contains the external `Behavior` production parametrized with the parameter `kind`. The latter is used to distinguish embedded languages and will become part of the concrete syntax. The top right of Fig. 3 shows a MontiArcAutomaton component model with two ports but without behavior. The `Automaton` grammar consists of the container production `Automaton` that contains multiple `Field` instances followed by multiple `Content` instances. The latter consist of arbitrary many states and transitions. Fields are either input data sources or output data sinks, states are names, and transitions consist of a source state name, a target state name, a guard, and assignments. The productions for guards and assignments are omitted as they are complex and do not contribute to embedding. The center of Fig. 3 depicts a mapping from production `Content` of `Automaton` to `Behavior` of `MAA`. The argument for this mapping is the keyword `automaton`, which is passed to be the `kind` of `MAA.Body` and helps MontiCore to distinguish which embedded production to select. It will also become a keyword in the concrete syntax of integrated models. Consequently, the integrated model (bottom right of Fig. 3) uses this keyword and arbitrary `Automaton.Content` elements for its `Behavior`. While this allows to embed partial syntax of behavior languages (**R1**) and to parse integrated models into combined ASTs, `Automaton` models expect `Field` instances for input and output, but should use ports when embedded. Also embedding does not integrate the well-formedness rules of the `Automaton` language. Both requires symbolic integration.

5.2 Integration of Symbols

The names used in a model to reference parts of the same or other models are symbolic references with certain meaning. For instance, the left-hand side of the assignment `result = true` of the transition depicted in Fig. 3 is only meaningful in an automaton if `result` is some form of `Field`. After embedding, MontiArcAutomaton prohibits fields in automata to avoid the underspecificaton from combining fields with ports. Consequently, the interpretation of `result` expecting to reference a field changes to referencing a port. Changing the symbolic interpretation of names in MontiArcAutomaton amounts to change the symbols it resolves when looking up a name. To this effect, MontiArcAutomaton provides infrastructure to add adapters between different symbols (such as fields and ports) to its symbol tables. Changing the interpretation of names in assignments from field references to port references requires that, whenever MontiArcAutomaton looks up a name expected to reference a field symbol, it returns a port symbol disguised as a field symbol instead (**R2**). With MontiCore, such change of interpretation is done by providing corresponding adapters (cf. Fig. 4). The `Port2Field` adapter registers to provide symbols of its `TARGET_KIND`, which corresponds to the `FieldSymbol` kind. Whenever MontiArcAutomaton tries to resolve a field symbol with a certain name, the adapter will return a port symbol of that name. This symbol than can be used to perform checks (e.g., whether `true` can be assigned to the port referenced by `result`). With the required adapters in place, all well-formedness rules of the `Automaton` language can be reused although there are only simulated `Field` symbols to check.

5.3 Integration Infrastructure

MontiArcAutomaton supports configuration of embedded productions, provision of adapters, and specification of well-formedness rules with a Groovy-based internal DSL [10]. Groovy allows to omit syntactic sugar (such as brackets around method arguments and dots between object expressions). As it further is compatible to Java, it can interface instances of the modeling language classes of MontiCore directly and allows to reuse the stand-alone infrastructure of behavior languages.

Listing 1 shows a model of the Groovy behavior configuration modeling language (GBC). This model first defines the condition and keyword `automaton` for embedding of the production `Automaton.Content` into the MontiArcAutomaton ADL (ll. 1-2). It also references the stand-alone DSLTool instance of the `Automaton` language (l. 3). From this, it retrieves the language's symbol table infrastructure and well-formedness rules (**R3**). Afterwards, it adds the integration-specific well-formedness rule

```
1 name "automaton"
2 behavior "Automaton.Content"
3 tool new AutomatonDSLTool()
4 coco new NoFieldsInEmbeddedModels()
5 adapter new Port2FieldAdapter()
```

Listing 1. GBC model for integrating the Automaton behavior language into MontiArc-Automaton.

`NoFieldsInEmbeddedModels` (**R4**) regarding prohibition of `Field` instances in embedded models (l. 4) and the `Port2FieldAdapter` depicted in Fig. 4 to interpret field references as port references. The data types of `tool`, `coco`, and `adapter` must correspond to the data types specified with the fluent interface of the class `GBCBuilder` that GBC operates on. Fig. 5 shows this class with its quintessential members and related data types. The `GBCTool` is a DSLTool that extends the `MAATool`, processes GBC models, constructs `BehaviorConfiguration` instances from these and parametrizes the `MAATool` with their information. Thus, using the `GBCTool` with corresponding GBC models allows to parse and check component models with integrated behavior languages.

6 Discussion

The mechanisms currently also allow to integrate arbitrary languages into the host ADL. Whether resulting combined models can produce input-output behavior is not checked yet. This would require means to designate productions of the behavior languages' grammars as input and output elements. Such a designation also would allow to generate parts of the adapters. Furthermore, the mechanisms could validate integration of the behavior languages' dynamic semantics with the messaging semantics of the MontiArcAutomaton ADL. As MontiCore languages codify dynamic semantics via code generators, this requires proper extension of MontiArcAutomaton's code generator composition framework [8]: explication of the semantics of produced artifacts enables the composition mechanism to reason over the semantic validity of specific language combinations. Embedding behavior language parts can introduce syntactical conflicts and MontiCore detects these at composition time and reports on these.

7 Related Work

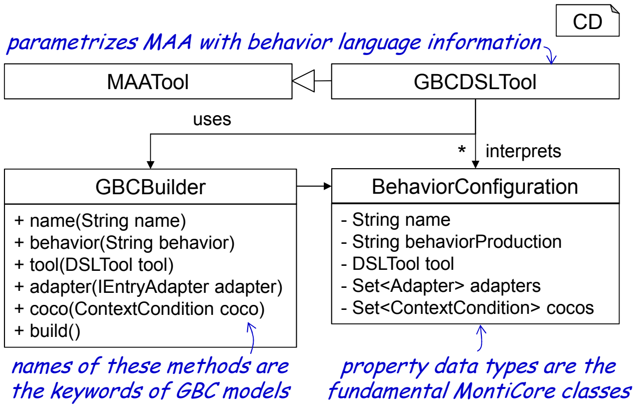

CD
parametrizes MAA with behavior language information
MAATool
GBCDSLTool
uses
*
interprets
GBCBuilder
+ name(String name)
+ behavior(String behavior)
+ tool(DSLTool tool)
+ adapter(IEntryAdapter adapter)
+ coco(ContextCondition coco)
+ build()
BehaviorConfiguration
- String name
- String behaviorProduction
- DSLTool tool
- Set<Adapter> adapters
- Set<ContextCondition> cocos
names of these methods are the keywords of GBC models
property data types are the fundamental MontiCore classes

Figure 5. The methods of class GBCBuilder define the syntax of the GBC.

The presented integration mechanisms are generalizable to other C&C ADLs. It requires the host ADLs is to provide a well-defined extension points for component behavior and the behavior languages to specify input-output behavior. However, most C&C ADLs disregard integration of component behavior DSLs [11]. To the best of our knowledge, similar integration is supported by AADL [12] and xADL [13] only. Both consider syntactic integration only, whereas MontiArcAutomaton also supports symbolic integration, reuse of well-formedness rules, and code generator composition [4] to translate integrated models into GPL artifacts.

To some extend, our approach relates to general language workbenches [14] as well. Most workbenches feature powerful language integration mechanisms, such as the abstract syntax embedding of Spoofax [15], and SugarJ [16]. These mechanisms are generic and their application to ADLs has yet to be defined for our purposes.

Our approach also relates to the byADL framework for ADL model-driven development [17]. The framework presents general meta model composition mechanisms operations providing great flexibility. Similar to general language workbenches, this freedom entails complexity and requires integrators to amass expertise in software language engineering. Our operations very specifically embed behavior into well-defined extension points of C&C ADL components and reduce the complexity for integrators.

8 Conclusion

We have presented a concept for syntactic and symbolic integration of behavior DSLs into components of a C&C ADL. It relies on well-defined extension points in the host ADLs abstract syntaxes allows to reuse abstract syntax and static semantics of behavior languages. Such integration facilities participation of domain experts as it enables to use the most appropriate modeling languages to describe component behavior.

References

[1] D. S. Wile, “Supporting the DSL Spectrum,” *Computing and Information Technology*, 2001.

[2] R. France and B. Rumpe, “Model-Driven Development of Complex Software: A Research Roadmap,” in *Future of Software Engineering 2007 at ICSE.*, 2007.

[3] N. Medvidovic and R. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages,” *IEEE Transactions on Software Engineering*, 2000.

[4] 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.

[5] H. Krahn, B. Rumpe, and S. Völkel, “Monticore: a framework for compositional development of domain specific languages,” in *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.

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

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

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

[9] H. Krahn, B. Rumpe, and S. Völkel, “MontiCore: Modular Development of Textual Domain Specific Languages,” in *Proceedings of Tools Europe*, 2008.

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

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

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

[13] L. Naslavsky, H. Z. Dias, H. Ziv, and D. Richardson, “Extending xADL with Statechart Behavioral Specification,” in *Third Workshop on Architecting Dependable Systems (WADS)*, 2004.

[14] S. Erdweg, T. van der Storm, M. Vlter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning, “The State of the Art in Language Workbenches,” in *Software Language Engineering*. Springer International Publishing, 2013.

[15] G. H. Wachsmuth, G. D. Konat, and E. Visser, “Language design with the spoofax language workbench,” *Software, IEEE*, vol. 31, no. 5, pp. 35–43, 2014.

[16] S. Erdweg, L. C. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser, “Library-based Model-driven Software Development with SugarJ,” in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 17–18.

[17] D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, “Developing Next Generation ADLs Through MDE Techniques,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 85–94.