Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language

Arvid Butting[1], Arne Haber[1,2], Lars Hermerschmidt[1,3], Oliver Kautz[1], Bernhard Rumpe[1], and Andreas Wortmann[1(✉)]

[1] Software Engineering, RWTH Aachen University, Aachen, Germany
{Butting,Haber,Hermerschmidt,Kautz,Rumpe,Wortmann}@se-rwth.de
[2] Schier Consult GmbH, Braunschweig, Germany
[3] AXA Konzern AG, Cologne, Germany
http://www.schier-consult.de,
http://www.axa.de

Abstract. Architecture description languages (ADLs) combine the benefits of component-based software engineering and model-driven development. Extending an ADL to domain-specific requirements is a major challenge for its successful application. Most ADLs focus on fixed features and do not consider domain-specific language extension. ADLs focusing on extensibility focus on syntactic augmentation only and neither consider semantics, nor the ADL's tooling. We present a systematic extension method for the MontiArc component and connector ADL that enables extending its syntax and infrastructure. The MontiArc ADL is built on top of the MontiCore workbench for compositional modeling languages and leverages its powerful language integration facilities. Based on these, we conceived systematic extension activities and present their application to customizing MontiArc for three different domains. This application of software language engineering to ADLs reduces effort for their extension and the presented method guides developers in applying it to their domain. This ultimately fosters the application of ADLs to real-world domain-specific challenges.

Keywords: Model-driven engineering · Architectural programming · Action languages · Software language composition

1 Introduction

Component-based software engineering (CBSE) is a software engineering methodology that advocates the vision of composing complex software systems from off-the-shelf components. Through this, the individual components are supposed to be reused more often, better evaluated, and hence more mature. Nonetheless, most approaches to CBSE rely on exchanging binary or source code components, which are noisy [33] solution domain [8] artifacts that are specific to the general programming language (GPL) they are formulated in. This complicates their reuse and comprehension.

Model-driven development (MDD) lifts models to primary development artifacts that are more abstract, closer to the solution domain, less noisy, better comprehensible, and automatically translatable into solution domain artifacts. Architecture description languages (ADL) [23] are modeling languages for the development of complex software systems. They combine the benefits of CBSE and MDD and have been developed for and applied to multiple challenging domains including automotive [3], avionics [6], and robotics [30]. For each of these domains, completely new ADLs with domain-specific syntax and semantics have been developed from scratch. This is expensive, which is why extending ADLs to specific requirements is one of the major challenges to their successful application [21]. However, most ADLs focus on fixed domain-specific challenges and do not support domain-specific extension. Where this is possible (such as with AADL [6] or xADL [25]), the extensions are mainly of syntactic nature. Leveraging software language engineering enables implementing better extensible ADLs. Based on state-of-the-art software language composition mechanisms [4], we have conceived the MontiArc ADL [17] for extensive systematic extension on top of the MontiCore [20] language workbench. The language engineering mechanisms of MontiCore enable to adjust MontiArc's syntax and infrastructure to domain-specific requirements. Core modeling elements of the MontiArc ADL have been introduced in [17] and an earlier variant extension method has been presented in [12]. Furthermore, the behavior language embedding mechanisms of the MontiArc derivative MontiArcAutomaton have been presented in [27]. This paper specifically contributes

- an augmented method for the structured extension of MontiArc with new syntax and semantics beyond behavior language embedding that considers reuse of well-formedness rules and no longer distinguishes between code generation and simulation; and
- three case studies describing applying this method to extending MontiArc for security architectures, robotics, and cloud-based systems.

In the following, Sect. 2 motivates the benefits of ADL extension by example, before Sect. 3 explains necessary preliminaries. Afterwards, Sect. 4 presents the MontiArc extension method and Sect. 5 describes its application to three domains. Subsequently, Sect. 6 highlights related work and discusses the approach, and Sect. 7 concludes.

2 Example

Consider a company developing distributed cloud systems for massive open online courses. For better abstraction and reuse, the company decides to model the structure of the systems using a component and connector (C&C) ADL. However, the company requires that the architecture's components can (a) be composed from other components to define logical hierarchies; (b) explicate service level requirements; and (c) replicate themselves to scale if necessary. Instead of creating a new ADL from scratch, the company decides to extend an ADL already supporting composed components. They analyze the necessary changes in the ADL's syntax and semantics and determine two extension requirements. **R1**: The syntax supports specifying and storing service level information for each component. Its semantics ensures that the service level for each composed component is at least as good as the sum of service levels of its subcomponents. **R2**: The syntax supports specifying and storing replication conditions per component. Its semantics includes this information to control component replication at runtime.

The base ADL that the company's software engineers will extend is domain-agnostic and features only the core concepts depicted in black in Fig. 1: In this metamodel, a component type has a name and an arbitrary number of incoming and outgoing ports, which define the interface of a component. Furthermore, it can declare subcomponents that have a name and a component type. Each port has a data type and a name. Component types define an arbitrary number of connectors to connect their subcomponents.

The base ADL defines its static semantics (well-formedness) by a set of individual rules and translational dynamic semantics (behavior) via code generation. The static semantics include that (a) at least one port of each component type's subcomponents is connected by at least one connector; (b) connected subcomponents (*i.e.,* sources and targets of connectors) are actually declared in the same component type; and (c) that the types of connected ports are compatible. The dynamic semantics of an ADL govern how messages are passed between components. This ADL employs event-driven message passing in which components start to compute whenever at least one message has arrived. The dynamic semantics are realized by translating components to Java classes implementing this behavior.

The extensions to the metamodel are annotated and highlighted in Fig. 1: Relative to this metamodel, **R1** is translated into an extension of the syntax and semantics for component types. The syntactic extension is realized as the new property `serviceLevel` of `ComponentType` and the semantic extension as a

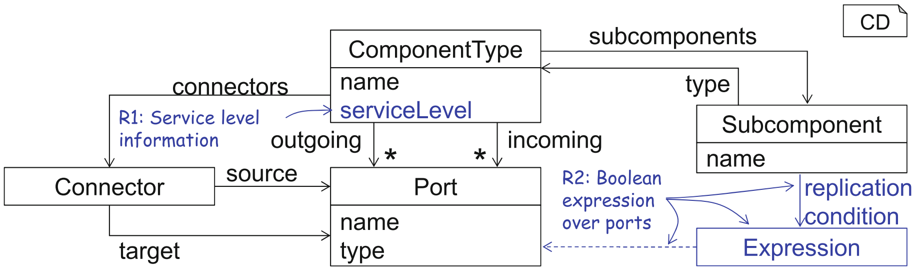

Fig. 1. Metamodel of a C&C ADL with extensions for R1 and R2.

new well-formedness rule. For **R2**, the syntax of subcomponents is extended by a replication expression. The company's engineers reuse a variant of OCL [11] as `Expression` language to enable reasoning over object structures. In conditions, they interpret names as references to ports (*e.g.,* a valid replication condition for a port `users` receiving lists of user data could be "`users.size() > 1`").

To ensure well-formedness of extended models, the engineers add new rules to extend the ADL's static semantics. These include

- service levels are positive numbers;
- the service level of a composed component is at least as high as the sum of service level of its subcomponents;
- the replication conditions respect the types of ports (*i.e.,* it ensures that its equations are type-compatible with the referenced ports); and
- the replication conditions evaluate to Boolean.

Realizing the intended replication behavior requires adding subcomponents at runtime when the conditions are fulfilled. However, in the base ADL's realization of its dynamic semantics, each component type is represented by a Java class yielding a single, fixed attribute for each subcomponent. To achieve flexible replication, the Java classes realizing component types should yield a set of subcomponents for each component type among their subcomponents instead. Moreover, they should feature a new method that checks the replication conditions whenever a message arrives and passes messages to new subcomponents as required. Hence, the company's software engineers extend the ADL's code generator accordingly. This small extension – a few properties, rules, and code generator adjustments – enables the company's engineers to customize the base ADL to their requirements and prevents creating a new ADL from scratch.

3 Preliminaries

MontiArc [17] is a component and connector ADL built on top of the MontiCore language workbench [20]. This enables leveraging MontiCore's powerful software language engineering capabilities, such as language composition and extension [13].

MontiCore employs context-free grammars (CFGs) for the integrated definition of concrete syntax and abstract syntax [20] of modeling languages. These CFGs describe which models are principally possible. Validating static semantics constraints not expressible with CFGs requires additional mechanisms. For such checks, MontiCore features a compositional *context condition* (CoCo) framework [32], where CoCos are well-formedness rules formulated in Java. Code generators implement the modeling languages' dynamic semantics. From a language's CFG, MontiCore generates the corresponding abstract syntax tree (AST) classes and infrastructure to parse textual models [10] into AST instances. The AST instances store the content of models, such as their elements and their relations to each other free from concrete syntax keywords. From each grammar, MontiCore automatically produces a model processing infrastructure that enables to parse models to operate on AST instances, for example, to apply model transformations or CoCo checks. For realization of dynamic semantics, MontiCore further features a template-based model-to-text code generation framework [29], which supports translating AST instances into arbitrary target representations. Moreover, MontiCore supports compositional language integration [1] via inheritance, embedding, and aggregation [13]. Language inheritance allows sublanguages to extend and override productions of its superlanguage. From this, MontiCore produces refined AST classes that inherit from the AST classes of the overridden production. Language embedding is realized by declaring *external* productions in a host grammar, which are abstract in the sense that they cannot be instantiated. To this effect, MontiCore's language configuration models define how productions from other languages are mapped to the external productions of the host grammar. Using this, MontiCore combines the individual parsers accordingly and produces integrated ASTs. Language aggregation enables to relate artifacts of different languages that are specified in separate artifacts.

MontiArc [12,17] is a C&C ADL and modeling infrastructure for the development of distributed systems. It is designed to provide the benefits of a comprehensible core architectural style that can be extended as necessary using the powerful language composition features of MontiCore [14]. Consequently, MontiArc provides a small core of language features that are easy to learn yet powerful enough to model complex software architectures. The language's infrastructure comprises code generators translating models into arbitrary GPL realizations. MontiArc is intentionally designed to be light-weight to keep the language easy to learn and flexibly adaptable. Thus it aims at providing only the most important modeling elements of C&C ADLs and focuses on language and tool chain extensibility. The provided elements are exactly the fundamental elements of architectural descriptions [23]: components, connectors, and configurations. Components are the units of computation in an architectural model and yield well-defined interfaces. Connectors connect the interfaces of components to realize component communication. A configuration is a graph of components and connectors that describes component composition. Following this principle, MontiArc facilitates modeling C&C software architectures with hierarchically structured, interconnected components. The interface of a component is defined

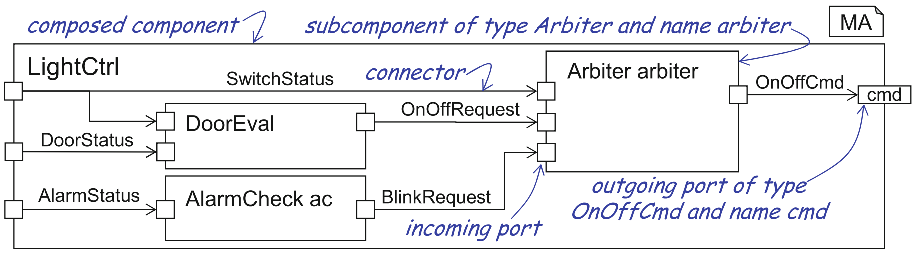

Fig. 2. MontiArc architecture for a light control system with three subcomponents.

by a set of unidirectional, named, and typed ports. Components receive messages via their incoming ports and emit messages via their outgoing ports. Unidirectional connectors connect exactly one source port to one or more target ports. To facilitate component reuse, MontiArc distinguishes between component types and component instances. A component type (denoted "component" in the following) defines the interface of its instances by a set of ports and may comprise component instances ("subcomponents") and connectors defining a configuration. If a component contains subcomponents, it is called composed. Otherwise it is called atomic. Atomic components perform the actual computations of a system. The behavior of a composed component is completely derived from the composition of the behaviors of its subcomponents according to its configuration. The behavior of atomic component has to be implemented by hand, *i.e.,* by providing GPL code implementations. MontiArc provides further language features, such as generic type parameters, configuration parameters, and syntactic sugar for automatically connecting all ports of the same type. A complete description of the MontiArc ADL is available in [12]. Figure 2, for instance, depicts the graphical representation of the component type `LightCtrl`. Listing 1.1 shows its corresponding textual definition. The component consists of four ports (l. 2), three subcomponents (ll. 4–6), and seven connectors (ll. 8–13). Connectors are unidirectional and connect one sending port with one or more receiving ports of compatible data types. The incoming port `switchStatus` of component `LightCtrl`, for instance, is connected to the same-named and same-typed incoming ports of the subcomponents `arbiter` and `doorEval` (l. 9).

```
 1 component LightCtrl {
 2   port in SwitchStatus, in AlarmStatus, in DoorStatus, out OnOffCmd cmd;   MA
 3
 4   component AlarmCheck ac;
 5   component DoorEval;
 6   component Arbiter arbiter;
 7
 8   connect arbiter.onOffCmd -> cmd;
 9   connect switchStatus -> arbiter.switchStatus, doorEval.switchStatus;
10   connect doorStatus -> doorEval.doorStatus;
11   connect alarmStatus -> alarmCheck.alarmStatus;
12   connect alarmCheck.blinkRequest -> arbiter.blinkRequest;
13   connect doorEval.onOffRequest -> arbiter.onOffRequest;
14 }
```

Listing 1.1. Component type `LightControl` with one configuration parameter. It consists of three incoming ports, one outgoing port, three subcomponents among which one is of an inner component type, and seven connectors (*cf.* [12]).

As MontiArc is realized as a MontiCore language, it employs the parsers and AST classes generated by MontiCore from its grammar to translate textual architecture models into AST instances. Using the AST, it applies the handcrafted workflows and model transformations as registered. Based on the transformed ASTs, it creates a symbol table infrastructure, which enables to resolve model (parts) across different models. Ultimately, MontiArc invokes its code generator to translate the (possibly transformed) ASTs into GPL artifacts that depend on classes of a runtime environment (RTE). A RTE consists of GPL artifacts of the same target language as the generated code and supports execution of generated artifacts (for instance, by realizing scheduling or message passing). The RTE's artifacts are independent of the generator's input models and are thus the same for any generated output. We treat the RTE as part of the generated code that remains static, independent of the generator's input models. MontiArc does not provide to define models in graphical syntax, but creating this can be achieved via translation to, e.g., EMF [28].

4 MontiArc Extension Method

Reuse is one of the prime enablers for efficient engineering. The language workbench MontiCore supports defining reusable languages that can be extended or combined to new languages [14]. We combine its language composition mechanisms with well-defined MontiArc extension points to extend the MontiArc ADL and its infrastructure. This enables customizing the ADL to requirements of specific domains and adding further language processing steps, while most infrastructure parts can be reused with the adjusted language directly. This section presents an integrated method to extend MontiArc that comprises structured activities to extend the ADL, model processing, and code generation. Extending the derived languages follows the same pattern. The method does not support creating a completely different ADL and infrastructure, as the result might not be applicable to the extension method anymore. For example, eliminating the component production by overriding might prevent further customization.

4.1 Extending the Syntax of MontiArc

The first step towards extending MontiArc's syntax is to analyze the intended extension's purpose: If the extension should change structural language elements (*e.g.,* components, ports, connectors), it requires inheriting from MontiArc to enable adding or refining language elements. To add a new component behavior modeling language, it requires embedding that language only. If the changes to MontiArc's syntax should enable adjusted model processing only, MontiArc's many places for stereotypes require even less extension effort. The related activities are illustrated in Fig. 3.

For introducing new modeling elements or refining existing ones, extension by inheritance starts with analyzing which MontiArc productions will be affected. For instance, introducing service level properties to components would require refining the production responsible for components. With MontiCore languages, refinement is realized via grammar inheritance. To make the new language elements accessible for well-formedness checking, language composition, or other processing, the relevant information must be added to the corresponding symbols also. The activities required to extend the symbol table are depicted in Fig. 4. For embedding of modeling languages to describe component behavior, MontiArc relies on MontiCore's language embedding capabilities. This includes that MontiArc provides an external grammar production for the embedding of

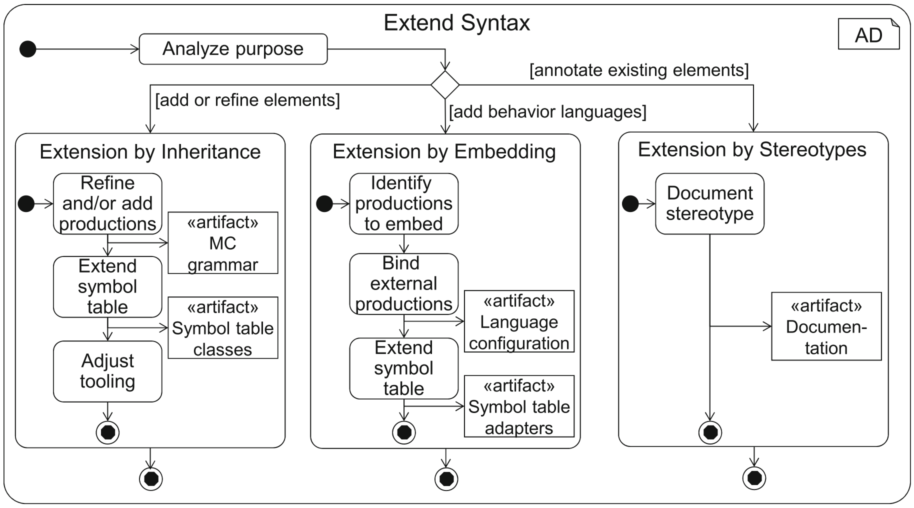

Fig. 3. Extending MontiArc's syntax: new language elements are introduced via inheritance, embedding, or as stereotypes.

grammar productions of embedded languages. First, the productions of the languages to be embedded must be identified. Afterwards, the mapping between the external production and the productions to be embedded is established. The mapping is defined in MontiArc's language configuration. During model processing, MontiCore then combines the parsers generated from the individual languages' grammars according to this mapping. This enables parsing components with embedded behavior models. However, these embedded models are usually unaware of their new operation context: for instance, embedded automata might expect to read inputs from variables. To interpret inputs and outputs of embedded models as references to ports, adapters between their symbols realize proper interpretation. Extension with new stereotypes amounts to providing proper documentation of the new stereotypes and their possible values. Please note that we support stereotypes only for minor and ad-hoc extensions. We advise to use metamodel extension, via inheritance or embedding, instead.

Extending MontiArc's ADL is coupled to extending its symbol table and introducing or refining productions as well as embedding behavior languages might require symbol table extension. After analyzing the cause for symbol table extension, one of the following activities is to be performed.

1. Type language adaptation: MontiArc supports using arbitrary type languages.
2. Reflect behavior language embedding: if the modeling elements of an embedded behavior language are relevant to the symbol table, *e.g.,* for checking inter-language well-formedness, these must be integrated.

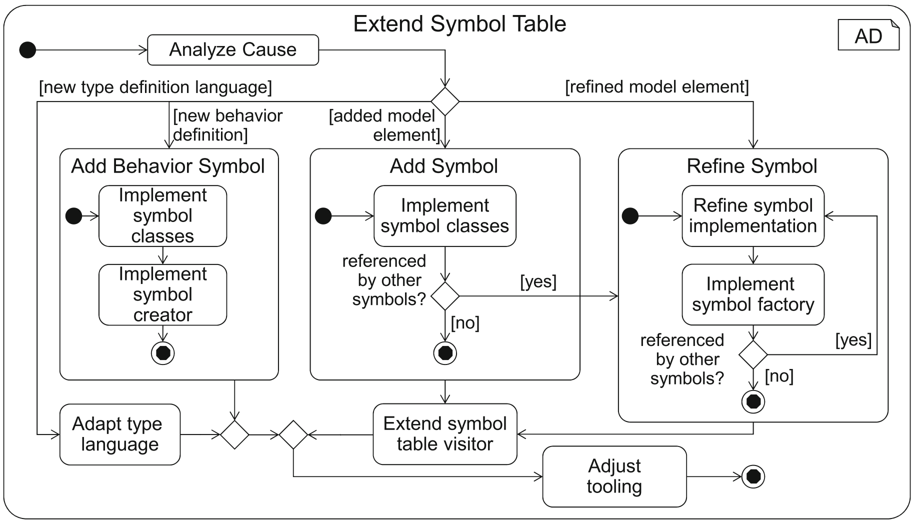

Fig. 4. Activities of extending MontiArc's symbol table.

3. New symbols: if language extension produced completely new modeling elements, such as the replication conditions described in Sect. 2, which require symbols as well, these must be added also.
4. Entry refinement: where modeling elements of MontiArc have been refined, for instance by adding a service level to components, this must be reflected at the corresponding symbols also.
5. Tooling adjustment: depending on the symbol table extension performed, MontiArc's tooling has to be extended accordingly.

Integrating a new type language into MontiArc requires the aggregation of both languages. This enables using the types defined in this language for MontiArc's ports and configuration parameters. Language aggregation is detailed in [32] and comprises the following activities: (1) Adapt symbols of the new type language to MontiArc's type symbol; (2) Create and register *qualifiers* [32] via subclassing. Qualifiers enable relating unqualified names (*e.g.,* `String`) to qualified names (*e.g.,* `java.lang.String`); and (3) Checking type properties requires loading the referenced model. To this end, MontiArc employs MontiCore's symbol resolving [32], which requires creating and registering *resolvers* via subclassing. These resolvers load symbols for qualified names. MontiArc's symbol table yields a dedicated extension point for convenient integration of embedded behavior languages' symbols. This requires creating a proper symbol kind for the behavior language. For example, when embedding automata, such a symbol might comprise information about data sources and data sinks the automaton models operate on. Additionally, qualifiers and resolvers must be provided. Where extension raises the need for integrating completely new symbols, these must be created and registered accordingly. After the entry classes have been created, MontiArc's symbol table visitor, which takes care of translating AST instances into symbols, must be extended via subclassing. In case the new model element is referenced by another element, the symbol representing the other element has to be refined accordingly. Refining a symbol entails refining its implementation via subclassing. The subclasses store the new information (*e.g.,* the service level) and must be accompanied by a registered qualifier, resolver, and symbol creator as explained above. Moreover, the symbol table visitor must be extended by subclassing to translate the refined AST properties into the corresponding symbol. Ultimately, the tooling has to be adjusted to use the refined factories, the extended symbol table visitor, and the created support classes (qualifier, resolver, *etc.*). For this, MontiArc employs dependency injection via the guice [31] framework. MontiArc contains a central registry module which can be overridden to change which implementations are bound to which MontiArc interface. This, for instance, enables to bind a subclass of MontiArc's symbol table visitor to the related interface.

4.2 Extend Model Processing

While extending MontiArc's syntax and symbol table enables introducing new model properties, refining existing ones, and adding stereotypes to models, using these modified elements requires adjusting MontiArc's model processing infrastructure. This may include extension with new workflows, model analyses, or model-to-model transformations as presented in Fig. 5. It does, however, not cover code generator extension, which is described in Sect. 4.3.

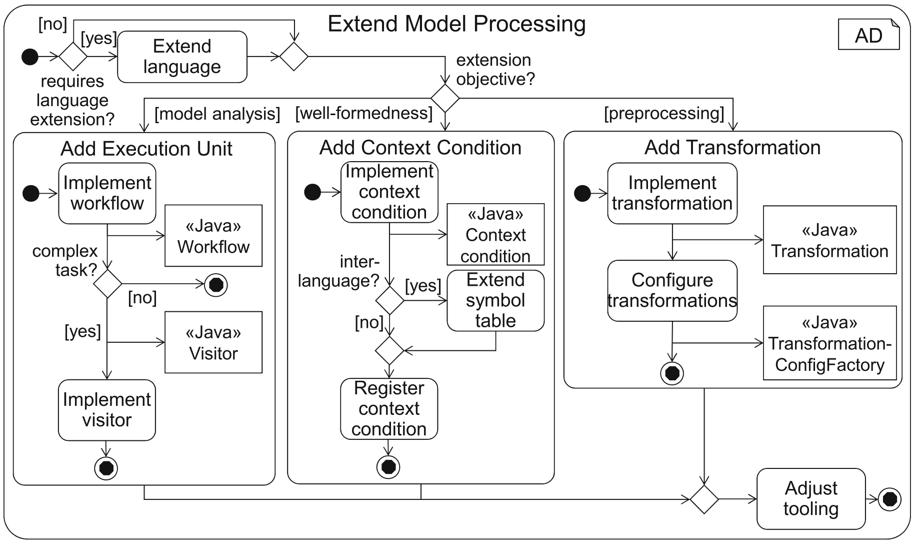

Fig. 5. Activities involved in extending MontiArc's model processing infrastructure.

Every extension of MontiArc's model processing infrastructure may start with extending its ADL as discussed in Sect. 4.1. If MontiArc should be extended with a new model analysis, a new MontiCore execution unit has to be added to its infrastructure. An execution unit is a wrapper for MontiCore workflows and creating a new execution unit requires registering a subclass of MontiCore's `DSLWorkflow` [19]. These workflows are executable units that perform calculations on the abstract syntax of a model using a visitor pattern variant. The static semantics of MontiArc are realized by a set of context conditions that ensure model well-formedness. If extension requires adding additional conditions, these can be implemented as subclasses of MontiCore's `ContextCondition` [32]. If the new condition relates models of different languages with another, it might be necessary to add adapters to the symbol table as presented in Sect. 4.1. Ultimately, the new context conditions must be registered by creating and binding a subclass of MontiArc's `ContextConditionCreator` using guice. If MontiArc is extended with a new model-to-model transformation, an artifact that executes the transformation has to be created. This artifact must implement one of the transformation interfaces for different abstract syntax elements of MontiArc. Configuration requires subclassing `TransformationConfigurationFactory` and binding it via guice.

4.3 Extend Code Generator

Syntactic extensions often aim at tailoring the language's dynamic semantics, *i.e.,* modified should change the behavior of the model (or generated code). Sometimes, the behavior aimed at can be reproduced by existing modeling elements. In this case, a transformation should be implemented as explained in Sect. 4.2. This section covers the case where semantics preserving transformations are neither possible nor desirable. For MontiArc, this entails adjusting its code generator to alter production of artifacts realizing the modified behavior.

Generally, MontiArc's code generation framework comprises a component-invariant run-time environment (RTE) and templates that produce component-specific artifacts. The RTE specifies properties invariant to individual component models, such as scheduling or message passing. The templates translate the abstract syntax of components to GPL artifacts interfacing the RTE. Hence, MontiArc provides templates for all abstract syntax concerns (*e.g.,* ports, connectors, subcomponents, *etc.*), which are registered at its central generator configuration. Creating a generator configuration consists of binding hook points with FreeMarker templates [29], which should be included (executed) at the hook point's location. By default, a unique start template is used for setting these configuration parameters for components. Figure 6 depicts an overview of the necessary activities for extending the code generator infrastructure after the syntactic extensions, if any, have been performed. If the existing RTE is insufficient for representing the new concepts (*e.g.,* scheduling and message passing remain unaffected), the generated API has to be adjusted, before the template parts can be adjusted.

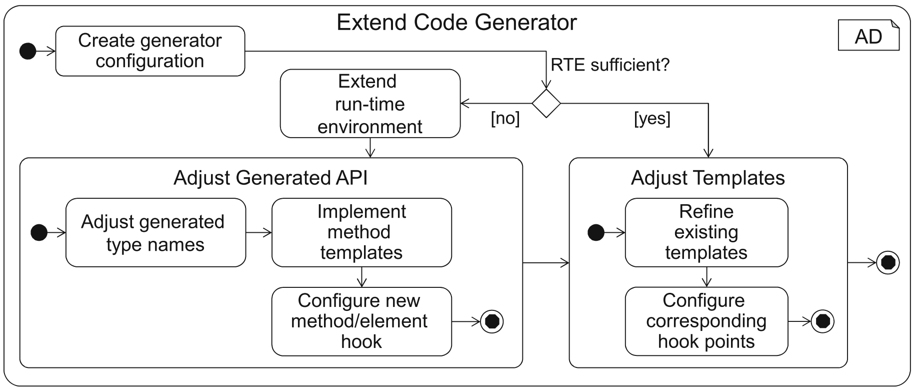
Extend Code Generator
AD
Create generator configuration
RTE sufficient?
Extend run-time environment
[no]
[yes]
Adjust Generated API
Adjust generated type names
Implement method templates
Configure new method/element hook
Adjust Templates
Refine existing templates
Configure corresponding hook points

Fig. 6. Code generator extension depends on reusability of the RTE.

Extending the RTE typically consists of (a) subtyping existing classes to refine commonly used method implementations; (b) adding additional methods to interfaces reflecting additional concepts captured by the new syntax; and (c) introducing new interfaces and classes representing new elements that cannot be captured by already existing classes and interfaces of the RTE. Extensions to the RTE entail adjusting the generated API. First, generated type names have to be adjusted with regard to referencing the newly created subclasses instead of the superclasses referenced before. Afterwards, method templates for the new methods added to the interfaces have to be implemented. This typically affects – but is not limited to – the generated code for components.

Ultimately, the generator's hook points have to be configured for new templates created for the new classes and interfaces added to the RTE in step (c) as well as for the new method templates. After incorporating possible RTE extensions, existing generated methods have to be adjusted to reflect the intended meaning of the changes performed on the ADL. To this effect, extension templates have to be implemented first. These templates are used for injecting code into preexisting methods generated by the former templates. Afterwards, the created templates are added to the method hooks in the generator configuration.

Where existing templates must be refined only, for instance to include translating new modeling elements, extension must provide new templates that are registered for hook points of related abstract syntax concerns. The lack of template inheritance in the underlying template engine causes this effort and ongoing work on code generator reuse might alleviate this [9]. However, these new templates can reuse existing templates in a white-box fashion and, thus, contribute new translation as appropriate.

5 Case Studies

MontiArc's extension method has successfully been applied to extend it for different domains. This section presents selected case studies.

MSA

```
1  component CashDeskLine {
2      port out PaymentRequest; // to Bank
3      component CashDeskUI ui { port out Sale; }
4      component CardReader reader { port out CardHolderData; }
5      component CashDesk cashDesk {
6          port in CardHolderData, in Sale, port out PaymentRequest;
7          trustlevel +1;
8          accesscontrol on;
9      }
10     identity weak ui -> cashDesk;
11     connect ui.sale -> cashDesk.sale;
12     connect encrypted reader.cardHolderData -> cashDesk.cardHolderData;
13     connect encrypted cashDesk.paymentRequest -> paymentRequest;
14 }
```

Listing 1.2. MontiSecArc architecture cash desk line found in supermarkets.

MontiSecArc (MSA) is an extension of MontiArc extended to the description of security architectures that enable the analysis of security flaws [18]. As such, it introduces modeling elements to specify security properties, such as trust levels, encrypted connectors, and identity links. To integrate these elements, which entail changes to abstract syntax, static semantics, and dynamic semantics, the MSA grammar inherits from MontiArc's grammar and introduces rules for the new concepts. It does not introduce new symbols, but refines the symbols for modified modeling elements to reflect the new properties and adds few context conditions related to these properties only. To produce proper Java artifacts, MSE also adjusts existing code generation by incorporating encryption into the RTE's message passing and overrides several templates where necessary. Listing 1.2 illustrates selected MSA features, such as the `trustlevel` (l. 7), which describes that the component provides protection against adversaries. Moreover, the component performs `accesscontrol` for all incoming ports (l. 8). The `weak identity` link from `ui` to `cashDesk` (l. 10) ensures that requests to the `sale` port are authenticated by a user logged in at the `ui` and cannot be spoofed by an adversary. Finally, two connectors are `encrypted` (ll. 12–13) to prevent adversaries from reading and modifying messages on these connections.

MAA

```
component BumpControl[int min = 1] {
  port in  Integer distance, in  Timer signal,
       out TimerCmd timer,   out Motor right, out Motor left;

  behavior automaton {
    state idle, drive, back, turn;
    initial idle                     / {right=Motor.STOP, left=Motor.STOP};
    idle  -> drive [distance < min]   / {right=FWD, left=FWD};
    drive -> back  [distance < 5*min] / {right=BWD, left=BWD, DOUBLE};
    // additional transitions
  }
}
```

Listing 1.3. MontiArcAutomaton architecture with embedded behavior model.

MontiArcAutomaton (MAA) is a framework for architecture modeling that extends the MontiArc infrastructure focusing on flexible embedding of component behavior languages and compositional code generation [27]. It extends MontiArc via grammar inheritance and embedding of various component behavior modeling languages. Consequently, it refines symbols of modified modeling elements and adds behavior symbols for the embedded languages. It also reuses all but one context condition of MontiArc and adds several context conditions for the new modeling elements. Moreover, it adds transformations to support various modeling shortcuts. MAA severely extends MontiArc's code generation framework to greatly facilitate behavior language embedding and features code generators producing Java and Python artifacts. As it does not extend MontiArc's scheduling and message passing, its Java generator reuses MontiArc's RTE without modification. For the Python generator, a new, compatible, RTE was devised. For the former, it provides extension templates only, for the latter it replaces all templates. The component `BumpControl` depicted in Listing 1.3 illustrates two of MAA's features: it introduces default parameters to component types (l. 1) and embeds a stand-alone language for I/O$^{\omega}$ automata to describe component behavior (ll. 5–10). To this effect, it binds the external productions inherited from MontiArc to automata productions for states (ll. 6–7) and transitions (ll. 8–10). To interpret names on transitions as component ports, which the automata are unaware of, MAA adds adapters between MontiArc's port symbols and automaton variable symbols as well as new context conditions (*e.g.,* to respect the direction of ports when reading). MontiArcAutomaton furthermore extends MontiArc's template for atomic components to delegate translation of embedded behavior language models to registered responsible code generators [27].

clArc is an infrastructure focusing on modeling cloud architectures. Its cloud ADL extends MontiArc with replication of ports and subcomponents to support load-balancing, port groups to enable message traceability, and service ports that describe requirements on the environment of the architecture. To introduce these elements clArc also extends MontiArc's grammar via inheritance, refines its symbols, and provides new context conditions on top of MontiArc's context conditions. Code generation produces event-driven Java implementations, hence clArc only refines MontiArc's templates for component hulls, ports, and subcomponent instantiation. Listing 1.4 illustrates clArc's new modeling elements: The port group `UserData` (l. 2), denotes that the following ports belong to a semantic unit for which calculations are performed only if all its ports have received at least one message. Replication, denoted by `[*]` (l. 3), enables instantiating a variable number of component instances at system runtime. The component `UserManagement` also requires a `service` (l. 5) to function properly, which is translated to a dependency to the eponymous Java interface.

```
1 component UserManagement {
2   port group UserData in User usr, in UpdateRequest req;            clarc
3   component UpdateStore store [*];
4   connect usr -> store.user;
5   connect req -> store.request;
6   service required clarc.db.NoSQL;
7 }
```

Listing 1.4. A clArc user management system with replicating subcomponents and port groups.

6 Discussion and Related Work

The systematic extension mechanism of MontiArc enables extending it to cover a great variety of architecture modeling concerns: Besides the variants presented above, it has been extended to support component behavior modeling with a variant of Java/P [16], architecture alignment checking [24], and delta modeling [15]. Following this method enables to reuse great parts of existing tooling, such as transformations, generators, and well-formedness rules. While the present extension mechanism is powerful, it requires MontiCore expertise to understand its language constituents and their interaction. As long as there is no general consensus on the shape of *language components* with interfaces for specific purposes, this remains necessary. Although many language workbenches [5] support extension mechanisms comparable to the mechanism presented here, none provides similar structural guidance to achieve such comprehensive tool chain integration.

Overall, science and industry have produced more than 120 ADLs [21]. These emerged from different domains and consequently focus on different challenges of architecture modeling. Although extensibility is “a key property of modeling notations” [22] most of these ADLs are so-called “first-generation ADLs” [22] that solely focus on technological challenges instead of domain-specific aspects or extensibility. Notable exceptions are the Architecture Analysis and Design Language (AADL) [6], π-ADL [26], and xADL [2]: AADL [6] features various modeling elements to describe hardware and software components of embedded systems. Similar to MontiArcAutomaton, AADL can also be extended with behavior modeling languages via sublanguages according to the behavior annex [7]. It does not support structured extension of its syntax or semantics aside from this. The xADL [25] also focuses on architecture extensibility and shares many features with MontiArc (*e.g.,* composed and atomic component types, instantiation, component behavior models). Moreover, it features modeling elements for product lines and variability not supported by MontiArc. Extension in xADL is syntactic and it neither supports non-invasive language aggregation, nor customization of its model processing infrastructure. The π-ADL [26] enables modeling structure and behavior of software architectures based on the π-calculus. It generally also supports to add behavior modeling capabilities as layers on top of its ADL. This ultimately produces a monolithic language composite whose individual languages are difficult to exchange. Moreover, it does not support structured extension of semantics composition of code generators for the individual behavior DSLs.

7 Conclusion

We have presented the MontiArc architecture modeling infrastructure, which leverages the results from software language engineering as realized with the MontiCore language workbench to enable extension of syntax and semantics. At its core, this infrastructure contains a light-weight ADL with extension points for behavior language embedding and integration of type languages. Its infrastructure comprises frameworks to integrate new well-formedness checks, workflows, model-to-model transformations, and code generation capabilities. We have presented an extension method that covers each of these aspects and enables customizing MontiArc to domain-specific requirements. This method alleviates the need for developing a specific ADLs from scratch, which we have illustrated with examples of MontiArc variants for three different domains, and greatly facilitates employing ADLs in different domains. This ultimately fosters their successful application in real-world scenarios.

References

1. Clark, T., Brand, M., Combemale, B., Rumpe, B.: Conceptual model of the globalization for domain-specific languages. In: Cheng, B.H.C., Combemale, B., France, R.B., Jézéquel, J.-M., Rumpe, B. (eds.) Globalizing Domain-Specific Languages. LNCS, vol. 9400, pp. 7–20. Springer, Cham (2015). doi:10.1007/978-3-319-26172-0_2
2. Dashofy, E.M., der Hoek, A.V., Taylor, R.N.: A highly-extensible, xml-based architecture description language. In: WICSA 2001. Proceedings of the Working IEEE/IFIP Conference on Software Architecture, p. 103. IEEE Computer Society, Washington, DC (2001)
3. Debruyne, V., Simonot-Lion, F., Trinquet, Y.: EAST-ADL — an architecture description language. In: Dissaux, P., Filali-Amine, M., Michel, P., Vernadat, F. (eds.) Architecture Description Languages. ITIFIP, vol. 176, pp. 181–195. Springer, Boston, MA (2005). doi:10.1007/0-387-24590-1_12
4. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, LDTA 2012, NY, USA. ACM, New York (2012)
5. Erdweg, S., et al.: The state of the art in language workbenches. In: Erwig, M., Paige, R.F., Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 197–217. Springer, Cham (2013). doi:10.1007/978-3-319-02654-1_11
6. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, Boston (2012)
7. Franca, R.B., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex-experiments and roadmap. In: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 377–382. IEEE Computer Society, Washington, DC (2007)
8. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering, FOSE 2007, pp. 37–54 (2007)
9. Greifenberg, T., Müller, K., Roth, A., Rumpe, B., Schulze, C., Wortmann, A.: Modeling variability in template-based code generators for product line engineering. In: Modellierung (2016)
10. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Textbased modeling. In: 4th International Workshop on Software Language Engineering, Informatik-Bericht, Nashville, vol. 4. Johannes-Gutenberg-Universität Mainz (2007)
11. Group, O.M: OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10–05-03) (2010)

12. Haber, A.: MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems. No. 24 in Aachener Informatik-Berichte, Software Engineering, Shaker Verlag (2016)
13. 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. Scitepress, Angers, France (2015)
14. Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., Wortmann, A.: Composition of heterogeneous modeling languages. In: Desfray, P., Filipe, J., Hammoudi, S., Pires, L.F. (eds.) MODELSWARD 2015. CCIS, vol. 580, pp. 45–66. Springer, Cham (2015). doi:10.1007/978-3-319-27869-8_3
15. Haber, A., Rendel, H., Rumpe, B., Schaefer, I.: Delta modeling for software architectures. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII, pp. 1–10. fortiss GmbH (2011)
16. Haber, A., Ringert, J.O., Rumpe, B.: Towards architectural programming of embedded systems. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI. Informatik-Bericht, vol. 2010-01, pp. 13–22. fortiss GmbH, Germany (2010)
17. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - architectural modeling of interactive distributed and cyber-physical systems. Technical report AIB-2012-03, RWTH Aachen University (2012)
18. Hermerschmidt, L., Hölldobler, K., Rumpe, B., Wortmann, A.: Generating domain-specific transformation languages for component & connector architecture descriptions. In: 2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp) (2015)
19. Krahn, H.: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering. No. 1 in Aachener Informatik-Berichte, Software Engineering, Shaker Verlag (2010)
20. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(5), 353–372 (2010)
21. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6), 869–891 (2013)
22. Medvidovic, N., Dashofy, E.M., Taylor, R.N.: Moving architectural description from under the technology lamppost. Inf. Softw. Technol. **49**(1), 12–31 (2007)
23. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**, 70–93 (2000)
24. Mir Seyed Nazari, P.: Architektur Alignment von Java Systemen. Master's thesis, RWTH Aachen University (2011)
25. 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, pp. 22–26. IET (2004)
26. Oquendo, F.: π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Softw. Eng. Notes **29**(3), 1–14 (2004)
27. Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Language and code generator composition for model-driven engineering of robotics component & connector systems. J. Softw. Eng. Rob. (JOSER) **6**, 33–57 (2015)

28. Ringert, J.O., Rumpe, B., Wortmann, A.: From software architecture structure and behavior modeling to implementations of cyber-physical systems. In: Software Engineering Workshopband (SE 2013). LNI, vol. 215, pp. 155–170 (2013)
29. Schindler, M.: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. No. 11 in Aachener Informatik-Berichte, Software Engineering, Shaker Verlag (2012)
30. Schlegel, C., Steck, A., Lotz, A.: Model-driven software development in robotics: communication patterns as key for a robotics component model. In: Chugo, D., Yokota, S. (eds.) Introduction to Modern Robotics. iConcept Press (2011)
31. Vanbrabant, R.: Google Guice: Agile Lightweight Dependency Injection Framework. Apress, New York (2008)
32. Völkel, S.: Kompositionale Entwicklung domänenspezifischer Sprachen. No. 9 in Aachener Informatik-Berichte, Software Engineering, Shaker Verlag (2011)
33. Wile, D.S.: Supporting the DSL spectrum. Comput. Inf. Technol. **4**, 263–287 (2001)