Composition of Heterogeneous Modeling Languages

Arne Haber[1], Markus Look[1], Pedram Mir Seyed Nazari[1](✉), Antonio Navarro Perez[1], Bernhard Rumpe[1], Steven Völkel[2], and Andreas Wortmann[1]

[1] Software Engineering, RWTH Aachen University, Aachen, Germany
`nazari@se-rwth.de`
`http://www.se-rwth.de`
[2] Volkswagen Financial Services, Braunschweig, Germany

Abstract. Model-driven engineering aims at managing the complexity of large software systems by describing their various aspects through dedicated models. This approach requires to employ different modeling languages that are tailored to specific system aspects, yet can be interpreted together to form a coherent description of the total system. Traditionally, implementations of such integrated languages have been monolithic language projects with little modularization and reuse of language parts.

This paper presents a method for engineering reusable language components that can be efficiently combined on the syntax level. The method is based on the concepts of language aggregation, language embedding, and language inheritance. The result is the ability to efficiently develop project-specific combinations of modeling languages in an agile manner.

1 Introduction

Engineering of non-trivial software systems requires reducing the conceptual gap between problem domains and solution domains [1]. Model-driven engineering (MDE) aims at achieving this by raising the level of abstraction from programming of a complete system implementation to abstract modeling of domain and system aspects. In this way, models are raised to the level of primary development artifacts. Different aspects of complex software systems require different modeling languages to be expressed with. The UML [2], for instance, contains seven structure modeling languages, with class diagrams probably being the most famous, and also seven behavior modeling languages, such as statecharts and activity diagrams. Integration of modeling languages for a software project either requires composing the languages specifically for this project a priori, or designing the independent languages with composition in mind - but without prior assumptions of the actual composition. The former approach yields monolithic language aggregates that are hardly reusable for different projects.

We propose an approach to syntax-oriented black-box integration of grammar-based textual languages developed around the notions of language

P. Desfray et al. (Eds.): MODELSWARD 2015, CCIS 580, pp. 45–66, 2015.
DOI: 10.1007/978-3-319-27869-8_3

aggregation, language embedding, and language inheritance [3,4]. This approach addresses all aspects of syntax-oriented language integration, namely concrete syntax, abstract syntax, symbol tables, and context conditions. It is based on previous work on syntactic modeling language integration [5] and introduces new mechanisms to inter-language model validation. Different aspects of these new mechanisms were briefly introduced at the GEMOC workshop at MODELS 2013 [6] and at the MODELSWARD conference [7]. This contribution extends the concepts and provides an in depth discussion of their implementation with the language workbench MontiCore [8] in detail.

At first, we motivate the need for language integration in MDE on the example of a robotic system in Sect. 2. Afterwards, Sect. 3 explains the concepts for language integration, and Sect. 4 their support through a language integration framework. Section 5 discusses concepts related to our work. Section 6 concludes this contribution with an outlook on future work and a summary.

2 Motivation

To illustrate our approach we first motivate the concepts of language aggregation, language embedding, and language inheritance by the example of a robot that is described by various, heterogeneous models. The techniques employed in this example are described in detail in the following sections. Throughout the example, different needs for language integration arise that we categorize as follows:

– Language aggregation integrates different modeling languages by mutually relating their concepts such that their models can be interpreted together, yet remain independent.
– Language embedding denotes the composition of different modeling languages by embedding concepts of one language into declared extension points of another. Models of the new language thereby contain concepts of both languages.
– Language inheritance is the definition of new languages on the basis of existing languages through reuse and modification of existing language concepts.

Consider a robot exploring uncharted areas, identifying obstacles, and storing these in a map. Our aim is to specify this system by using models in a way detailed enough to generate a significant amount of its implementation automatically.

To this end, various system aspects, such as its overall architecture, the data it operates on, and its deployment onto a runtime infrastructure, need to be addressed individually by appropriate modeling languages, including architecture models, data models, and deployment models. Yet, many of those aspects are not independent, but mutually related. For instance, the components of a software architecture operate on inputs defined via data models and may employ component behavior models to describe output behavior. In addition, some aspects of the system are of a more general nature and apply to a wide

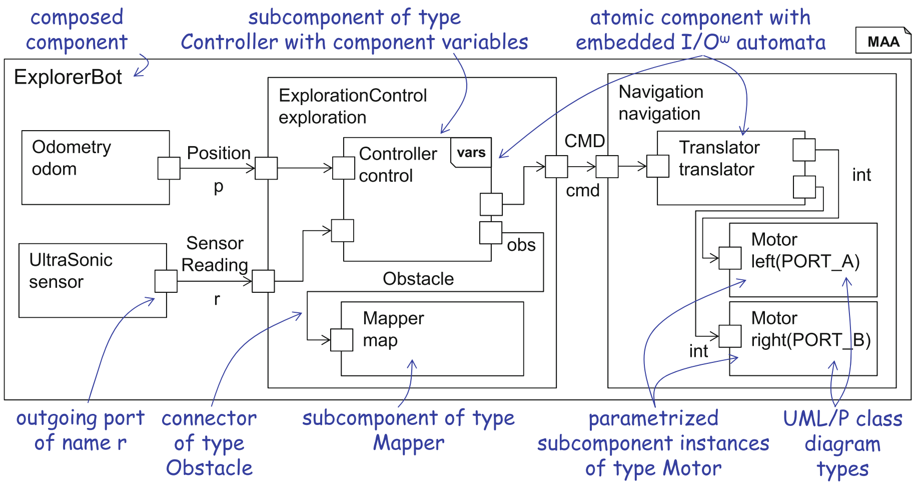

composed component
subcomponent of type Controller with component variables
atomic component with embedded I/Oω automata
MAA
ExplorerBot
ExplorationControl exploration
Navigation navigation
Odometry odom
Position
p
Controller control
vars
CMD
cmd
Translator translator
int
UltraSonic sensor
Sensor Reading
r
obs
Obstacle
Motor left(PORT_A)
Mapper map
Motor right(PORT_B)
int
outgoing port of name r
connector of type Obstacle
subcomponent of type Mapper
parametrized subcomponent instances of type Motor
UML/P class diagram types

Fig. 1. The MontiArcAutomaton software architecture of the exploration robot uses modeling elements inherited from MontiArc, references aggregated UML/P CDs for data types, and embeds I/O$^\omega$ automata to describe component behavior.

range of system kinds. Conversely, other aspects are specific to the application domain at hand. For instance, architectural distribution is not only relevant in robotics, but also in web systems, whereas aspects such as graceful degradation are more specific to robotics. It is desirable to separate the language concepts for general aspects from those for specific aspects in order to facilitate the modularization of languages into reusable language components. Besides, such reuse of existing general languages reduces the need for developers to learn new notations and concepts.

In the following, we model the software architecture and the domain model of our system using multiple modeling languages. In doing so, we stress the different needs of language integration arising from our scenario.

2.1 Software Architecture

Robotic systems typically consist of different components for specific tasks, such as sensing, navigation, motion, or planning. We model the overall software architecture of the robot with the component and connector architecture description language (ADL) [9] MontiArcAutomaton [10–12] that is derived from the ADL MontiArc [13] and includes *extensions* to model component behavior with application-specific modeling languages [10].

Figure 1 shows the graphical representation of the robot's software architecture model, formulated in terms of hierarchically composed components. Components realize the system's functionality by interacting through directed connectors over which they exchange messages. They comprise of an interface of typed, directed ports, configuration parameters, and type parameters, and of a body containing component behavior. Composed components contain a

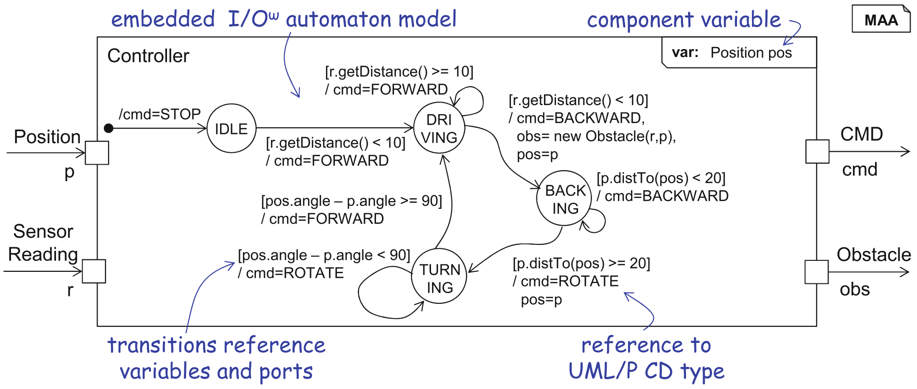


Fig. 2. Atomic component `ExplorationControl` with embedded I/O$^\omega$ automaton.

hierarchy of subcomponents and their behavior emerges from the behavior of these subcomponents. The behavior of atomic components is specified by source code artifacts. Connectors are at each end connected to typed ports that in sum represent a component's interface. While these elements are defined by MontiArc and reused in MontiArcAutomaton via language inheritance, MontiArcAutomaton also introduces new extensions, two of which are shown in the following example. First, atomic components can contain *embedded* behavior models of other languages instead of a reference to a implementation artifact. Second, atomic components may contain variables as well.

In our example, the software architecture model describes the logical software architecture of the exploration robot `ExplorerBot` (Fig. 1). The software architecture comprises of several atomic and composed components to provide sensor data (`Odometry`, `Ultrasonic`), control the robot (`Controller`), store obstacles on a map (`Mapper`), and propel the robot (`Navigation` with subcomponents `Translator` and two `Motors`). Components and connectors are inherited from MontiArc. To be reusable with multiple target platforms, the atomic components controller and translator employ behavior models. The extension points for behavior models are a feature of MontiArcAutomaton to describe the input-output behavior of atomic components. In this case, the behavior of both components is modeled with I/O$^\omega$ automata and embedded into MontiArcAutomaton. The component `Controller` is depicted in Fig. 2 that shows its embedded automaton. The automaton comprises of four states and eight transitions that react on messages received by the component's input ports, store values to variables, and emit messages on its output ports.

The language of I/O$^\omega$ automata was developed independently of MontiArcAutomaton and vice versa. Consequently, automata are unaware of component & connector concepts, such as components or ports while MontiArcAutomaton is unaware of the I/O$^\omega$ language. Nonetheless, automata reference to MontiArcAutomaton's ports and data types and efficient modeling requires that the well-formedness of such references is checked. Checking the validity of assignment `obs = new Obstacle(r,p)`, for instance, requires checking whether `obs` is an

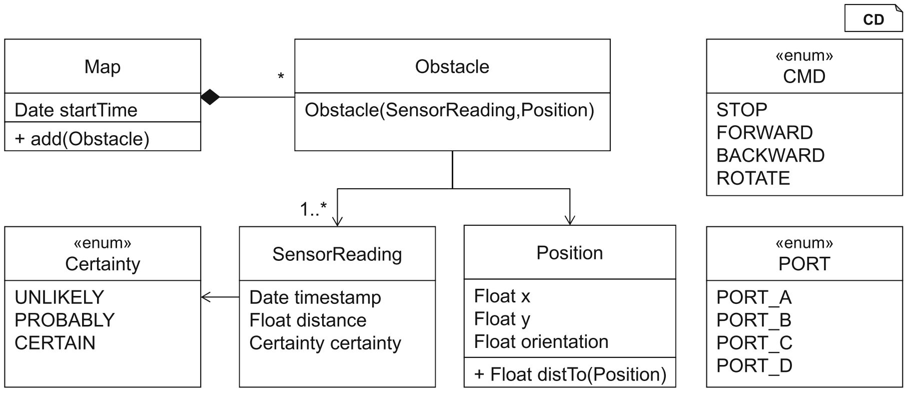


Fig. 3. Domain model `SensorData` for sensor measurements with data types `CMD` and `PORT` used by component type `Motor` and `Control`.

outgoing port or a variable, whether `obs` is of type `Obstacle`, and whether the data type `Obstacle` provides a constructor that accepts two arguments of types `SensorReading` and `Position`, respectively. This requires checking properties of three languages: the component & connector ADL, its behavior languages, and the language of its domain model. Before this is explained, the following section illustrates the domain model used by `Explorerbot`.

2.2 Domain and Data Modeling

Nearly all object-oriented systems operate in the context of a domain model. Such models describe the application's real world context in terms of classes and associations between classes. Their most prominent role is to serve as the basis for the application's fundamental data structures that are used for computation, communication, and persistence.

Class diagrams (CD) are the foundational modeling language in which domain models are described. We formulate them by means of a textual syntax defined by the UML/P [3,14,15], a variant of the UML focused on precise semantics and applicability to generative software engineering. Figure 3 shows a graphical representation of the domain model for our exploration robot example. It consists of classes representing the messages received by the system and the values they convey. A `Map` consists of a set of obstacles (class `Obstacle`), that may contain multiple `SensorReadings` for a specific `Position`. Each `SensorReadering` is associated with a `Certainty` to reflect the reading's quality. The domain model furthermore contains the two enumerations `CMD` and `PORT` used by components of type `Motor` and `Control`. The types `Obstacle`, `SensorReading`, `Position`, `CMD`, and `PORT` are referenced directly by the `ExplorerBot` software architecture. It is evident that integration concepts between these different and heterogeneous models are required. In the following section we define such concepts on the language level.

3 Language Integration Concepts

In [6] we already gave a first realization of the language integration concepts defined above and outlined their implementation for grammar-based languages in the language workbench MontiCore [8]. Here, we describe these concepts as well as their application in detail. In particular, we give a detailed description of the integration concepts, the integrated abstract syntax trees (AST), and of references between AST nodes. Please note that the corresponding parser integration mechanisms and the resulting challenges are already discussed in [8].

The following concepts use MontiCore's extended grammar format, which serves to systematically derive both the concrete as well as the abstract syntax of a language. Additionally, language processing infrastructure such as parsers and pretty-printers are derived. Described briefly, every production of a grammar corresponds to a generated AST node class of the same name. The nonterminals of the production become the set of attributes forming the signature of that class. In addition, grammars can define abstract productions and interface productions which can be extended, or implemented respectively. Thus, these can be used anywhere the extended/implemented production's nonterminals are used. Abstract productions differ from normal productions since they have to be extended by at least one normal production. Interface productions are basically the same without defining concrete syntax. An in-depth description of the MontiCore grammar format is given in [8].

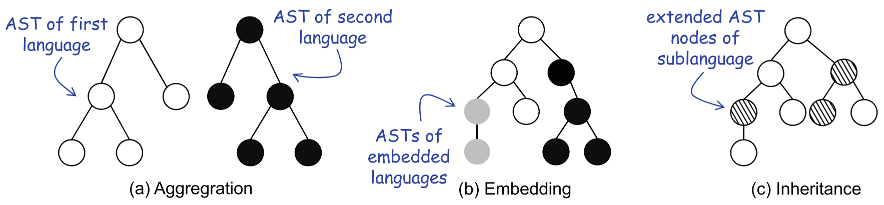
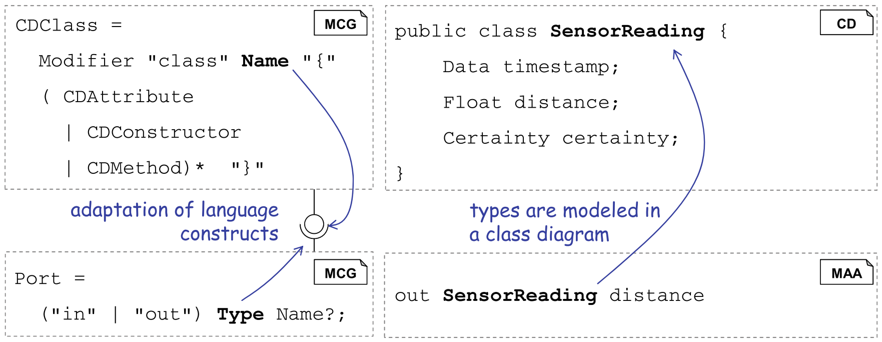


Fig. 4. The resulting ASTs for aggregation, embedding, and inheritance. Aggregation results in separate ASTs for each model. Embedding results in a single AST with subtrees embedded at the leaves of the host language. Inheritance results also in a single AST containing extended nodes of the sublanguage (cf. [6]).

3.1 Language Aggregation

Language aggregation integrates multiple languages into a so called *language family*, such that models of these languages are kept in seperate artifacts but interpreted together, enabling loose coupling but mutual referencing each others elements, as shown in Sect. 2.1. There, port declarations of the MontiArcAutomaton model reference type definitions given by the seperate class diagram.

Figure 5 shows how aggregation works on a conceptual level and how concrete aggregations can be defined within the MontiCore language workbench. The left half shows two grammars (MCG) while the right half shows a class diagram and part of a MontiArcAutomaton model that correspond to their respective


MCG
CD
adaptation of language constructs
types are modeled in a class diagram
MCG
MAA

```
CDClass =
  Modifier "class" Name "{"
  ( CDAttribute
    | CDConstructor
    | CDMethod)*  "}"
```

```
public class SensorReading {
    Data timestamp;
    Float distance;
    Certainty certainty;
}
```

```
Port =
  ("in" | "out") Type Name?;
```

```
out SensorReading distance
```

Fig. 5. The mechanism for aggregating languages. By adaptation between elements of two independent languages referencing between them is achieved. The right half of the figure shows concrete models of aggregated languages referencing each other.

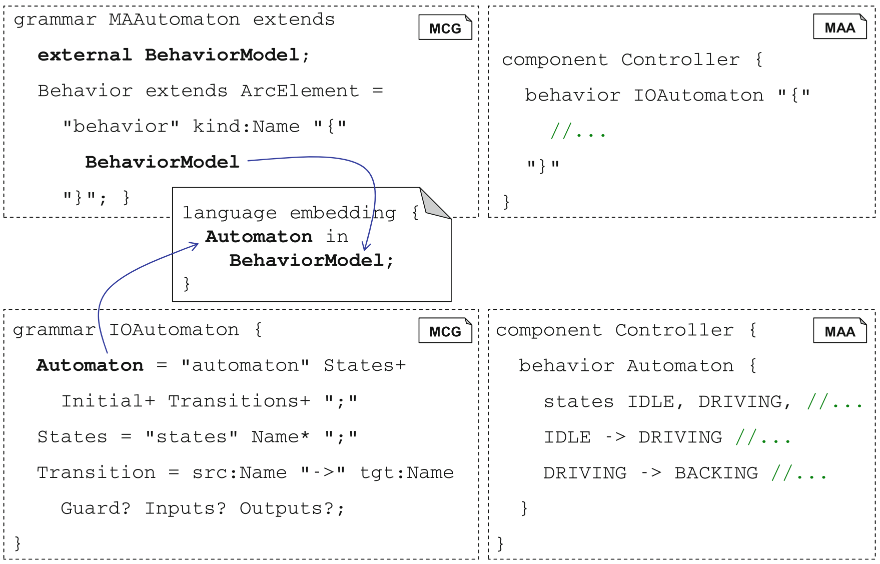
grammar on the left. The upper left part shows an excerpt of the grammar for UML/P CDs. In particular, it shows the production of the `CDClass` nonterminal which defines a class as consisting of multiple subelements, especially a name defininf a possible identifier. Curly brackets enclose the possible subelements. The upper right part contains an instance of the production in concrete syntax in which a `SensorReading` class is defined. The lower left part shows an excerpt of the MontiArcAutomaton grammar in which the production for a component `Port` declaration is defined. Both `Type` and `Name` define possible identifiers for types and instances names similar to the naming scheme used in Java. As explained earlier, the `Type` of a port is a name interpreted as a reference to a data type. The lower right part shows an instance of the production referencing a `SensorReading` type.

Via language aggregation CD and ADL models can be combined such that the type references in the ADL model are interpreted as references to class definitions in the CD. The technical realization of language aggregation works in two steps. First, every model of every language is parsed individually, resulting in an AST for each model as shown in Fig. 4. In our example, the type reference in the AST of the architecture model is represented by an AST node of type `Name` containing the name of the reference, whereas the class definition in the AST of the CD is represented by an AST node of type `CDClass`. Second, the references are related to each other by a symbol table. Conceptually, the symbol table manages a kind of link between AST nodes. Technically, the links are implemented by adapter classes to allow for flexible linking to AST nodes from other languages. The details of this are described in Sect. 4.

Language aggregation is adequate for modeling different aspects of a system, each of which can be understood on their own. Each aspect is then described by individual model documents in specialized modeling languages. Through aggregation, these models are related to each other without infringing a tight coupling between them. Thereby, models can be reused in different combinations and in a modular way.

3.2 Language Embedding

Language embedding combines languages such that they can be used in a single model. To this end, an embedding language incorporates elements from other languages at distinguished extension points. Even though this gives the impression of tight coupling, the individual languages are still developed independently and integrated in a black-box way. References between elements of embedding and embedded languages work similarly to language aggregation. Figure 2 shows an example of MontiArcAutomaton with embedded I/O$^\omega$ automata.



```
grammar MAAutomaton extends
  external BehaviorModel;
  Behavior extends ArcElement =
    "behavior" kind:Name "{"
      BehaviorModel
    "}"; }
```

```
component Controller {
  behavior IOAutomaton "{"
    //...
  "}"
}
```

```
language embedding {
  Automaton in
    BehaviorModel;
}
```

```
grammar IOAutomaton {
  Automaton = "automaton" States+
    Initial+ Transitions+ ";"
  States = "states" Name* ";"
  Transition = src:Name "->" tgt:Name
    Guard? Inputs? Outputs?;
}
```

```
component Controller {
  behavior Automaton {
    states IDLE, DRIVING, //...
    IDLE -> DRIVING //...
    DRIVING -> BACKING //...
  }
}
```

Fig. 6. The mechanism for embedding languages. By declaring an external nonterminal `BehaviorModel` and a separate mapping artifact, nonterminals of arbitrary languages, such as `Automaton` can be embedded. The right half of the figure shows a concrete model with the embedded element.

Figure 6 shows how the embedding is accomplished within the MontiCore language workbench. The upper left part again shows an excerpt of the MontiArcAutomaton grammar with the production of the nonterminal `Behavior`. The production describes a component behavior with a name followed by a `BehaviorModel` enclosed in curly brackets.

The `BehaviorModel` is defined as an external nonterminal. Such external productions act as the extensions points into which elements from other languages can be embedded. In fact, the language is incomplete as long as its external productions have not been bound to external language elements. Note that neither the grammar nor the external production contain any information about filling the external nonterminal, leaving the binding to a later stage. The lower left part shows an excerpt of the I/O$^\omega$ automaton grammar containing the definition of an `Automaton` nonterminal that comprises further nonterminals. Again, the

`IOAutomaton` grammar does omit any explicit reference to language embedding. This is specified in the language configuration model, shown in the middle part of Fig. 6, which maps `Automaton` to the external `BehaviorModel`.

It is also possible to embed several languages into a single external production by mapping several external nonterminals to it. After parsing, the resulting AST consists of different nodes of the different languages, as shown in Fig. 4. Nodes from embedded languages manifest as subtrees attached in place of the node representing the external production. Language embedding is especially useful when the language developer does not want to force the use of a specific language but allows choosing the sub-language later.

3.3 Language Inheritance

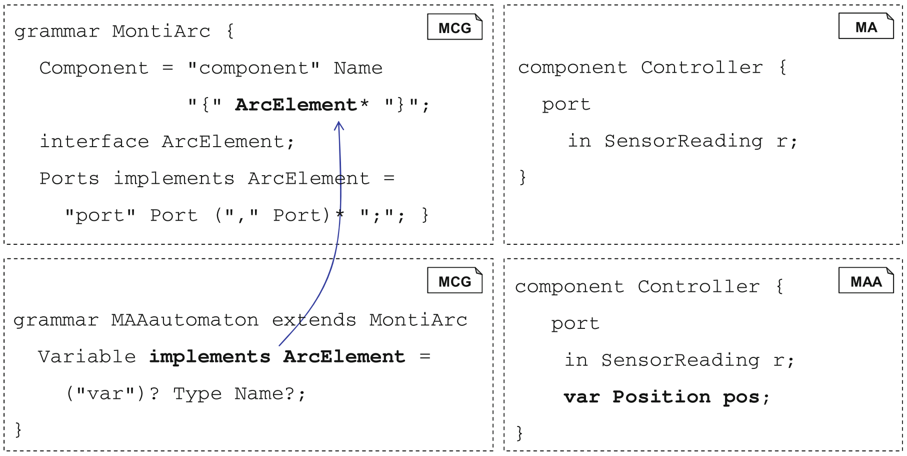
Language inheritance can be used to extend or refine an existing language. For this purpose, MontiCore allows to define new languages on the basis of existing languages by reusing, modifying and overriding their productions. The example in Sect. 2.1 illustrates how MontiArcAutomaton extends the MontiArc ADL and adds robotics-specific extensions.

Figure 7 illustrates how this extension is defined within MontiCore's grammar format. The upper left part shows a production (with details omitted) from the MontiArc grammar that specifies the nonterminal interface `ArcElement`, which all productions usable in a component body implement (cf. the production `Ports` that references the production `Port` introduced in Fig. 5). Each production that implements `ArcElement` thus can be used in a component's body.

The upper right part shows part of a model that conforms to the production shown in the upper left part, i.e., a component with name and a single port. The lower left part shows an excerpt of the textual MontiArcAutomaton grammar extending the MontiArc grammar. The name of the extended grammar is followed by the keyword `extends` and a reference to the extended grammar. The MontiArcAutomaton grammar introduces a new production `Variable` that also implements `ArcElement`, which is available due to inheritance. It is mandatory to keep all elements of a production that have been present in the parent production. Instead, it is also possible to leave some out, reorder them, add new elements in between, or even remove all elements. The lower right part shows a model element that corresponds to the production on the lower left.

Productions of extended languages (or "parent" language) are "virtually'" copied into the extending new language where they can be referenced from new productions. In addition, new productions can individually extend productions from the parent language and thereby inherit that production's interface. This means that the extending production can be used anywhere the nonterminal from the extended production is used. The resulting new AST nodes consequently implement the signatures of their parent counterparts.

The right part in Fig. 4 illustrates the structure of ASTs from inheriting languages. The generated parser for the sublanguage is able to parse text corresponding to the parent language as well as text corresponding to the sublanguage, and consequently creates an AST containing node types from both



```
grammar MontiArc {
  Component = "component" Name
              "{" ArcElement* "}";
  interface ArcElement;
  Ports implements ArcElement =
    "port" Port ("," Port)* ";"; }
```

```
component Controller {
  port
    in SensorReading r;
}
```

```
grammar MAAautomaton extends MontiArc
  Variable implements ArcElement =
    ("var")? Type Name?;
}
```

```
component Controller {
   port
    in SensorReading r;
    var Position pos;
}
```

Fig. 7. The mechanism for inheritance between languages. By declaring an extension, languages can inherit from each other and are able to override or implement productions. The right half of the figure shows a concrete model using the inheritance.

languages. Since parent grammars and nodes are referenced by names, name collisions can occur. To prevent this, the language designer may use full qualified names formed by the respective grammar's package, the grammar's name and the name of the production.

Language inheritance is particularly useful for reusing existing concepts of languages while extending them with new concepts. It is applicable when the inheriting language is conceptually similar to the parent language.

3.4 Context Conditions

Context conditions are well-formedness rules for languages inexpressible by their context-free grammars, e.g., restraining a class in a CD from having two members with the same name. These context conditions can impose restrictions between multiple models of the same or different languages. Thus, we distinguish four types of context conditions [3] that have to be integrated differently:

1. **Language-internal Intra-model Context Conditions** only consider a single model of a single language, e.g., to check whether a class of a CD contains two members with the same name.
2. **Language-internal Inter-model Context Conditions** emerge from references between models of the same language, e.g., to check whether a class can instantiate another with certain parameters.
3. **Cross-language Intra-model Context Conditions** consider relations emerging from the embedding of one language's model into the model of another language, e.g., to check whether the return type of an embedded SQL statement matches the return type of the embedding method declaration (cf. Fig. 7).

4. **Cross-language Inter-model Context Conditions** check conditions between models of different languages, e.g., whether the type of a component's port is declared within the imported CDs.

4 Language Integration Framework

In this section, we describe a framework to (1) implement *symbol table infrastructures* that relate model elements in composed languages in a black-box way and to (2) configure them for concrete language compositions based on the concepts introduced above.

Section 3 described how language embedding and language inheritance manifest in the ASTs of processed models that are instances of combined languages (cf. Fig. 4). Both mechanisms make information about embedded or inherited model elements directly available in the AST for further model processing, such as code generation. However, this is not the case for models of aggregated languages in which elements of one language reference elements of another language by name. Here, the AST nodes only contain the raw name of the referenced model element. The same holds for embedding and embedded languages that refer to each other through raw names as well. Consequently, additional infrastructure is necessary to translate raw name references into information about referenced model elements.

In the following, we describe an infrastructure named symbol table that (a) allows to acquire information from referenced models as well as (b) to transparently interpret elements of one language as elements of another. Compared to traditional symbol table techniques, our realization must be able to translate these different kinds of concepts between languages. For example, automata know about states and input signals, whereas Java knows nothing about these. To integrate these languages nonetheless, the concept of a state must be translated into Java in a meaningful way. For instance, states could be mapped to an enumeration or also to subclasses (as, e.g., in the state pattern [16]). To keep both languages, independent, we cannot define this translation in either language. Instead, we need to define it as a separate artifact during language integration. The ability to do this is one key feature of our symbol table framework.

4.1 Symbol Table Concepts

A *symbol table* is a data structure that is used to store and to resolve identifiers within a language [4]. An identifier, such as a name, is associated with further information from the corresponding language element. In this way detailed information may be gathered from the symbol table by resolving an element using its name. The result of a name search is a symbol. In the following we define the core concepts of a symbol table structure[1].

[1] Please note that part of the nomenclature introduced in [7] was changed to reflect the refined concepts more precisely.

Definition 1 (Symbol and Kind). A symbol, i.e., an entry in the symbol table, is a representation of essential information about a (named) model element. It has a specific kind and a well defined signature determined by the model element it denotes, e.g., variable, method, state.

Every symbol has a name (resp. simple name), e.g., `SensorReading`, which usually is unique within the scope (see below) the symbol is defined in. The qualified name (resp. full name) of a symbol additionally includes, among others, its package name. For example, `de.se.SensorReading` is the qualified name where `de.se` is the package name and `SensorReading` the simple name. Above all, the qualified name of a symbol is important to resolve a *referenced symbol* that is defined in another model. Generally, every symbol is defined exactly once, but may be referenced several times. The symbol definition contains the whole information about the symbol, whereas a symbol reference only contains information needed to resolve the definition.

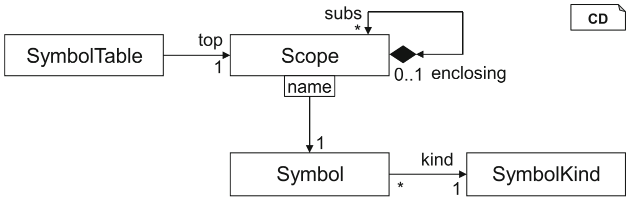


Fig. 8. Symbol table, hierarchical scopes and related symbols.

Definition 2 (Symbol Visibility). The visibility of a symbol is the region where the symbol is accessible through its name.

The visibility of a symbol can be shadowed by other symbols (*shadowing*). For example, a variable defined in a Java method shadows a same-named field defined in the enclosing class. Some languages provide *access modifiers* to set the visibility depending on the context, e.g., in Java, the access modifier `private` narrows the symbol visibility down to the class scope. Access modifiers determine whether specific model elements may be accessed *from outside*, i.e., from other models whereas shadowing is *within* a model or model hierarchy. A combination of both is also possible, for example, a Java field in a class can shadow protected and public fields of the super class.

Definition 3 (Scope). A scope holds a logical grouping of symbol definitions and limits their visibility. Usually, scopes are attached to nonterminals that open scopes and are thus organized hierarchically, which leads to a *scope-tree*. If a scope redefines (i.e., shadows) names from its enclosing scope(s), it is a *shadowing scope*, else, it is a *visibility scope*.

The structural relation of the symbol table elements is depicted in Fig. 8. A `Scope` associates symbols (`Symbol`) with their name. Scopes may have an arbitrary number of sub-scopes and an optional enclosing scope to represent a scope-tree.

The `SymbolTable` mainly consists of that scope-tree and points to its root. Every symbol has exactly one kind (`SymbolKind`).

The different symbol table kinds we described in [7] are now substituted by a combination of symbol shadowing and the aforementioned access modifiers:

1. **Encapsulated** symbol are only visible within their defining scope. For example, a local variable that is defined in a if-statement in Java, is only visible within that statement, not in the enclosing method or class.
2. **Imported** symbols are symbols that are imported from another scope in order to make them visible in the importing scope. Simply put, scopes can import symbols of their enclosing scope. For example, a field defined in a Java class is visible in all its methods (i.e., sub-scopes). If the enclosing scope is from another model, e.g., a super class in Java, not all symbols are visible, depending on what symbols the enclosing scope *exports*, as described next.
3. **Exported** symbols are defined in a corresponding scope and are publicized to their environment. Whether a symbol is exported can be set by access modifiers. In Java, for example, all non-private symbols are exported. However, those symbols may not necessarily be used from everywhere equally: a protected field symbol is exported, but can only be imported by sub-classes and classes in the same package.
4. **Forwarded** symbols are both imported and exported. For example, an imported symbol that represents a protected method inherited from a superclass is also exported, and hence, can be imported by the corresponding subclasses.

4.2 Symbol Table Components

The previously described symbol table and scope structure has to be created for each language. The MontiCore language workbench [17] provides infrastructure for a uniform development of technical symbol table components for modeling languages. The most important classes and interfaces are depicted in Fig. 9.

Each concrete modular modeling language is presented by the interface `ILanguage`. It offers the technical components needed to create the symbol table with the scopes for an instance of that language. Its symbol creators (subclasses of `ConcreteASTAndScopeVisitor`) are used to set up the scope hierarchy of a model. Then, symbols for model elements are created and organized in the symbol table, i.e., in the corresponding scope. The registered `IInheritedSymbolCalculatorClients` are used to compute whether symbols from imported scopes are shadowed by locally defined symbols. An `IQualifierClient` has to be provided for each element kind of a language which instances may be referenced within the current or another model, such as a referenced type of a field. The concrete qualifier client is used to calculate the qualified name of a referenced symbol with the corresponding kind. Resolver clients (`IResolverClient`) have to be provided for each symbol kind that may be referenced within the current scope hierarchy. The registered deserializers

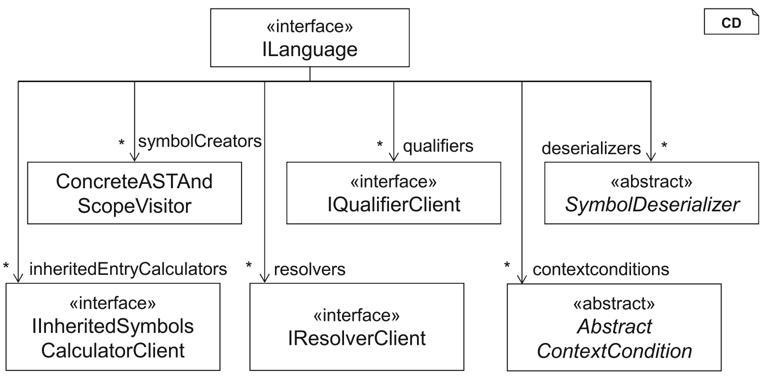


Fig. 9. Technical components of a symbol table (cf. [4]).

(`SymbolDeserializer`) load serialized symbols from externally referenced models. They are used to resolve the corresponding symbol definitions for symbol references that represent a referenced model element. Associated context conditions that extend the abstract class `AbstractContextCondition` are used to check if processed models are well formed.

This way a concrete `ILanguage` module offers all means to process models or model parts of a certain language and produce a corresponding symbol table. The provided infrastructure additionally alleviates inter-model relations that allow to resolve external information defined in related models. How to combine these components in several ways to realize the language integration concepts presented in Sect. 3 is described in the following subsection.

4.3 Configuration of Language Compositions

Language integration requires different effort depending on the type of integration. The composition takes place hierarchically to enable application of mechanisms in the best possible order. Figure 10 shows the different language concepts required to achieve this compositionality.

Language aggregation of two or more existing modeling languages is implemented in `LanguageFamily` instances. These gather the different independent `ModelingLanguages` together with the inter-language infrastructure, such as resolvers, qualifiers and adapters for symbol table integration, as well as factories and inter-language context conditions. Language families are used by MontiCore's `DSLTools` to process sets of heterogeneous but related models. The MontiArcAutomaton language family, for instance, comprises a modeling language for architecture models and a modeling language for CDs.

A `ModelingLanguage` is a black-box language and contains language-specific information such as the file ending. It may contain either a single language or a composition of embedded languages, such as MontiArcAutomaton components with embedded I/O$^\omega$ automata. Therefore, modeling languages contain a hierarchy of `ILanguage` interfaces. Based on this, MontiCore creates the infrastructure

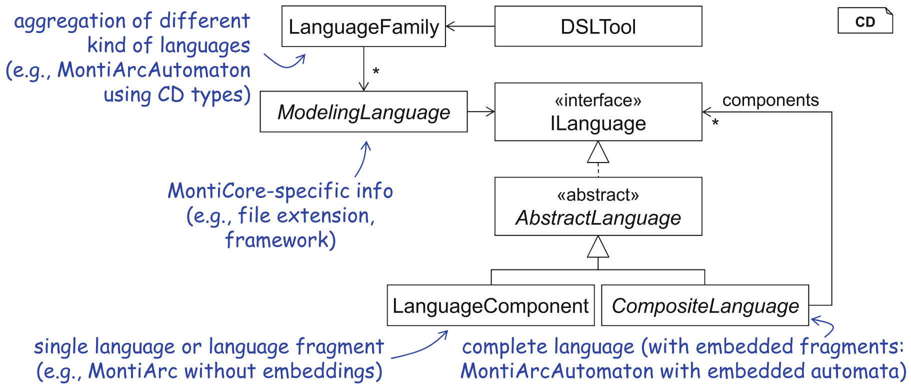

CD
LanguageFamily
DSLTool
ModelingLanguage
«interface» ILanguage
«abstract» AbstractLanguage
LanguageComponent
CompositeLanguage
components
*
*
aggregation of different kind of languages (e.g., MontiArcAutomaton using CD types)
MontiCore-specific info (e.g., file extension, framework)
single language or language fragment (e.g., MontiArc without embeddings)
complete language (with embedded fragments: MontiArcAutomaton with embedded automata)

Fig. 10. Technical realization of MontiCore's language composition mechanisms (cf. [3,4]).

to parse model instances accordingly. This infrastructure contains the correct combination of parser and lexer for the model at hand.

For single languages, modeling languages contain only a single `LanguageComponent` that contains the symbol table infrastructure and context conditions necessary. A `LanguageComponent` contains the information required to process symbols of the respective single language, i.e., how symbols are created, deserialized, qualified, resolved, which context conditions are available, and which symbol types are exported (see Sect. 4.2). For embedded languages, modeling languages contain a hierarchy of `CompositeLanguages`. These are composed of the embedded languages that themselves are represented by an implementation of `ILanguage`. Language components and composite languages are implemented as `AbstractLanguage` that provide common functionality used by language components and composite languages. They implement the interface `ILanguage` to allow utilization in a composite [16]. Using the composite pattern for embedding allows to reuse the resulting language combinations easily in different contexts, e.g., to embed activity diagrams and I/O$^\omega$ automata into components. Composite languages and language families can be considered as the glue between languages and their symbol tables as they hold the required adapters, resolvers, qualifiers, and context conditions for their specific composition.

Figure 10 also shows, that the order of language aggregation is arbitrary and depends on the language engineer. Whether a subset of the embedded languages should define a new `ModelingLanguage` solely depends on the desire to reuse this combination. It might, for example, be useful to combine MontiArcAutomaton and I/O$^\omega$ automata first, and to reuse the resulting combination with different, additional component behavior languages. Please note that language inheritance is not reflected in this structure, as the resulting combined abstract syntax does not necessarily require any interaction on symbol level. If this however is necessary, usually a new'main' symbol has to be created that contains the additional information resulting from the language extension. Thus, MontiArcAutomaton

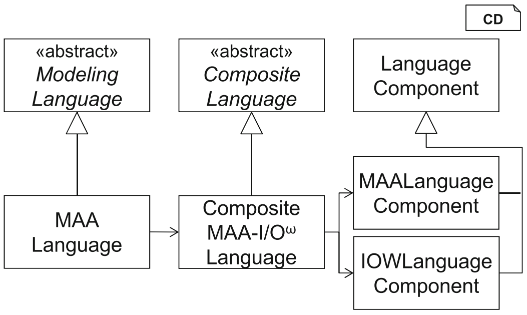


Fig. 11. Language composition for the MontiArcAutomaton `MAALanguage` (cf. [3]).

introduces a new component symbol to contain the new language features. To be reusable with existing language integration infrastructure of the inherited language, these need to be adapted to its symbols accordingly. Hence, MontiArcAutomaton component symbols need to be adapted to MontiArc component symbols. Similar to the development of the symbol table of a new language, symbols, symbol creators, qualifiers, and resolvers have to be registered for elements added by the inheriting language. The inheriting language can also reuse context conditions of the inherited language and add new ones.

Figure 11 illustrates the language composition mechanisms on the MontiArcAutomaton language `MAALanguage`. The language allows to model components with embedded I/O$^\omega$ automata illustrated in Sect. 2.2. As this embedding happens on the concrete syntax, there is no need to reference models of other languages by name. Therefore, the language is implemented as a modeling language that contains the composite language `CompositeMAAAutomata` which realizes the embedding. `CompositeMAAAutomata` is composed of language components for `MontiArc`, and `IOAutomaton` respectively, and contains adapters between the two languages as well as inter-language context conditions. Language-internal context conditions are defined in the language component. Cross-language inter-model context conditions are defined in language families, whereas cross-language intra-model context conditions are defined in composite languages. Adaptation between symbols of two languages, such as the types of automaton messages and ports of embedding MontiArcAutomaton, requires the composite language to provide adapters, qualifiers, and resolvers for type pairs. Adaptation between aggregated languages of a language family requires to configure these elements in the language family instead. Figure 12 shows the elements required to adapt type symbols of MontiArcAutomaton to type symbols of a CD language. Integration requires to provide a new adapter factory marked responsible to create symbols of a certain type (here MontiArcAutomaton type symbols for CD types). When a CD type needs to be qualified or resolved for embedded I/O$^\omega$ automata, the factory produces an adapter which behaves like a MontiArcAutomaton type symbol, but delegates all methods to the adapted CD type symbol.

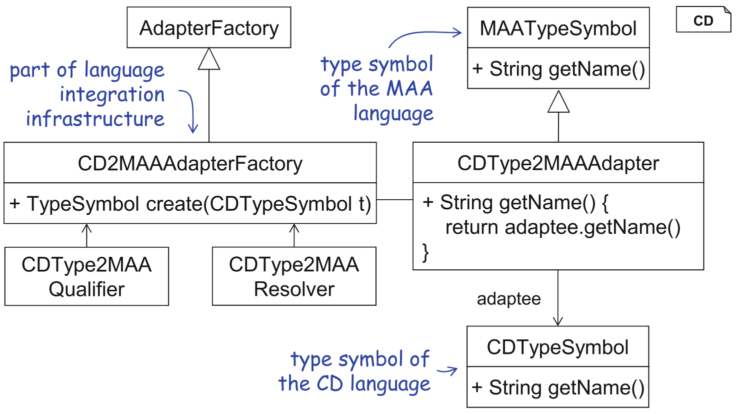


Fig. 12. Adaptation between type symbols of MontiArcAutomaton and those of the CD language.

Using adaptation on the levels of composite languages and language families allows to develop languages without consideration of a posteriori integration. As the languages are free from integration premises, they can be composed arbitrarily.

5 Related Work

We have presented three mechanisms for the integration of modeling languages. Integration takes place on the syntactical level and enables language aggregation, language embedding, and language inheritance. Related to our contribution are other studies and approaches on general syntax-oriented language integration. We do neither discuss language integration for specific language families [18,19] as these are usually specifically created to be integrated, nor do we discuss semantic language integration [20–23].

A study on language composition mechanisms distinguishes the mechanisms: "language extension, language restriction, language unification, self-extension, and extension composition" [24]. The authors' notion of language extension also requires that languages can be composed a-posteriori and distinguishes language extension from language integration. Our approach to language extension allows to overwrite nonterminals from the extended language in order to reduce expressiveness. The proposed notion of "language unification" matches our definition of language aggregation, where two independent languages can be used "unchanged by adding glue code only". In their definition of "self-extension", the authors start from a different definition of "language embedding" than we do: there, language embedding is that a "domain-specific language is embedded into a host language by providing a host-language program that encapsulates the domain-specific concepts and functionality" [25], which defines the use of domain-specific programs and is hardly recognizable as language embedding.

Accordingly, the author's definition of "self-extension" requires that "the language can be extended by programs of the language itself while reusing the language's implementation unchanged"–which also allows to "embed" languages as strings into the host language, e.g., SQL queries or regular expressions in Java. MontiCore does not provide an explicit "self-extension" mechanism, but supports it by embedding action languages allowing definition of programs. MontiCore languages also support the language extension composition mechanisms denoted as incremental extension and language unification as defined in [24].

A recent study on language workbenches [26] provides an overview of existing tools and their features. In particular, Ensō [27], Más [28], MetaEdit+ [29], MPS [30], Onion [26], Rascal [31], Spoofax [32], SugarJ [33], Whole Platform [34] and XText [35] are reviewed. This review considers four dimensions: syntax, validation, semantics and editor services. According to the overview all of the presented tools are able to achieve syntactical composition via different mechanisms. Nevertheless the composition on the validation depends on the validation features of the respective workbench. Only MPS, SugarJ and XText provide validation for naming and type checking similar to our approach to syntax-oriented language integration, namely concrete syntax, abstract syntax, symbol tables, and context conditions. In [36] the composition of languages in MPS is shown in more detail. To compose types new type definition rules have to be applied to infer types via unification. Since MontiCore is not projectional and uses independent parsers we define these connections between AST elements via our adaptation mechanisms and not via generic type definition rules. Furthermore, [36] distinguishes between language combination, extension, reuse and embedding. While language combination and reuse are similar to our notion of aggregation, language extension corresponds to our language inheritance, and language embedding is congruent with our concept of embedding.

The authors of [37] highlight cross-language context conditions as an important source for errors, propose to develop reusable cross-language context conditions and sketch how these can be implemented with their language workbench. It remains to be discussed how context conditions checking semantic properties specific to a language family can be designed for reuse.

Another approach to deal with the complexity of language integration is to employ domain-specific embedded languages (DSELs) in a host language (e.g., Scala) [38]. Regarding our example, this circumvents the problems arising from using data types between languages and allows to reuse existing development infrastructure. These approaches focus on syntax-oriented integration as well, but language reuse is limited to languages of the same host language and often DSELs lack explicit meta-models usable for integration purposes.

Attribute grammars [39] allow to enrich grammar symbols with computation rules. Research in attribute grammars led to promising results regarding language integration, such as Forwarding [40], and produced capable language workbenches as well [23]. Using multiple inheritance of attribute grammars to integrate is another interesting approach [41] to language integration that suffices to fulfill the language composition mechanisms identified in [24].

The authors of [42] propose a semi-automatic lifting from meta models to ontologies to ease language integration. After constructing ontologies representing the languages' meta models, the authors suggest nine refactoring patterns to make concepts implicit in the meta models explicit in the ontologies. The resulting ontologies should reduce language integration to ontology matching. While this process could ease language aggregation, neither language extension, nor language embedding require matching of symbols.

6 Conclusion

Engineering of complex software systems requires MDE where language integration can help to deal with the heterogeneity of modeling languages involved. We introduced language aggregation, language embedding, and language inheritance by example. These language integration techniques allow integration of languages without stipulating possible integration partners or mechanisms a priori. This enables to compose languages with minimal effort.

We have illustrated how these integration mechanisms are implemented in MontiCore. To achieve cross-language resolution of names, the symbol table, language families, and modeling languages were introduced. Language families and modeling languages contain the glue to enable cross-language model usage. This glue is implemented in the form of adapters between entries of symbol tables. The presented concepts and framework have been evaluated with various languages for different domains, such as architectural modeling [13], modeling the evolution of such architectures [43] modeling variability and evolution in software product lines [44–47], modeling cloud architectures [48] and architectural as well as behavioral modeling for cyber-physical systems [10,49,50].

In order to use the symbol table infrastructure out-of-the-box, in the future, it could be extended by generic default implementations. Following the MDE approach, it should also be examined whether (language-specific) parts of the symbol table can be generated from the language and its models directly. Moreover, it should be investigated whether the different language integration definition mechanisms (e.g., grammars for embedding, symbol table for aggregation) can be unified. Furthermore, it might be possible to generate the cross language infrastructure from enriched models of the respective language as well.

References

1. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering (FOSE 2007) (2007)
2. Object management group: OMG unified modeling language (OMG UML), superstructure version 2.3 (2010). http://www.omg.org/spec/UML/2.3/Superstructure/PDF/. Accessed 05 May 2010
3. Schindler, M.: Eine Werkzeuginfrastruktur zur Agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker (2012)

4. Völkel, S.: Kompositionale Entwicklung domänenspezifischer Sprachen. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag (2011)
5. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: modular development of textual domain specific languages. In: Paige, R.F., Meyer, B. (eds.) Proceedings of Tools Europe. Lecture Notes in Business Information Processing, vol. 11. Springer, Heidelberg (2008)
6. Look, M., Navarro Perez, A., Ringert, J.O., Rumpe, B., Wortmann, A.: Black-box integration of heterogeneous modeling languages for cyber-physical systems. In: Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC), Miami, Florida, USA (2013)
7. 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, Angers, France, Scitepress (2015)
8. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Softw. Tools Technol. Trans. (STTT) **12**(5), 353–372 (2010)
9. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)
10. Ringert, J.O., Rumpe, B., Wortmann, A.: MontiArcAutomaton: modeling architecture and behavior of robotic systems. In: Workshops and Tutorials Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Karlsruhe, Germany (2013)
11. Ringert, J.O., Rumpe, B., Wortmann, A.: Architecture and behavior modeling of cyber-physical systems with MontiArcAutomaton. Number 20 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag (2014)
12. Ringert, J.O., Rumpe, B., Wortmann, A.: Multi-platform generative development of component and connector systems using model and code libraries. In: 1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp 2014). CEUR Workshop Proceedings, Valencia, Spain, vol. 1281, pp. 26–35 (2014)
13. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - architectural modeling of interactive distributed and cyber-physical systems. Technical report AIB-2012-03, RWTH Aachen (2012)
14. Rumpe, B.: Modellierung mit UML, 2nd edn. Springer, Heidelberg (2011)
15. Rumpe, B.: Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2nd edn. Springer, Heidelberg (2012)
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Boston (1995)
17. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Technical Report Informatik-Bericht 2006–04, Software Systems Engineering Institute, Braunschweig University of Technology (2006)
18. Barja, M.L., Paton, N.W., Fern, A.A.A., Williams, M.H., Dinn, A.: An effective deductive object-oriented database through language integration. In: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB) (1994)
19. Groenewegen, D., Visser, E.: Declarative access control for WebDSL: combining language integration and separation of concerns. In: Proceedings of the 8th International Conference on Web Engineering 2008 (ICWE 2008) (2008)

20. Grönniger, H., Rumpe, B.: Modeling language variability. In: Calinescu, R., Jackson, E. (eds.) Monterey Workshop 2010. LNCS, vol. 6662, pp. 17–32. Springer, Heidelberg (2011)
21. Hedin, G., Magnusson, E.: JastAdd - an aspect-oriented compiler construction system. Sci. Comput. Program. **47**(1), 37–58 (2003)
22. Wende, C., Thieme, N., Zschaler, S.: A role-based approach towards modular language engineering. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 254–273. Springer, Heidelberg (2010)
23. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Electron. Notes Theor. Comput. Sci. **203**(2), 103–116 (2008)
24. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the 12th Workshop on Language Descriptions, Tools, and Applications (2012)
25. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of the 5th International Conference on Software Reuse 1998 (1998)
26. Erdweg, S., et al.: The state of the art in language workbenches. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 197–217. Springer, Heidelberg (2013)
27. van der Storm, T., Cook, W.R., Loh, A.: The design and implementation of object grammars. Sci. Comput. Program. **96**, 460–487 (2014)
28. Más website. http://www.mas-wb.com
29. Kelly, S., Lyytinen, K., Rossi, M.: Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080. Springer, Heidelberg (1996)
30. Dmitriev, S.: Language oriented programming: the next programming paradigm. JetBrains onBoard **1** (2004). https://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf
31. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain specific language for source code analysis and manipulation. In: Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation 2009 (SCAM 2009) (2009)
32. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. SIGPLAN Not. **45**(10), 444–463 (2010)
33. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: library-based syntactic language extensibility. ACM SIGPLAN Not. **46**(10), 391–406 (2011)
34. Solmi, R.: Whole platform. Ph.D. thesis, University of Bologna (2005)
35. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (2010)
36. Voelter, M.: Language and IDE modularization and composition with MPS. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2011. LNCS, vol. 7680, pp. 383–430. Springer, Heidelberg (2013)
37. Tomassetti, F., Vetro, A., Torchiano, M., Voelter, M., Kolb, B.: A model-based approach to language integration. In: Proccedings of the 5th International Workshop on Modeling in Software Engineering (MiSE) (2013)
38. Hofer, C., Ostermann, K.: Modular domain-specific language components in scala. ACM SIGPLAN Not. **46**(2), 83–92 (2010)
39. Knuth, D.F.: Semantics of context-free languages. Math. Syst. Theory **2**(2), 127–145 (1968)

40. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 128–142. Springer, Heidelberg (2002)
41. Mernik, M.: An object-oriented approach to language compositions for software language engineering. J. Syst. Softw. **86**(9), 2451–2464 (2013)
42. Kappel, G., et al.: Lifting metamodels to ontologies: a step to the semantic integration of modeling languages. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 4199, pp. 528–542. Springer, Heidelberg (2006)
43. Haber, A., Hölldobler, K., Kolassa, C., Look, M., Rumpe, B., Müller, K., Schaefer, I.: Engineering delta modeling languages. In: Proceedings of the 17th International Software Product Line Conference (2013)
44. Haber, A., Rendel, H., Rumpe, B., Schaefer, I.: Evolving delta-oriented software product line architectures. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 183–208. Springer, Heidelberg (2012)
45. Haber, A., Kutz, T., Rendel, H., Rumpe, B., Schaefer, I.: Delta-oriented architectural variability using MontiCore. In: ECSA 2011 5th European Conference on Software Architecture: Companion Volume (2011)
46. Haber, A., Rendel, H., Rumpe, B., Schaefer, I.: Delta modeling for software architectures. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII (2011)
47. Haber, A., Rendel, H., Rumpe, B., Schaefer, I., van der Linden, F.: Hierarchical variability modeling for software architectures. In: Proceedings of International Software Product Lines Conference (SPLC 2011) (2011)
48. Navarro Pérez, A., Rumpe, B.: Modeling cloud architectures as interactive systems. In: 2nd International Workshop on Model-Driven Engineering for High Performance and CLoud computing (MDHPCL) (2013)
49. Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., Wortmann, A.: A new skill based robot programming language using UML/P statecharts. In: Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA), Karlsruhe, Germany (2013)
50. Ringert, J.O., Rumpe, B., Wortmann, A.: From software architecture structure and behavior modeling to implementations of cyber-physical systems. In: Wagner, S., Lichter, H. (eds.) Software Engineering 2013 Workshopband. Volume 215 of LNI, GI, Köllen, pp. 155–170. Druck+Verlag GmbH, Bonn (2013)