Software Language Engineering in the Large: Towards Composing and Deriving Languages

Katrin Hölldobler, Bernhard Rumpe, Andreas Wortmann

Software Engineering, RWTH Aachen University, Aachen, Germany `www.se-rwth.de`

Abstract

Suitable software languages are crucial to tackling the ever-increasing complexity of software engineering processes and software products. They model, specify, and test products, describe processes and interactions with services and serve many other purposes. Meanwhile, engineering suitable modeling languages with useful tooling also has become a challenging endeavor - and far too often, new languages are developed from scratch. We shed light on the advances of modeling language engineering that facilitate reuse, modularity, compositionality, and derivation of new languages based on language components. To this end, we discuss ways to design, combine, and derive modeling languages in all their relevant aspects. We illustrate the application of advanced language engineering throughout the paper, which culminates in the example of deriving complete domain-specific transformations language from existing language components.

Keywords: Software Language Engineering, Language Composition, Language Derivation

The limits of my language
mean the limits of my world

– Ludwig Wittgenstein

1. Motivation

Using models to understand and shape the world is a very foundational technique that has already been used in ancient Greece and Egypt. Scientists model to comprehend while engineers model to design (parts of) the world. Although modeling has been employed for ages in virtually all disciplines it is fairly recent that the form of models is made explicit in *modeling languages.*

Computer science invented this approach to enable formality and a precise understanding of what is a well-formed model for the communication between humans and machines.

Programming languages in general, SQL [1], XML [2], and the Unified Modeling Language (UML) [3, 4, 5] in particular have been developed to enable such highly precise communication. Despite these efforts, it is obvious that researchers and practitioners of many domains are dissatisfied by solving domain-specific problems with general purpose languages or unified languages that try to cover all domains. The general aspiration of such languages creates a conceptual gap between the problem domains and the solution domains giving rise to unintended complexities [6]. As a result, Domain-Specific Languages (DSLs) and Domain-Specific Modeling Languages (DSMLs) [7, 8, 9, 10] were created to meet domain-specific needs. Due to the ongoing digitization of virtually every domain in our life, work, and society, the need for more specific languages arises. It is apparent, that we need to be able to accommodate new and changing domains with appropriate domain-specific languages – ideally on-the-fly. This raises three questions:

1. How to design new DSLs that fit specific purposes?
2. How to engineer a DSL from predefined components?
3. How to derive DSLs from existing DSLs?

In this paper, we discuss means to efficiently engineer DSLs and to enable their composition to facilitate language engineering in the large. One particular mechanism towards this is language derivation, in which new languages are derived from existing ones. We present and discuss such a mechanism for the derivation of transformation languages as well as its application in detail. Using this can prevent designing new languages from scratch each time and facilitates efficient engineering of languages from reusable components. The discussed mechanisms to compose and derive languages are the core of what we call *Software Language Engineering* (SLE) [11] today: the discipline of engineering software languages, which are not only applied to computer science, but to any form of domain that deals with data, their representation in form of data structures, smart systems that need control, as well as with smart services that assist us in our daily life. This paper extends our previous work presented in [12, 13] by a refined software language derivation mechanism for domain-specific transformation languages and a detailed description of the derived MontiArc transformation language. Overall, the contributions of this paper are

- A presentation of the state of software language engineering with an example using the MontiCore language workbench (Section 2).
- An investigation of software language composition mechanisms (Section 3).
- A detailed presentation of a language derivation mechanism that presents its derivation rules and a comprehensive example describing the derivation of the MontiArcTL transformation language (Section 4).
- A discussion of challenges in language engineering (Section 5).

2. Language Engineering

Model-Driven Development [14, 15, 6] lifts abstract models to primary development artifacts to facilitate software analysis, communication, documentation, and transformation. Automated analysis and transformation of models require that these adhere to contracts. Such automation is feasible, if models conform to contracts in the form of languages specifications. For many popular modeling languages, such as UML [3], AADL [16], or Matlab/Simulink [17], research and industry have produced useful analyses and transformations. These rely on making the constituents and concerns of languages machine processable. To this effect, the discipline of SLE investigates disciplined and systematic approaches to the design, implementation, testing, deployment, use, evolution, and recovery of modeling languages.

Similar to research in natural languages, SLE commonly defines languages as the set of sentences they can produce [18]. Operationalizing languages, however, requires more precise characterizations. To this effect, languages usually are defined in terms of their syntax (possible sentences) and semantics (meaning) [19]. Syntax comprises concrete syntax (words) and abstract syntax (structure), while semantics comprises static semantics (well-formedness) and dynamic semantics (behavior) [18]. The technical realizations of modeling languages often implement the latter concretization. As "software languages are software too" [20], their technical realizations are as diverse as implementations of other software kinds. This complicates comprehensibility, maintenance, evolution, testing, deployment, and reuse.

To shed light on this diversity of language realization mechanisms, this section presents different mechanisms to define modeling languages and highlights selected language development environments employing these mechanisms. Afterwards, we illustrate the development of a language to represent a variant of UML class diagrams that will serve as running example for the subsequent sections.

2.1. Engineering Modeling Languages

Research has produced various means to develop solutions for representing the different concerns of modeling languages. Lately, two different language implementation techniques have been distinguished:

1. Internal modeling languages can be realized as fluent APIs [7] in host programming languages whose method names resemble keywords of the language. Omitting syntactic sugar (such as dots and parentheses) as supported by modern programming languages (*cf.* Groovy, Scala) enables creating chains of method calls that resemble models. This method is suitable for language prototyping and promotes reusing the host language's tooling (such as parsers, editors, compilers, *etc.*). The expressiveness of the modeling language depends on the host programming language.

2. External modeling languages feature a stand-alone syntax that requires tooling to process its models into machine-processable representations. While this creates additional effort over internal languages, external languages can leverage a greater language definition flexibility. However, language-related tooling must be provided by the language engineer.

The majority of modeling language research focuses on external languages, which yield greater flexibility in language design. Consequently, research has produced more solutions to the definition of external languages, which is why we focus on their realization techniques in the following.

Engineering language syntaxes historically is related to the development and processing of (context-free) grammars [21], which are sets of derivation rules that at least enable describing the languages' abstract syntaxes. Many approaches to grammar-driven language engineering [22] support specifying a language's concrete syntax in the same grammar as well [23]; hence, they enable efficient language development and maintenance. Metamodels are another popular means to develop the abstract syntax of languages [9]. Here, classes and their relations structure the syntax of a language. While these do not support the integration of the concrete syntax (and, hence, always require providing editors), they enable reifying references between model elements that are name-based in grammars, as first level references.

Concrete syntaxes are either textual [24, 25], graphical [26], or projectional [9]. Textual syntaxes require parsing, whereas graphical and projectional syntaxes (*e.g.,* forms enabling editing the abstract syntax directly [27]) usually are bound to specific editors. In contrast, textual syntaxes enable reusing established software engineering tooling, such as editors or version control systems.

Whether the well-formedness of models is subject to their syntax or their static semantics is subject to debate [18, 28]. Nonetheless, various techniques have been established to enforce well-formedness of models with respect to properties that cannot be captured by grammars or metamodels (*e.g.,* preventing two class members of the same name). Popular approaches to well-formedness checking are programming language rules and Object Constraint Language (OCL) [29] constraints. Both require a model's internal representation and raise errors if these are not well-formed according to the individual rule. As OCL is a modeling language itself, it requires interpreting it or translating the constraints to programming language artifacts actually executed on the models under development.

Executing models is a popular way to realize their dynamic semantics. This can have the form of interpretation [30] or transformation [31]. With the former, a software (the interpreter) processes the models and executes according to their description. This interpreter can be part of the models or a separate software. Transformations process models and translates these into other formalisms with established semantics, such as a programming language. Model-to-text (M2T) transformations [32] read models of a specific language and translate these to plain text (such as programming language code), whereas model-to-model (M2M) transformations [32] translate models from an input modeling language to an output modeling language. The former lends itself for ad-hoc transformation development using template engines or string concatenation (as the output language is not required) but lacks the structure and verifiability of M2M transformations.

Language workbenches [33] – such as GEMOC Studio [34], Neverlang [35], MontiCore [36], MPS [27], Rascal [37], or Spoofax [38] – are software development environments supporting language engineering. Based on an, usually fixed, integration of language definition constituents, they facilitate creating languages and corresponding tooling. For instance, GEMOC Studio [34] employs ECore [39] metamodels for abstract syntax, OCL for static semantics, and Kermeta [30] for weaving interpretation capabilities into its languages. Concrete syntax can, *e.g.,* have the form of Xtext [40] grammars or Sirius [41] editors. The meta programming system (MPS) features projectional language engineering on top of a metamodel and combines this with well-formedness checking and execution through M2M transformations. Neverlang [35] supports grammar-based language definition and focuses on combining these with language processing tools. It executes models via interpretation.

Other approaches to engineering reusable languages and language modules focus on defining languages through attribute grammars [42], which support encoding semantics (such as behavior) into the abstract syntax itself. While this enables incremental language engineering [43], a more compact definition of languages and enforces considering language reuse with respect to syntax and semantics, it usually forces language engineers to master additional formalisms and tooling (*e.g.,* a grammar-embedded behavior language [44, 45]) instead of being able to reuse established mechanisms (such as Xtend [46] or FreeMarker [47]). Similarly, ASF+SDF environment enables specifying syntax through SDF in a BNF-like fashion and supports algebraic specification of semantics with ASF [48]. While this is even more expressive than attribute grammars, it again demands specific metalanguage tooling that language engineers rarely are familiar with.

The next section illustrates engineering of a textual modeling language for class diagrams (CDs) with the MontiCore language workbench.

2.2. Language Engineering with MontiCore

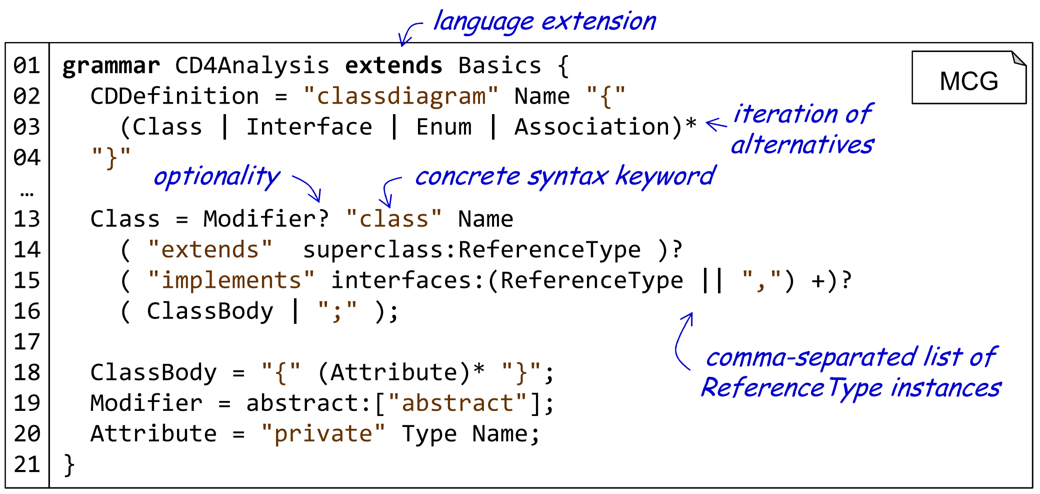
MontiCore [24, 49, 36] is a language workbench for the efficient engineering of compositional modeling languages. The concrete and abstract syntax of languages are defined as extended context-free grammars (CFG). From these grammars, MontiCore generates parsers and abstract syntax classes. The parser enables processing textual models into instances of the languages' abstract syntax classes. To ensure well-formedness of parsed models, MontiCore also supports a Java-based well-formedness checking framework. With this, Java context conditions process these models to check their well-formedness prior to applying M2M transformations [12] or template-based code generators [50]. MontiCore supports language inheritance, language embedding, and language aggregation [49] to reuse and combine languages with little effort. Language inheritance resembles language extension as presented in [51, 52], but in contrast to operating on attribute grammars, MontiCore operates on CFGs, focuses on code generation, and leaves the choice of model behavior implementation language to the developer (although FreeMarker [47] is supported out-of-the-box).

Various MontiCore languages have been engineered for and applied to different domains[1], including automotive [53], cloud computing [54], smart homes [55], robotics [50], and software engineering [4, 5] itself. For the latter, we have developed the UML/P family of modeling languages [5], which is a subset of UML [3] that is refined to enable pervasive model-driven engineering. The UML/P includes the class diagram for analysis (CD4A) modeling language, which is used to illustrate language engineering techniques in the following.

Consider the excerpt of the `CD4Analysis` MontiCore grammar depicted in Figure 1, which defines the syntax of the CD4A modeling language. The grammar comprises productions that define the integrated concrete and abstract syntax of language elements. After the keyword `grammar` and its name, the grammar extends the grammar `Basics` to reuse previously defined productions (l. 1), such as rules for types and names. Afterwards, the productions follow that characterize this variant of UML class diagrams. Each rule is defined by a head (before the "=", *e.g.,* `CDDefinition` in l. 2) and a body. The head of a rule declares a new language element and the body defines its properties. To this effect, the body contains terminals (*e.g.,* `"classdiagram"` in l. 2) and non-terminals (*e.g.,* `Class` in l. 3). Different operators (*e.g.,* "$\star$" in l. 3, "?" in l. 13, and "+" in l. 15) define the quantity or presence of a part in a rule's body.

Based on this grammar, the CD4A model, shown as an excerpt in Figure 2, can be created. It describes a simplified banking system consisting of a package declaration (l. 1), an abstract `Account` class to describe different types of accounts (ll. 3-7), an interface to model employees (l. 17) and its implementation (ll. 18-20), as well as multiple associations (*e.g.,* l. 55). From this grammar, MontiCore produces a parser and an abstract syntax class for

[1]See `http://monticore.de/languages/`


language extension
MCG
iteration of alternatives
optionality
concrete syntax keyword
comma-separated list of ReferenceType instances

```
grammar CD4Analysis extends Basics {
  CDDefinition = "classdiagram" Name "{"
    (Class | Interface | Enum | Association)*
  "}"
…
  Class = Modifier? "class" Name
    ( "extends"  superclass:ReferenceType )?
    ( "implements" interfaces:(ReferenceType || ",") +)?
    ( ClassBody | ";" );

  ClassBody = "{" (Attribute)* "}";
  Modifier = abstract:["abstract"];
  Attribute = "private" Type Name;
}
```

Figure 1: An excerpt of a MontiCore grammar for the CD4A language.

each rule. The latter captures the rule's right-hand side by providing members capable of storing its content. Additionally, MontiCore's infrastructure applies context conditions – manually created Java rules defined relative to the abstract syntax of CD4A – to determine the well-formedness of models. This is used, for instance, to ensure uniqueness of class names within a diagram.

3. Composing Modeling Languages

Model-Driven Development is successful when initiated bottom-up [56], *i.e.*, developers employ modeling languages considered suitable for their domain-specific challenges instead of using predefined, monolithic general-purpose modeling techniques. For efficient language engineering, evolution, validation, and maintenance, these languages should be retained as independent as possible. Ultimately, however, combining such languages mandates their efficient *composition* [18]. Considering, for instance, the software of the smart and modular factories imagined within Industry 4.0, which demand integrating business processes, domain models, behavior models and failure models of the automation systems, assembly plan models, manipulator kinematics, and more. Integrating these modeling languages into a combined software requires well-defined operations for their composition.

Software engineering itself is another prime example of a domain leverag-

```
package banking;                                                    CD4A
classdiagram BankingSystem {
  abstract class Account {
    long number;
    int balance;
    int overdraft;
  }
…
  interface Employee;
  class Consultant implements Employee {
    String name;
  }
…
  association [1] Account <-> [[name]] Consultant;
}
```

Figure 2: An example of a CD4A model describing a lightweight banking system.

ing language composition to facilitate development, evolution, and maintenance. To this effect, research and industry have produced languages for (1) modeling structure and behavior of the software under development, such as UML [3]; (2) describing database interaction, such as SQL [1] or HQL [57]; (3) describing software build processes, such as Maven's Project Object Models [58]; (4) describing configuration of product lines [59], such as feature diagrams [31]; (5) describing model changes in a structured fashion, such as delta modeling languages [60] (6) extending models with additional, external information (tagging languages [61]); (7) coordinating the use of different modeling languages, such as the BCOol language [62]; (8) transforming models of other languages, such as ATL [63] or the FreeMarker [47] template language; and (9) describing the syntax and semantics of modeling languages, such as ECore [39], Kermeta [30], Melange [64], or MontiCore [24].

Consequently, structured reuse of language parts is crucial to enable efficient SLE. While research on language integration has produced reuse concepts and relations to language definition concerns [18], the diversity of language realization techniques has spawned very different reuse mechanisms [51]. Generally, we distinguish *language integration*, which produces a new language, from existing languages, from *language coordination*, in which the sentences of two languages (*i.e.,* their models) are related to achieve common goals [18].

For integrating languages, concepts such as merging of metamodels [64], inheriting and embedding of grammars [52, 49], importing or combining of metamodel and grammar elements [65, 66, 67, 68, 69] as well as interoperability by translating between different metamodels using horizontal model transformations [70, 71] have been conceived. These mechanisms enable a white-box integration to extend and refine existing abstract syntaxes to domain-specific requirements but rarely consider including integration of other language concerns. For instance, efficient creation of models conforming to languages produced through merging, inheritance, or importing of parts of other languages requires creating or extending proper editors. Even when editors for the base languages exist, this requires handcrafting editing capabilities for the extensions. Similar challenges arise for reusing language semantics. As these usually are realized through model interpretation or model transformation, the corresponding tools of extended languages must also be extended. Yet, there are only a few approaches that support compositional semantics realizations, such as code generator composition mechanisms [50] or code generator reuse by translating between metamodels [70, 72, 71]. Furthermore, when integrating different grammars, ambiguities in parsing can arise as discussed in [73].

Coordination of modeling languages is less invasive, but mandates means to reason over models of coordinated languages – either for their joint analysis or their joint execution. The former, for instance, requires checking the validity of feature models or model transformations with respect to the referenced models. To prevent tying the referencing languages to abstract syntax internals of the referenced languages, abstraction mechanisms, such as the symbol tables of MontiCore [49] have been developed. Joint execution of models of different languages requires exposing and combining their execution mechanisms. Where languages originate from the same language workbench, this integration has been addressed (*e.g.,* by exposing the executable interfaces of model elements [74]). Truly heterogeneous, generalizable coordination has yet to be achieved.

In the next section, we sketch how applying language integration mechanisms to the CD language enables preparing it for code generation.

3.1. Extending and Refining a MontiCore Language

To enable describing software-related properties of domain models more precisely (*e.g.,* their behavior), we extend the CD4A language by additional language constructs such as methods and their bodies. On the other hand,

```
01 grammar CD4Code extends CD4Analysis {                              MCG
02   start CDDefinition;
03
04   ClassBody = "{" (Attribute | Method)* "}";
05   Method = returnType:Type Name Parameters "{" Statement* "}";
06   Parameters = "(" (Parameter || ",")* ")";
07   Parameter = Type Name;
08   interface Statement;
…    // various implementations of interface Statement
82 }
```

Figure 3: An excerpt of the CD4Code extension of CD4A.

modeling abstract classes should be prevented without breaking existing CD4A-processing tooling.

For the former, the `CD4Code` grammar extends `CD4Analysis` (*cf.* Figure 1) and introduces methods with return types and bodies. For the latter, we introduce a new well-formedness rule that rejects models with abstract classes. An excerpt of the newly created `CD4Code` language is shown in Figure 3. It extends the `CD4Analysis` language (l. 1) and reuses its start rule (l. 2). In addition, the `ClassBody` rule is overridden and adds methods (ll. 4-8) and leverages MontiCore's `interface` productions to enable various statement implementations for method bodies (*e.g.,* assignments, conditionals, loops, *etc.*).

The rule `Class` of `CD4Analysis` also features an optional `Modifier`, which is translated to a field of the corresponding type in the abstract syntax representing `Class`. Consequently, reused CD4A tooling expecting a `Modifier` field (*e.g.,* a model checker) will not break as the field still is present in the abstract syntax. However, due to the new well-formedness rule, there will never be a `CD4Code` diagram instance with an "abstract" modifier. Ambiguity between nonterminal names is detected by MontiCore and handled by choosing the first occurrence replacing all further references to this nonterminal [36]. Further ambiguities are detected by the underlying ANTLR parser generator [75].

4. Deriving Modeling Languages

Software engineering leverages modeling languages to mechanize working with models of other languages, such as transformation languages [63, 76], delta modeling languages [60], or tagging languages [61]. Such languages have in common that they are either overly generic or are specifically tied to a *base language* (*i.e.,* the languages whose models are transformed or tagged). The former requires developers to learn completely new languages that are independent of a (possibly well-known) base language, while the latter raises the challenge of engineering and maintaining specific languages as well as their specific tooling (editors, analyses, transformations), which is hardly viable.

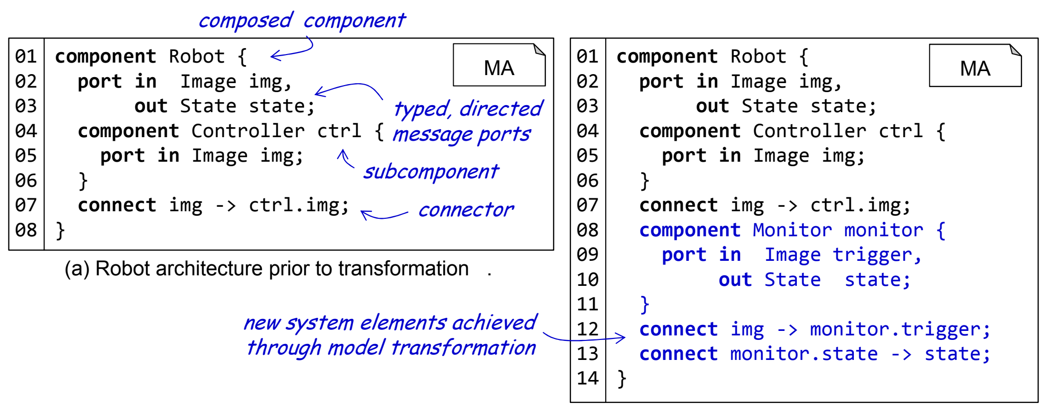
To address the latter, methods to develop new languages by deriving their syntaxes from related base languages have been developed. These methods rely on processing the base languages' (abstract) syntaxes and creating new (abstract) syntaxes from them. Where the base languages are defined by grammars, such derivation can produce derived concrete syntaxes. For metamodel-based language definition, this would require deriving editor (parts) instead. Automating the creation of well-formedness rules and behavior implementations of derived languages is more challenging as both may differ from the base languages completely. Where, for example, Statecharts describe state-based behavior, a transformation language derived from Statecharts describes how to translate Statechart models into something else. The behaviors of both languages are unrelated. The same holds for their well-formedness rules.

After an example illustrating the benefits of deriving modeling languages from other languages, this section presents a mechanism to derive Domain-Specific Transformation Languages (DSTLs) and demonstrates it by deriving the grammar MontiArcTL.

4.1. Example for Modeling Language Derivation

Consider developing software architectures for cyber-physical systems. A recent study on architecture description languages (ADLs) [77] identified over 120 different ADLs for different domains and use cases [78]. These range from industrial, comprehensive languages for specific domains (such as AADL [16] for avionics) to less complex, academic languages (such as ACME [79]). Nonetheless, learning a new ADL requires considerable effort.

Many of the ADLs identified in [78] have in common that they enable decomposing complex architectures into hierarchically composed, interconnected components that operate individually and exchange messages through explicit ports. An example for such an architecture, formulated in the MontiArc ADL [80], is depicted in Figure 4 (a), which sketches the composed


composed component
typed, directed message ports
subcomponent
connector
MA
new system elements achieved through model transformation
MA

```
component Robot {
  port in  Image img,
       out State state;
  component Controller ctrl {
    port in Image img;
  }
  connect img -> ctrl.img;
}
```

(a) Robot architecture prior to transformation .

```
component Robot {
  port in Image img,
       out State state;
  component Controller ctrl {
    port in Image img;
  }
  connect img -> ctrl.img;
  component Monitor monitor {
    port in  Image trigger,
         out State  state;
  }
  connect img -> monitor.trigger;
  connect monitor.state -> state;
}
```

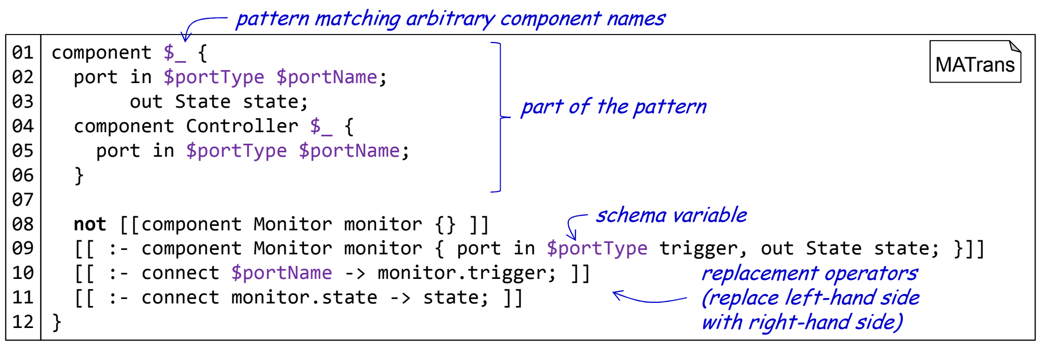
(b) After transformation, Robot includes a monitor and related connectors.

Figure 4: `Robot` architecture before and after transformation.

component `Robot` receiving an input `Image` via its incoming port `img` and passes that to its `Controller` that decides on its reaction. A typical pattern for such systems is to include run-time monitoring, which can be achieved through integrating a monitor component and connectors as depicted in Figure 4 (b). For each affected component, a new sub-component of type `Monitor` should be introduced that receives the same input as the containing component. The monitor evaluates the input and determines the containing component's state based on the input. This state is emitted through the containing component's `state` port, such that other components can take this state into consideration.

Instead of integrating this infrastructure manually for every component, it should be added automatically through appropriate transformations. Learning a general-purpose transformation language to describe transformations for the ADL of choice requires additional effort. Creating a transformation language that uses the syntax of the ADL of choice can reduce this effort accordingly. Obviously, using the exact same syntax is not feasible as transformation languages usually describe patterns of replacing model elements. Hence, such languages require means to describe patterns with static and variable parts. Ideally, architecture developers can describe these patterns in familiar syntax.

For the integration of monitoring infrastructure as illustrated in Figure 4 (b), a transformation should check whether the affected component does not contain a monitor yet and integrate a monitor and connectors accordingly.



```
01 component $_ {
02   port in $portType $portName;
03        out State state;
04   component Controller $_ {
05     port in $portType $portName;
06   }
07
08   not [[component Monitor monitor {} ]]
09   [[ :- component Monitor monitor { port in $portType trigger, out State state; }]]
10   [[ :- connect $portName -> monitor.trigger; ]]
11   [[ :- connect monitor.state -> state; ]]
12 }
```

Figure 5: A domain-specific transformation that integrates monitoring infrastructure into composed components using the base ADL's syntax.

Figure 5 depicts such a pattern in a DSTL derived from MontiArc. The pattern begins with matching arbitrary, composed components that contain a `Controller` sub-component that receives inputs from its containing component (ll. 1-6). This yields an outgoing `State` port, and does not contain a `Monitor` sub-component yet (l. 8). If such a component is met, a `Monitor` with corresponding ports and connectors is added to the containing component (ll. 9-11).

Evidently, this pattern resembles the input language very closely: it uses the same concrete syntax to describe patterns and replacements, *i.e.,* it uses vocabulary the developers are already familiar with [81, 82]. However, transformations require greater flexibility (*e.g.,* to match arbitrary composed components) than the base languages to enable more powerful patterns. This relaxation [83, 84] requires to ease constraints of the base language, to, for instance, enable omitting base language model elements in patterns that are irrelevant to it.

Deriving such languages often enables to retain large parts of the syntax of the base language and, hence, reduces the effort of learning the new language. Therefore, the following sections explain how to derive a DSTL and detail the resulting MontiArcTL grammar.

4.2. Elements of a DSTL

This section describes the elements that a derived DSTL supports. Based on this, the derivation of DSTLs is explained in the subsequent sections. The explanation partly reproduces the explanation given in our previous work [76], presents new derivation rules, and provides additional details.

4.2.1. Concrete Syntax and Schema Variables for Pattern

First of all, a derived DSTL needs to be able to describe the model part being transformed, *i.e.,* the pattern. As DSTLs are tailored to their corresponding DSLs, the syntax for describing patterns is taken from the DSL and complemented with transformation specific operators explained in this section. Consequently, every model itself is a valid pattern described using the DSTL. In addition, a DSTL relaxes the constraints of its DSL such that a pattern does not necessarily need to be a complete model [83, 84]. Instead, every model element defined by a non-terminal in the DSL is allowed as a top-level element in the pattern. For example, a pattern for a MontiArc model can describe connectors without specifying its surrounding component. Furthermore, only properties of the model element relevant to the pattern need to be described. For instance, if a component's ports are irrelevant for the transformation they need not be specified within the pattern even though the component in the model could have ports. Omitted properties do not specify their absence, instead, there is nothing said about these properties. To explicitly prevent the occurrence of certain model elements within the model, negation can be used (*cf.* Section 4.2.3). These elements are explained subsequently. Finally, DSTLs provide schema variables for abstraction purposes, *i.e.,* they can be used instead of specifying the pattern element and thus only specify the presence of a certain element, and bind pattern elements to variables. Schema variables consist of a "`$`" sign and a name, *e.g.,* `$portName` in Figure 5. Furthermore, using the application constraint explained in Section 4.2.7, constraints based on the schema variables can be expressed. Finally, schema variables can serve as a placeholder for values calculated and assigned to them as described in Section 4.2.6.

There are three possible way to use schema variables. The simplest form regards names in the model, *i.e.,* usages of the non-terminal `Name` in the grammar of the DSL. In MontiArcTL, this form can be used, for instance, to match component names and port names, for instance. Instead of a concrete name, a schema variable is used. In Figure 5, this form is used for the names of the ports (`$portName`). Using the same schema variable multiple times – as in the example – expresses that the corresponding name occurrences in the model must be identical.

The second way of using schema variables is to bind pattern elements. For this purpose, the pattern element is enclosed in double square brackets and preceded by a schema variable, *e.g.,* `$C [[ component Monitor`

`{} ]]`.

The third way is suitable for model elements defined by non-terminals such as components ports or connectors. In this case, a schema variable is used instead of a pattern element. This form consists of the schema variable and the type of the variable which is the name of the non-terminal in the DSL's grammar. Therefore, the pattern just expresses that an element corresponding to the type of the schema variable must be present in the model but does not restrict its structure, *e.g.,* `MAComponent $C`.

A match for the pattern is found in a model if a corresponding model element for every element of the pattern is found. If no match for the pattern exists the transformation is not applicable.

4.2.2. Replacement Operator

Graph transformations [85] typically express transformations by describing two graphs, the left-hand side (LHS) and the right-hand side (RHS). The LHS describes the pattern, while the RHS describes the same model part after applying the transformation. Elements only present in the graph of the LHS are deleted, elements present on both sides are kept and elements only present on the RHS are created. Separating these graphs complicates writing and understanding transformations. All elements that should be kept need to be repeated on the RHS and the modification needs to be understood by comparing both graphs. Thus, an integrated notation that only marks added and deleted elements is preferable as realized by several tools such as Henshin [86, 87] or eMoflon [88]. Derived DSTLs also use an integrated notation and provide a replacement operator for this purpose. The operator has the following syntax:

```
"[[" Element?  ":-" Element?  "]]"          (1)
```

The operator replaces the element specified on the left of `:-` by the one on its right. Furthermore, both sides can be left blank. If there is an element on the left-hand side only it is removed, if it is on the right-hand side only it is added. For example, in Figure 5, the operator is used to add a monitor component and connecting ports (ll. 9-11). Besides replacing, adding and removing elements, moving elements is possible using the replacement operator. Therefore, this operator is used in conjunction with schema variables. In this case, an element is bound to a schema variable and the replacement operator is used to remove it in one location and add it to another location within the model. Due to the schema variable, the bound element is added instead of a newly created one.

4.2.3. Negative Elements

Another feature derived DSTLs require is to prevent the presence of certain model elements, for instance, those elements that should be added. To express this, DSTLs provide an operator that marks elements as forbidden. The operator has the following syntax:

```
"not" "[[" Element "]]"
```
(2)

Thus, an element is enclosed in brackets and marked with the keyword `not` to mark it as a negative and thus forbidden element. If negative elements are present within the pattern a match for this pattern is found if a match for the “simple” pattern elements is found and no match for the negative elements is found. In this case, the transformation is applicable. For example, the operator is used in Figure 5 (l. 8) to ensure no monitor component is present before applying the transformation.

For simplicity and usability reasons negative elements are not allowed to be nested. However, this limitation is compensated by the application constraint explained in Section 4.2.7.

4.2.4. Collection Operator

A further feature of the derived DSTLs is the collection operator similar to the *set nodes* in PROGRES [89] or *multi objects* in Fujaba [90]. Using this operator, it is possible to match several similar structures within the model. In contrast, to set nodes and multi objects, the collection operator allows to match subpatterns multiple times. The operator has the following syntax:

```
"list" "[[" Element "]]"
```
(3)

The operator encloses a pattern element – and attached child elements – in double square brackets and marks it with the keyword `list`. Other approaches provide collection operators as well [91, 92, 93, 94, 95, 96, 97], however, these do not use the concrete syntax of the DSL for the operator. The collection operator is greedy, *i.e.,* it matches its pattern as often as possible but at least once. In case the collection operator is used, the transformation is applicable if a match for the "simple" pattern elements, no match for the negative elements and at least one match for the pattern inside the collection operator is found. An example for the collection operator is provided in Figure 6. Here, the pattern matches a component and all connectors defined inside it.

MATrans

```
component $_ {
  list [[ connect $_ -> $_ ; ]]
}
```

Figure 6: Example for the collection operator.

4.2.5. Optionality Operator

The DSTLs also provide an optionality operator, which enables marking pattern elements as optional. Optional elements are matched if they are present within the model. However, in case no match for an optional pattern element is found, the transformation is still applicable. Thus, this operator is useful to reduce the number of necessary transformations as this operator allows to handle two cases – the element is present or absent – within the same transformation. The operator has the following syntax:

$$\texttt{"opt" "[[" Element "]]"} \tag{4}$$

Thus, this operator encloses a model element in brackets and marks it with the keyword `opt`. If an optional element is present in the pattern, the transformation is applicable if the transformation without the optional part is applicable. However, in case a match for the optional element is present it is matched by the transformations and – if combined with the replacement operator – modified. An example for the optional operator is shown in Figure 7. This transformation matches an arbitrary component and a port of type `State` of this component, if present.

MATrans

```
component $_ {
  opt [[ port in State $_ ; ]]
}
```

Figure 7: Example for the optionality operator.

4.2.6. Variable Assignment

Derived DSTLs also feature an assignment block that allows calculating values for schema variables that only occur on the RHS. This is useful for values calculated based on other pattern elements, *e.g.,* uncapitalize component types for deriving instance names [98]. The assignment block has the following syntax:

```
"assign" "{"
   (SchemaVar "=" Expression ";")+            (5)
"}"
```

An example of an assignment is shown in Figure 8. Here, the transformation adds an instance name to a subcomponent [98] and derives the name based on the component's type.

```
component $type [[ :- $name ]] ;

assign { $name = uncapitalize($type) }
```
MATrans

Figure 8: Example for the assignment block.

4.2.7. Application Constraint

Finally, the application constraint allows constraining the applicability of a transformation. Therefore, the application constraint allows formulating an expression that needs to be fulfilled. All variables used in the pattern are available within the application constraint. Thus, if a pattern element needs to be further constrained, a variable needs to be attached to it. The application constraint has the following syntax:

```
"where" "{"
   BooleanExpression            (6)
"}"
```

```
component $type {}

where { $type.equals("Monitor") }
```
MATrans

Figure 9: Example for application constraints.

An example of an assignment is shown in Figure 9. The transformation matches a component but uses a variable for the component's type. For demonstration purposes, it uses the application constraint to constrain the possible type of the component.

4.3. Deriving a Domain-Specific Transformation Language

Transformation languages support pattern descriptions and replacement operations over elements of modeling languages. Domain-specific transformation languages realize both using the syntax of the DSL they are derived from. In [76], we presented rules to derive DSTLs from a given DSL. Such a DSTL consists of a common base grammar providing DSL-independent elements and of a grammar derived from the base language that defines the DSL-specific elements. This section presents the corresponding derivation rules to create the non-terminals for the DSTL's operators and introduces new derivation rules to complement [76]. As presented in [12, 76] models of the DSTL, *i.e.,* transformations, are translated to Java implementations that are used to apply the corresponding transformations.

4.3.1. Rule 1: Grammar Structure

The quintessential element of a DSTL is its grammar, which our derivation mechanism populates with individual non-terminals for the different transformation operations. This rule distinguishes between monolithic and modular base grammars. For monolithic base grammars, it derives a single new DSTL grammar. For modular, *i.e.,* inheriting, grammars, it retains the inheritance relations and creates DSTL grammars for each super grammar. The resulting DSTL grammar then inherits from the DSTL grammars derived from the super grammars.

Rule 1. *For a base grammar L, create a new grammar TL. If L is monolithic, TL inherits from $TFCommons$, i.e.,*

```
grammar TL extends TFCommons { }
```

Otherwise, for all of L's super grammars $SL_1 \dots SL_n$, create the according DSTL $TSL_1 \dots TSL_n$ and create a new grammar header that inherits from these grammars, i.e.,

$$\texttt{grammar TL extends } TSL_1\texttt{, ..., } TSL_n \texttt{ \{ \}}$$

Here, the names TSL_1, ..., TSL_n are derived from their respective base grammars by adding the suffix `TL` (for "transformation language").

The grammar `TFCommons` is a basis used for all DSTLs either directly or transitively. This grammar provides DSL-independent non-terminals, such as `TfIdentifier` that is used for schema variables and replacements of names, as well as the non-terminals for the assignment block and the application constraints as explained later on.

4.3.2. Rule 2: Concrete Syntax and Schema Variables

For each non-terminal and each relevant keyword of the base grammar, we create an interface non-terminal in the DSTL's grammar. Keywords represented in the abstract syntax of the DSL are changeable and thus considered as relevant. For instance, the keyword `component` of a component in MontiArc is not changeable – and hence has no representation in the abstract syntax – but a method's visibility keyword may be changed from `public` to `private` and could be reflected by Boolean attributes in the abstract syntax. Implementation of the respective interfaces enables using not only the base DSL's concrete syntax but also schema variables and other operations in its place. In the following, L denotes the base DSL's grammar, TL the DSTL under construction, N, K, and I are non-terminals and k a *relevant keyword*. To capture the different alternatives for the DSL's non-terminals, this rule is split into six sub-rules.

Rule 2a. *For each interface non-terminal $N \in L$, create a new interface non-terminal N in the DSTL TL. If $N \in L$ extends an interface non-terminal $I \in L$, then $N \in TL$ extends the interface non-terminal $I \in TL$.*

Rule 2b. *For each normal non-terminal $N \in L$, create a new interface non-terminal N in the DSTL TL. If $N \in L$ implements an interface non-terminal $I \in L$, then $N \in TL$ also extends the interface non-terminal $I \in TL$. If $N \in L$ extends a normal non-terminal $Y \in L$, then $N \in TL$ extends the interface non-terminal $Y \in TL$.*

The first two sub-rules ensure that inheritance relations between non--terminals of the DSL are represented in the DSTL. The next rule enables patterns over keywords in the DSTL. Here, the interface non-terminal's name is derived from the keyword's name by capitalizing its first letter, *e.g.,* a keyword `abstract` yields a DSTL non-terminal `Abstract`.

Rule 2c. *For each relevant keyword $k \in L$, create a new interface non-terminal K in TL.*

To flexibly match underspecified parts of the base DSL, the DSTL must support using patterns in place of the DSL's concrete syntax. As we reconstructed the DSL's structure using interfaces, appropriate patterns can easily be integrated through interface implementations as presented below:

Rule 2d. *For each normal non-terminal $N \in L$, create a new non-terminal $N\_Pat \in TL$, such that:*

```
N_Pat implements N =
        SyntaxOfN
          | "N" SchemaVar
          | SchemaVar "[[" SyntaxOfN "]]";,
```

where $SyntaxOfN$ is a modified copy of N's rule body. This copy is modified to replace (1) all occurrences of relevant keywords $k \in L$ with their corresponding interface non-terminals; and (2) all occurrences of the non--terminal `Name` with the non-terminal `TfIdentifier`, which is defined in the base grammar `TFCommons`.

This rule produces a disjunction of three parts. The first part is a copy of the DSL's syntax and, thus, allows describing the pattern by using the same vocabulary as used in the model. Due to the derivation of interface non-terminals described above, referenced non-terminals within the copied syntax point to the created interface non-terminals. Consequently, the representation of the base grammar's non-terminals does not rely on the base grammar's productions but references the derived interface non-terminals of the DSTL. This enables using the base grammar's concrete syntax (including schema variables) as well as the operators defined in the following. The second part of the disjunction provides the option to use a schema variable without specifying any properties of the matched model element (*cf.* Section 4.2.1). The last part combines schema variables with copied syntax and facilitates binding pattern elements to variables (*cf.* Section 4.2.1).

Interface non-terminals do not define the concrete syntax on their own. Hence, the representation of interface non-terminals omits $SyntaxOfN$. Moreover, transformation developers should be enabled to underspecify which specific implementation of a grammar rule must be present in a model. Thus, their representation introduces schema variables of the interface non-terminal's kind:

Rule 2e. *For each interface non-terminal $N \in L$, create a new non-terminal $N\_Pat \in TL$, such that:*

```
N_Pat implements N =
      "N" SchemaVar | SchemaVar "[[" N "]]";
```

The concrete syntax of a keyword is the keyword itself. Thus, there are no schema variables for keywords and the created DSTL elements omit corresponding alternatives:

Rule 2f. *For each relevant keyword $k \in L$, create a new non-terminal $K\_Pat \in TL$, such that:*

```
K_Pat implements K = k;
```

4.3.3. Rule 3: Replacement Operator

The replacement operator enables to create, update, and (re)move model elements in an integrated notation. We derive a specific replacement operator for each non-terminal and relevant keyword of the base grammar to support this create, read, update and delete functionality. As replacing model elements differs from replacing keywords, this rule is separated into two subrules accordingly.

Rule 3a. *For each normal non-terminal or interface non-terminal $N \in L$, create a new non-terminal $N\_Rep \in TL$, such that*

```
N_Rep implements N = [[ lhs:N? :- rhs:N? ]];
```

These *N_Rep* non-terminals enable CRUD for model elements. The second part of this rule is responsible for supporting CRUD for keywords.

Rule 3b. *For each relevant keyword $k \in L$, create a non-terminal $K\_Rep \in TL$, such that:*

```
K_Rep implements K =
        [[ k :- ]] | [[ :- k ]];
```

4.3.4. Rule 4: Negation Operator

Automated model transformation also needs to be able to react on omitted model elements. This, for instance, is useful to ensure the creation of new model elements only in case they are not already present. To support this, we derive rules for each non-terminal and relevant keyword that support specifying its negation. Again, this rule is separated into two parts responsible for model elements and keywords, respectively.

Rule 4a. *For each normal non-terminal and interface non-terminal $N \in L$, create a new non-terminal N_Neg, such that:*

```
N_Neg implements N = not [[ N ]];
```

Rule 4b. *For each relevant keyword $k \in L$, create a new non-terminal $K\_Neg \in TL$, such that:*

```
K_Neg implements K = not [[ k ]];
```

4.3.5. Rule 5: Collection Operator

The patterns introduced so far enable matching a single occurrence of a modeling element or keyword only. For better expressiveness, we introduce a collection operator that matches the contained pattern one or multiple times. The following rule creates the required non-terminals to use collection operators.

Rule 5. *For each normal non-terminal or interface non-terminal $N \in L$, create a new non-terminal $N\_List \in TL$, such that:*

```
N_List implements N = list [[ N ]];
```

For keywords, such a collection operator would match models supporting the same keywords at the same position several times. As this should be prevented in general [99], a collection operator for keywords is not supported.

4.3.6. Rule 6: Optional Operator

Optional occurrences of model elements are a specific refinement of collections that support zero or one of the specified model elements only. As optionality can be applied to keywords, this rule is separated into parts responsible for model elements and keywords also.

Rule 6a. *For each normal non-terminal or interface non-terminal $N \in L$, create a new non-terminal $N\_Opt \in TL$, such that:*

```
N_Opt implements N = opt [[ N ]];
```

Rule 6b. *For each relevant keyword $k \in L$, create a new non-terminal $K\_Opt \in TL$, such that:*

```
K_Opt implements K = opt [[ k ]];
```

With these rules in place, transformations can act upon optional model elements and keywords by matching the contained pattern at most one time. This supports transforming elements that might be present in a model.

4.3.7. Rule 7: Starting Rules for DSTLs

Finally, each DSTL requires a dedicated starting rule combining the modeling elements specific to the DSTL with DSTL-independent application constraints and assignments. To this end, we combine the interfaces derived via rules 2-6 into the disjunction `AlternativeOfNTs` and concatenate application constraints (`Where` clauses) as well as assignments (`Assign` block). Application constraints and assignments leverage expressions and statements imported from a language provided by the transformation framework.

Rule 7. *Create a non-terminal* `TFRule` *such that:*

$$\texttt{TFRule = (}AlternativeOfNTs\texttt{)}\star\ \texttt{Where?}\ \ \texttt{Assign?;}$$

This lifts all modeling elements of the base language to the DSTLs top level, which enables describing transformations beginning with nested elements of the base language. This enables defining transformations that match, for instance, ports of a component without having to model containing modeling elements of the host language. This allows for more efficient transformation modeling. Moreover, it enables specifying patterns over different modeling elements that do not share a common parent element. This especially enables addressing different models in a single transformation.

4.4. Deriving and Applying a Transformation Language

Figure 10 illustrates deriving and applying a transformation language using MontiTrans. MontiTrans is a generator that implements the derivation rules described in Section 4.3 and serves to demonstrate the applicability of our approach. It generates the DSTL grammar and the infrastructure to generate Java implementations for transformations described using the DSTL. MontiTrans thus automates the development of DSTLs for language developers and provides a generator for transformation developers to turn their transformations into executable Java code.

For language developers, developing a DSTL according to the derivation described in Section 4.3 consists of applying the MontiTrans generator and providing the resulting DSTL including the created generator to transformation developers.

To apply a transformation, it is translated into a Java class that realizes a pattern matching algorithm and is capable of modifying the model according to the modifications described by the transformation. To ease the application

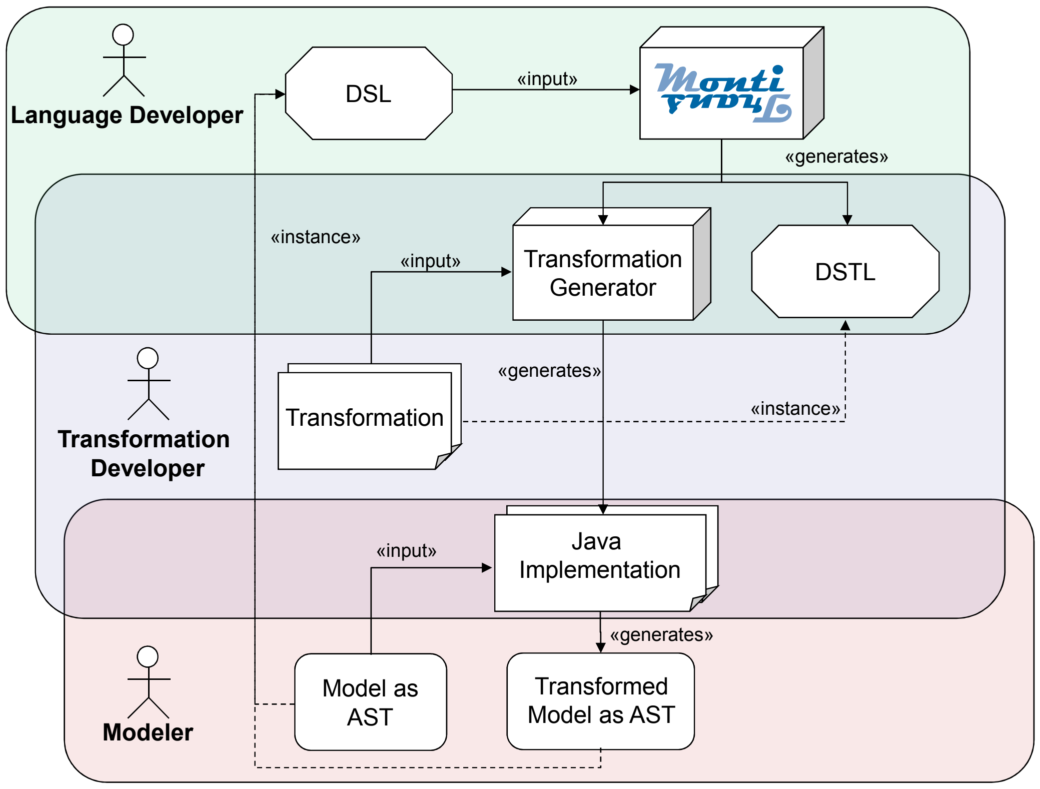


Figure 10: Overview of deriving and applying a transformation language using MontiTrans.

of generated transformation implementations they share a uniform interface as illustrated by Figure 11: Every generated transformation implementation provides a constructor to pass one or multiple models to the transformation. Furthermore, methods to match the pattern and modify the model are provided by every transformation: (1) the `doPatternMatching` method executes the pattern matching part of the transformation and return true in case the pattern was found, (2) the `doPatternReplacement` method can be used to modify the model after the pattern was matched, and (3) the `doAll` method first executes the pattern matching and afterwards modifies the model. Finally, for every schema variable defined within a transformation corresponding access methods are generated. These can be used to retrieve the matched elements from the transformation (`get`) or to predefine the pattern elements for the transformation (set). Thus, these methods turn schema

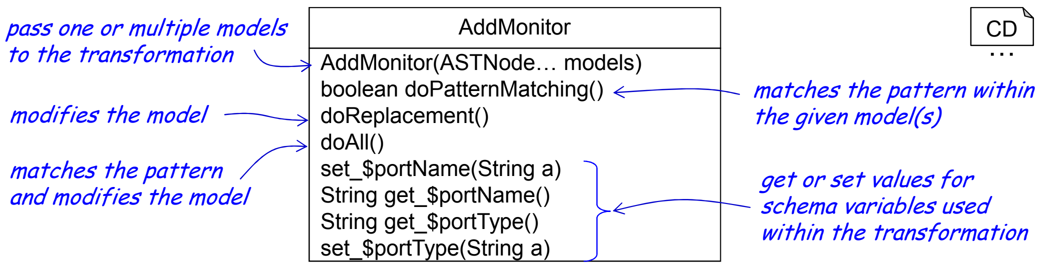

CD
AddMonitor
AddMonitor(ASTNode… models)
boolean doPatternMatching()
doReplacement()
doAll()
set_$portName(String a)
String get_$portName()
String get_$portType()
set_$portType(String a)
pass one or multiple models to the transformation
modifies the model
matches the pattern and modifies the model
matches the pattern within the given model(s)
get or set values for schema variables used within the transformation

Figure 11: Java class generated for the exemplary MontiArcTL transformation shown in Figure 5.

variables into parameters of a transformation.

Creating Java transformation implementations that share a uniform interface facilitates using transformations. Furthermore, these implementations can easily be provided as libraries and used within different projects, such as, for instance, different generators.

4.5. Deriving the MontiArcTL DSTL

Derivation of the MontiArcTL grammar begins with deriving transformation languages for the languages it extends. As MontiArc inherits from the `Types` language provided by MontiCore to describe types and literals, our automated transformation language derivation first derives the `TypesTL` transformation language (*cf.* Section 4.3.1). Afterwards, it derives the MontiArcTL grammars as depicted in Figure 12 (right). While the MontiArc grammar contains further elements (*e.g.,* to model components, ports, or connectors), these are not considered yet.

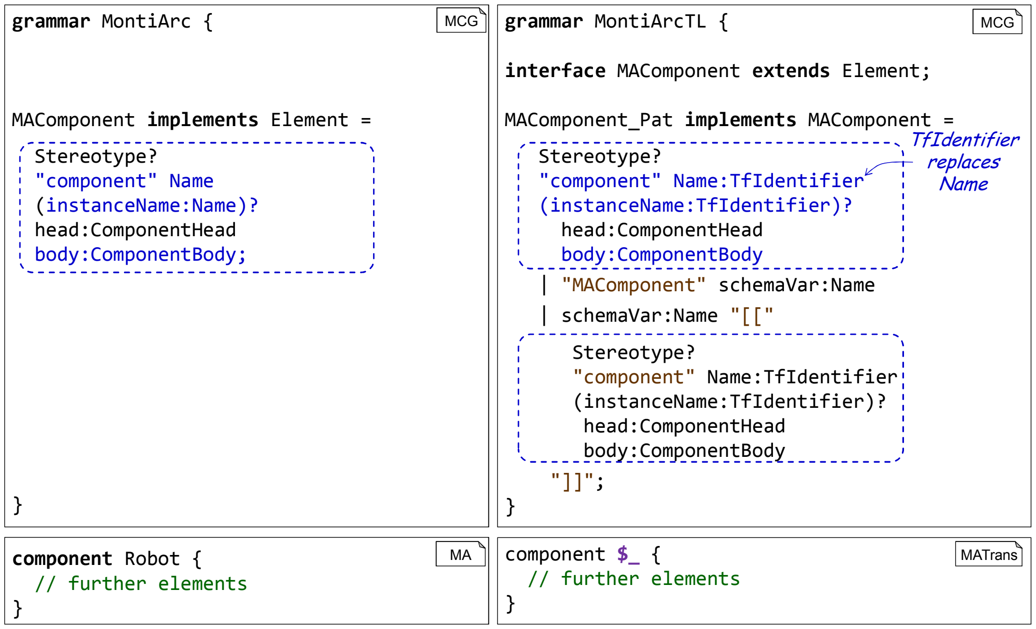
```
grammar MontiArc extends Types {                              MCG
  // further elements
}
```

```
grammar MontiArcTL extends TypesTL {                          MCG
}
```

Figure 12: Applying derivation rule 1.

The second derivation rule (*cf.* Section 4.3.2) iterates over interface non-terminals and normal non-terminals of the MontiArc grammar and creates an interface non-terminal of the same name in MontiArcTL for each. Relations between interface non-terminals and normal non-terminals are reflected by non-terminal extensions between the created interface non-terminals in MontiArcTL. Lifting non-terminals of the base language to interfaces in the transformation language enables providing different implementations (such as patterns) at the same place. Applying rule 2 also creates a first implementation for each new interface non-terminal in MontiArcTL: the pattern capable of matching the processed non-terminals. This pattern is a disjunction of three parts that enables matching either (1) the concrete syntax of the input non-terminal; (2) a schema variable matching a complete instance of the input non-terminal; or (3) a schema variable matching instances of this non-terminal with specific properties. To this end, it also replaces occurrences of names with instances of `TfIdentifier` to ensure schema variables can be used and replacements of names can be expressed.

For the input non-terminal `MAComponent` of MontiArc as depicted in Figure 13 (left), this results in the interface `MAComponent` and its pattern `MAComponent_Pat`. This non-terminal, for instance, enables matching the component `Robot` with the pattern depicted bottom-left in Figure 13 with the pattern depicted bottom-right.



```
grammar MontiArc {


MAComponent implements Element =
  Stereotype?
  "component" Name
  (instanceName:Name)?
  head:ComponentHead
  body:ComponentBody;

}
```

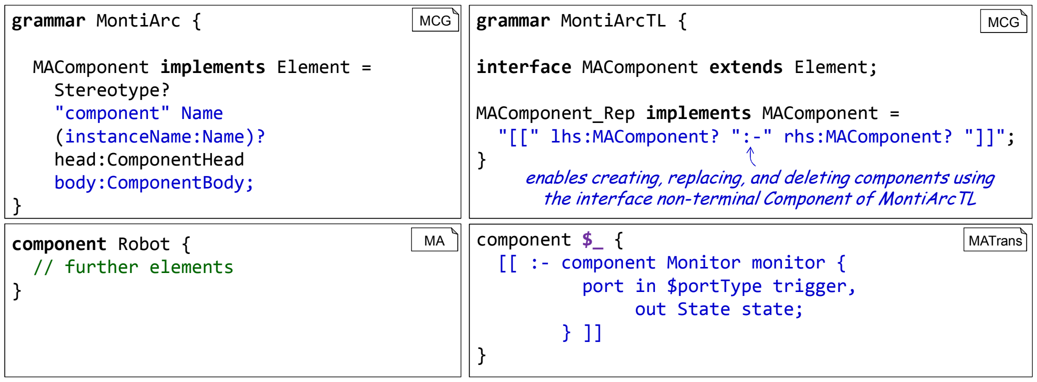
```
grammar MontiArcTL {

interface MAComponent extends Element;

MAComponent_Pat implements MAComponent =
  Stereotype?
  "component" Name:TfIdentifier
  (instanceName:TfIdentifier)?
    head:ComponentHead
    body:ComponentBody
  | "MAComponent" schemaVar:Name
  | schemaVar:Name "[["
      Stereotype?
      "component" Name:TfIdentifier
      (instanceName:TfIdentifier)?
       head:ComponentHead
       body:ComponentBody
    "]]";
}
```

```
component Robot {
  // further elements
}
```

```
component $_ {
  // further elements
}
```

Figure 13: Applying derivation rule 2.

To create, replace, or delete instances of non-terminals, the third derivation rule (*cf.* Section 4.3.3) creates a replacement operator for each normal non-terminal and interface non-terminal of MontiArc as depicted in Figure 14: for the non-terminal `MAComponent` (top left) derivation creates the non-terminal `MAComponent_Rep` (top right). This rule introduces a replacement pattern that supports omitting its left-hand side or its right-hand side, and, thus, enables creating or deleting instances of the `MAComponent` non-terminal. To enable using this pattern in places where components can be matched, the non-terminal `MAComponent_Rep` also becomes an implementation of MontiArcTL's `MAComponent` interface. Applying this pattern to component `Robot` (bottom left) enables, for instance, integrating a new subcomponent to take care of monitoring the `Robot`'s behavior (bottom right).


MCG
MCG
MA
MATrans
enables creating, replacing, and deleting components using the interface non-terminal Component of MontiArcTL

```
grammar MontiArc {

  MAComponent implements Element =
    Stereotype?
    "component" Name
    (instanceName:Name)?
    head:ComponentHead
    body:ComponentBody;
}
```

```
grammar MontiArcTL {

interface MAComponent extends Element;

MAComponent_Rep implements MAComponent =
  "[[" lhs:MAComponent? ":-" rhs:MAComponent? "]]";
}
```

```
component Robot {
  // further elements
}
```

```
component $_ {
  [[ :- component Monitor monitor {
          port in $portType trigger,
               out State state;
        } ]]
}
```

Figure 14: Applying derivation rule 3.

Similarly, the derivation rules 4-6 (*cf.* Section 4.3.4 - 4.3.6) create new non-terminals to negate patterns, list pattern instances, and handle optionality in MontiArcTL, respectively. For MontiArc's non-terminal `MAComponent`, this produces the three rules `MAComponent_Neg`, `MAComponent_List`, and `MAComponent_Opt` as depicted in Figure 15 (bottom right).

After applying these rules, MontiArcTL contains all patterns derived from the interface non-terminals and normal non-terminals of MontiArc. To enable specifying any patterns, derivation rule 7 (*cf.* Section 4.3.7) creates the `TFRule` starting non-terminal for MontiArcTL. `TFRule` is an iteration of disjunctions over all interfaces derived from MontiArc's non-terminals followed by an assignments block and a where clause. This non-terminal enables describing transformations over arbitrary many patterns, restricting their appearances through the where clause if necessary, and computing complex calculations without polluting the patterns.

MA

```
component Robot {
  component Monitor monitor {
    port in Trigger trigger,
         out State state;
  }
}
```

MATrans

```
component $_ {
  not [[component Monitor monitor { } ]]
  list [[component $_ { } ]]
  opt [[component Logger log { } ]]

}
```

MCG

```
grammar MontiArc {

  MAComponent implements Element =
    Stereotype?
    "component" Name
    (instanceName:Name)?
    head:ComponentHead
    body:ComponentBody;

}
```

MCG

```
grammar MontiArcTL {

interface MAComponent extends Element;

MAComponent_Neg implements MAComponent =  // Rule 4
  "not" "[[" MAComponent "]]";

MAComponent_List implements MAComponent = // Rule 5
  "list" "[[" MAComponent "]]";

MAComponent_Opt implements MAComponent =  // Rule 6
  "opt" "[[" MAComponent "]]";
}
```

Figure 15: Applying derivation rules 4, 5, and 6.

4.6. Related Language Derivation Mechanisms

There are various approaches to engineer domain-specific transformation languages. These, however, provide semi-automated support only [82, 83, 100, 101] or require fully manual language engineering [84, 102, 103]

Semi-automated approaches often derive pattern languages based on the input languages' metamodels [82, 83]. While this enables specifying the left-hand side and right-hand side of the transformation, the DSTL developer must provide a concrete syntax manually (*e.g.,* through an editor). Similarly, there are approaches to derive graphical DSTLs from graphical DSLs [100, 101]. Here, concrete syntax and abstract syntax must be integrated manually as well.

Manual engineering of DSTLs often begins bottom-up, such that complex DSTLs are composed of individual transformation modules [102, 103]. Lacking automated integration, manually engineering DSTLs from modules is generally as complex as engineering new DSTLs from scratch. A tool to overcome this challenge is AToMPM [104, 105], which supports automated derivation of graphical DSTLs only. The ideas presented in [106] also support automated derivation of DSTLs. An implementation is yet to be presented. Another approach is presented in [48] here a language definition can be accompanied by a set of rewriting rules. These rewriting rules provide a similar functionality as a derived transformation language such as rewrit-

```
grammar MontiArc {                                    MCG

  MAComponent implements Element = /*…*/;
  Port implements Element = /*…*/;
  Connector implements Element = /*…*/;
}
```

```
grammar MontiArcTL {                                  MCG

  TFRule =
    (MAComponent | Port | Connector | /*…*/ )*
    TFAssignments? TFWhere?;

  // Inherited from TFCommons
  TFAssignments = "assign" "{" Assign* "}";

  Assign =
    variable:Name "=" value:Expression ";" ;

  TFWhere =
    "where" "{" constraint:Expression "}";
}
```

```
component Robot {                                     MA
  // further elements
}
```

```
component $name {                                     MATrans
}

assign { $name = $name + "Monitored"; }
where { !$name.contains("Monitored") }
```

Figure 16: Applying derivation rule 7.

ing for patterns, list operator or conditional rewriting rules described using the languages concrete syntax. However, the intention of the approach presented in [48] differs as the underlying assumption is that all models of a language are treated similarly and transformations are used to compile, i.e., generate the corresponding code and thus rewriting rules can be developed in conjunction with the grammar. In contrast, our approach not only addresses compiler or generator developers but in addition modelers that should be able to express transformations for their models, e.g. to refactor or systematically update their models. Transformations can be developed independently of the grammar and thus be added without changing the grammar. It is also possible to develop different sets of transformations for different target platforms [98]. Furthermore, the generated code is Java code instead of C code, which integrates nicely with the MontiCore language workbench. The generated pattern matching process is realized by graph transformation rather than term rewriting and not limited to one input model. Matches for patterns can even be distributed over several input models.

Other uses for language derivation are tagging, delta-modeling, and specification of edit operations. Tagging languages [61], for instance, enable attaching information (tags) to models of the language the tagging language is derived from. This enables functionality similar to stereotypes but prevents polluting the base model. Delta-modeling languages enable describing transformations for specific models using the base language's syntax [107, 60]. In contrast to transformations, deltas rarely are generalizable to sets of models. Similarly, model-editing language [108, 109] derive syntax-preserving edit operations for models of the base language.

Generating transformations from example models is another way to automatically derive transformations. However, these do not lead to reusable DSTLs [110, 111, 112, 113, 114, 115].

5. Discussion

Research in software language engineering investigates the efficient engineering, deployment, use, and evolution of software languages to support software engineers and domain experts to efficiently model future systems. DSLs foster innovation and efficiency in software engineering. They have become crucial innovation drivers in many disciplines, including Automotive [116], Avionics [16], Civil Engineering [117], Industry 4.0 [118], Robotics [119], and Software Engineering.

Yet, many successful DSLs are engineered ad-hoc and proprietary. Without concentrated research on foundations, concepts, methods, and tools for software language engineering, software engineering practitioners and researchers will be unable to leverage these benefits to deploy the future's smart, distributed, cyber-physical systems. The adoption of DSLs raises three challenges:

1. The lack of commonly accepted foundations of software languages that practitioners and researchers of different domains can use, rely, and advance upon;

2. The systematic reuse of DSLs and DSL components to efficiently engineer new languages from existing ones;

3. The coordinated use of multiple DSLs each handling specific system aspects that enable domain experts to contribute their expertise to complex projects; and

Regarding commonly accepted foundations for SLE, there are some agreements in some schools of thoughts. These include, for instance, agreeing on the conceptual constituents of a software language, such as concrete syntax, abstract syntax, static semantics, and dynamic semantics [18], as well as agreeing on specific implementation technologies, such as EMF ECore [39]. While the former leaves much room for interpretation, the latter only touches the abstract syntax. Commonly accepted technological foundations for implementation of concrete syntax, static semantics, and dynamic semantics are yet to be identified. Consequently, many language workbenches stipulate different notions of language components and feature different implementation technologies [33].

Flexible reuse of DSLs and DSL components from different technological spaces to construct new languages would greatly benefit from commonly accepted foundations. Without these, reuse is restricted to specific technological spaces, such as between ECore-based languages or MontiCore languages. In these spaces, different approaches to reuse languages – either through composition [51], integration [18], or derivation (*cf.* Section 4.3) – have been achieved. The derivation mechanism explained in this paper focusses on transformation languages and illustrates that deriving languages from another might be a worthwhile investigation as well. It has been applied to deriving delta languages [60] and tagging languages [61] as well, but its generalization requires further investigation.

Recently, concepts of reuse for DSLs have been presented that operate on staging different reuse mechanisms according to the VCU [120] (variability, customization, use) model [121]. While these can support reuse in the same technological space, truly flexible reuse of DSLs and DSL components across language workbench boundaries requires for efficiently bridging their respective technological spaces or agreeing on a "lingua franca" for DSL definition – similar to many general-purpose programming languages being compatible to the JVM (*e.g.,* Scala, Groovy).

The efficient coordinated use of multiple DSLs for different system aspects currently is successful for apriori integrated language families (such as the UML [3] or SysML [122] languages) only. For heterogeneous, independent languages, their coordinated reuse still requires comprehensive efforts to integrate both the DSLs and related software tools. Research in solutions to the coordinated execution of models focuses on exchanging symbols between the DSLs and having a mechanism to control the flow of symbols between models of the different DSLs [62]. Coordinated use of DSLs would require an understanding of possible DSL manifestations that is yet to be achieved.

6. Conclusion

Ludwig Wittgenstein once postulated that the limits of his language are the limits of his world. Modern programming languages are suitable to describe structure, operations, and data, while general-purpose modeling languages, such as UML, are suitable for specifying structure, architecture, and behavior of software systems. However, both kinds of languages suffer from being designed for software engineers and raise a gap between the problem domains (such as medicine, physics, robotics) and the solution domain of software engineering.

Software is eating the world and domains are being digitalized with increasing velocity. Consequently, an increasing number of non software experts (such as physicists, mechanical engineers, or even lawyers) have to manage to encode their information, knowledge, methods, and procedures digitally. Thus, providing suitable languages to these domain experts is crucial. This includes models of various unforeseen forms and calls for a strong and active field of software language engineering.

Software language engineering envisions a systematic way of developing language components, integrating and composing these into larger languages, modifying and extending these as desired, as well as to facilitate the evolution of digitalized domains. We discussed these language engineering techniques on three levels: domain-specific transformation models support domain experts in modifying their models without being forced to learn overly generic transformation languages (level 1). The DSTL itself is generated based on a DSL using a language derivation mechanism (level 2). This, in turn, is engineered by means of a grammar-based language workbench – in this case, MontiCore (level 3). Only on the level of language workbenches, language engineering techniques become feasible.

Despite these principles being understood to some extent, it still takes more time and joint research efforts to industrially capitalize on these advances.

References

[1] C. J. Date, H. Darwen, A Guide to the SQL Standard, Vol. 3, Addison-Wesley New York, 1987.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau,

Extensible markup language (XML), World Wide Web Journal 2 (4) (1997) 27–66.

[3] Object Management Group, Unified Modeling Language (UML), Version 2.5, http://www.omg.org/spec/UML/2.5/PDF/, [Online; Accessed: 12.10.2017] (2015).

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

[5] B. Rumpe, Agile Modeling with UML: Code Generation, Testing, Refactoring, Springer International, 2017.

[6] R. France, B. Rumpe, Model-driven Development of Complex Software: A Research Roadmap, Future of Software Engineering (FOSE '07) (2) (2007) 37–54.

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

[8] P. Hudak, Domain Specific Languages, Handbook of programming languages 3 (39-60) (1997) 21.

[9] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, G. Wachsmuth, DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, dslbook.org, 2013.

[10] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: An annotated bibliography, ACM Sigplan Notices 35 (6) (2000) 26–36.

[11] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, Addison-Wesley, 2008.

[12] L. Hermerschmidt, K. Hölldobler, B. Rumpe, A. Wortmann, Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions, in: Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15), Vol. 1463 of CEUR Workshop Proceedings, 2015, pp. 18–23.

[13] K. Hölldobler, A. Roth, B. Rumpe, A. Wortmann, Advances in Modeling Language Engineering, in: International Conference on Model and Data Engineering, Springer, 2017, pp. 3–17.

[14] C. Atkinson, T. Kuhne, Model-driven development: a metamodeling foundation, IEEE software 20 (5) (2003) 36–41.

[15] B. Selic, The pragmatics of model-driven development, IEEE software 20 (5) (2003) 19–25.

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

[17] J. Abell, MATLAB and SIMULINK. Modeling Dynamic Systems, CreateSpace Independent Publishing Platform, 2016.

[18] T. Clark, M. G. J. van den Brand, B. Combemale, B. Rumpe, Conceptual Model of the Globalization for Domain-Specific Languages, in: Globalizing Domain-Specific Languages, Springer, 2015, pp. 7–20.

[19] D. Harel, B. Rumpe, Meaningful Modeling: What's the Semantics of "Semantics"?, IEEE Computer 37 (10) (2004) 64–72.

[20] J.-M. Favre, D. Gasevic, R. Lämmel, E. Pek, Empirical language analysis in software linguistics., in: SLE, Springer, 2010, pp. 316–326.

[21] D. E. Knuth, Semantics of context-free languages, Theory of Computing Systems 2 (2) (1968) 127–145.

[22] P. Klint, R. Lämmel, C. Verhoef, Toward an Engineering Discipline for Grammarware, ACM Transactions on Software Engineering and Methodology (TOSEM) 14 (3) (2005) 331–380.

[23] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, MontiCore: A Framework for the Development of Textual Domain Specific Languages, in: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume, 2008, pp. 925–926.

[24] H. Krahn, B. Rumpe, S. Völkel, MontiCore: a Framework for Compositional Development of Domain Specific Languages, International Journal on Software Tools for Technology Transfer (STTT) 12 (5) (2010) 353–372.

[25] G. H. Wachsmuth, G. D. P. Konat, E. Visser, Language Design with the Spoofax Language Workbench, IEEE Software 31 (5) (2014) 35–43.

[26] S. Ellner, W. Taha, The semantics of graphical languages, in: PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, ACM Press, New York, NY, USA, 2007, pp. 122–133.

[27] M. Völter, K. Solomatov, Language modularization and composition with projectional language workbenches illustrated with MPS, Software Language Engineering, SLE 16 (2010) 3.

[28] H. Grönniger, B. Rumpe, Modeling Language Variability, in: Workshop on Modeling, Development and Verification of Adaptive Systems, LNCS 6662, Springer, 2011, pp. 17–32.

[29] M. Richters, M. Gogolla, On formalizing the UML object constraint language OCL, ER 98 (1998) 449–464.

[30] J.-M. Jézéquel, O. Barais, F. Fleurey, Model Driven Language Engineering with Kermeta, GTTSE 9 (2009) 201–221.

[31] K. Czarnecki, Generative Programming-Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models, Ph.D. thesis, Technical University of Ilmenau (1998).

[32] T. Mens, P. van Gorp, A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science 152 (2006) 125 – 142.

[33] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al., Evaluating And Comparing Language Workbenches: Existing Results And Benchmarks For The Future, Computer Languages, Systems & Structures 44 (2015) 24–47.

[34] B. Combemale, J. Deantoni, O. Barais, A. Blouin, E. Bousse, C. Brun, T. Degueule, D. Vojtisek, A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio, in: 8th Transformation Tool Contest, CEUR, 2015.

[35] E. Vacchi, W. Cazzola, Neverlang: A framework for feature-oriented language development, Computer Languages, Systems & Structures 43 (2015) 1–40.

[36] B. Rumpe, K. Hölldobler, MontiCore 5 Language Workbench. Edition 2017, Aachener Informatik-Berichte, Software Engineering Band 32, Shaker Verlag, 2017.

[37] P. Klint, T. van der Storm, J. Vinju, RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation, in: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE, 2009, pp. 168–177.

[38] L. Kats, E. Visser, The Spoofax Language Workbench, in: W. R. Cook (Ed.), Proceedings of the ACM international conference on Object oriented programming systems languages and applications, ACM, 2010, p. 444.

[39] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.

[40] M. Eysholdt, H. Behrens, 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, SPLASH '10, ACM, New York, NY, USA, 2010, pp. 307–309.

[41] V. Viyović, M. Maksimović, B. Perisić, Sirius: A rapid development of DSM graphical editor, in: Intelligent Engineering Systems (INES), 2014 18th International Conference on, IEEE, 2014, pp. 233–238.

[42] P. Deransart, M. Jourdan, B. Lorho, Attribute grammars: definitions, systems and bibliography, Vol. 323, Springer Science & Business Media, 1988.

[43] M. Mernik, V. Žumer, Incremental programming language development, Computer Languages, Systems & Structures 31 (1) (2005) 1–16.

[44] I. F. Jr., T. Kosar, I. Fister, M. Mernik, Easytime++: A case study of incremental domain-specific language development, ITC 42 (1) (2013) 77–85.

[45] M. Challenger, M. Mernik, G. Kardas, T. Kosar, Declarative specifications for the development of multi-agent systems, Computer Standards & Interfaces 43 (2016) 91–115.

[46] L. Bettini, Implementing domain-specific languages with Xtext and Xtend, Packt Publishing Ltd, 2016.

[47] J. Radjenovic, B. Milosavljevic, D. Surla, Modelling and implementation of catalogue cards using FreeMarker, Program 43 (1) (2009) 62–76.

[48] M. G. J. van den Brand, J. Heering, P. Klint, P. A. Olivier, Compiling language definitions: the ASF+ SDF compiler, ACM Transactions on Programming Languages and Systems (TOPLAS) 24 (4) (2002) 334–368.

[49] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, A. Wortmann, Composition of Heterogeneous Modeling Languages, in: Model-Driven Engineering and Software Development, Vol. 580 of Communications in Computer and Information Science, Springer, 2015, pp. 45–66.

[50] J. O. Ringert, A. Roth, B. Rumpe, A. Wortmann, Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems, Journal of Software Engineering for Robotics (JOSER) 6 (1) (2015) 33–57.

[51] S. Erdweg, P. G. Giarrusso, T. Rendel, Language Composition Untangled, in: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, LDTA '12, ACM, New York, NY, USA, 2012.

[52] M. Mernik, An object-oriented approach to language compositions for software language engineering, Journal of Systems and Software 86 (9) (2013) 2451–2464.

[53] C. Berger, B. Rumpe, Engineering Autonomous Driving Software, in: C. Rouff, M. Hinchey (Eds.), Experience from the DARPA Urban Challenge, Springer, Germany, 2012, pp. 243–271.

[54] A. Navarro Pérez, B. Rumpe, Modeling Cloud Architectures as Interactive Systems, in: Model-Driven Engineering for High Performance

and Cloud Computing Workshop, Vol. 1118 of CEUR Workshop Proceedings, 2013, pp. 15–24.

[55] T. Kurpick, M. Look, C. Pinkernell, B. Rumpe, Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings, in: Modelling of the Physical World Workshop (MOTPW'12), ACM, 2012, pp. 2:1–2:6.

[56] J. Whittle, J. Hutchinson, M. Rouncefield, The State of Practice in Model-Driven Engineering, Software, IEEE 31 (3) (2014) 79–85.

[57] W. Iverson, Hibernate: A J2EE (TM) Developer's Guide, Addison-Wesley Professional, 2004.

[58] F. P. Miller, A. F. Vandome, J. McBrewster, Apache Maven, Alpha Press, 2010.

[59] P. Clements, L. Northrop, Software Product Lines, Addison-Wesley,, 2002.

[60] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, I. Schaefer, C. Schulze, Systematic Synthesis of Delta Modeling Languages, Journal on Software Tools for Technology Transfer (STTT) 17 (5) (2015) 601–626.

[61] T. Greifenberg, M. Look, S. Roidl, B. Rumpe, Engineering Tagging Languages for DSLs, in: Conference on Model Driven Engineering Languages and Systems (MODELS'15), ACM/IEEE, 2015, pp. 34–43.

[62] M. E. V. Larsen, J. Deantoni, B. Combemale, F. Mallet, A behavioral coordination operator language (BCOoL), in: Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on, IEEE, 2015, pp. 186–195.

[63] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Science of computer programming 72 (1-2) (2008) 31–39.

[64] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Melange: A Meta-language for Modular and Reusable Development of DSLs, in: 8th International Conference on Software Language Engineering (SLE), Pittsburgh, United States, 2015, pp. 25–36.

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

[66] A. Johnstone, E. Scott, M. G. J. van den Brand, Modular Grammar Specification, Science of Computer Programming 87 (2014) 23 – 43.

[67] L. V. Reis, V. O. D. Iorio, R. S. Bigonha, An on-the-fly grammar modification mechanism for composing and defining extensible languages, Computer Languages, Systems & Structures 42 (2015) 46 – 59, special issue on the Programming Languages track at the 29th ACM Symposium on Applied Computing.

[68] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, J. Vinju, Modular language implementation in Rascal - experience report, Science of Computer Programming 114 (2015) 7 – 19, LDTA (Language Descriptions, Tools, and Applications) Tool Challenge.

[69] A. M. Şutî, M. G. J. van den Brand, T. Verhoeff, Exploration of modularity and reusability of domain-specific languages: an expression DSL in MetaMod, Computer Languages, Systems & Structures 51 (2018) 48 – 70.

[70] H. Kern, Model Interoperability between Meta-Modeling Environments by using M3-Level-Based Bridges, Ph.D. thesis, University of Leipzig (2016).

[71] G. Kardas, E. Bircan, M. Challenger, Supporting the Platform Extensibility for the Model-Driven Development of Agent Systems by the Interoperability Between Domain-Specific Modeling Languages of Multi-Agent Systems., Computer Science and Information System 14 (2017) 875–912.

[72] J. M. Gascueña, E. Navarro, A. Fernández-Caballero, R. Martínez-Tomás, Model-to-model and Model-to-text: Looking for the Automation of VigilAgent, Expert Systems 31 (3) (2014) 199–212.

[73] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, E. Visser, Disambiguation Filters for Scannerless Generalized LR Parsers, in: R. N. Horspool (Ed.), Compiler Construction, Springer Berlin Heidelberg, 2002, pp. 143–158.

[74] J. Deantoni, Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination, in: Architecture-Centric Virtual Integration (ACVI), 2016, IEEE, 2016, pp. 12–18.

[75] T. Parr, The definitive ANTLR 4 reference, Pragmatic programmers, The Pragmatic Programmers, 2014.

[76] K. Hölldobler, B. Rumpe, I. Weisemöller, Systematically Deriving Domain-Specific Transformation Languages, in: Conference on Model Driven Engineering Languages and Systems (MODELS'15), ACM/IEEE, 2015, pp. 136–145.

[77] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on software engineering 26 (1) (2000) 70–93.

[78] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, A. Tang, What Industry Needs from Architectural Languages: A Survey, Software Engineering, IEEE Transactions on 39 (6) (2013) 869–891.

[79] D. Garlan, R. T. Monroe, D. Wile, Acme: Architectural Description of Component-Based Systems, Foundations of component-based systems 68 (2000) 47–68.

[80] A. Butting, A. Haber, L. Hermerschmidt, O. Kautz, B. Rumpe, A. Wortmann, Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language, in: Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017, Springer International Publishing, 2017, pp. 53–70.

[81] B. Rumpe, I. Weisemöller, A Domain Specific Transformation Language, in: Workshop on Models and Evolution (ME), 2011.

[82] T. Baar, J. Whittle, On the Usage of Concrete Syntax in Model Transformation Rules, in: Perspectives of Systems Informatics, LNCS, Springer, 2007, pp. 84–97.

[83] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, M. Wimmer, Explicit Transformation Modeling, in: Models in Software Engineering, Vol. 6002 of LNCS, Springer, 2010, pp. 240–255.

[84] E. Syriani, J. Gray, H. Vangheluwe, Modeling a Model Transformation Language, in: I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, J. Bettin (Eds.), Domain Engineering, Springer, 2013, pp. 211–237.

[85] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Systems Journal 45 (3) (2006) 621–645.

[86] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations, in: Proceedings of MoDELS'10, Vol. 6394 of LNCS, Springer, 2010, pp. 121–135.

[87] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, M. Tichy, Henshin: A Usability-Focused Framework for EMF Model Transformation Development, Springer, 2017, pp. 196–208.

[88] E. Leblebici, A. Anjorin, A. Schürr, Developing eMoflon with eMoflon, in: International Conference on Theory and Practice of Model Transformations, Springer, 2014, pp. 138–145.

[89] A. Schürr, A. J. Winter, A. Zündorf, Graph grammar engineering with PROGRES, in: W. Schäfer (Ed.), Software Engineering - ESEC '95, Vol. 989 of LNCS, Springer, 1995, pp. 219–234.

[90] L. Geiger, A. Zündorf, Tool Modeling with Fujaba, Electronic Notes in Theoretical Computer Science 148 (1) (2006) 173–186.

[91] R. Grønmo, S. Krogdahl, B. Møller-Pedersen, A Collection Operator for Graph Transformation, Software & Systems Modeling 12 (1) (2013) 121–144.

[92] C. Fuss, V. E. Tuttlies, Simulating Set-Valued Transformations with Algorithmic Graph Transformation Languages, in: A. Schürr, M. Nagl, A. Zündorf (Eds.), Applications of graph transformations with industrial relevance, Vol. 5088 of LNCS, Springer, 2008, pp. 442–455.

[93] M. Minas, B. Hoffmann, An Example of Cloning Graph Transformation Rules for Programming, Electronic Notes in Theoretical Computer Science 211 (2008) 241–250.

[94] D. Balasubramanian, A. Narayanan, S. Neema, F. Shi, R. Thibodeaux, G. Karsai, A Subgraph Operator for Graph Transformation Languages, Electronic Communications of the EASST 6.

[95] A. Rensink, Nested Quantification in Graph Transformation Rules, in: A. Corradini (Ed.), Graph Transformation, Vol. 4178 of LNCS, Springer, 2006, pp. 1–13.

[96] B. Hoffmann, D. Janssens, N. van Eetvelde, Cloning and Expanding Graph Transformation Rules for Refactoring, Electronic Notes in Theoretical Computer Science 152 (2006) 53–67.

[97] J. de Lara, C. Ermel, G. Taentzer, K. Ehrig, Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets, Electronic Notes in Theoretical Computer Science 109 (2004) 17–29.

[98] K. Adam, K. Hölldobler, B. Rumpe, A. Wortmann, Modeling Robotics Software Architectures with Modular Model Transformations, Journal of Software Engineering for Robotics (JOSER) 8 (1) (2017) 3–16.

[99] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, S. Völkel, Design Guidelines for Domain Specific Languages, in: Domain-Specific Modeling Workshop (DSM'09), Techreport B-108, Helsinki School of Economics, 2009, pp. 7–13.

[100] R. Grønmo, Using concrete syntax in graph-based model transformations, Ph.D. thesis, University of Oslo (2009).

[101] R. Grønmo, B. Møller-Pedersen, Concrete Syntax-based Graph Transformation, research Report 389 (2009).

[102] E. Syriani, H. Vangheluwe, B. LaShomb, T-Core: a framework for custom-built model transformation engines, Software & Systems Modeling (2013) 1–29.

[103] J. Sánchez Cuadrado, E. Guerra, J. de Lara, Towards the Systematic Construction of Domain-Specific Transformation Languages, in:

Modelling Foundations and Applications, Vol. 8569 of LNCS, Springer, 2014, pp. 196–212.

[104] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. van Mierlo, H. Ergin, AToMPM: A Web-based Modeling Environment, in: MODELS'13: Invited Talks, Demos, Posters, and ACM SRC, 2013, pp. 21–25.

[105] J. Corley, E. Syriani, H. Ergin, Evaluating the cloud architecture of AToMPM, in: 2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2016, pp. 339–346.

[106] T. Reiter, E. Kapsammer, W. Retschitzegger, W. Schwinger, M. Stumptner, A Generator Framework for Domain-Specific Model Transformation Languages, in: 8th International Conference on Enterprise Information Systems (ICEIS), 2006, pp. 27–35.

[107] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, I. Schaefer, Engineering Delta Modeling Languages, in: Software Product Line Conference (SPLC'13), ACM, 2013, pp. 22–31.

[108] M. Rindt, T. Kehrer, U. Kelter, Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools, in: Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), 2014, pp. 35–39.

[109] T. Kehrer, G. Taentzer, M. Rindt, U. Kelter, Automatically Deriving the Specification of Model Editing Operations from Meta-Models, in: Theory and Practice of Model Transformations: 9th International Conference, ICMT 2016, Springer, 2016, pp. 173–188.

[110] Y. Sun, J. Gray, J. White, A demonstration-based model transformation approach to automate model scalability, Software & Systems Modeling 14 (3) (2015) 1245–1271.

[111] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer, Model Transformation By-Example: A Survey of the First Wave, in: Conceptual Modelling and Its Theoretical Foundations, Vol. 7260 of LNCS, Springer, 2012, pp. 197–215.

[112] P. Langer, M. Wimmer, G. Kappel, Model-to-Model Transformations By Demonstration, in: L. Tratt, M. Gogolla (Eds.), Theory and Practice of Model Transformations, Vol. 6142 of LNCS, Springer, 2010, pp. 153–167.

[113] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, W. Schwinger, An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example, in: A. Schürr, B. Selic (Eds.), Model Driven Engineering Languages and Systems: 12th International Conference, Springer, 2009, pp. 271–285.

[114] Y. Sun, J. White, J. Gray, Model Transformation by Demonstration, in: A. Schürr, B. Selic (Eds.), Model Driven Engineering Languages and Systems: 12th International Conference, Springer, 2009, pp. 712–726.

[115] M. G. Nanda, S. Mani, V. S. Sinha, S. Sinha, Demystifying Model Transformations: An Approach Based on Automated Rule Inference, in: S. Arora (Ed.), Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, ACM, 2009, p. 341.

[116] P. Cuenot, P. Frey, R. Johansson, H. Lönn, M.-O. Reiser, D. Servat, R. T. Koligari, D. Chen, Developing Automotive Products Using the EAST-ADL2, an AUTOSAR Compliant Architecture Description Language, in: Embedded Real-Time Software Conference, Toulouse, France, Vol. 40, Citeseer, 2008.

[117] S. Zhang, J. Teizer, J.-K. Lee, C. M. Eastman, M. Venugopal, Building information modeling (BIM) and safety: Automatic safety checking of construction models and schedules, Automation in Construction 29 (2013) 183–195.

[118] A. Wortmann, B. Combemale, O. Barais, A Systematic Mapping Study on Modeling for Industry 4.0, in: Conference on Model Driven Engineering Languages and Systems (MODELS'17), IEEE, 2017, pp. 281–291.

[119] A. Nordmann, N. Hochgeschwender, S. Wrede, A survey on domain-specific languages in robotics, in: International Conference on Simula-

tion, Modeling, and Programming for Autonomous Robots, Springer, 2014, pp. 195–206.

[120] J. Kienzle, G. Mussbacher, O. Alam, M. Schöttle, N. Belloir, P. Collet, B. Combemale, J. Deantoni, J. Klein, B. Rumpe, VCU: The Three Dimensions of Reuse, in: Conference on Software Reuse (ICSR'16), Vol. 9679 of LNCS, Springer, 2016, pp. 122–137.

[121] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, A. Wortmann, Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering, Computer Languages, Systems & Structures 54 (2018) 139–155.

[122] S. Friedenthal, A. Moore, R. Steiner, A Practical Guide to SysML: The Systems Modeling Language, Morgan Kaufmann, 2014.