JOT JOURNAL OF OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

[BEH+20] A. Butting, R. Eikermann, K. Hölldobler, N. Jansen, B. Rumpe, A. Wortmann:
A Library of Literals, Expressions, Types, and Statements for Compositional Language Design.
In: Journal of Object Technology, 19(3), pp. 3:1-16, AITO, Oct. 2020.
www.se-rwth.de/publications/

A Library of Literals, Expressions, Types, and Statements for Compositional Language Design

Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann
RWTH Aachen University, Software Engineering, Aachen, Germany

ABSTRACT Many modeling languages share common concepts, such as types, expressions, statements, or literals. Nonetheless, these essential language concerns are often developed again and again, independently, and with minor changes here and there. A well-designed and extensible library of such, tried-and-tested, language components can facilitate engineering new languages through their reuse. We utilize the powerful language composition techniques of the MontiCore language workbench to conceive such a library, present its components, and their rationale. This fosters language engineering in MontiCore and can inspire more efficient language engineering in the technological spaces of other language workbenches.

KEYWORDS Software Language Engineering; Language Reuse; Language Modularity

1. Introduction

Model-driven development (MDD) (Selic 2003; Völter et al. 2013) lifts models to primary modeling artifacts to reduce the conceptual gap (FR07 2007) between the problem domain challenges and solution domain implementations. Automation is one of the pillars of efficient software engineering and to make models accessible to automation, they must conform to machine-processable modeling languages, such as the Unified Modeling Language (UML) (Rum16 2016), Object Constraint Language (OCL) (Richters & Gogolla 2000), Systems Modeling Language (SysML) (Friedenthal et al. 2014), MATLAB Simulink (Dabney & Harman 2004) and many more. This need gave birth to the discipline of software language engineering (SLE) (Kleppe 2008; HRW18 2018), which investigates the conception, engineering, maintenance, and evolution of modeling languages. Another pillar of efficient software engineering is the capability to reuse software parts successfully tried and tested in different contexts, to create new solutions on the shoulders of giants. As "software languages are software too" (Favre 2005), they can greatly benefit from reusing tried and tested software language parts as well.

There are various kinds or categories of modeling languages that either directly or indirectly depend on each other, such as SysML extending UML. However, this extension is codified in natural language standards documents, but not technically through reusing UML parts in SysML. Consequently, every implementation of, in this case, SysML, needs to make sure to conform to the UML parts it is – by way of standards – meant to reuse or extend. Moreover, most of the tooling, such as analysis, syntheses, and editors, available for UML needs to be reimplemented for SysML as well. This, of course, also holds for the variety of different variants of UML (Dévai et al. 2014; Rum16 2016; Starrett 2016), OCL (Cabot & Gogolla 2012; Demuth & Wilke 2009; Rum16 2016), and SysML (WBCW20 2020; Wolny et al. 2020).

Other kinds of languages share concepts more loosely, but could benefit from explicit reuse of language parts (and corresponding tooling) as well. For instance, there are over 120 architecture description languages (ADLs) (Medvidovic & Taylor 2000), such as Architecture Analysis & Design Language (Feiler & Gluch 2012) or EAST-ADL (Etzel & Bauer 2019), out of which many share the language concepts of components, ports, and connectors (Malavolta et al. 2013). Similarly, many action languages and programming languages relying on the Java Virtual Machine, such as Clojure (Halloway 2009), Groovy (Koenig et al. 2007), Kotlin (Jemerov & Isakova 2017),

JOT reference format:
Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. *A Library of Literals, Expressions, Types, and Statements for Compositional Language Design*. Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution - No Derivatives 4.0 International (CC BY-ND 4.0)
http://dx.doi.org/10.5381/jot.2020.19.3.a4

and Scala (Odersky et al. 2008) share the same concepts and syntax. Reusing and extending the shared concepts between different languages and technological spaces (Kurtev et al. 2002) instead of reimplementing these again and again could allow language engineers to address more advanced challenges. To facilitate engineering and reusing language components, many language workbenches (Fowler 2005; Erdweg et al. 2015) support various language composition and reuse mechanisms (Erdweg et al. 2012) addressing reuse of different combinations of concrete and abstract syntax, well-formedness rules, code generators, transformations, and other language-related tooling. A collection of reusable components (Wor19 2019) for core language concerns, such as literals, types, and expressions, which are part of most modeling languages, still is missing. To mitigate this, we investigated the modularization of these core language concerns and present the resulting language components in the technological space of the MontiCore (HR17 2017; HMSNRW16 2016) language workbench. Building upon these can greatly facilitate language engineering and may help language engineers to 'see further'.

In the remainder, Section 2 presents concepts for language modularity before introducing novel language components to facilitate language reuse for a variety of language kinds and domains. Section 3 illustrates the use of these components by example, Section 4 highlights related work, and Section 5 discusses observations related to engineering and using these components. Section 6 concludes.

2. Collection of Language Components for Language Reuse

This section first introduces the MontiCore (HR17 2017; HMSNRW16 2016) language workbench and its features facilitating modularity in language engineering. Thereafter, the different components for literals, expressions, types, and statements are explained. All language components discussed in this paper are part of the MontiCore project, which is available on GitHub[1]. Table 1 (in the Appendix) provides an overview of all the different language components presented in this paper. Furthermore, usages of these language components in different modeling and programming languages are discussed in the case study in Section 3 and summarized in Table 2 (in the Appendix).

2.1. MontiCore Language Workbench

MontiCore provides an EBNF-like context-free grammar format to define languages. For a given grammar, MontiCore generates infrastructure necessary to efficiently engineer languages in a modular fashion. It has been applied to the engineering of modeling languages for a variety of domains, including automotive (DGH+19 2019), cloud services (Eikermann et al. 2017), robotics (AHRW17b 2017), systems engineering (DJR+19 2019), and more.

The generated infrastructure for a given grammar includes parser and lexer, Java classes for the abstract syntax tree (AST), an infrastructure to implement context conditions (language well-formedness rules), visitors (HMSNRW16 2016) to develop

[1] MontiCore project on GitHub: https://github.com/MontiCore/monticore/

MG

```
1  grammar IOAutomata extends CommonLiterals,
2                             CommonStatements,
3                             CommonExpressions,
4                             AssignmentExpressions{
5
6    Automaton  = "automaton" Name "{"
7                 (State | Transition)*
8                 "}" ;
9
10   State      = ["initial"]? ["final"]?
11                "state" Name ";" ;
12
13   Transition = from:Name "->" to:Name
14                ("[" guard:Expression "]")?
15                ("/" "{"
16                action:MCBlockStatement*
17                "}")? ";" ;
18 }
```

Figure 1 Exemplary grammar of an I/O automata language.

and compose analyses, and symbol tables (HMSNR15 2015; MSNRR15 2015; MSNRR16 2016) to combine models of different languages.

A MontiCore grammar defines both the abstract syntax and concrete syntax of a language. To this end, it comprises productions that define nonterminals. A production consists of a left-hand side (LHS) and a right-hand side (RHS) separated by an = sign. The LHS is the nonterminal that the production defines while the RHS is the production's body and defines both the abstract and concrete syntax. Figure 1 depicts a MontiCore grammar of a compact language for finite input/output automata (RRW14a 2014), while Figure 2 shows a corresponding automaton model. The automata grammar comprises three productions defining the syntax of the nonterminals `Automaton`, `State`, and `Transition`. MontiCore generates one AST class per production. Its attributes result from the production body. Stored terminals map to attributes while nonterminal usages map to compositions.

IOAutomata

```
1  automaton PingPong {
2    initial state Ping;
3    state Pong;
4
5    Ping - returnBall > Pong [ballIsHit];
6    Pong - returnBall > Ping;
7  }
```

Figure 2 Exemplary automaton for the language of Figure 1.

The body of a production consists of terminals and nonterminals. Terminals are surrounded by quotation marks, *e.g.*, `"automaton"` (l. 6) in Figure 1. Both terminals and nonterminals can have different multiplicities, *i.e.*, by appending a question mark '?' (l. 10) they become optional while '*' (l. 7) allows arbitrary many (including zero) occurrences and '+' enforces at least one occurrence. Alternatives are separated by '|' (l. 7) and grouping can be achieved by parenthesizing parts

using round brackets. Terminals whose presence is relevant for the abstract syntax can be parenthesized in square brackets, yielding a Boolean attribute in the abstract syntax. Optional elements are mapped to Java optionals and multiple occurrences to Java lists.

Besides "normal" nonterminals, MontiCore provides interface, abstract and external nonterminals. Abstract and external nonterminals are not detailed here but detailed information on these is available (HR17 2017). Interface nonterminals begin with the keyword `interface` (*cf.* Figure 8, l. 3). They do not specify concrete syntax themselves. Instead, interface nonterminals are implemented by other nonterminals (*cf.* Figure 8, l. 5). For interface nonterminals, a production body can be used to restrict possible implementing nonterminals (HR17 2017). Conceptually, interface nonterminals are an extension of alternatives. Whenever an interface nonterminal is used in a production body, every interface implementation can be parsed. Thus, instead of `A = B | C;` one can use interface nonterminals to define `interface A; B implements A; C implements A;`. The concrete syntax for these two examples does not differ. However, for interface nonterminals an AST interface instead of a class is generated and thus the relation between A and B, and A and C is mapped to inheritance instead of composition in the abstract syntax.

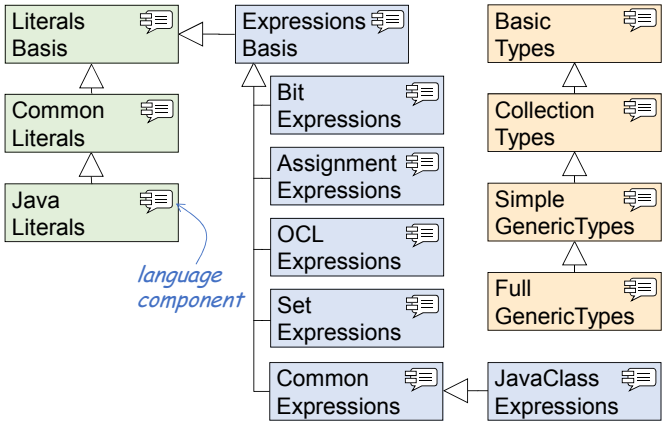

Figure 3 Relations of the language components for expressions, types, and literals.

Using MontiCore, languages can be developed efficiently by reusing the modular (parts of) other languages. To this end, MontiCore provides grammar extension mechanisms. As depicted in Figure 1 (ll. 1-4), the grammar of the automata language uses this concept. With the keyword `extends` followed by one or multiple comma-separated grammars, a grammar can extend other grammars. As a consequence, all nonterminals defined by productions of the super grammars are available in the current grammar. In the automata language, the transition production uses the nonterminal `Expression` that is not defined locally but defined in the super grammar `ExpressionsBasis`. If a grammar is designed for reuse only and does not define a complete language itself, it is marked as a component grammar by adding the keyword `component` (*cf.* Figure 8, l. 1).

The start nonterminal of a grammar is by default the first nonterminal in the grammar (HR17 2017). However, sometimes this is not feasible, *e.g.,* when an inherited nonterminal should be the start nonterminal of a language. To address this, the start nonterminal can be set explicitly as follows: `start State;`.

When extending a grammar, it is possible to extend productions of the super grammars. This is possible for normal and interface productions. In both cases, conceptually a new alternative to the existing body or implementation is created. Thus, all nonterminals and especially interface nonterminals can serve as extension points. To further control the priority of the newly added alternative, it is possible to add a priority in angle brackets (*cf.* Figure 8, l. 5). The higher the number within the brackets the higher is the alternative's priority in the parser.

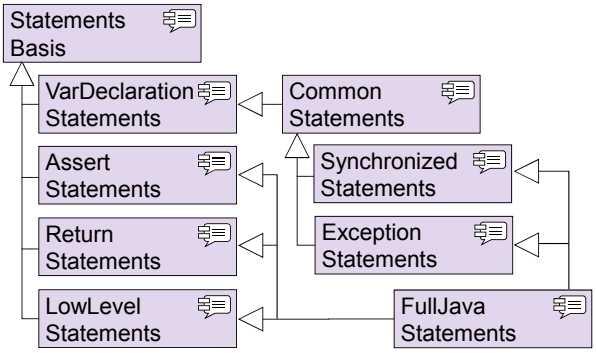

Figure 4 Language components for statements.

2.2. Literals

MontiCore provides many literals by default to support developing new modeling languages. Thus, these literals provide essential, usually atomic, elements of common languages, such as numbers, character strings, or boolean values. To facilitate integration, MontiCore provides these literals in a modularized fashion, enabling the selection of appropriate literals for a respective modeling language. Figure 3 (left) depicts the three literals language components and their inheritance hierarchy. `LiteralsBasis` provides a basic interface for using literals. It is implemented in `CommonLiterals`, resulting in the most common set of literals for language development in MontiCore. Furthermore, the `JavaLiterals` extend these again to provide Java-compliant literals.

MG

```
1 component grammar LiteralsBasis  {
2   interface Literal;
3 }
```

Figure 5 Grammar of `LiteralsBasis`.

The most basic variant of literals in MontiCore is `LiteralsBasis`. This grammar provides a general `Literal` interface (*cf.* Figure 5, l. 2). As a result, literals are modularized, which increases interchangeability and reuse of language components. The interface can be used in modeling languages to enable all implementing literals. To define specific literals, it is possible to implement this interface. An example is provided in `CommonLiterals`.

```
1  component grammar CommonLiterals extends Basics,
2                                    LiteralsBasis {
3    interface SignedLiteral;
4    interface NumericLiteral extends Literal;
5    interface SignedNumericLiteral
6              extends SignedLiteral;
7
8    NullLiteral implements Literal, SignedLiteral
9    = "null";
10
11   BooleanLiteral implements Literal, SignedLiteral
12     = src:["true" | "false"];
13
14   CharLiteral implements Literal, SignedLiteral
15     = Char;
16
17   StringLiteral implements Literal, SignedLiteral
18     = String;
19
20   NatLiteral implements NumericLiteral = Digits;
21
22   SignedNatLiteral implements SignedNumericLiteral
23     = (negative:["-"])? Digits;
24
25   BasicLongLiteral implements NumericLiteral
26     = Digits key("l" | "L");
27
28   SignedBasicLongLiteral
29     implements SignedNumericLiteral
30     = (negative:["-"])? Digits key("l" | "L");
31
32   BasicFloatLiteral implements NumericLiteral
33     = pre:Digits "." post:Digits key("f" | "F");
34
35   SignedBasicFloatLiteral
36     implements SignedNumericLiteral
37     = (negative:["-"])? pre:Digits
38       "." post:Digits key("f" | "F");
39
40   BasicDoubleLiteral implements NumericLiteral
41     = pre:Digits "." post:Digits;
42
43   SignedBasicDoubleLiteral
44     implements SignedNumericLiteral
45     = (negative:["-"])? pre:Digits "." post:Digits;
46 }
```

Figure 6 Grammar of `CommonLiterals`.

The grammar `CommonLiterals` extends `LiteralsBasis` (Figure 6, ll. 1-2) and uses the provided interface to specify literals. It provides a reduced subset of literals, which abstracts from special cases (*e.g.,* hexadecimal representation in Java), offering a suitable foundation for most modeling languages. Additionally, the grammar introduces an interface `SignedLiteral` (l. 3), which provides an additional prefix for literals, enabling negative numbers. Lines 4-6 categorize these interfaces into corresponding (`Signed-`) `NumericLiterals`. The remainder of Figure 6 defines basic literals by implementing the available interfaces. These literals consist of primitive data types (similar to Java) and strings in their signed and unsigned variants. We have slightly simplified this grammar to preserve clarity. For instance, all token definitions (*i.e.,* `Char`, `String`, and `Digits`) have been omitted, since these only specify the allowed character sequences and do not contribute to language modularity. Lines 8-18 define a `NullLiteral` as well as Boolean values, characters, and strings. These literals have no differences in signs, and thus both implement the `Literal` and the `SignedLiteral` interface. As a result, they are available in both cases. Lines 20-45 define numeric literals. MontiCore supports natural numbers (ll. 20-25), long numbers (ll. 25-30), float numbers (ll. 32-38), and double numbers (ll. 40-45). However, as these values have a signed and unsigned variant, there are two productions each to cover both cases. The main difference between these cases is that signed values also have an optional preceding `"-"` sign. According to the requirements, the corresponding literals implement the `NumericLiteral` or `SignedNumericLiteral` interface.

```
1  component grammar JavaLiterals
2                       extends CommonLiterals {
3    IntLiteral    implements NumericLiteral <100>
4      = source:Num_Int;
5
6    LongLiteral   implements NumericLiteral  <99>
7      = source:Num_Long;
8
9    FloatLiteral  implements NumericLiteral <100>
10     = source:Num_Float;
11
12   DoubleLiteral implements NumericLiteral <100>
13     = source:Num_Double;
14 }
```

Figure 7 Grammar of `JavaLiterals`.

The `JavaLiterals` grammar defines Java-compliant literals. Additionally, it extends `CommonLiterals` to leverage the literals defined there (*cf.* Figure 7, ll. 1-2). Furthermore, new literals for the numeric values int, long, float, and double are specified (ll. 3-13). The advantage of this grammar is a comfortable usage of literals in a Java-like syntax. This includes extended forms of representation, *e.g.,* the hexadecimal representation of numbers.

2.3. Expressions

Expressions usually build complex structures built on literals and sub-expressions. They can be used in various contexts. To flexibly choose which kinds of expressions a language uses a decomposition into different grammars is useful. Thus, for the MontiCore language workbench the decomposition presented in Figure 3 was developed. The most basic variant `ExpressionsBasis` is depicted in Figure 8. This language component provides the central expression interface `Expression` as well as two basic implementations `NameExpression` and `LiteralExpression`.

When extending this language component, the newly created language enables the formulation of expressions that are simple variables such as `foo` or `bar`, or values. The kind of values expressible is not yet determined as the literals interface is used (cf. l. 9) and only the `LiteralsBasis` grammar is extended. This means that no `Literal` implementations are used so far.

Additional implementations of the `Expression` interface are provided by the components `AssignmentExpressions`,

```
component grammar ExpressionsBasis
                  extends LiteralsBasis, Basics {
  interface Expression;

  NameExpression    implements Expression <350>
    = Name;

  LiteralExpression implements Expression <340>
    = Literal;
}
```

Figure 8 Grammar of ExpressionsBasis.

`CommonExpression`, `BitExpression`, and `JavaClassExpression` each of which bundles a set of closely related expression implementations.

The grammar of the `AssignmentExpressions` language component is shown in Figure 9. It provides typical expressions for assignments using operators such as '=' or '+=' as well as increment or decrement operation as prefix or suffix operators. These expressions are bundled in their own language component as they produce side effects which is not feasible in every context *e.g.,* functional languages such as OCL. Furthermore, this grammar provides algebraic signs. Using this grammar enables to formulate expressions such as `-a`, `a = b`, or `a += ++b - c`. As this grammar does not provide any implementation of the `Literal` interface, it is still not determined which kind of values are possible within the expressions. For instance, `a = 1` is not yet expressible, as this would require an implementation of the `Literal` interface that provides numbers.

```
component grammar AssignmentExpressions
                      extends ExpressionsBasis {
  IncSuffixExpression implements Expression <220>
    = Expression "++";

  DecSuffixExpression implements Expression <220>
    = Expression "--";

  PlusPrefixExpression implements Expression <210>
    = "+" Expression;

  MinusPrefixExpression implements Expression <210>
    = "-" Expression;

  IncPrefixExpression implements Expression <210>
    = "++" Expression;

  DecPrefixExpression implements Expression <210>
    = "--" Expression;

  AssignmentExpression implements Expression <60>
    = left:Expression
    operator:[ "="   | "+="   | "-="  | "*="
             | "/="  | "&="   | "|="  | "^="
             | ">>=" | ">>>=" | "<<=" | "%=" ]
    right:Expression;
}
```

Figure 9 Grammar of `AssignmentExpressions`.

For clarity reasons, the grammars of `CommonExpression`, as well as the `BitExpression`, and `JavaClassExpression`, are not presented in this paper. The `CommonExpression` language component provides mathematical expressions such as addition and multiplication, comparisons such as *greater than* or *equals*, and Boolean expressions such as *and*-expressions. Furthermore, the bracket expression, as well as the conditional expression, is provided by this language component. These expressions are free of side effects and frequently used in modeling languages. When using this language component, expressions such as `a <= b`, `(a == b)`, `a && b`, or `a + b` are expressible. Again, no literal implementations are provided thus, the decision on literals is not determined by this language component.

The `BitExpression` language component provides bit operations such as shift expressions and bitwise expressions such as *and* and *or* expressions. These expressions are rarely used for modeling and thus separated from other, more frequently used expressions. When using this language component, expressions such as `a « b` or `a & b` are expressible.

The `JavaClassExpression` language component builds upon the `CommonExpression` language component. `JavaClassExpression` provides typical Java expressions such as array expressions, type cast, and `instanceof` expressions or expressions that use `super` or `this`. These expressions are less useful for modeling but needed when designing programming languages such as Java.

The presented collection of expression components provides a modular foundation for creating individual expression languages. By composing an adequate subset of these components, an OCL variant (Cabot & Gogolla 2012; Demuth & Wilke 2009; Rum16 2016) can be implemented without much effort.

```
component grammar BasicTypes extends Basics {

  interface MCType;

  MCQualifiedName = part:(Name || ".")+;

  MCImportStatement = "import" MCQualifiedName
    ("." Star:["*"])? ";" ;

  MCPrimitiveType implements MCType
    = primitive: [ "boolean" | "byte"
                 | "short"   | "int"
                 | "long"    | "char"
                 | "float"   | "double" ];

  interface MCObjectType extends MCType;

  MCQualifiedType implements MCObjectType
    = MCQualifiedName;

  MCReturnType = MCVoidType | MCType;

  MCVoidType = "void";
}
```

Figure 10 Grammar of `BasicTypes`.

```
component grammar CollectionTypes extends BasicTypes
{
  interface MCGenericType extends MCObjectType;

  MCListType implements MCGenericType <200>
    = "List" "<" MCTypeArgument ">";

  MCOptionalType implements MCGenericType <200>
    = "Optional" "<" MCTypeArgument ">";

  MCMapType implements MCGenericType <200>
    = "Map" "<" key:MCTypeArgument ","
    value:MCTypeArgument ">";

  MCSetType implements MCGenericType <200>
    = "Set" "<" MCTypeArgument ">";

  interface MCTypeArgument;

  MCBasicTypeArgument implements
     MCTypeArgument <200> = MCQualifiedType;

  MCPrimitiveTypeArgument implements
     MCTypeArgument <190> = MCPrimitiveType;
}
```

Figure 11 Grammar of `CollectionTypes`.

2.4. Types

Type systems are available in a variety of (programming) languages and facilitate programming because typing errors can already be detected at compile time. To express type usages, a library of language components for modeling types was developed for MontiCore-based languages. It consists of four language components that are in an inheritance relationship. The most basic language component is `BasicTypes`, which provides the central interface nonterminal `MCType` (*cf.* Figure 10, l. 3). Besides this interface nonterminal, this language component provides nonterminals that enable modeling primitive types (ll. 10-14) and qualified types (ll. 18-19). The latter are realized with the nonterminal `MCQualifiedName` that contains only an optional qualifier. Therefore, non-qualified types are expressible as well.

Furthermore, the grammar provides a return type (l. 21), which can be an `MCType` or *void*. These nonterminals are bundled as they form a relativity small yet useful collection of types for modeling. When using this language component, types such as `int`, `Person`, or `java.lang.String` are expressible.

```
component grammar SimpleGenericTypes
                      extends CollectionTypes {
  MCBasicGenericType implements MCGenericType <20>
    =(Name || ".")+
    "<" (MCTypeArgument || ",")* ">";

  MCCustomTypeArgument
    implements MCTypeArgument <20> = MCType;
}
```

Figure 12 Grammar of `SimpleGenericTypes`.

```
component grammar StatementsBasis {
  interface MCBlockStatement;
  interface MCStatement extends MCBlockStatement;
  interface MCModifier;
}
```

Figure 13 Grammar of `StatementsBasis`.

The `CollectionTypes` language component builds upon the basic types language component as visualized in Figure 11. This language component adds implementations to the `MCType` interface nonterminal that enables to model four kinds of generics: `Set`, `List`, `Map`, and `Optional`. Via the `MCTypeArgument`, qualified and primitive types can be used as arguments for the aforementioned generics. Furthermore, these generics cannot be nested as the purpose of this language component is to provide some commonly used collection types but limit their functionality such that it is useful for high-level models. Using this language component, types such as `List<Integer>`, `Set<char>`, or `Map<java.lang.String, Person>` are expressible.

The language component `SimpleGenericTypes` extends the language component `CollectionTypes` (cf. Figure 12). This language extends the expressible types with custom generics. `MCBasicGenericType` enables modeling generics of arbitrary classes with arbitrary arguments. The latter is achieved by the nonterminal `MCCustomTypeArgument`. When using this language component, types such as `Person<String>` or `Map<Person<String>, Integer>` are expressible. These types, however, do not cover all possible types from Java, as Java additionally supports inner types of generic types, which is not expressible using the language component `SimpleGenericTypes`, *e.g.,* types such as `a.b.C<D>.E.F<G>.H` are not expressible. If these kinds of types are required, the `FullGenericTypes` language component, whose grammar is not included here, can be used. This language component is part of the MontiCore project on GitHub.

```
component grammar VarDeclarationStatements extends
     StatementsBasis, BasicTypes, ExpressionsBasis {

 LocalVarDeclaration implements MCBlockStatement
  = MCModifier* MCType (VariableDeclarator || ",")+;

 VariableDeclarator
  = DeclaratorId ("=" VariableInit)?;

 DeclaratorId = Name (dim:"[" "]")*;

 interface VariableInit;

 SimpleInit implements VariableInit = Expression;

 ArrayInit  implements VariableInit
  = "{" (VariableInit || ",")* (",")? "}";
}
```

Figure 14 Grammar of `VarDeclarationStatements`.

```
1  component grammar CommonStatements extends VarDeclarationStatements {
2    MCJavaBlock  implements MCStatement = "{" MCBlockStatement* "}";
3
4    JavaModifier implements MCModifier =
5      modifier:["static"  | "protected"  | "public" | "private"  | "transient" | "final" | "abstract"
6              | "native" | "threadsafe" | "const"  | "volatile" | "strictfp"  | "synchronized"];
7
8    IfStatement implements MCStatement
9      = "if" "(" cond:Expression ")" thenStatement:MCStatement ("else" elseStatement:MCStatement)?;
10
11   ForStatement implements MCStatement = "for" "(" ForControl ")" MCStatement;
12
13   interface ForControl;
14
15   CommonForControl implements ForControl = ForInit? ";" cond:Expression? ";" (Expression || ",")*;
16
17   ForInit = ForInitByExpressions | LocalVariableDeclaration;
18
19   ForInitByExpressions = (Expression || ",")+;
20
21   WhileStatement implements MCStatement = "while" "(" cond:Expression ")" MCStatement;
22
23   DoWhileStatement implements MCStatement = "do" MCStatement "while" "(" cond:Expression ")" ";";
24
25   SwitchStatement implements MCStatement
26     = "switch" "(" Expression ")" "{" SwitchBlockStatementGroup* SwitchLabel* "}";
27
28   EmptyStatement implements MCStatement = ";";
29
30   ExpressionStatement implements MCStatement = Expression ";";
31
32   SwitchBlockStatementGroup = SwitchLabel+ MCBlockStatement+;
33
34   interface SwitchLabel;
35
36   ConstantExpressionSwitchLabel implements SwitchLabel = "case" constantExpression:Expression ":";
37
38   EnumConstantSwitchLabel implements SwitchLabel = "case" enumConstantName:Name ":";
39
40   DefaultSwitchLabel implements SwitchLabel = "default" ":";
41 }
```

Figure 15 Grammar of `CommonStatements`.

2.5. Statements

MontiCore provides a library of modular statement language components, depicted in Figure 4, to offer language developers different options for embedding Java-like statements in a modeling language. To this end, language developers can select the appropriate subset of statements or add their own statements.

MontiCore offers a basic grammar that provides the necessary interfaces for a modular statement definition. Figure 13 contains a listing of this grammar, named `StatementsBasis`. Line 2 specifies the interface `MCBlockStatement`, which represents the most extensive set of statements. The interface `MCStatement` further extends this (*cf.* l. 3). The difference between the two interfaces is that `MCBlockStatement` allows any kind of statements, while `MCStatement` depicts a subset that excludes block statements. For instance, this is demonstrated via variable declarations which, according to Java, are used exclusively in blocks. Finally, line 4 introduces the interface `MCModifier`. It provides access to more advanced properties of model elements, such as attribute visibility and accessibility.

Grammar `VarDeclarationStatements` in Figure 14 specifies statements for declaring variables. It uses the predefined interfaces from `StatementsBasis` as well as `MCBasicTypes` and `ExpressionBasis` (ll. 1-2). A `LocalVarDeclaration` (ll. 4-5) is part of the `MCBlockStatements` and thus implements the corresponding interface. These declarations may have several modifiers, followed by an `MCType` and at least one `VariableDeclarator`. This declarator (ll. 7-10) contains an identifier consisting of the variable's name and its dimension, expressed by `"[" "]"` for multidimensional variables, such as arrays. Furthermore, variables can optionally be initialized via the inherited expressions (ll. 12-17). For multidimensional variables, several expressions are provided as a comma-separated list in curly brackets.

The grammar `CommonStatements` defines basic statements such as method calls, variable assignments, conditions, and loops. The grammar (*cf.* Figure 15) extends `VarDeclarationStatements`. Line 2 defines `MCJavaBlock`, a statement that may contain additional statements in curly

brackets. Furthermore, the interface `MCModifier` from the inherited grammar `StatementBasis` is implemented by explicit `JavaModifiers` (ll. 4-6). In general, this grammar enables a standard Java-like statement syntax. This includes if-clauses (ll. 8-9), for loops (ll. 11-19), and while loops (ll.21-23). Furthermore, `SwitchStatements` (ll. 25-26) enable case distinctions in Java-customary syntax. The specified statements use expressions to establish and verify their conditions.

Further statement grammars are not presented in detail for the sake of clarity. Instead, we briefly introduce these according to their main features. The grammar `ReturnStatements` enables specifying the return of methods. Therefore, an optional expression is provided, which evaluates to the return value. `SynchronizedStatements` supports the definition of `synchronized` expressions. This, *e.g.,* enables thread safety for method calls. Grammar `ExceptionStatements` introduces additional `try-catch` blocks. These reference exceptions by their name and can thus handle these. Grammar `LowLevelStatements` enables several basic control statements such as `break` and `continue` commands, as well as labelling of statements. Finally, `FullJavaStatements` joins the defined statement sub-languages, leveraging their modular structure. Only existing grammars must be integrated without the necessity of introducing new productions. Overall, `FullJavaStatements` provides access to an extensive, Java-compatible, collection of statement components.

3. Case Study

The presented language components can be used to realize languages without the effort of re-implementing commonly used parts again. Instead, "off-the-shelf" language components can be reused and combined to form the basis for engineering a new language. This section describes the engineering of the syntax for a light-weight ADL for which the grammar is depicted in Figure 16.

MG

```
1  grammar LightADL extends BasicTypes, IOAutomata {
2
3    Component implements CmpElement
4      = "component" Name "{" CmpElement* "}";
5
6    interface CmpElement;
7
8    Port implements CmpElement = "port"
9      direction:[ "in" | "out" ] MCType Name ";";
10
11   Connector implements CmpElement
12     = "connect" src:Name "->" tgt:Name ";";
13
14   StateBasedBehavior implements CmpElement
15     = "behavior" "{" (State | Transition)* "}"
16 }
```

Figure 16 Exemplary grammar of the light-weight ADL.

The light-weight ADL (LightADL) grammar reuses the language components for automata (*cf.* Section 2) and basic types (*cf.* Section 2) by extending the grammars of the respective language components (l. 1). Through this, other language components (*e.g.,* for literals, expressions, and statements) are reused transitively as well. A component of the ADL begins with the keyword `component`, followed by a name and a list of component elements (ll. 3-4). The component elements are realized as interface production (l. 6) to foster extensibility with further kinds of component elements. The syntax of every kind of component element is realized as grammar production implementing the interface production. Possible component elements of the ADL are transitively contained components (ll. 3-4), directed and typed ports (ll. 8-9), through which a component can exchange messages with other components via connectors (ll. 11-12). The types of ports are realized by reusing the interface nonterminal `Type`. In this ADL, components may further contain behavior descriptions (ll. 14-15) in form of automata. For realizing these, the syntax of `States` and `Transitions` is reused from the automata language component.

MG

```
1  grammar FullADL extends LightADL,
2          FullJavaStatements, SimpleGenericTypes {
3    start Component;
4  }
```

Figure 17 Exemplary grammar of the complex ADL.

For a different application, the light-weight ADL is extended to create an ADL with a more sophisticated type system and more options for using statements to describe the actions of a behavior automaton's transitions. The grammar of the resulting `FullADL` language is depicted in Figure 17. `FullADL` extends the `LightADL`, and also the language components for `JavaStatements` and `SimpleGenericTypes`. Through this, the port definitions can use generic types. Further, the statements in action blocks of automaton transitions can be any Java statements. This is due to the fact that `LightADL` uses the interface nonterminal `MCType` of the `BasicTypes` language component for specifying port types and `SimpleGenericTypes` provides further implementations for `MCType`. Analogously, the interface nonterminal `BlockStatement` that the automata language uses for statements in action blocks is provided with further implementations in the `FullADL` through extending the language component `JavaStatements`. The grammar further reuses the start production `Component` of the `LightADL`. The remaining body of the grammar is empty, as all grammar productions are reused from super grammars and no new productions are defined.

Figure 18 depicts an overview of the language components of this case study and their interrelations. Language components that are reused from the presented language component library are colored gray. For clarity, the figures omits relations between reused language components and transitively reused language components. `IOAutomata`, `LightADL`, and`FullADL` are *complete* languages, which means that they yield a parser. This is indicated in Figure 18 via a dedicated icon.

While the `FullADL` does not introduce a single new nonterminal, the `LightADL` introduces five nonterminals, and

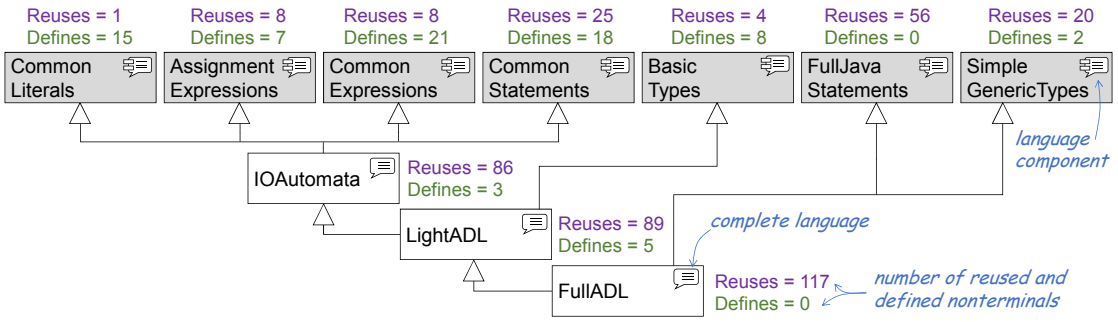

Figure 18 Language components of the case study, including reused (colored gray) language components. Relations between reused language components and transitive reuses are omitted.

`IOAutomata` introduces three nonterminals. However, the number of introduced nonterminals is only a small portion of the number of nonterminals that these languages use. Most nonterminals are reused from inherited language components. Both the number of nonterminals that a language component reuses from other language components and the number of nonterminals that a language component defines are depicted in Figure 18 next to each language component. These numbers exclude token productions and count nonterminals, which are inherited multiple times through diamond inheritance, only once. The number of defined and reused nonterminals per language component of the presented language component library is depicted for literals, types, and expressions in Figure 20. The numbers for statements are depicted in Figure 19.

The case study demonstrates by example that a large proportion of nonterminals can be reused for building a new language. This does not only save effort in re-engineering the syntax again, but also has the advantage that reused parts of language infrastructure can be developed, tested, evolved, and maintained individually.

4. Related Work

The functionalities and features of language workbenches have already been studied in detail over the past decades (Erdweg et al. 2013). Utilizing language composition, several allow providing composable language modules similar to the proposed language components. In the following, we discuss the potential of modern language workbenches to provide respective language modules and describe existing libraries of composable modules to the best of our knowledge.

Spoofax (Wachsmuth et al. 2014) is a language workbench, which automatically derives integrated Eclipse editors from a language definition. It supports the modularization of a modeling language into different grammars, so-called modules. Different concepts of such modules can be reused via imports, enabling language extension. Thus, Spoofax offers fundamental features for language composition. Even if basic modules of public projects sometimes resemble each other, there are no standard modules or libraries similar to the language components provided by MontiCore.

Xtext (Bettini 2016) is an open-source framework for developing modeling languages. It enables the specification of a grammar, which is further processed into an Ecore metamodel and corresponding tooling for modeling. Xtext also automatically derives textual Eclipse editors. In general, Xtext supports reusable language components. Single grammar inheritance is supported, but neither non-terminal interfaces nor non-terminal inheritance. Hence, Xtext also offers basic concepts for language extension but does not cover the entire field. Besides a default grammar for terminals, Xtext also offers the expression language Xbase (Efftinge et al. 2012). Xbase relies on the general-purpose language Java, thus fostering the integration of Java-like expressions into a custom modeling language.

Neverlang (Vacchi & Cazzola 2015) is a framework for developing modular languages. It enables the composition of grammars with placeholders as extension points. For this purpose, production rules can assign undefined non-terminals, which are implemented by extending sub-languages. Furthermore, Neverlang supports language variability by using the Common Variability Language (Méndez-Acuña et al. 2016). Although we are not aware of any language components, Neverlang supports the required functionality to provide these.

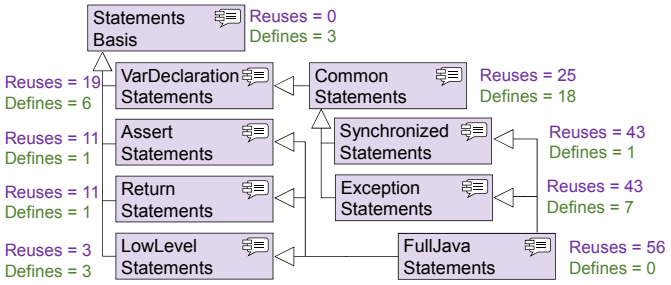

Figure 19 Numbers of reused and defined nonterminals for statement language components of the library.

The Meta Programming System (MPS) (Voelter & Pech 2012) is a language workbench that fosters the development of modeling languages with projectional editors. In projection editors, changes are performed directly on the AST without the necessity of parsing a model. MPS supports a wide range of editorial representations, such as textual, symbolic, or graphical. It enables language composition by extending abstract syntax elements (Voelter & Pech 2012). For these elements, MPS offers modular standard libraries, which contain expressions,

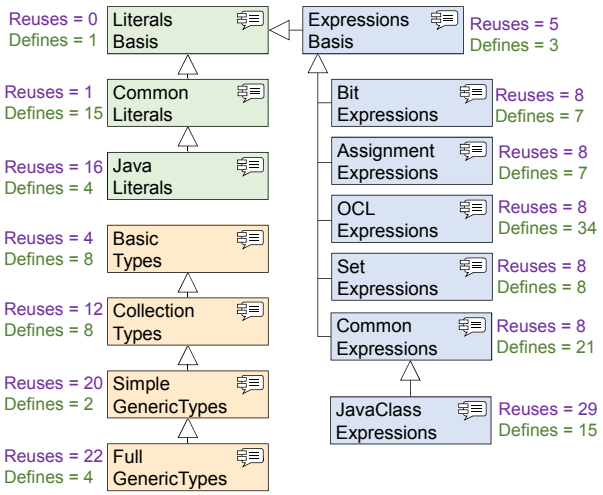

Figure 20 Numbers of reused and defined nonterminals per types, literals, and expression language component.

literals, and variables. Hence, this approach is similar to the language components presented for MontiCore, although it covers a different technological area.

mbeddr (Voelter et al. 2012) is a framework for the development of embedded modeling languages. Based upon MPS, mbeddr directly inherits its language composition features. It is specialized in language engineering for embedded systems, and, thus, offers the possibility to extend C artifacts by DSL snippets. Existing language modules are, therefore, primarily designed for embedded systems. These cover a recreation of the language C, as well as SI units and state machines for behavior specification. Thus, mbeddr also aims at offering language components for a particular scope. These languages are self-contained without further modularisation and separation of concerns.

Whole Platform (Solmi 2005) is a language workbench based on the Eclipse IDE. Based on grammars for language development, the Whole Platform also supports various projection representations, such as tables, tree views, and mathematical representations (Erdweg et al. 2013). The language workbench provides various languages supporting language development, such as composable types or a transformation language. Generally, the Whole Platform enables language product lines and language composition. Furthermore, there is a standard library of modeling languages for embedding and extension. However, to the best of our knowledge, these primarily cover complete languages (e.g., Java, XML). Therefore, the Whole Platform differs in scope and granularity from MontiCore's approach of modular language components.

5. Discussion

The presented library of language components for different kinds of literals, expressions, types, and statements is based on our experiences in engineering languages for various purposes in different applications. However, the language component library does obviously not claim to be complete. Further realizations for literals, expressions, types, and statements can be integrated into the language components library anywhere within the inheritance hierarchies. We identified many further language parts that are reusable for engineering various languages such as, *e.g.,* cardinalities, stereotypes, SI units, and modifiers. Presenting the language components for these in detail, however, is beyond the scope of this paper.

The language component library has been developed to avoid conflicting nonterminals. Multiple inheritance of grammars in general, however, can lead to ambiguities in combination with a flat namespace of nonterminal names that should be avoided.

Picking a suitable granularity for language components is crucial. Coarse-grained components prevent using parts of it for an application individually without reusing the parts that are not needed in this application. Fine-grained components, on the other and, increase the overhead in engineering and managing these individually.

Identifying language components in terms of reusable units requires foresight by language developers: If a language component depends on another language component by inheriting from it, it is impossible to use this language component without the other one. The rationale for engineering modular languages, therefore, is to develop language features as separate language components and postpone binding these to their environment, *i.e.,* to other language components, as late as possible.

An extensive application of language modularization can be leveraged to realize product lines of language (BEK+18a 2018; BEK+19 2019), in which each feature of a language is associated with a language component realizing this feature. Upon a selection of features, the respective language components can be composed to yield a language variant. This fosters systematic reuse of language components for families of similar languages.

A formal understanding of the OCL (Richters & Gogolla 1998), an OCL metamodel (Richters & Gogolla 1999), and the presented language component library with the modularization techniques of MontiCore are the perfect basis for building a modular OCL. The modular OCL can, *e.g.,* include imperative statements (Büttner & Gogolla 2014) based on statement language components from the presented library. Further, a modular OCL engineered with MontiCore can be embedded into other languages similar to the SOIL (Büttner & Gogolla 2011) approach.

While the focus of this paper lies in the modularization of language grammars, the modularization of other parts of language infrastructure is important as well. This can include, *e.g.,* a modular visitor infrastructure for traversing the abstract syntax. To give meaning to a model, language components can, *e.g.,* apply modular code generators (BEK+18a 2018), graph transformations (Kuske et al. 2002), or (domain-specific) model transformations (Hoe18 2018). Only providing, for instance, modular analyses (*e.g.,* type checks) and transformations (*e.g.,* code generators) accompanying the modular syntax makes language components truly modular.

6. Conclusion

Reusing tried and tested software parts in different contexts is one of the key success factors of software engineering. With software languages being software too, the efficient reuse of

software language components in different contexts reuse can facilitate engineering truly domain-specific languages. Hence, software language reuse can be a key enabler of their adoption and, thus, the adoption of MDD in general. To advance the use of software languages, we have developed concepts for the modularization of the ubiquitous language components of literals, expressions, types, and statements. Leveraging these facilitates engineering novel languages, liberates software language engineers from needing to reinvent the wheel, and ultimately may promote the adoption of MDD in different contexts.

References

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

Bettini, L. (2016). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd.

Butting, A., Eikermann, R., Kautz, O., Rumpe, B., & Wortmann, A. (2018, September). Modeling Language Variability with Reusable Language Components. In *Int. Conf. on Systems and Software Product Line (SPLC'18)*. ACM.

Butting, A., Eikermann, R., Kautz, O., Rumpe, B., & Wortmann, A. (2019, June). Systematic Composition of Independent Language Features. *Journal of Systems and Software*, *152*, 50–69.

Büttner, F., & Gogolla, M. (2011). Modular Embedding of the Object Constraint Language into a Programming Language. In *Brazilian symposium on formal methods* (pp. 124–139).

Büttner, F., & Gogolla, M. (2014). On OCL-Based Imperative Languages. *Science of Computer Programming*, *92*, 162–178.

Cabot, J., & Gogolla, M. (2012). Object Constraint Language (OCL): a Definitive Guide. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems* (pp. 58–90).

Dabney, J. B., & Harman, T. L. (2004). *Mastering Simulink*. Pearson.

Dalibor, M., Jansen, N., Rumpe, B., Wachtmeister, L., & Wortmann, A. (2019, September). Model-Driven Systems Engineering for Virtual Product Design. In L. Burgueño et al. (Eds.), *Proceedings of MODELS 2019. Workshop MPM4CPS* (pp. 430–435). IEEE.

Demuth, B., & Wilke, C. (2009). Model and Object Verification by Using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia* (pp. 687–690).

Dévai, G., Kovács, G. F., & An, Á. (2014). Textual, Executable, Translatable UML. In *OCL@ MoDELS* (pp. 3–12).

Drave, I., Greifenberg, T., Hillemacher, S., Kriebel, S., Kusmenko, E., Markthaler, M., … Wortmann, A. (2019, February). SMArDT modeling for automotive software testing. *Software: Practice and Experience*, *49*(2), 301-328.

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., & Hanus, M. (2012). Xbase: Implementing Domain-Specific Languages for Java. *ACM SIGPLAN Notices*, *48*(3), 112–121.

Eikermann, R., Look, M., Roth, A., Rumpe, B., & Wortmann, A. (2017). Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In *Software Architecture for Big Data and the Cloud* (pp. 207–226). Elsevier.

Erdweg, S., Giarrusso, P. G., & Rendel, T. (2012). Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications* (pp. 1–8).

Erdweg, S., Van Der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., … others (2013). The State of the Art in Language Workbenches. In *International conference on software language engineering* (pp. 197–217).

Erdweg, S., Van Der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W. R., … others (2015). Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, *44*, 24–47.

Etzel, C., & Bauer, B. (2019). Modeling and Analysis of Partitions on Functional Architectures Using EAST-ADL. In *International Conference on Model-Driven Engineering and Software Development* (pp. 298–319).

Favre, J.-M. (2005). Languages evolve too! Changing the Software Time Scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)* (pp. 33–42).

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

Fowler, M. (2005). *Language Workbenches: The Killer-App for Domain Specific Languages.*

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

Friedenthal, S., Moore, A., & Steiner, R. (2014). *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann.

Halloway, S. (2009). *Programming Clojure*. Pragmatic Bookshelf.

Heim, R., Mir Seyed Nazari, P., Rumpe, B., & Wortmann, A. (2016, July). Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conf. on Modelling Foundations and Applications (ECMFA)* (pp. 67–82). Springer.

Hölldobler, K. (2018). *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Shaker Verlag.

Hölldobler, K., Mir Seyed Nazari, P., & Rumpe, B. (2015). Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures. In *Domain-Specific Modeling Workshop (DSM'15)* (pp. 23–30). ACM.

Hölldobler, K., & Rumpe, B. (2017). *MontiCore 5 Language Workbench Edition 2017*. Shaker Verlag.

Hölldobler, K., Rumpe, B., & Wortmann, A. (2018). Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, *54*, 386–405.

Jemerov, D., & Isakova, S. (2017). *Kotlin in Action*. Manning Publications Company.
Kleppe, A. (2008). *Software Language Engineering: Creating Domain-specific Languages Using Metamodels*. Pearson Education.
Koenig, D., Glover, A., King, P., Laforge, G., & Skeet, J. (2007). *Groovy in Action*. Manning Publications Co.
Kurtev, I., Bézivin, J., & Aksit, M. (2002). Technological Spaces: An Initial Appraisal. *CoopIS, DOA*, *2002*.
Kuske, S., Gogolla, M., Kollmann, R., & Kreowski, H.-J. (2002). An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In *International conference on integrated formal methods* (pp. 11–28).
Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., & Tang, A. (2013). What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, *39*(6), 869–891. doi: 10.1109/TSE.2012.74
Medvidovic, N., & Taylor, R. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*.
Méndez-Acuña, D., Galindo, J. A., Degueule, T., Combemale, B., & Baudry, B. (2016). Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures*, *46*, 206–235.
Mir Seyed Nazari, P., Roth, A., & Rumpe, B. (2015). Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table. In *Domain-Specific Modeling Workshop (DSM'15)* (pp. 37–42). ACM.
Mir Seyed Nazari, P., Roth, A., & Rumpe, B. (2016, March). An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference* (Vol. 254, pp. 133–140). Bonner Köllen Verlag.
Odersky, M., Spoon, L., & Venners, B. (2008). *Programming in scala*. Artima Inc.
Richters, M., & Gogolla, M. (1998). On Formalizing the UML Object Constraint Language OCL. In *International conference on conceptual modeling* (pp. 449–464).
Richters, M., & Gogolla, M. (1999). A Metamodel for OCL. In *International conference on the unified modeling language* (pp. 156–171).
Richters, M., & Gogolla, M. (2000). Validating UML models and OCL constraints. In *International Conference on the Unified Modeling Language* (pp. 265–277).
Ringert, J. O., Rumpe, B., & Wortmann, A. (2014). *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag.
Rumpe, B. (2016). *Modeling with UML: Language, Concepts, Methods*. Springer International.
Selic, B. (2003). The pragmatics of model-driven development. *IEEE software*, *20*(5), 19–25.
Solmi, R. (2005). *Whole Platform* (Unpublished doctoral dissertation). Citeseer.
Starrett, C. (2016). xtUML: Current and Next State of a Modeling Dialect. In *EXE@ MoDELS* (pp. 33–37).
Vacchi, E., & Cazzola, W. (2015). Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, *43*, 1–40.
Voelter, M., & Pech, V. (2012). Language Modularity with the MPS Language Workbench. In *Software Engineering (ICSE), 2012 34th International Conference on* (pp. 1449–1450).
Voelter, M., Ratiu, D., Schaetz, B., & Kolb, B. (2012). mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity* (pp. 121–140).
Völter, M., Stahl, T., Bettin, J., Haase, A., & Helsen, S. (2013). *Model-driven software development: technology, engineering, management*. John Wiley & Sons.
Wachsmuth, G. H., Konat, G. D. P., & Visser, E. (2014). Language Design with the Spoofax Language Workbench. *IEEE Software*, *31*(5), 35–43.
Wolny, S., Mazak, A., Carpella, C., Geist, V., & Wimmer, M. (2020). Thirteen years of SysML: a systematic mapping study. *Software and Systems Modeling*, *19*(1), 111–169.
Wortmann, A. (2019, November). Towards Component-Based Development of Textual Domain-Specific Languages. In L. Lavazza, H. Mannaert, & K. Kavi (Eds.), *International Conference on Software Engineering Advances (ICSEA 2019)* (pp. 68–73). IARIA XPS Press.
Wortmann, A., Barais, O., Combemale, B., & Wimmer, M. (2020, January). Modeling Languages in Industry 4.0: an Extended Systematic Mapping Study. *Software and Systems Modeling*, *19*(1), 67–94.

About the authors

Arvid Butting received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2014 and 2016. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, software architectures, and model-driven development. You can contact the author at butting@se-rwth.de.

Robert Eikermann received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2012 and 2014. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, behavior languages, and model-driven software development. authorcontacteikermann@se-rwth.de

Katrin Hölldobler received her Ph.D. from RWTH Aachen University in 2018. Currently, she is a postdoctoral researcher at the Department for Software Engineering at RWTH Aachen University. Her research interests cover software engineering, software language engineering, model-driven development, and model transformation. You can contact the author at hoelldobler@se-rwth.de.

Nico Jansen received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2015 and 2018.

Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, software architectures, and model-based software and systems engineering. You can contact the author at jansen@se-rwth.de.

Bernhard Rumpe is heading the Software Engineering department at the RWTH Aachen University, Germany. Earlier he had positions at INRIA Rennes, Colorado State University, TU Braunschweig, Vanderbilt University, Nashville, and TU Munich. His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods as well as model-engineering based on UML/SysML-like notations and domain specific languages. He also helps to apply modeling, e.g. to autonomous cars, human brain simulation, BIM energy management, juristical contract digitalization, production automation, cloud, and many more. He is author and editor of 34 books including "Agile Modeling with the UML" and "Engineering Modeling Languages: Turning Domain Knowledge into Tools". You can contact the author at rumpe@se-rwth.de.

Andreas Wortmann received his Ph.D. from RWTH Aachen University in 2016. Currently, he is a tenured researcher at the Department for Software Engineering at RWTH Aachen University. His research interests cover software engineering, software language engineering, model-driven development, and robotics. He is a member of IEEE and its Technical Committee on Software Engineering for Robotics and Automation and serves on the board of the European Association for Programming Languages and Systems (EAPLS). You can contact the author at wortmann@se-rwth.de.

Appendix

Table 1 Overview of Language Components and their Intention

Language Component	Intention
CommonLiterals	provide a limited but useful collection of values usable in various modeling languages
JavaLiterals	provide additional values needed to support literals allowed in Java
CommonExpressions	provide side effect-free expressions commonly used in modeling languages
Assignment-Expressions	provide expressions with side effects that assign new values to variables
BitExpressions	provide expressions with side effects that operate on bits
JavaClassExpressions	provide expressions concerning classes needed in programming languages such as Java
BasicTypes	provide a limited but useful collection of types usable in various modeling languages
CollectionTypes	provide predefined generic types for Collections, Optionals and Maps useful for modeling
SimpleGenericTypes	provide custom generics that can be nested
FullGenericTypes	provide additional generic types such as inner types of generic types needed for programming languages such as Java
AssertStatements	provide the assert statement as known from Java constraints
CommonStatements	provide typical statements, such as method calls, assignment of variables, loops, conditions, and blocks
ExceptionStatements	provide statements for exceptions, including a Java-like try-catch notation
FullJavaStatements	provide exact Java statements by combining the other statement language components
LowLevelStatements	provide low-level control statements
ReturnStatements	provide return statements for Java methods
Synchronized-Statements	provide a statement for Java-like synchronization
VarDeclaration-Statements	provide statements concerning variable declaration and initialization

Table 2 Overview of Language Components and their Usage

Language Component	Used, e.g., in	Used, e.g., for
CommonLiterals	Java	Values for fields, variables or loops
	IOAutomaton	Values in Guards, Actions
JavaLiterals	Java	Values for Fields and Variables
CommonExpressions	Java	Conditions, comparisons, variable/field usages
	IOAutomaton	Expressions in Guards, Actions of Transitions
AssignmentExpressions	Java	assignments of fields/variables in expressions
	IOAutomaton	Expressions in Guards, Actions of Transitions
BitExpressions	Java	manipulating bits within expressions
JavaClassExpressions	Java	using `this` or `.class` within expression
BasicTypes	Java	Types of fields, variables, parameters, return types
	LightADL	Types of ports
ColletionTypes	Java	Types of fields, variables, parameters, return types
	FullADL	Types of ports
SimpleGenericTypes	Java	Types of fields, variables, parameters, return types
	FullADL	Types of ports
FullGenericTypes	Java	Types of fields, variables, parameters, return types
AssertStatements	Java	using asserts within method bodies
CommonStatements	Java	Method calls, assignments, if, loops, switch statements, and blocks
	IOAutomaton	Statements in Actions of Transitions
ExceptionStatements	Java	exceptions including try, catch, finally, and throw
	FullADL	Statements in Actions of Transitions
FullJavaStatements	Java	all statements
	FullADL	Statements in Actions of Transitions
LowLevelStatements	Java	Loops and conditionals
	FullADL	Statements in Actions of Transitions
ReturnStatements	Java	returns in method bodies
	FullADL	Statements in Actions of Transitions
SynchronizedStatements	Java	synchronize statements
	FullADL	Statements in Actions of Transitions
VarDeclarationStatements	Java	declaring local variables
	FullADL	Statements in Actions of Transitions