[JLR25] N. Jansen, A. Lüpges, B. Rumpe:
Lessons Learned from Developing the MontiCore Language Workbench: Challenges of Modular Language Design.
In: Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering, pp. 112-127, SLE '25,
ISBN 979-8-4007-1884-7, DOI 10.1145/3732771.3742717, ACM, Jun. 2025.

# Lessons Learned from Developing the MontiCore Language Workbench: Challenges of Modular Language Design

Nico Jansen
jansen@se-rwth.de
Software Engineering
RWTH Aachen University
Germany

Alex Lüpges
luepges@se-rwth.de
Software Engineering
RWTH Aachen University
Germany

Bernhard Rumpe
rumpe@se-rwth.de
Software Engineering
RWTH Aachen University
Germany

## Abstract

In software language engineering, composition and modular design are essential milestones to advance reuse in this domain. Although language engineering improves quality and development efficiency by incorporating reusable components and tooling, some disadvantages and limitations can inhibit sophisticated language development. Current research usually focuses on the conceptual advantages but neglects the often technical drawbacks resulting from the necessity for complex interactions for seamless integration. In this paper, we report on our experiences applying language composition with the language workbench MontiCore and elaborate on the conceptual and technical drawbacks of the intended compositional tooling. Using concise application examples, we demonstrate the challenges of a compositional parser, the complexity of the generated infrastructure, and how they scrape the limits of the underlying target programming languages and their compilers. This critical examination of the implications of modular language construction demonstrates the pitfalls of language composition in the large and should reveal potential for future research.

***CCS Concepts:*** • **Software and its engineering → Domain specific languages**; **Abstraction, modeling and modularity**.

***Keywords:*** Software Language Engineering, Language Composition, Domain-Specific Languages, Reuse, Modularization

**ACM Reference Format:**
Nico Jansen, Alex Lüpges, and Bernhard Rumpe . 2025. Lessons Learned from Developing the MontiCore Language Workbench: Challenges of Modular Language Design. In *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering (SLE '25), June 12–13, 2025, Koblenz, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3732771.3742717



*SLE '25, Koblenz, Germany*

ACM ISBN 979-8-4007-1884-7/2025/06
https://doi.org/10.1145/3732771.3742717

## 1 Introduction

The engineering of sophisticated software and software-intensive systems becomes increasingly complex. Approaches such as model-driven engineering (MDE) [48] aim to provide a suitable solution by representing the system under development on an abstract level, tailored for a specific purpose [49]. Models increasingly serve as central development artifacts [20] in developing complex cyber-physical systems (CPS) in various domains such as automotive [5], avionics [37], robotics [55], or medicine [47]. They adhere to modeling languages that formalize the concrete and abstract syntax and additionally define well-formedness rules and a formal mapping into a semantic domain [11] that determines the meaning of these models [31]. Modeling languages that are tailored to a specific application domain are also called domain-specific languages (DSLs) [9]. The development, evolution, and maintenance of such languages are investigated in the field of software language engineering (SLE) [36]. Research focuses on techniques, tools, and methodologies for the efficient provision of modeling and programming languages as well as appropriate tooling, such as parsers, well-formedness checking, analysis, interpreters, or code generators [9].

Since DSLs evolve, they incur the same development, maintenance, and troubleshooting expenses as any sizeable software project [19]. This becomes even more serious as the languages grow and new features are added. These features must be integrated into existing ones without breaking the already established functionality. Similar to classic software engineering, reusability is also becoming increasingly crucial in SLE in tackling these issues [6]. Over the years, various language composition techniques have been developed, including entire libraries of reusable language components [3] and corresponding compositional design patterns [16]. These enable the efficient development of large languages [28] and entire language families (e.g., UML [26] or SysML [25] derivates) from smaller building blocks with less effort and higher quality. Accordingly, reuse includes not only the concrete and abstract syntax of the incorporated components but also their parsers, as well as generated infrastructure and hand-written extensions.

In recent years, advances in this area have shown that reusability is a key factor in SLE. The respective techniques are also becoming increasingly sophisticated as they are designed in the context of mature language workbenches such as Xtext [18], MPS [53], and MontiCore [29] and implemented in their respective ecosystems. The corresponding generated language tooling is thus tailored directly to language composition and provided by the framework accordingly. This means individual language modules can be efficiently integrated and their infrastructure reused according to a library concept [39]. Approaches such as SCOLAR [45] even go one step further and attempt language composition across the various ecosystems, thus dealing with largely heterogeneous modules to be integrated seamlessly. The ongoing trend shows that reuse is still a relevant research topic in SLE and provides or at least promises many advantages for the future. These include faster development by embedding existing and tested components, which directly impacts quality. This also results in lower engineering costs compared to an utterly from scratch development. Furthermore, building on a solid basis of language components facilitates the subsequent integration of independent functionalities. SysML v2 [25] follows this notion with KerML [23], which is intended to facilitate the integration of other domain-specific formalisms in the future, which were also built based on the KerML metaconcept.

Despite the many advances and publications on language libraries [3] and their application in constructing increasingly sophisticated language families and tooling [28], there are still major challenges in modular language design. Neither are these challenges always apparent nor do their weaknesses reside at a conceptual level. They are often an intrinsic part of the composition problem to be solved, or they may be due to limitations of the underlying general-purpose programming language in which a DSL is implemented. Since many publications often report on new findings and successes in their application [15], this paper aims to provide insights into the challenges of applying language composition techniques, conceptually and in practice. One apparent example is that the provided software infrastructure is becoming more complex and more complicated for developers to use methodically. Our observations are based on practical experience with MontiCore, a language workbench with substantial composition capabilities. We report on our insights from the last years of intensive application of modular language construction and highlight still challenging issues. Thus, we elaborate on issues in compositional SLE and how they were addressed in MontiCore. We systematically highlight the underlying challenge and discuss solutions, solution strategies, or at least workarounds if no general solution is conceptually feasible. As some of MontiCore's solution approaches may entail additional challenges, we also discuss these and differentiate between implementation-specific and conceptual risks. Furthermore, there are general concerns, for instance, regarding the scalability of the artifact size of the underlying programming language, which can be mitigated but not solved entirely. Although the findings originate from SLE within the MontiCore ecosystem, the underlying problems are often more profound. They are intended to highlight the potential pitfalls of compositional SLE as well as indicate still-existing research gaps. Using a simple running example, we show where current technologies reach their limits or where the promised advantages are overshadowed by disadvantages that are not always entirely obvious. Using these findings, we compare related approaches from other language frameworks to highlight the strengths and weaknesses of different techniques. Thus, the main contributions of this paper are:

- An application-oriented identification of challenges in modular language construction based on the MontiCore language workbench
- An in-depth analysis of the root causes of these issues
- A comparison to alternatives from other language ecosystems
- A pointer for future SLE research into which pain points require further attention

The remainder of this paper is structured as follows. In section 2, we give a short overview of state-of-the-art language workbenches and related work. Our running example for highlighting challenges in the application of modular language design is introduced in section 3. We elaborate on these challenges in section 4. The results of this paper are discussed in section 5 and compared to alternative solutions in section 6. Finally, section 7 concludes.

## 2 State of the Art and Related Work

Various language workbenches provide support for modular language design [17], as shown by the various workbench challenges of [17]. Modularity of languages affects not only the syntax, but also concerns validation, semantics, and editor services. In this section, various language workbenches and features are introduced, with a more concrete comparison on a feature basis occurring in Section 4.

Due to the author's proficiency, we reduce the examined language composition mechanisms to three of the textual language composition mechanisms supported by MontiCore [7]: *Language inheritance*, *language extension*, and *language embedding*. Similar to object-oriented inheritance, language inheritance allows a language to `extend` other languages by reusing their language elements, i.e. their nonterminals and terminals. It is even possible to override the nonterminal productions of a super language. If the inheritance is conservative, i.e. all valid models of an inherited language are also valid in the new language, we speak of *language extension*. For example, adding only optional elements to a nonterminal is such a conservative extension.

Similar compositions can be carried out with the Eclipse Modeling Framework (EMF)-based Melange language workbench [14], where the syntax of various languages can be merged (similar to language embedding), inherited in the sense of classical object-oriented programming, or reduced by means of a slice operation. By using aspects in combination with the EMF, semantics can be executed via the metamodel. Due to being based on the EMF, reusability is even possible with (some) other tools, such as Xtext.

Xtext [18] explicitly defines a separation between the concrete syntax and the abstract syntax via the metamodel, in which the language defining `EPackage` utilizes the `returns` keyword to distinguish the returned type of a rule. Similarly, Xtext supports language inheritance (via the `"with"` keyword) and language embedding (via the EPackage import mechanism). To work around keyword ambiguities, Xtext allows an optional caret (^) prefix for identifiers which is stripped during the parsing[1].

Spoofax [35] is a language workbench for textual DSLs with Eclipse IDE support. A language is specified declaratively using multiple meta-DSLs, with the syntax using the Syntax Definition Formalism 3 DSL, the type system and name bindings using Statix, and the semantics using Stratego. From a language definition, Spoofax generates language-specific Eclipse IDE plugins.

A fourth, existing language composition mechanism is *language aggregation*, in which a model is described in multiple artifacts of different languages. This mechanism does not lead to a unified, composed language and is therefore not discussed in this paper. The concept of parameterized language composition [44] allows another dimension of language variability by substituting holes in the language definition during the composition. A limited realization of language parameterization using inheritance and interfaces with MontiCore is presented in [4].

Many workbenches use ANTLR to generate the backing parsers [22], which does not inherently support the composition of language definitions [17, 42]. Nevertheless, the adaptive LL(*) parsing approach is the backbone for various kinds of composition of multiple workbenches.

In our work we focus on a language workbench that supports textual notation DSLs and uses free-form editing, i.e. modelers are able to develop models like conventional source code [17]. Other notations and editor forms, such as graphical or tabular ones exist, which share some challenges but alleviate others due to their core design. MPS, for example, achieves the composition of physical units and state machines into C code using projectional editing [54], which allows a developer to view and edit the same state machine in different views, such as various tables, graphical, etc.

[1] https://eclipse.dev/Xtext/documentation/301_grammarlanguage.html

## 3 Running Example

UML statecharts [26] describe the behavior of a system and can be applied in many scenarios. The preconditions of transitions are often modeled using OCL constraints, which illustrates a perfect use case for language composition. For this example, we want to model statecharts consisting of states, transitions, and preconditions. We utilize the MontiCore language workbench [29] to (a) describe a monolithic language and (b) compose an expression and an automaton language into a modular statecharts-with-preconditions language. More precisely, a language consists of its concrete syntax, abstract syntax, well-formedness rules, and semantics. The concrete and abstract syntax can be described using a MontiCore grammar DSL, with the well-formedness rules and semantics being given as Java code [29].

```
1  grammar StatechartsMonolith extends MCBasics {   MG
2   Statechart = "statechart" Name
3                "{" (S | T)* "}";
4   S = "state" Name ";";
5   T = from:Name "->" to:Name
6         ("[" precondition:Expr "]")? ";";
7   Expr = Name
8         | "("  Expr ")"
9         | "!" Expr
10        | left:Expr "&&" right:Expr
11        | left:Expr "||" right:Expr;
12 }
```

**Figure 1.** Monolithic definition of a MontiCore DSL for statecharts with expressions as preconditions of transitions.

The monolithic grammar definition of Figure 1 is named `StatechartsMonolith` (cf. l. 1). This grammar is not entirely monolithic, as it extends the `MCBasics` grammar [29], which defines a `Name` nonterminal for identifiers and further nonterminals for the handling of comments. The start rule `Statechart` (cf. ll. 2-2) accepts the terminal `"statechart"`, followed by an identifier with the `Name` nonterminal and any number of states and transitions by referencing the `S` and `T` nonterminals respectively. Finally, a left-recursive `Expr` nonterminal for simple expressions is defined. The left-most, `Name` referencing, alternative has the highest priority, e.g., `isActive`, followed by parentheses around an expression, e.g., `(isActive)`, the negation operator `!`, the logical and operator `&&`, and finally the logical or operator `||`.

For comparison, we also provide the equivalent modular MontiCore grammar for expressions and automata in Figure 2, with the composed statecharts with preconditions grammar given in Figure 4. In the `Exprs` component grammar, the `Expr` interface extension point (cf. l. 2) is defined, and with the `component` keyword the grammar definition is marked as a component grammar, for which no parser, etc. is generated. The name referencing expression is declared in line 3 and the parentheses expression in line 4. The operator priority of the `Expr` nonterminal interface is determined via the priority given in the angled brackets. Next, the `BExprs` grammar

binds the logical not `!`, logical and `&&`, and the logical or `||` operator expressions to the expression extension point.

Figure 4 shows the MontiCore grammar combining the expressions and automaton grammars of Section 3. The statecharts grammar extends both the `BExprs` and `Automaton` grammars in line 1 and specifies that the `Aut` production is the start rule of this language in line 2. Composite grammars can specify a start production which is defined in either a component grammar or in the composite grammar itself. Subsequently, line 3 replaces the *automaton* keyword with the keyword *statechart* (cf. C6). Finally, we extend the transition production by embedding an optional expression as the transition's precondition. While this transition extension is conservative, i.e. automata transitions without a precondition are still correct statecharts transitions, the keyword replacement means that automata models are not syntactically correct models of the statecharts language. Thus, the composition of the statecharts language is not conservative. In Figure 3 the relationship between the statecharts grammar and its super languages can be seen, with the extension of the transition AST class `ASTT` of the automata grammar by an identically named class in the statecharts grammar. The interface extension `implements AutElem` of the transition production is implicit and such implementing behavior cannot be overridden.

The relationship between the modular language components and their AST classes is visualized in Figure 3, with the `Expr` grammar defining the `ASTExpr` interface in the top right corner. The AST classes of the various nonterminal productions that implement this interface can be seen in both the `Expr` and `BExpr` grammars. The `Automata` grammar is defined in lines 13-19 of Figure 2 and its AST classes are shown in the top left corner of Figure 3. Its `Aut` nonterminal is composed of instances of the `AutElem` interface nonterminal, which are the state and transition nonterminals `S` and `T` respectively.

We assume that the expressions, boolean expressions, and automaton languages are already provided as part of a language library. This enables DSL designers to re-use these language components.

```
1  component grammar Exprs extends MCBasics {                    MG
2   interface Expr;
3   NameE implements Expr<350> = Name;
4   ParenthE implements Expr<310> = "(" Expr ")";
5  }
6  grammar BExprs extends Exprs {                                 MG
7   NotE implements Expr<190> = "!" Expr;
8   AndE implements Expr<120> = left:Expr
9         "&&" right:Expr;
10  OrE  implements Expr<117> = left:Expr
11        "||" right:Expr;
12 }
13 grammar Automaton extends MCBasics {                           MG
14  Aut = "automaton" Name "{" AutElem*  "}";
15  interface AutElem;
16  S implements AutElem = "state" Name ";";
17  T implements AutElem = from:Name "->"
18        to:Name ";" ;
19 }
```

**Figure 2.** Definition of a library of re-useable and modular MontiCore DSLs for expressions and automata.

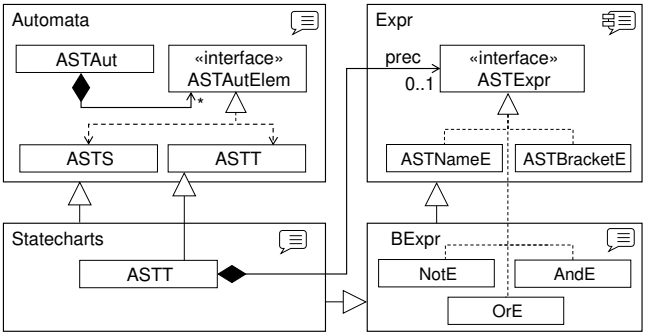


**Figure 3.** The languages of the running example along with their language inheritance relationships and nonterminals.

```
1 grammar Statecharts extends BExprs, Automaton {                 MG
2  start Aut;
3  replacekeyword "automaton":"statechart";
4  @Override T = from:Name "->" to:Name
5              ("[" precondition:Expr "]")? ";"
6 }
```

**Figure 4.** Definition of the statecharts language re-using the expression DSL and extending the automata DSL.

## 4 Challenges

The design and development of modular languages and their tooling presents additional challenges. In this section, we present some of the challenges we have tackled during the continuous development of the MontiCore language workbench. We also present possible solutions we have found. Design choices made by other language workbenches in relation to these challenges are presented in Section 6.

**C1: Grammar Defined in Multiple Places** Splitting the language definition into multiple grammar files makes it more difficult for language designers to understand the language as a whole. By providing clear documentation with each modular grammar, this problem can be mitigated somewhat. For example, the documentation of our expression component grammar[2] would explicitly mention the `Expr` interface, the available expressions, and how to bind additional nonterminals to the `Expr` interface. Extensive use of MontiCore in a teaching environment has shown that the

[2]https://github.com/MontiCore/monticore/blob/dev/monticore-grammar/src/main/grammars/de/monticore/expressions/Expressions.md

documentation is often not enough, be it due to an overwhelming amount of information, its difficulty to find, or other reasons. Thus, MontiCore provides LSP-based editor support for its grammar DSL, providing code completion of available productions, jumps to the definition of a production, and displays of a production's documentation on hover.

MontiCore generates a parser for every grammar that is not marked with the `component` keyword. In our example, we thus generate parser infrastructure for the `BExpr`, `Automaton`, and `Statecharts` grammars. In addition, access to the compositional parser is delegated via the mill design pattern (cf. C12), such that in our example the developers of automaton tooling can parse transitions with preconditions if the mills have been initialized correctly.

By building upon ANTLR [43, 42], the MontiCore languages can stand on the shoulders of a giant which enables entrusting ANTLR with solving many of the challenges of non-modular parser construction, leaving us only with the modular challenges. The MontiCore generator thus combines all terminals and nonterminals into a single ANTLR g4 file, taking MontiCore-specific features, such as interfaces, keyword replacement, etc. into account.

**C2: Ambiguity in Production Rules** Ambiguity may arise in the names of non-terminals defined in different component grammars. Non-specific names, such as `Element`, increase the chance for this issue to arise. While a solution would be possible via slicing (cf. C5) or Java-like packages, language designers using MontiCore are instead discouraged from employing ambiguous naming, obviating the necessity for solving this problem. In our running example, the nonterminal describing elements of an automaton is thus called `AutElem`, the prefix `Aut` relating to the `Automaton` grammar. Thus, MontiCore uses the conceptual solution of avoiding ambiguous names. When dealing with ambiguity in the generated code, e.g. due to explicitly overridden nonterminals, the technical solution of fully qualified class names is used.

**C3: Black Box Reuse of Nonterminals** By reusing the syntax definition from language libraries, language designers are able to include language components without the effort of re-engineering parts of a language. This composition of languages expands the set of accepted models. In our example, the statecharts language has the need for expressions. By decoupling the definition of the abstract syntax (in the `Exprs` grammar) from some concrete implementations (in the `BExprs` grammar), language elements, such as boolean expressions, can be offered in a bundle. Language designers of modular languages can specify the need for, e.g., an expression, but leave the concrete available expression implementation up to language designers using their modular language. In object oriented programming, the concept of interfaces is used to allow such an abstraction, which we extend as a conceptual solution for black-box reuse.

With MontiCore, language designers utilize nonterminal interfaces to describe abstractions of nonterminals. Interface nonterminals can be either defined without a right-hand side, thus with no information except their existence, or with a required abstract structure. The `Expr` interface of the expressions component language is one such example without a right-hand side. Nonterminals can implement interfaces and thus act as a viable interpretation of the given interface. For example, the `NameE` nonterminal of our example is an implementation of the `Expr` interface. In the generated AST class diagram, interfaces are represented by a Java interface and `implements` relationships. During the parser generation, all implementing nonterminals of an interface are normalized into one rule for each interface with the nonterminals as alternates. Figure 5 illustrates this normalization with the example of the `NameExpr` and `AndE` productions. For expressions, the order of operation is especially important.

The concept of interfaces, while being a key factor of black-box language reuse in MontiCore, introduces several new problems:

- Increased complexity of a system, resulting in longer build and startup times (C18).
- Temptation to use simple interface names, such as `Element`, leading to the ambiguous naming of nonterminals (C2).
- Necessity to declare an operator precedence for recursive nonterminals (C4).
- The difference between a language's abstract syntax and its normalized ANTLR syntax (C9).
- Interfaces are not a part of the curriculum of formal language design and thus parse errors must be understandable (C10).

**C4: Precedence with Black-Box Reuse** With interface nonterminals, the first nonterminal implementing an interface has an implicit higher precedence than the second nonterminal. When combining multiple component grammars, this implicit precedence is obscure and limiting.

In MontiCore, language designers can thus assign an explicit precedence to nonterminals. In our example, `NameE` has a higher precedence (`Expr<350>`) than `AndE` (`Expr<120>`), resulting in the order for `Expr'`.

Due to the use of a number for the precedence, gaps have to be left during language design to facilitate further extensions. Instead of using the less-than relation of natural numbers, a direct relation between productions would eliminate the need for gaps (i.e., `NameE>AndExpr`), but reduce readability. ANTLR itself supports operator precedence by evaluating expressions from the left to right, but it does not support interfaces, requiring this transformation step.

**C5: Importing a Slice of a Language** Language libraries often provide a wide range of language elements to support all kinds of use cases. Often language designers only need a

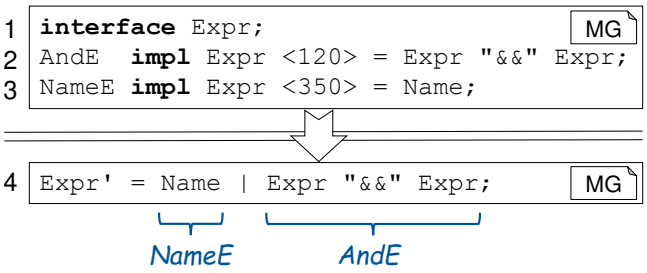


**Figure 5.** Transformation step to normalize MontiCore interface nonterminals to enable black-box grammar reuse.

certain subset of those elements, creating the need for slicing an imported grammar.

With modular language imports, MontiCore always includes all defined terminals and nonterminals. During the design we have opted to not support selected slicing of imported languages, instead sensible component boundaries must be drawn during the design of component libraries to enable a partial selection. If necessary, language designers can override nonterminals with a nonsensical keyword that is impossible to parse. Both import slicing as well the bundling of multiple component grammars in one library pose another risk for the accidental use of sliced language components (C13).

**C6: Transmute Keywords into new Domains**

The concrete syntax of a textual DSL is strongly influenced by the keywords of the language [10], such as `"state"` or `"automaton"`. Especially when reusing component languages into the domain of a DSL, a language designer may want to rename keywords.

In our running example, we rename the implicit keyword `"automaton"` of the automaton grammar to `"statechart"` to indicate a different domain. As keywords can act like reserved words for a language, an identifier may not be possible if the identifier matches a keyword, for example, an automaton-state may not be called `"automaton"`. With language composition, the list of reserved keywords must be carefully managed (cf. C8). With MontiCore, we provide an option to replace a keyword in only the concrete syntax via the `replacekeyword` statement. The AST is not affected by this. If applied, the parser and its inverse, the pretty printer, will be built using the provided replacement keyword. Since the generated parser itself is not modular (cf. 1), this keyword replacement is trivial. Pretty printers, which act like model-to-text transformations turning an AST into a textual model [29], are based on the modular visitor infrastructure (cf. 16). Thus, replacing keywords of a superlanguage requires the generation of and delegation to modular pretty printers of the superlanguage (C12).

While it would be possible to use an overridable nonterminal for each terminal, this would also affect the AST and add further complexity (C18).

Instead, we decided to add this mechanism to replace terminals only in MontiCore's parser generation as a grammar transformation step, and thus included within the generated ANTLR parsers. The divergence between the ANTLR parser and the generated infrastructure is similar to C9 and has to be considered when reporting parser errors (C10).

**C7: Non-Conservative Language Extension**

Replacing a keyword is one such non-conservative extension since models of the original language are no longer valid models of the new composite language. So instead of replacing, it is thus possible to add a new alternative keyword. If we were to apply this in our example, the models could both start with `"statechart"` or `"automata"`, and the exemplary statecharts with preconditions language would be a conservative language extension of its components. While the optional precondition of the statecharts `T` nonterminal is a conservative extension, the replaced keyword causes the non-conservative language extension.

Language designers have to be aware if a non-conservative language extension is desired. The building of new elements of a super language will fail due to missing an attribute in tooling code of the super language, despite delegating the instantiation to the new language's element (cf. C12). Language analysis tools, such as within the generator or editor, can be used to detect this during language design.

**C8: Handling Reserved Keywords** When composing languages, the set of the language's reserved keywords is combined into a single set. This can be an unintuitive reason for non-conservativeness (cf. C7). Instead of providing compatibility by modifying the concrete textual syntax of a language by replacing a keyword, it is instead possible to contextualize the keyword [42, 10]. With contextual keywords, the lexical disambiguation is performed by means of the context in which a keyword occurs. In MontiCore, this is enabled via the `nokeyword` statement [7]. The context in MontiCore's case is the position within the parse tree. Thus, contextual keywords only act like keywords at certain positions within a language. While contextual keywords improve language composition for language designers, the complexity of the underlying ANTLR parser increases significantly, resulting in C9.

Another solution we provide is the functionality to specify lexer modes, in which only a limited set of keywords are retained. While this feature was originally used for languages such as XML, it can also be used with language embedding to separate the keywords of DSLs by effectively using a different lexer for each language.

**C9: Divergence of Declared Syntax and Parser** MontiCore uses ANTLR internally to generate the parser of a language and the generated ANTLR parser constructs a parse tree by tracking the individual rule contexts, which are valid abstract syntax tree representations [42]. But ANTLR does not support language composition, interfaces (cf. C3), overridden nonterminals (cf. C7), replaced keywords (cf. C6),

etc. MontiCore thus has to translate the syntax of language into a valid flattened ANTLR language. Reusing these rule context data structures and exposing them to language tool developers is not feasible, as they are not compatible with the previously mentioned addition of interfaces, a modular visitor infrastructure (cf. C16), symbol table infrastructure, etc. The rule normalization step of interfaces (cf. C3) inlines interfaces in the ANTLR grammar, and to support a modular visitor infrastructure methods to facilitate double dispatch have to be added (cf. C16). We therefore generate our own AST classes and need to construct an instance of this data structure during parsing. The straightforward way is to use semantic actions to build our AST at the same time as ANTLRs rule contexts are created. Figure 6 shows the ANTLR rule for the state nonterminal production (line 1) using semantic actions. ANTLR's rule context is expanded by an `ASTS` variable `ret` (line 2) and a variable `b` containing a new state builder. We use the mill pattern in line 3 to retrieve a new instance of the builder (cf. C12). Next, the `STATE` terminal is expected (line 5), followed by a reference to the `name` token which is then referenced via `t_name` in a semantic action to set the builder's name (line 6). The last element of the rule is the token of a semicolon. Finally, the builder's `build()` method is called and its value is returned.

```
1 S = "state" Name ";";                                   MG
2 s returns [ASTS ret]                                    G4
3 @init{b = AutomataMill.sBuilder();}
4 @after{return b.build();}
5 :STATE
6  t_name=name {b.setName(t_name.getText());}
7  SEMICOLON;
```

**Figure 6.** Example of the methodology for building the AST of the state nonterminal production (line 1) within the ANTLR parser. Highlighted are the parts of the rule required for the two-phased parsing and AST creation approach.

Special attention must be taken with left recursive productions, such as expressions, because ANTLR does not permit a semantic predicate in the first position of a left-recursive rule. Consequently, the builder initialization must be included in the semantic action subsequent to the first token or nonterminal reference.

With MontiCore, we have experienced that the performance penalties due to parser lookahead are drastically higher in combination with semantic actions. Due to the presence of semantic actions, the LL(*) analysis of the ANTLR-generated parser was no longer able to compute the closure of the current state within the augmented transition network (ATN) during lookahead computation [43] in a straightforward manner. This resulted in the textual input of an expression `!(f1 && f2) && ...` repeated 30 or more times taking over an hour to parse.

The ATN for the `AutElem` interface nonterminal is shown in Figure 7. The upper half shows the ATN states with semantic actions, while the lower half shows the same production without semantic actions. Both ATNs start with a square decision node, followed by either a reference to the state or transition rules. The semantic actions in the upper half then end the epsilon transition closure, while the ATN in the lower half is able to continue the closure computation (depending on the state and transition rules). For left recursive expressions, both the cost for non-trivial closure computation and the complexity of the ATN increase.

In order to ensure that no semantic actions inhibit the LL(*) algorithm, we move the AST creation into a second step after the parsing. The lower part of Figure 7 illustrates the augmented transition network (ATN) with a postponed AST creation. Notably, the action transitions are not present, allowing the closure computation to continue without being interrupted by semantic actions [43]. By removing the AST-creating semantic actions from the parser generation, i.e. the unhighlighted parts of lines 2-7 of Figure 6, we have to translate the rule context of the ANTLR-generated parser into MontiCore's AST. We accomplish this by traversing the rule context tree after the ANTLR generated parser has finished parsing. Transformation steps, such as for interfaces or constant groups, have to be translated as well.

This separation between the abstract syntax data structure and the concrete syntax further allows us to optimize the rules, e.g. by removing ambiguities and thus abstracting the details of parser construction from the language designer. The parser generator is also able to add specific, syntactically invalid, alternatives for common errors that are difficult for ALL(*) parsers to recover from [46, 42]. For example, a missing semicolon at the end of a state declaration can lead to incomprehensible errors being displayed to the user. By only including these labeled failure alternatives only within the parser, they are not present in the AST, where they would represent unnecessary information.

Another benefit of this two-phased approach to parsing and AST creation is the reduction of MontiCore-specific Java code in the (generated) ANTLR parsers. This allows the use of ANTLR-specific tooling, reduces the effort to adapt MontiCore grammar parsers in non-Java environments, and reduces the size of the generated ANTLR parser class. Previously, we were able to construct grammars, such as the domain-specific transformation language (DSTL) for Java-like models, which broke Java's class size limit for their parser class due to their number of semantic actions.

**C10: Understandable Parser Errors**

We claim that language designers should never be exposed to the ANTLR parser, but incorrect input models lead to

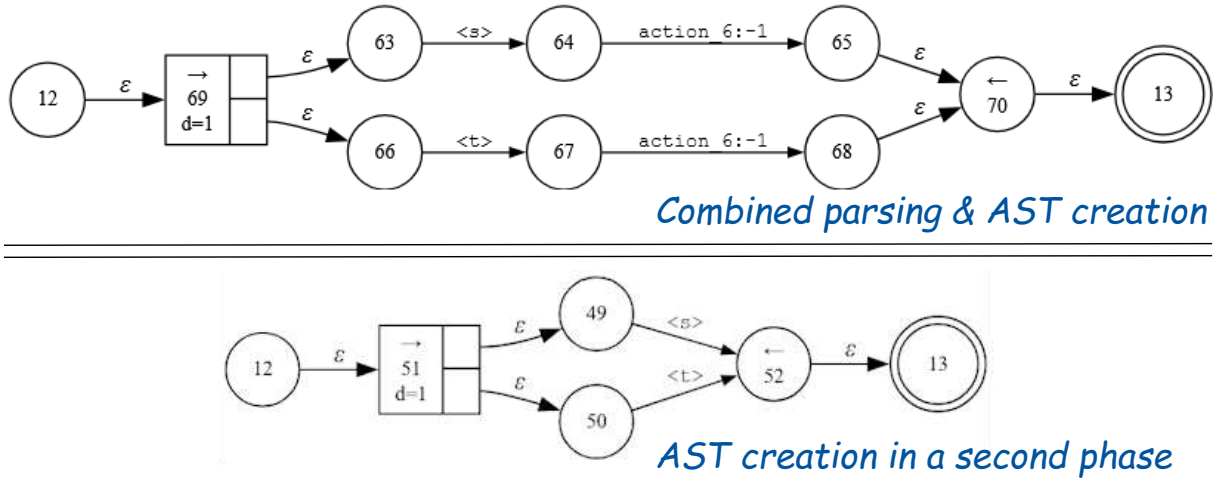


**Figure 7.** The augmented transition networks (ATNs) of the `Aut` nonterminal production with the combined parsing and AST creation (upper half) and the creation delegated to a second phase (lower half) with a lower number of non-epsilon transitions.

ANTLR parser-specific error messages and parser stacks. In addition, modelers are not necessarily the language designer, and thus may not understand the language in its entirety. MontiCore therefore translates these error messages back into a format that can be understood by the model developer with minimal language knowledge and without knowledge of ANTLR. Due to the described differences between the ANTLR rules and MontiCore productions (cf. C9), e.g. due to interfaces, this translation step is especially necessary. Similarly, when reporting which terminals are valid next candidates, contextualized keywords (C8) have to be considered.

**C11: Black Box Reuse of Code**

Generally, language composition comprises more than just integrating concrete and abstract syntax. It involves all facets of the incorporated components, including generated and manually extended artifacts, as well as additional tooling. This poses a challenge to black-box reusability, as functionalities can (and should) only be implemented against the API of the current language level [16]. While language developers can certainly design a DSL for extensibility, it is neither possible nor practical to foresee all subtypes in future composition scenarios.

The MontiCore generator synthesized Java classes and interfaces from the abstract syntax [29]. In our automaton example (see Figure Figure 2), the nonterminals `Aut`, `S`, and `T` result in the Java classes `ASTAut`, `ASTS`, and `ASTT` in the `automaton._ast` Java package. The `AutElem` nonterminal interface results in a similarly named `ASTAutElem` Java interface. The AST classes of the interface implementing productions also implement the AST interface, i.e., `ASTS` and `ASTT` extend the `ASTAutElem` interface. Extends relationships between productions are similarly represented using Java's `extends` inheritance between the AST classes. Overriding of nonterminals results in a new nonterminal AST class for the overriding language, extending the original AST class. In our example, the statechart's `statecharts._ast.ASTT` class extends the `automaton._ast.ASTT` class. Ambiguous class names between grammars are avoided in the generated code by using fully qualified names and package names that include the grammar name.

The black box composition of code written against the derived AST, symbol table infrastructure, etc., introduces new challenges, as developers must expect gaps and redefined language elements:

- Instantiation of new AST elements that may be overridden is one such challenge. For composed languages, the instantiation must be delegated (C12).
- Addition of new operations on the composed AST or symbol table of the model (C16).
- Java's method invocation is based on the compile-time type of the arguments (C17).
- The amount of synthesized classes leads to an increased compilation and startup time (C18).

**C12: Instantiation of Components**

Considering our modular running example, the developer of the `Automaton` DSL (cf. Figure 2) has only knowledge about the known language components up to this level (in this case, `Automaton` itself and the incorporated `MCBasics` for general token definitions). Consequently, only the corresponding generated artifacts of these languages are accessible implementation-wise. While this fact is sufficient and desired for implementing corresponding tooling for the `Automaton` DSL, a language engineer cannot anticipate that the production for transition is overridden and extended on the `Statecharts` level (cf. Figure 4). Let us further assume that an algorithm has been provided for the `Automaton` DSL

that completes missing transitions in the model. This simple functionality should also apply to the `Statecharts` in a black-box fashion. Even if the transition itself is altered, the analysis and addition algorithm stays the same. However, this poses the challenge that, for an automaton, only corresponding transitions of this level can be created. The constructor call

```
new ASTT()
```

would, therefore, instantiate the wrong type for Statecharts, which disrupts black-box reusability. While modern software engineering practices discourage the use of constructors anyway and instead recommend design patterns such as the builder pattern [21], this only shifts the problem to the next level of indirection since builders also have to be instantiated again.

To tackle this issue, MontiCore provides a static builder for builders mechanism [16], which is configured for the corresponding language level and can adaptively provide the correct instance of an object. Mills recognize the language hierarchy and can be exchanged via static delegates [29]. In this way, they establish an API at their own language level, which can be used in composition scenarios due to the delegate's adaptive interchangeability. This allows a language developer to implement against the provided API of their own mill without compromising black-box reusability at other levels. Applied to our example, the call

```
AutomatonMill.tBuilder()
  // input parameters
  .build();
```

always yields the correct transition instance at both the automaton and Statecharts levels. In MontiCore, the mill is versatilely used, among others, for the builder, visitors, and the symbol table, but introduces additional challenges. The static implementation is a problem when aggregating multiple languages simultaneously (C14) and the need to use the mill is not obvious to new tooling developers (C15).

**C13: Usage of Non-Included Language Components**

By defining multiple component languages within one project and then publishing the languages as a single library, it is a common occurrence to inadvertently develop code against data structures or infrastructure of a component language, which is not part of the composite target language. For example, adding an expression using the `new` operator to an AST element of a Haskell model. The Java compiler will not find this defect, as an expression using the `new` keyword is still an expression, but the visitor infrastructure (cf. C16) will be unable to correctly traverse ASTs containing these expressions at runtime. As the visitor infrastructure is used for i.e. the symbol table, type-system, etc., finding the causes for issues caused by incorrectly initialized mills is unintuitive. Within the MontiCore ecosystem, the MontiCore language component (MLC) language can be used to describe which artifacts, such as source files, a language component consists of and which relationships between language components are allowed. An MLC-related tool is then used to check for illegal relationships. Another solution are Java 9 modules, but those are restricted to the Java ecosystem.

**C14: Simultaneous Multi Language Usage**

Mills, by their nature, delegate to a single static instance. Thus, the simultaneous use of two languages containing expressions is therefore impossible, since the expression mill will only be able to delegate to the mill of one composite language. Switching between two languages with MontiCore therefore requires that the static state of the mills is also switched. To avoid the violation of the tight coupling principle by using static variables and other disadvantages of a global, static state, dependency injection [13] is a valid alternative. Depending on the injection framework used, multiple instances of an expression mill would be possible.

**C15: Consistent Usage of the Mill Pattern**

The biggest challenge with mills for new tooling developers is to see the need for them. When developing tooling for a particular language, the use of mills is not inherently intuitive and, worse, not required. For example, an automaton tooling developer might write `new AutomataTraverser()` in their tool that reports the number of transitions and the closure of a state. When used with automata models, the tooling works. But when it is used with statechart models, transitions with preconditions would not be counted and target states of these transitions would not be included in the closure. MontiCore's modular visitor infrastructure reports the occurrence of incorrect traversers at runtime, i.e. an automata traverser instead of a traverser of the statecharts language, but these errors only occur with the composition of a language and its tooling. The original developer of the automata tool may not be aware of these issues. Hiding the internal code, i.e. traverser implementations, constructors of AST classes, etc., from the tool developers, by means of visibility, modules, or splitting of an API, are possible solutions.

**C16: Compositional Visitor Pattern**

In SLE, it is often necessary to traverse the AST several times after parsing and to perform operations or checks on certain node types. Appropriate design patterns, such as visitors [21] or tree iterators, are suitable for this. Thus, MontiCore provides a DSL-specific realization of the visitor pattern, for traversing the AST and symbol table. This enables processing each node of the parsed model and executing analyses or calculations without having to adapt their classes. This promotes the separation of concerns [40] and is particularly useful when integrating many different functionalities. For each type of AST node (i.e., ultimately for each nonterminal `X`) and each symbol kind, the language workbench generates four methods by default [16]:

- `visit(ASTX node)` is called when entering the node and contains the execution logic

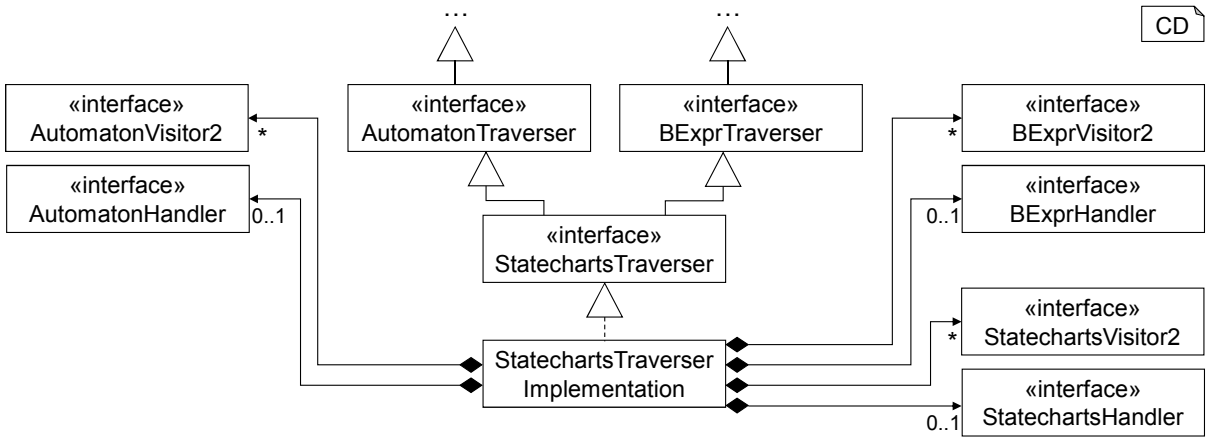


**Figure 8.** Simplified illustration of the compositional visitor infrastructure generated for the `Statecharts` language.

- `endVisit(ASTX node)` is called when leaving the node (also contains execution logic)
- `traverse(ASTX node)` defines the (default) traversal strategy of all nodes to be visited
- `handle(ASTX node)` manages proper execution order by calling `visit`, `traverse`, and `endVisit`

The actual realization of the visitor pattern in MontiCore languages has a compositional structure. This means a `Visitor2` interface is generated for each composed sublanguage, containing the `visit` and `endVisit` methods. These interfaces usually have to be implemented to execute operations on the AST. Additionally, a `Handler` interface exists for each sublanguage comprising the corresponding handle and traverse methods. It allows modifying the standard depth-first traversal strategy and invocates the `visit` / `endVisit` methods. While these two interfaces operate independently, a `Traverser` interface reflects the concrete inheritance hierarchy of all incorporated sublanguages. The language-specific handlers and visitors are then hooked into instances of these traversers. A traverser thus represents the concrete interaction point with all AST nodes and operates as a delegator and orchestrator for the language-specific interfaces for concrete operations.

Figure 8 presents a simplified example of this compositional architecture applied to the modular `Statecharts` running example. Each incorporated sublanguage has a corresponding traverser interface, realizing the inheritance hierarchy of the components. Additionally, a corresponding class realization (cf. `StatechartsTraverserImplementation`) is generated. Instances of this class represent the access point for the AST. They comprise an optional handler of each sublanguage for altering the traversal strategy on respective node types. Analogously, the traversers can carry multiple visitor instances of each sublanguage for defining node-specific operations. These lists of visitors also allow for parallel execution of different functionalities during a single traversal step.

Please note the visitor interface has the suffix "2". It helps distinguish it from an earlier visitor infrastructure in MontiCore, which did not yet separate between traverser and visitor [32]. This previous version had two main disadvantages: (i) A configured `"Visitor1"` only knows the currently included node types to be traversed. It could not handle unknown node types when incorporating this visitor in a new language composition scenario. Thus, a language engineer had to manually create a new class, extending the given visitor and implementing the most recent `"Visitor1"` interface to enable proper traversal of the new AST structure. In contrast, a traverser is always provided directly via the mill, which means that the correct, most language-specific variant is always provided adaptively. That is, the `Visitor2` implementations are just attached to the traverser, enabling black-box reusability. (ii) The second disadvantage was technical, such that diamond inheritances at a grammatical level were multiplied at the implementation level of the visitors. This, in combination with successive overloading and overriding methods, led to very long compile and class loading times for large languages.

For modular language construction, the presented evolution of the visitor pattern has already shown that the preference for delegation over inheritance had a positive impact both conceptually and technically. Nevertheless, this change can only mitigate the problem and not eliminate it entirely. The generated infrastructure is still large and complex, with further possible optimization options (cf. C 18). MontiCore itself also uses the generated visitor infrastructure to realize further generated functionalities. For example,

well-formedness rules (so-called context conditions), the construction of the symbol table, and the automatically provided pretty printers are based on visitors. In practice, however, this current architecture has become a sweet spot between efficiency and usability.

**C17: Language Extension and Java's Parameter Based Method Selection**

MontiCore supports non-terminal interfaces and inheritance on non-terminals. Consequently, the synthesized AST node types also inherit from each other. However, for the visitor infrastructure, this also requires ensuring that all nodes are visited in their most concrete type during traversal to execute the correct behavior. The generated architecture is based on Java, which only supports single dispatching. Thus, a traverser cannot directly call the handle method when reaching a new node during climbdown, as otherwise, it would not be possible to resolve the most concrete type. As an example, consider the modular `Statecharts` grammar introduced in Figure 4. When traversing a transition (ll. 4f.), we can have an arbitrary expression as a precondition. As only the `Expr` interface is exposed at this level, a traverser cannot determine via single dispatching whether it is a `NotE`, `AndE`, or `OrE` expression. For this purpose, the traversers in MontiCore emulate a double dispatching step by performing a kind of handshake with the respective AST node, which is always called in its most concrete type due to polymorphism [8]. Therefore, each AST class[3] has an `accept(traverser)` method, which receives the current traverser instance as a parameter. Applied to our example, this means that the traverser now calls `precondition.accept(this)` within the traverse method of a statechart's transition to transfer itself to the concrete object. This expression then executes `traverser.handle(this)` from its concrete type, which ultimately provides the relevant type of the expression back to the traverser. The emulated double dispatching, together with the default depth-first climbdown algorithm of the generated visitor pattern, ensures a correct traversal of the AST by construction. Nevertheless, this emulation results in additional runtime overhead and, as it requires additional methods, slightly slower class loading performance. As these effects are rather negligible, we consider this solution comparatively better than direct class cast in the traversers themselves. However, this handshake regularly confuses novice developers initially, as its use and purpose are apparently not obvious.

**C18: Build and Startup Overhead**

A drawback of the generated infrastructure is the amount of overloaded, i.e., equally named, methods with a default implementation, which substantially increases the class loading time due to the slower method resolution of the Java virtual machine[4]. Table 1 shows the count of methods of various languages, i.e. the running examples, the language for defining grammars, a SysML v2 language in its textual notation, a modular language for UML statecharts, and the derived language for domain-specific transformations of said statecharts. The factor between the methods and nonterminals varies between languages due to the traverser class containing separate visit methods for AST nodes, symbols, and scopes. However, despite a certain overhead and variance, the number of methods roughly correlates to the number of nonterminals. For example, the SysML v2 MontiCore DSL counts 302 nonterminals in total, resulting in 556 `handle`, `traverse`, `visit`, and `endVisit` methods. Nevertheless, the lookup complexity of overloaded methods also corresponds to the number of artifacts to look into with respect to the inheritance hierarchy, thus ultimately resulting in a worst-case complexity of `m x n` (with m = *number of artifacts* and n = *number of nonterminals*). Thus, increasing the number of integrated languages and, analogously, the number of generated Traversers, through which the inheritance hierarchy is realized, in turn, increases the classloading time. This drawback naturally depends on the underlying Java infrastructure and is not in itself a conceptual problem of the visitor mechanism. This renders it a purely technical problem, which, however, comes into effect with large language families. It is also a good example of why modularizing languages should not be overdone, as it only slows down tooling unnecessarily.

In MontiCore, this issue occurs several times due to the combination of different features. Especially for derived languages such as domain-specific transformation languages (DSTLs) [30], where six additional nonterminals are constructed from each original nonterminal (see Table 1), this initializing is noticeable with more than ten seconds. Additionally, for interpretative semantics, MontiCore generates similar methods to the visitor infrastructure, with a return value, which again increases the number of methods.

| Language | `Traverser#visit` methods | Nonterminals |
|---|---|---|
| Example Monolith | 19 | 4 |
| Example Modular | 42 | 12 |
| Grammar Modular [5] | 375 | 230 |
| SysMLv2 Modular[6] | 556 | 302 |
| Modular Statecharts | 266 | 131 |
| DSTL Modular Statecharts | 1304 | 1060 |

**Table 1.** Number of overloaded methods within the generated traversers and the total count of nonterminals of some languages designed using MontiCore.

[3]The same holds for the classes of the symbol table, which also support inheritance.

[4]https://docs.oracle.com/javase/specs/jvms/se20/html/jvms-5.html#jvms-5.4.3.3

On a technical level, different variants of the visitor implementation could minimize this issue. As it mainly arises due to method overloading, this factor could be eliminated, for example, by uniquely naming each of the `visit`, `endVisit`, `handle`, and `traverse` methods. Unfortunately, this variant has its own disadvantages. On the one hand, the generated visitor pattern is only a basic structure and not a complete implementation in itself. The actual behavior must be added by the tooling developer, who naturally has a high degree of interaction with the interfaces and method signatures of this generated infrastructure. Unique method names could be unintuitive or, at worst, make the methods unrecognizable. Furthermore, name clashes due to identically named or overwritten nonterminals from several language definitions must be taken into account and adequately considered by the language workbench.

**C19: Compositional Context Checking**

Context-free grammars are used to define the syntax of a language with MontiCore. The definition of context-sensitive restrictions, such as that the source of a transition must exist, is not possible in context-free grammars. Similarly, some constraints are easier to write in a context-sensitive way, such as allowing each state's name at most once. With MontiCore, such context conditions (CoCos) can be provided by hand-written Java code, each CoCo implementing a visitor of the language. Syntactically correct models of a language, which pass all CoCos, are also called well-formed. A generated `CoCoChecker` class accepts a collection of CoCos and traverses the AST, checking each CoCo against appropriate AST classes. This approach reuses the visitor infrastructure and thus shares its challenges, advantages, and disadvantages (cf. C16). When composing languages, all CoCos of the component languages can be added to a CoCoChecker of the composed language. For example, the automata CoCo that ensures that source states must exist applies to the statecharts language. But not all CoCos of a super language can be inherited. For example, an automata CoCo that disallows duplicate transitions by limiting source-target pairs may not apply to the statecharts language, because now source-precondition-target triples must be considered.

With MontiCore, a language designer thus has to explicitly add CoCos to the CoCoChecker of a language. It is possible to add only selected CoCos or all CoCos of another language.

**C20: Modular Type System**

Type-checking is a special kind of well-formedness checking, in which a type-system is utilized to compare i.a., expressions in terms of assignability. The MontiCore type-system is built upon the visitor infrastructure [29]. It constructs a type-representation of given types of a model (e.g., the primitive type `int` in a class diagram) and deduces the type-representation of an expression (e.g., the expression `[int] + [int]` results in an `int`).

For our example language, we require a type-visitor per grammar, that maps AST nodes of a grammar to their respective type. For expressions, the resulting type depends on the terms and operators of the expression. In our example, the type visitor of the `Exprs` grammar thus handles name and parentheses expressions. The type of a parentheses expression is the type of the inner expression, but the type of a name expression is the type of the referenced element. We then use MontiCore's modular symbol infrastructure [12, 29] to resolve the named element and the type of the element. As the expressions of our example are only boolean expressions, the type-system for the `BExprs` grammar appears trivial. Although, the semantics (cf. C21) of the statecharts grammar heavily influence its type-system, which has to be considered during the type-system implementation of the super languages. For example, a state could be referenced in an expression and this expressions holds iff the state was visited before, selected by a user, etc. We thus have to heavily rely on patterns for modular language design, such as the realThis and delegate patterns [16], to support such semantics.

Other workbenches use DSLs to define the type-system instead of hand-written Java code. MontiCore's typechecking infrastructure can be considered a hand-written extension which, due to its complexity, is provided next to the stable expression language library of MontiCore. The inclusion of extra rules and operators thus can only be accomplished via hand-written code and the application of the realThis pattern, but one is not restricted by the limits of a DSL. Class2MC [7] is an optional extension to the type-system, which provides type definitions based upon Java types loaded at runtime.

In a more complex composite language, such as with full Java-like expressions and an included SysMLv2 SI-Units library [24], a simple +infix operator's type deduction and semantic are more complicated: If any term is a string, the result is the concatenated string of both terms. If SI-Units are present, the result is another SI-unit, iff both units are compatible. For example, meters and seconds are not compatible with the plus operator, but kilometers and meters are. If both terms are numeric, numeric promotion according to the Java language specification[7] occurs. For example, the expression `0.4 + 2`, the addition of a double and an int, results in the widening of the int, resulting in a double as the expression's type. If none of the previous cases apply, the type-system of the given language is unable to derive the type of the expression, and the type-check would report this as an error.

The type-system and symbol infrastructure are thus building blocks for modular language workbenches [12] that use the existing mill and visitor infrastructure, and, sharing a large set of their challenges (cf. C12 and C16).

**C21: Composed Semantics**

[7] https://docs.oracle.com/javase/specs/jls/se20/html/jls-5.html#jls-5.6

The research area of DSL composition is mainly focused on the composition of concrete and abstract syntax, as well as the generated tooling to a certain extent. However, the semantics (e.g., denotational, operational, or translational) of languages should also be taken into account during composition. Prominent realizations can be found in code generators or interpreters.

Generally, composing code generators is still in its infancy. In MontiCore, first advances [34, 41] exists building upon the symbol infrastructure to connect CRUD-like access operations with symbols. Other generators or generator components can then embed the concrete accessor snippets via addressing the corresponding symbol. Another option is the development of compatible generators sharing a large interface. However, these approaches either need a strong synchronization between the generator developers or yield to be overly generic [45]. Genesys [33] is another framework for building property conform code generators using services to communicate access information, but it only works within the jABC [50] ecosystem. Thus, although there are initial approaches to generator composition, there is still much potential in providing a general and reusable mechanism.

Another option for realizing semantic mapping is based on interpreters. They directly operate on the parsed model (i.e., the AST) and define the execution behavior. Such interpreters can be realized similarly to the visitor pattern and, thus, support the same composition mechanism. Consequently, they inherit the same advantages and disadvantages (cf. C16). Additionally, issues from the type-systems arise (cf. Section C20) as, during interpretation, implicit and explicit type conversions must be performed. Reusing results from already performed type-checking can definitely support this conversion on the interpreter level. However, this highly depends on the compatibility of the type-worlds of the modeling language and the underlying programming language.

## 5 Discussion and Open Challenges

The presented paper showcases the lessons learned during the continuous development and research with the language workbench MontiCore [29] regarding the challenges of modular language design and language composition. We have found, that the benefits of modular language design [7, 6] come with challenges, some of which are not immediately apparent. Large, modular language families incur a cost in terms of software system complexity and both build-time and run-time performance, which results in necessary compromises between the level of modularity, its costs, and other requirements. Our goal is to help developers of language workbenches and language designers in their decision-making regarding these compromises, as well as possible solutions. With MontiCore, one of the main requirements is the black-box re-use of language definitions and hand-written Java tooling. The resulting complexity of the visitor infrastructure (cf. C16) remains an area of research but is the fundamental building block for features, such as analysis, pretty printers, the type-system, etc. Particularly in combination with the TOP mechanism, which allows hand-written code to be included within generated code, the modular infrastructure of MontiCore demonstrates its ability to facilitate easy black-box re-use. Language workbenches with different requirements may elect to use DSLs and generators to generate code for a specific composite language instead [35]. We follow a blended approach with MontiCore, as we generate a non-modular parser (cf. C9). Another approach for merging hand-written behavior provided as code into a composite system could be code adaption [38]. With MontiCore, we expect performance improvements due to Java's Project Leyden[8], which aims, among other things, to improve the startup time of Java applications.

MontiCore, like any other workbench used for research purposes, is going to continue to evolve along with its requirements [27]. Due to its usage in industry settings and production systems, larger breaking changes should be used cautiously.

***Threats to Validity.*** As usual for application-oriented investigations in the software engineering domain, our results are subject to threats to validity, according to [56]. The most apparent aspect is that we report on our experience of language composition in the context of mainly one language workbench. This is a threat to external validity. Some of our findings might not be generalizable or might not be applicable in other language workbenches. However, with MontiCore [29], we have chosen a framework that emphasizes compositionality in language development. To the best of our knowledge, comparable workbenches such as Xtext [18] or Spoofax [35] support either equivalent or slightly less compositional techniques [17], while MPS [53] uses a different editor approach, i.e. projectional editing. Thus, similar issues to those discussed should also apply to other ecosystems. All four language workbenches are also based on Java at their core, which means they are subject to the same fundamental infrastructural challenges. Solely, problems related to learnability or accessibility could be less applicable in these frameworks, as they have many examples and tutorials as systems that are heavily used in industry. An important topic for future research would be investigating corresponding challenges for language workbenches based on completely different technology stacks, such as Langium[9].

Other risks relate to internal validity. The examples shown here are deliberately kept simple to increase comprehensibility. However, some challenges discussed only show their full impact with much larger and more complex language compositions, such as within a realization of SysML v2 or derived

[8] https://openjdk.org/projects/leyden
[9] https://langium.org/

transformation languages. Another threat is that our report is not a single snapshot but has been recorded across several major versions of MontiCore over the years. Nevertheless, all the problems and most of their effects are already apparent from the examples presented here. The example project attached allows to reproduce our results. Finally, it must be considered that the authors are expert users of MontiCore and language development in general. On the one hand, this means that many of the problems documented here deal with intricacies that not every practitioner encounters directly. On the other hand, by using MontiCore in various teaching environments, we could also include feedback from new users and students.

## 6 Comparable Approaches

The pitfalls in modular language construction originate from our experience with the MontiCore language workbench. While some of these challenges are generally part of ongoing research, different language frameworks follow distinct approaches. Therefore, we discuss notable solutions from other language workbenches and compare these to the stated challenges of modular language design.

A different way to handle ambiguous nonterminal names (cf. C2) is to always use fully qualified names for rule references, such as `Automaton.S`, as Xtext [18] does. Another solution is to explicitly rename the incorporated productions during the import. Rascal [2] allows such renaming via an alias to the fully qualified production. Similarly, the Melange language workbench provides a `renaming` operation [14] to rename the EMF objects, such as EPackages and EClasses. With MontiCore, language designers are discouraged from employing ambiguous naming, obviating the necessity for solving this problem.

Similar to MontiCore, lots of other language workbenches [35, 17] support the modular definition of language components, but import all elements of a language. Reusing only a subset of productions of a grammar (cf. C5) is possible in Melange via its `slice` operator. Only the EClasses mentioned in the slice operation and, recursively, their superclasses and referenced classes are retained. Rascal differentiates between `extending` a language and `importing` only one language element. The language extension of Rascal is transitive.

When dealing with keywords between languages (cf. C8), Xtext provides another means of disambiguation by allowing an optional, silently removed caret on the model level(^) in front of all identifiers. This way reserved keywords can be circumvented. However, this approach simultaneously shifts the responsibility for the problem to the modeler. Spoofax [35] avoids the problem of lexical disambiguation by using a scannerless parsing approach in which only character classes are terminals [52]. The main disadvantages of scannerless parsers are the increased lookahead, which results in a decreased efficiency, and the explicit handling of whitespaces in the context-free grammar. Similar to the normalization of MontiCore interface nonterminals, the syntax of a language must be normalized into its scannerless equivalents and syntax errors must be translated back into the concrete syntax. MontiCore includes certain aspects of scannerless parsing and provides an option to translate selected tokens, such as the logical right shift operator `>>>`, into their scannerless form using the `splittoken` statement [29]. Projectional language workbenches, such as MPS [53], circumvent the problem of modular parser construction entirely and thus are not affected by the parser-specific challenges [17].

Other workbenches, such as Spoofax use DSLs to define the semantics, including a type system, instead of hand-written Java code. MontiCore's typechecking infrastructure can be considered a hand-written extension. A DSL, e.g. Spoofax's Statix [51], on the other hand, is domain-specific and thus abstracts away implementation specifics. With DSLs, the automatic derivation of proofs for completeness and correctness of the type system is also possible.

The framework of scope graphs [51] further allows well-formedness checks (cf. C19) to be defined in a formalized way. Racket [1] provides macros and MPS provides a constraint rule language[10], in which expressions to be evaluated are provided along with the error messages.

## 7 Conclusion

In this paper, we have analyzed open challenges in applying language composition in the large. Even if some of these issues are based on intricacies and are not immediately apparent, we have investigated them in an application-orientated context. Our investigations explore various relevant aspects of a language tool using the language workbench MontiCore as an example. We address challenges in compositional parsers, ASTs including their creation, as well as generated tooling, such as visitors for navigation and operation execution. It becomes evident that the increasing complexity of the infrastructure in the background rises rapidly as the number of incorporated languages increases. This jeopardizes reusability in large projects, especially where language composition promises its greatest advantages. We have even been able to construct language definitions that are valid in themselves but can no longer be processed by the default Java compiler due to their sheer artifact size. As SLE continues to advance as a research field and has a major impact on both industrial developments and research software, we are contributing to a better understanding of current challenges. This provides indications of open topics for future research. It can also assist practitioners in better understanding technical intricacies in the background and thus assess in which cases modular language construction is reasonable in current SLE endeavors.

[10] https://www.jetbrains.com/help/mps/constraintrules.html

## Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 459671966

## References

[1] Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for domain-specific languages. *Proc. ACM Program. Lang.*, 4, OOPSLA, Article 229, (Nov. 2020), 29 pages. doi:10.1145/3428297.

[2] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju. 2015. Modular language implementation in rascal – experience report. *Science of Computer Programming*, 114, 7–19. LDTA (Language Descriptions, Tools, and Applications) Tool Challenge. doi:https://doi.org/10.1016/j.scico.2015.11.003.

[3] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. 2020. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology (JOT)*, 19, 3, (Oct. 2020), 3:1–16. Lars Hamann, Richard Paige, Alfonso Pierantonio, Bernhard Rumpe, and Antonio Vallecillo, (Eds.) http://www.se-rwth.de/publications/A-Library-of-Literals-Expressions-Types-and-Statements-for-Compositional-Language-Design.pdf.

[4] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, Gothenburg, Sweden, (Sept. 2018). http://www.se-rwth.de/publications/Modeling-Language-Variability-with-Reusable-Language-Components.pdf.

[5] Hans Blom et al. 2013. EAST-ADL: An architecture description language for Automotive Software-Intensive Systems. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design*, 456. doi:10.4018/978-1-4666-3922-5.ch023.

[6] Benoit Combemale, Betty H.C. Cheng, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, (Eds.) 2015. *Globalized Domain Specific Language Engineering. Globalizing Domain-Specific Languages*. Springer, 43–69. doi:10.1007/978-3-319-26172-0_4.

[7] Arvid Butting. 2023. *Systematic Composition of Language Components in MontiCore. Aachener Informatik-Berichte, Software Engineering, Band 53*. Shaker Verlag, (Feb. 2023). ISBN: 978-3-8440-8936-3. http://www.se-rwth.de/phdtheses/Diss-Butting-Systematic-Composition-of-Language-Components-in-MontiCore.pdf.

[8] Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys (CSUR)*, 17, 4, 471–523.

[9] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, (Eds.) *Globalizing Domain-Specific Languages*, LNCS 9400, (2015). Springer. http://www.se-rwth.de/publications/Globalizing-Domain-Specific-Languages2.pdf.

[10] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. 2016. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, (Nov. 2016). https://www.crcpress.com/Engineering-Modeling-Languages/Combemale-France-Jezequel-Rumpe-Steel-Vojtisek/p/book/9781466583733.

[11] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)* (LNCS 5795). Springer, 670–684. http://www.se-rwth.de/publications/Variability-within-Modeling-Language-Definitions.pdf.

[12] Benoit Combemale, Jeff Gray, and Bernhard Rumpe. 2024. Model modularity for reuse, libraries and composition: symbol management is key. *Journal Software and Systems Modeling (SoSyM)*, 23, 3, (June 2024), 525–526. doi:10.1007/s10270-024-01190-0.

[13] Shigeru Chiba and Rei Ishikawa. 2005. Aspect-oriented programming beyond dependency injection. In *European Conference on Object-Oriented Programming*. Springer, 121–143.

[14] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2015). ACM, Pittsburgh, PA, USA, 25–36. ISBN: 978-1-4503-3686-4. doi:10.1145/2814251.2814252.

[15] Kay Dickersin. 1990. The Existence of Publication Bias and Risk Factors for Its Occurrence. *Jama*, 263, 10, 1385–1389.

[16] Florian Drux, Nico Jansen, and Bernhard Rumpe. 2022. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology (JOT)*, 21, 4, (Oct. 2022), 4:1–13. http://www.se-rwth.de/publications/A-Catalog-of-Design-Patterns-for-Compositional-Language-Engineering.pdf.

[17] Sebastian Erdweg et al. 2013. The state of the art in language workbenches: Conclusions from the language workbench challenge. In *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings 6*. Springer, 197–217.

[18] Moritz Eysholdt and Heiko Behrens. 2010. 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*, 307–309.

[19] J-M Favre. 2005. Languages evolve too! Changing the Software Time Scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 33–42. doi:10.1109/IWPSE.2005.22.

[20] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, (May 2007), 37–54. http://www.se-rwth.de/publications/Model-driven-Development-of-Complex-Software-A-Research-Roadmap.pdf.

[21] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. 1995. Elements of Reusable Object-Oriented Software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company*.

[22] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. 2008. Classification of concrete textual syntax mapping approaches. In *Model Driven Architecture – Foundations and Applications*. Ina Schieferdecker and Alan Hartman, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 169–184. ISBN: 978-3-540-69100-6.

[23] Object Management Group. 2023. Kernel Modeling Language (KerML), Version 1.0 Beta 1. https://www.omg.org/spec/KerML/1.0/Beta1/PDF [Online; accessed 2024-06-05]. (June 2023).

[24] Object Management Group. 2019. OMG Systems Modeling Language (OMG SysML), Version 1.6. https://www.omg.org/spec/SysML/1.6/PDF [Online; accessed 2024-06-05]. (Nov. 2019).

[25] Object Management Group. 2023. OMG Systems Modeling Language (OMG SysML), Version 2.0 Beta 1. https://www.omg.org/spec/SysML/2.0/Beta1/Language/PDF [Online; accessed 2024-06-05]. (June 2023).

[26] Object Management Group. 2017. OMG Unified Modeling Language (OMG UML), Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF [Online; accessed 2024-06-05]. (Dec. 2017).

[27] Morane Gruenpeter et al. 2021. Defining Research Software: a controversial discussion. Version 1. (Sept. 2021). doi:10.5281/zenodo.5504016.

[28] Malte Heithoff, Nico Jansen, Jörg Christian Kirchhof, Judith Michael, Florian Rademacher, and Bernhard Rumpe. 2023. Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*

(SLE 2023). Association for Computing Machinery, Cascais, Portugal, (Oct. 2023), 194–207. doi:10.1145/3623476.3623527.

[29] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48*. Shaker Verlag, (May 2021). ISBN: 978-3-8440-8010-0. http://www.monticore.de/handbook.pdf.

[30] Katrin Hölldobler. 2018. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen. Aachener Informatik-Berichte, Software Engineering, Band 36*. Shaker Verlag, (Dec. 2018). ISBN: 978-3-8440-6322-6. http://www.se-rwth.de/phdtheses/Diss-Hoelldobler-MontiTrans-Agile-modellgetriebene-Entwicklung-von-und-mit-domaenenspezifischen-kompositionalen-Transformationssprachen.pdf.

[31] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, 37, 10, (Oct. 2004), 64–72. http://www.se-rwth.de/staff/rumpe/publications20042008/Meaningful-Modeling-Whats-the-Semantics-of-Semantics.pdf.

[32] Katrin Hölldobler and Bernhard Rumpe. 2017. *MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32*. Shaker Verlag, (Dec. 2017). ISBN: 978-3-8440-5713-3. http://www.se-rwth.de/publications/MontiCore-5-Language-Workbench-Edition-2017.pdf.

[33] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. 2008. Genesys: service-oriented construction of property conform code generators. *Innovations Syst Softw Eng*, 4, 361–384. doi:https://doi.org/10.1007/s11334-008-0071-2.

[34] Nico Jansen and Bernhard Rumpe. 2023. Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2023). Association for Computing Machinery, Cascais, Portugal, (Oct. 2023), 163–168. doi:10.1145/3623476.3623530.

[35] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*. Martin Rinard, (Ed.) Reno, NV, USA.

[36] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Pearson Education.

[37] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. 2022. Correct and Sustainable Development Using Model-based Engineering and Formal Methods. In *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*. IEEE, (Sept. 2022). http://www.se-rwth.de/publications/Correct-and-Sustainable-Development-Using-Model-based-Engineering-and-Formal-Methods.pdf.

[38] Marco Konersmann, Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Valdes Voufo. 2024. Towards a Semantically Useful Definition of Conformance with a Reference Model. *Journal of Object Technology (JOT)*, 23, 3, (July 2024), 1–14. doi:10.5381/jot.2024.23.3.a5.

[39] Charles W Krueger. 1992. Software Reuse. *ACM Computing Surveys (CSUR)*, 24, 2, 131–183.

[40] Vinay Kulkarni and Sreedhar Reddy. 2003. Separation of Concerns in Model-Driven Development. *IEEE software*, 20, 5, 64–69.

[41] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. 2016. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference* (LNI). Vol. 254. Bonner Köllen Verlag, (Mar. 2016), 133–140. http://www.se-rwth.de/publications/An-Extended-Symbol-Table-Infrastructure-to-Manage-the-Composition-of-Output-Specific-Generator-Information.pdf.

[42] Terence Parr. 2013. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf. ISBN: 9781934356999.

[43] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive ll(*) parsing: the power of dynamic analysis. *SIGPLAN Not.*, 49, 10, (Oct. 2014), 579–598. doi:10.1145/2714064.2660202.

[44] Luis Pedro, Vasco Amaral, and Didier Buchs. 2008. Foundations for a Domain Specific Modeling Language Prototyping Environment: A compositional approach. In *Proceedings of the 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM)*. (Oct. 2008), 20–27.

[45] Jérôme Pfeiffer and Andreas Wortmann. 2021. Towards the Black-Box Aggregation of Language Components. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 576–585. doi:10.1109/MODELS-C53483.2021.00088.

[46] Sérgio Queiroz de Medeiros and Fabio Mascarenhas. 2018. Error recovery in parsing expression grammars through labeled failures and its implementation based on a parsing machine. *Journal of Visual Languages & Computing*, 49, 17–28. doi:https://doi.org/10.1016/j.jvlc.2018.10.003.

[47] Stefan Rüther, Thomas Hermann, Maik Mracek, Stefan Kopp, and Jochen Steil. 2013. An Assistance System for Guiding Workers in Central Sterilization Supply Departments. In (PETRA '13). ACM. ISBN: 9781450319737.

[48] Bran Selic. 2003. The Pragmatics of Model-Driven Development. *IEEE software*, 20, 5, 19–25. doi:10.1109/MS.2003.1231146.

[49] Herbert Stachowiak. 1973. *Allgemeine Modelltheorie*. Springer.

[50] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. 2007. Model-Driven Development with the jABC. In *Hardware and Software, Verification and Testing: Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers 2*. Springer, 92–108.

[51] Hendrik Van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proceedings of the ACM on Programming Languages*, 2, OOPSLA, 1–30.

[52] Eelco Visser. 1997. Scannerless Generalized-LR Parsing. Tech. rep. P9707. Programming Research Group, University of Amsterdam, (July 1997).

[53] Markus Voelter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1449–1450.

[54] Markus Voelter, Jos Warmer, and Bernd Kolb. 2015. Projecting a Modular Future. *IEEE Software*, 32, 5, 46–52. doi:10.1109/MS.2014.103.

[55] Dennis Leroy Wigand, Arne Nordmann, Niels Dehio, Michael Mistry, and Sebastian Wrede. 2017. Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case. *Journal of Software Engineering for Robotics*.

[56] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer.

Received 05 March 2025