Chapter 10
Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques—The MontiCore Approach

Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann

Abstract Composing modelling languages and analysis tools still require significant efforts to properly consider syntax and semantics as well as related analyses and syntheses. This composition ideally should be defined on individual language components that can be composed when needed. Only when model-based analysis infrastructures can be composed in accordance to their related language definitions and can be reused in a black-box fashion without modification, can we foster automation in language engineering and integration. In this chapter, we demonstrate object-oriented language engineering concepts that enable composing models of heterogeneous languages using the language workbench MontiCore. This composition includes the concrete syntax and abstract syntax as well as analysis infrastructures and analyses. We demonstrate in detail how the MontiCore infrastructure enables (de)composing languages and related model-based analysis techniques such that the analyses can be reused with other languages with minimal effort. Several of the provided techniques are based on adaptations of the well-known concepts of object-oriented development, such as inheritance and the extension and the visitor patterns. This can reduce the effort of engineering truly domain-specific modelling languages significantly.

This case-study chapter illustrates concepts introduced in Chap. 4 and addresses Challenge 1 in Chap. 3 of this book.

A. Butting · K. Hölldobler · B. Rumpe (✉)
RWTH Aachen, Aachen, Germany
e-mail: butting@se-rwth.de; hoelldobler@se-rwth.de; rumpe@se-rwth.de

A. Wortmann
Universität Stuttgart, Stuttgart, Germany
e-mail: andreas.wortmann@isw.uni-stuttgart.de

R. Heinrich et al. (eds.), *Composing Model-Based Analysis Tools*,
https://doi.org/10.1007/978-3-030-81915-6_10

10.1 Introduction

Many engineering domains moved to use explicit modelling languages to enable domain experts to contribute to the engineering of systems. Ideally, these modelling languages and their parts can be reused in and tailored to different contexts, such that deploying precise *domain-specific modelling languages* (DSMLs) becomes less challenging. Despite efforts in software language engineering [Kle08, HRW18], the composition of modelling languages, especially their analyses and syntheses, is far from solved in general (cf. Chap. 3 of this book [Hei+21]) and still requires significant manual efforts. This hinders the deployment of the most suitable domain-specific modelling languages for experts, who instead have to cope with overly generic modelling languages, such as the *unified modeling language* (UML) [Obj15] or the *systems modeling language* (SysML) [Obj12], and tailor these through profiles or modelling guidelines. Both introduce a conceptual gap [FR07] between the experts' problem domain of discourse (e.g., material science, kinematics, geometry) and the solution domain of software engineering through which the domain experts need to work around the limitations of these languages.

The efforts for efficiently engineering DSMLs can be reduced if the languages support modularity and their infrastructure follows this modularity. When the infrastructure for analyses is derived in ways that foster modularity, composition of this infrastructure can be automated as well. This eases reusing DSML (parts) and their analysis in the context of other languages and can foster the application of DSMLs in general. In this chapter, we therefore demonstrate core concepts to compose models from heterogeneous sublanguages. This includes the syntax (concrete and abstract) as well as the infrastructures to define the syntax and the analyses to operate on these composed languages. We demonstrate in detail how the MontiCore infrastructure allows to decompose a number of analysis techniques, both for functional and for extra-functional properties.

The contributions of this chapter, hence, are:

- A method for engineering modelling languages based on modular syntax definitions
- Generation of a visitor-based framework for modular model-based analyses
- Families of modular languages for expressions and literals

In the remainder, Sect. 10.2 introduces MontiCore, before Sect. 10.3 applies it to engineering modular languages and analysis infrastructures. Section 10.4 highlights related work, and Sect. 10.5 discusses our approach. Section 10.6 concludes.

10.2 Preliminaries

This section introduces the MontiCore [HR17] language workbench [Erd+15] and its features used for engineering compositional languages as explained in subsequent sections.

MontiCore is a language workbench that provides an EBNF-like grammar format to define languages from which it generates much of the infrastructure necessary to efficiently engineer modular languages. It has been applied to the engineering of modelling languages for a variety of domains, including automotive [Dra+19], cloud services [Eik+17], robotics [Ada+17b], systems engineering [Dal+19], and more.

For a given grammar, MontiCore generates infrastructure for the language. This includes parser and lexer, Java classes for the *abstract syntax tree* (AST), an infrastructure to implement context conditions (language well-formedness rules), visitors [Hei+16] to develop and compose analyses, and symbol tables [HMR15, MRR15, MRR16] to combine models of different languages. The general procedure to process a model is depicted in Fig. 10.1. First, the model is transferred to its internal representation, i.e., the AST, by the parser and lexer. Next, the AST is processed by functions, which can include well-formedness checks, analyses, or transformations. The resulting AST as well as the analysis results are used to produce the output, which can be generated code, models, or analysis reports.

A MontiCore grammar defines the abstract and concrete syntax of a language. It consists of 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 10.2 depicts a MontiCore grammar of a compact language for finite automata, while Fig. 10.3 shows a corresponding automaton model. This grammar consists of three productions defining the nonterminals Automaton, State, and Transition. MontiCore generates one AST class for each production. Its attributes are defined by the production body. Stored terminals map to attributes, while nonterminal usages map to compositions.

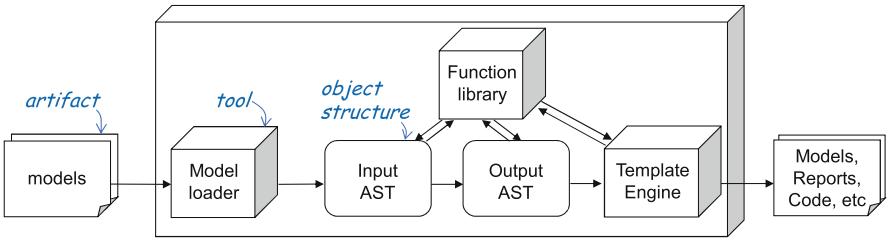

Fig. 10.1 MontiCore's tool chain for processing models comprises fully generated components (parser, lexer, etc.) and modular infrastructures for tools that are handcrafted (well-formedness rules, model transformations)

```
1  grammar Automata extends ExpressionsBasis,
2                           CommonLiterals {
3    Automaton = "automaton" Name "{"
4                (State | Transition)* "}" ;
5
6
7    State = ["initial"]? ["final"]? "state" Name ";";
8
9    Transition = from:Name "-" ("[" guard:Expression "]")?
10               input:Name ">" to:Name ";";
   }
```

Fig. 10.2 Exemplary grammar of an automata language

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

Fig. 10.3 Exemplary automaton for the language of Fig. 10.2

The body of a production consists of terminals and nonterminals. Terminals are surrounded by quotation marks, e.g., `"automaton"` in line 3 of Fig. 10.2. Both terminals and nonterminals can have different multiplicities, i.e., by appending a question mark `?` it becomes optional, while `*` allows arbitrarily many (including zero) occurrences and + enforces at least one occurrence. Alternatives are separated by `|`, and grouping can be achieved by parenthesising parts using round brackets. Terminals whose presence is relevant for the abstract syntax can be parenthesised in square brackets, yielding a Boolean attribute in the abstract syntax. Optionals 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 in [HR17]. Interface nonterminals are marked using the keyword interface (cf. Fig. 10.4, line 3). They do not specify concrete syntax themselves. Instead, interface nonterminals are implemented by other nonterminals (cf. Fig. 10.4, line 5). For interface nonterminals, a production body can be used to restrict possible implementing nonterminals [HR17]. Concep-

```
1  component grammar ExpressionsBasis extends LiteralsBasis {
2
3    interface Expression;
4
5    NameExpression implements Expression<350> = Name;
6
7
8    LiteralExpression implements Expression<340> = Literal;
   }
```

Fig. 10.4 Component grammar providing basic syntax elements for expressions

tually, interface nonterminals are an extension of alternatives. Whenever interface nonterminals are used in a production body, every interface implementation is possible. 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 the relation between A and B and A and C is mapped to inheritance instead of composition in the abstract syntax.

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 Fig. 10.2 line 1, the grammar of the automata language already uses this concept. By using 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 inherited grammars (also referred to as super grammars) are available in the current grammar. In the automata language, this is used for the transition production as it uses the nonterminals `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 language itself, it can and should be marked as a component grammar by adding the keyword `component` (cf. Fig. 10.4, line 1).

The start nonterminal of a grammar is by default the first nonterminal in the grammar [HR17]. However, there are situations in which this is not feasible, e.g., when extending an existing grammar and one of its nonterminals should be the start nonterminal of the currently developed language. To address this, it is possible to configure the start nonterminal explicitly as follows: `start State`. In this case, `State` is used as the start nonterminal.

When extending a grammar, it is possible to extend productions of the super grammars. This is possible for normal as well as for interface productions. In both cases, conceptually a new alternative to the existing body resp. implementations 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. Fig. 10.4, line 5). The higher the number within the brackets, the higher is the alternative's priority in the generated parser.

10.3 Compositional Language Engineering

MontiCore provides means to support modular definition of languages and means to realise language composition [HR17]. Modularisation fosters language reusability and reduces co-evolution, as the commonalities in different languages can be extracted to individual language modules that multiple languages rely on.

Grammar inheritance can be leveraged to decompose the syntax of a language into modules (cf. Sect. 10.2). Further, MontiCore supports the definition of component grammars to indicate that a grammar contains a reusable col-

lection of pieces of syntax rather than a complete language. For instance, the automata grammar presented in Fig. 10.2 uses grammar inheritance to decouple the definitions of the automata language syntax in terms of states and transitions from the syntax of expressions and of literals. This is realised by extending the grammars `ExpressionsBasis` and `CommonLiterals`. A language module, also referred to as language component, is defined by its grammar but also contains all artefacts generated from the grammar, all handwritten extensions to the generated artefacts, and handwritten language tooling such as, e.g., model-based analyses. Therefore, modularisation has to be carried out for all these constituents as well.

As describing modular analyses on languages requires modular language syntax, the following first introduces some of MontiCore's means for modular grammar definitions, before introducing the modular visitor infrastructure. Afterwards, the application of this infrastructure for modular analyses is demonstrated by example.

10.3.1 Modular Syntax Definition

Expressions, types, literals, and statements are typical elements in modelling or programming languages. However, every language requires a well-suited variant of these concepts. Thus, these concepts are natural candidates for being encapsulated into individual language components that can be reused by any language.

To facilitate this, MontiCore offers a multitude of modular base grammars each of which contributes syntax to define expressions, literals statements, or types. Figs. 10.4, 10.5, 10.6, and 10.7 demonstrate this concept. The ExpressionsBasis (Fig. 10.4) and CommonExpressions (Fig. 10.5) grammars provide syntax for expressions.

```
component grammar CommonExpressions extends ExpressionsBasis {

  LogicalNotExpression implements Expression <190> =
    "!" Expression;

  PlusExpression implements Expression <170> =
    left:Expression operator:"+" right:Expression;

  EqualsExpression implements Expression <130> =
    left:Expression operator:"==" right:Expression;
}
```

Fig. 10.5 Component grammar describing the syntax of basic expressions

```
component grammar LiteralsBasis  {
  interface Literal;
}
```

Fig. 10.6 Grammar providing a syntax extension point for literals

```
1 component grammar CommonLiterals extends LiteralsBasis {
2
3   BooleanLiteral implements Literal =
4     source:["true" | "false"];
5
6
7   SignedNatLiteral implements Literal =
8     (negative:["-"])? Digits;
}
```

Fig. 10.7 Basic Boolean and integer literals

The ExpressionsBasis grammar is a component grammar providing building blocks for the syntax of expressions. At its core, it contains an interface nonterminal Expression acting as extension point for different syntactical constructs that realise expressions. ExpressionsBasis only provides the syntax for names (cf. NameExpression) and values (cf. LiteralsExpression). Name (not depicted) is a token for Java-like identifiers, such as Java method names. Literal is an inherited interface nonterminal provided by LiteralsBasis (cf. Fig. 10.6), which only provides this interface nonterminal but no implementations. Thus, the decision what kind of literals are used and how these are defined is delayed to further grammars extending the ExpressionsBasis grammar.

CommonExpressions extends ExpressionsBasis and adds three novel implementations to the Expression nonterminal providing syntax for some basic expressions. While the LogicalNotExpression and EqualsExpression are commonly used for Boolean expressions, the PlusExpression is commonly used for number expressions. However, it can also be used to represent, e.g., String concatenation. These grammars define only the syntax of the expression; their evaluation is performed at a later stage in language processing.

All three grammar productions in the grammar for common expressions introduce potential left recursion through inheritance with the interface nonterminal Expression. MontiCore can handle the ambiguity introduced by this left recursion, inter alia, through the parser priorities (cf. Sect. 10.2). The grammar CommonLiterals (cf. Fig. 10.7) extends the LiteralsBasis and introduces Boolean literals (ll. 3–4) and integer literals (ll.6–7).

An example of how to use ExpressionsBasis and add pre-built Literal implementations is presented in Fig. 10.2. The automata grammar extends both the ExpressionBasis grammar and the CommonLiterals. Through this multiple inheritance, the Literal nonterminal in the ExpressionsBasis grammar is implemented by the nonterminals introduced in CommonLiterals. An excerpt of the AST data structure that MontiCore produces from the grammar Automata is depicted in Fig. 10.8. As a result of this extension, true and false as well as integer numbers can be used in guards of an automaton through the LiteralExpression.

Other possible extensions are to either add further implementations of interface nonterminals Expression or Literal in the automata grammar or use further pre-built grammars that provide additional implementation such as CommonExpressions. Figure 10.9 gives an overview of MontiCore's pre-built grammars for expressions,

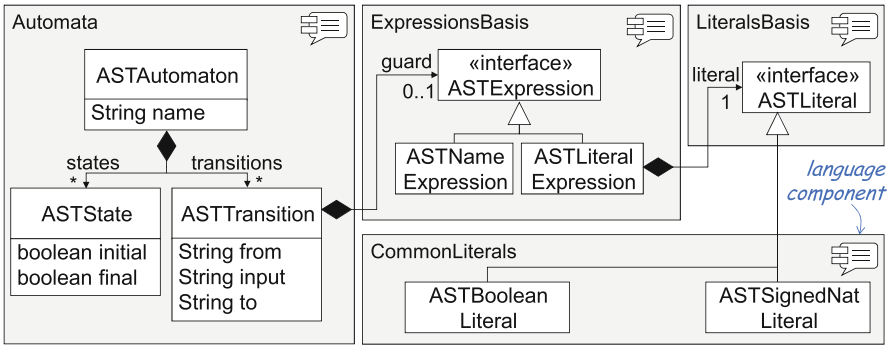

Fig. 10.8 Excerpt from the AST generated from the Automata grammar

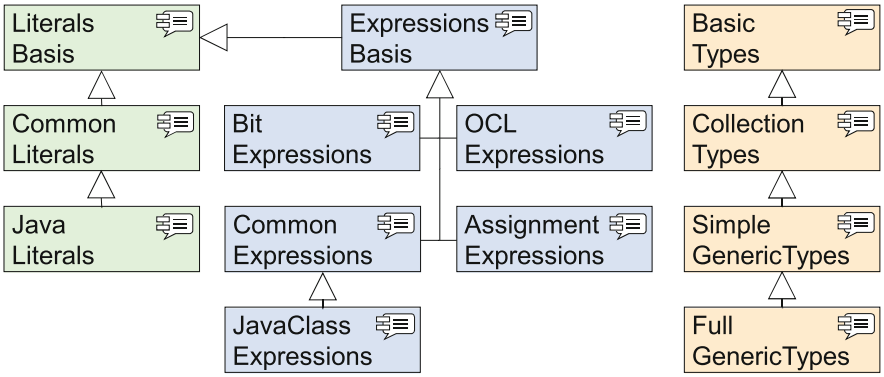

Fig. 10.9 Structure of base grammars for expressions, literals, and types

literals, and types. Types and literals grammars each are in a linear inheritance relationship, where each grammar extends the syntax provided by their parent grammar conservatively [HR17]. To this end, if a language uses types (or literals), it can be post hoc extended with more syntax for types (or literals) by additionally inheriting from a grammar that (transitively) extends the type (or literal) grammar that was originally used.

The various application purposes for expressions prevent a linear inheritance hierarchy for expression grammars: For example, it should be possible for a language to use only the syntax for assignment expressions without bit expressions (which include, e.g., shift operators). At the same time, other languages should be able to use only bit expressions without assignment expressions. However, all expression grammars extend the basis grammar for expressions, and all syntax these add is available by implementing the Expression interface. Therefore, through multiple inheritance with different expression grammars, a combination of expression syntaxes can be made available as well.

In summary, the essence for modular syntax definitions as suggested in this approach is to provide a grammar that only provides an interface nonterminal (in the following called interface grammar). An interface grammar can be extended by other grammars. Hereby, two kinds of extensions are conceivable: (1) Grammars extending the interface grammar can provide further pre-built syntax options that other languages can use. (2) Through inheriting from the interface grammar, a language can delay the decision, which implementations should be used. Languages engineers, thus, can design grammars that extend those interface grammars and by this, specify that some sort of literals, statements, or types are used within their developed language and where they are used. Later, this is resolved through multiple inheritance from this language and the grammar(s) that extend(s) the interface grammars for, e.g., literals, statements, or types. For example, the ExpressionsBasis extends the LiteralsBasis, but through multiple inheritance in the Automata grammar (cf. Fig. 10.2), expressions used in automata can use literals provided by CommonLiterals.

Figure 10.10 provides two example language components that utilise the language components shown in Fig. 10.9. `RoboJAction` is a domain-specific language for modelling actions in the context of service robotics applications similar to this approach [Ada+17a]. The language lends notation elements from Java but only supports basic types, literals of reduced complexity as well as a subset of possible expression implementations. These notation elements are reused by inheriting from several language components. The language further introduces novel syntax elements for realising domain-specific concepts.

The second example is the *object constraint language* (OCL) that combines the language components `CommonLiterals`, `BitExpressions`,

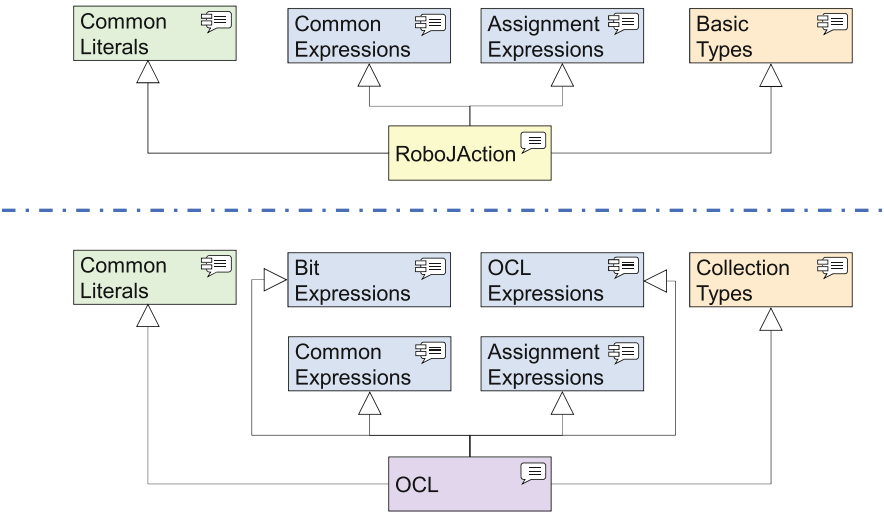

Fig. 10.10 Combining base grammars

AssignmentExpressions, CommonExpression, OCLExpressions, and CollectionTypes. JavaLight and an OCL are considered as complete modelling languages. However, in case more complex types, literals, or expressions are needed, it is possible to extend those languages and combine them with additional language components such as SimpleGenericTypes or JavaLiterals.

10.3.2 Modular Analysis Infrastructure

The modularity for syntax presented in the previous section would be of limited use without modularity in analyses, transformations, and further operations implemented against the syntax. For this purpose, MontiCore generates composable visitors [Hei+16]. From each grammar, a Visitor interface prefixed with the name of the grammar is generated. This interface provides four methods handle, visit, traverse, and endVisit for each nonterminal of the given grammar. A depth-first traversal of the AST of the grammar is already included via default implementations of the handle and traverse methods. The handle methods encapsulate the handling of the nonterminals and call the corresponding visit, traverse, and endVisit methods for the nonterminals. The traverse method is responsible for traversing child nodes of the nonterminal. To implement an analysis for models of a given language, language engineers can focus on implementing the analysis using the visit and endVisit methods. By default, visit and endVisit methods have an empty default implementation and only have to be implemented if it is intended to use these for the implementation of the analysis. The visitor interfaces provide methods for the current grammar only. However, they extend the corresponding visitor interfaces of all extended grammars, and through this, all visitor methods for inherited nonterminals are available as well.

In addition to visitor interfaces, MontiCore generates delegator visitors that are composed of other visitors. These visitors only handle the traversal themselves but delegate the visit and endVisit to registered visitors. By default, one visitor per super grammar can be registered. Figure 10.11 depicts the visitor interface and the delegator visitor that MontiCore generated for the automata grammar in Fig. 10.2.

10.3.3 Composed Analyses

With the modular analysis infrastructure, MontiCore enables language engineers both (1) to describe monolithic analyses across different syntax modules and (2) to reuse analyses as part of reusing a language component. A monolithic analysis across modular syntax can be realised by implementing a visitor interface and using the visitor methods of all (including inherited) nonterminals. This kind of analysis, thus, enables to optionally reuse all visitor infrastructure parts from

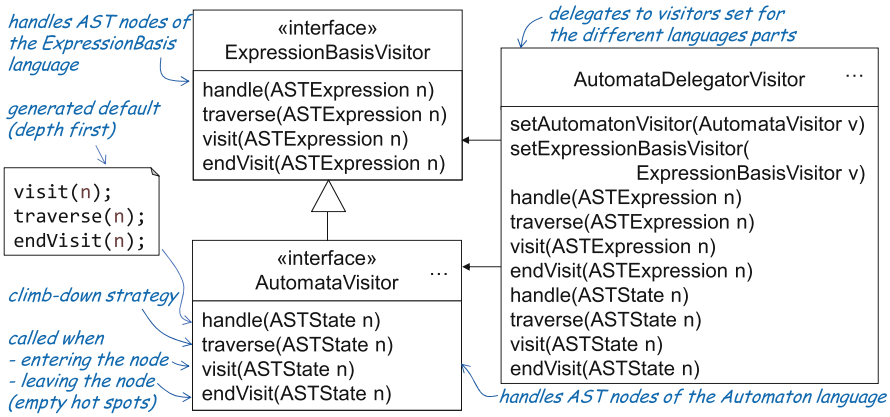

Fig. 10.11 Visitors generated for the example in Fig. 10.2

inherited language components while being able to override and customise parts of it whenever this is required or desired. A monolithical analysis is specifically suitable in situations in which the kind of analysis that is required from inherited language parts has a low potential for being reused in different contexts. If analysis parts that operate on inherited language parts are intended to be reused in a different context, we recommend to realise such parts as individual, modular analyses.

An example for a monolithic analysis on the automata language presented in Sect. 10.2 is to calculate the *effective degree* of all states of an automaton model. By effective degree, we denote the number of incoming and outgoing transitions of a state, which have a satisfiable guard condition, i.e., a guard condition that does not always evaluate to $false$. This analysis can be realised as a class EffectiveStateDegrees implementing the interface AutomataVisitor (cf. Fig. 10.11) as depicted in Fig. 10.12. Through transitive inheritance, the visitor methods, e.g., for ASTExpressions, are reused without modification. Only visitor methods that perform parts of the analysis' calculations are overridden. The traverse method for automata is overridden to first handle the traversion of all states of the automaton, before handling all transitions. The purpose of this is that the degree of each state can be initialised with 0 in the visit method of the ASTState. The visit method of ASTTransition is overridden as well. It initialises a Boolean variable isTraversable with $true$. By this, each transition is initially regarded as traversable. If a transition in the model has a guard condition, the AST nodes of this condition are visited by the traversal strategy before invoking the endVisit method of the transition. Thus, by overriding visit methods of expressions and literals, the isTraversable variable can be adjusted specific to each guard. As the automata language in this example uses the ExpressionsBasis language, a guard can only comprise either a single name or a Boolean or integer literal. For this example, we consider expressions comprising either the Boolean value $false$ or negative integers as unsatisfiable. This is realised by overriding the visit methods of the respective AST classes in the EffectiveStateDegrees. The overridden endVisit method for transitions increments

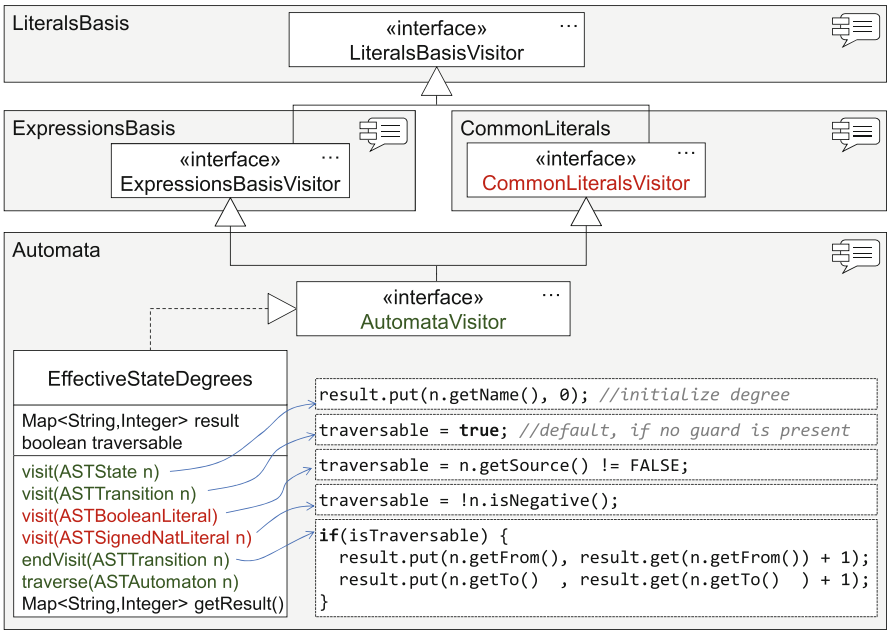

Fig. 10.12 Example for a monolithic analysis across several language components

the degree of source and target states if isTraversable is *true*. As the employed evaluation for expressions has a low potential of being reused in other language components than for automata, the developers decided to realise this as monolithical analysis.

Using delegator visitors enables reusing visitors for the individual language parts involved and, thus, to develop analyses and other operations on the AST modularly. As depicted in Fig. 10.11, a delegator visitor has a setter method for visitors of each (transitive) parent grammar as well as traversal and visit methods for all nonterminals of all grammars. An example for a modular operation on the example automata language is a model complexity analysis as depicted in Fig. 10.13. This analysis counts all instances of abstract syntax elements of a model that introduce

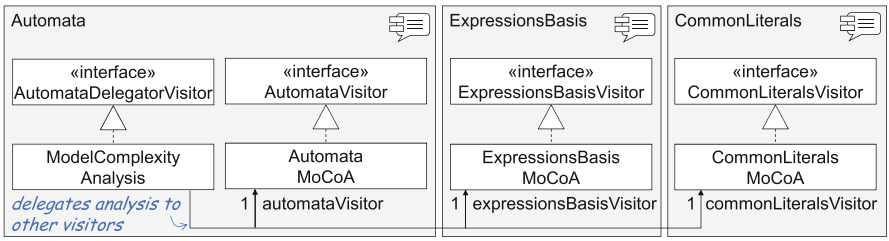

Fig. 10.13 Example for composing modular analyses via delegation visitor

concrete syntax. For each language component in the example that introduces concrete syntax, a class realising the model complexity analysis (suffixed MoCoA) is implemented. These classes implement the visitor interfaces and override their methods to count the syntax elements. The ExpressionsBasisMoCoA, for instance, is capable of counting model elements of expressions only. The CommonLiteralsMoCoA counts boolean and integer values only. The AutomataMoCoA counts all syntax elements of an automaton model except the expressions in the guards.

We distinguish different forms of composing analyses as explained in Chap. 4 of this book [Hei+21]. Combining these modular analyses can be achieved by employing a delegator visitor. In the example, the class ModelComplexityAnalysis extends the delegator visitor for automata and manages delegates for each analysis module. The effect of this is that the delegator visitor delegates the execution of the handle method for an AST node to the delegate, which is responsible for this node. Through this, the model complexity analysis takes into account all automaton model elements including those of the guard condition. Instead of this modular analysis, the ModelComplexityAnalysis could be realised as monolithical analysis as well. But as stated above, this would prevent reusing the analysis modules for literals and expressions for model complexity analyses in other contexts. Furthermore, modular analyses enable reusing foreign analysis parts conveniently. If, for example, the engineers of the automata language decide to use the CommonExpression language component with an individual analysis module instead of ExpressionBasis, the only adjustment in the analysis is to exchange the delegate object.

Conducting analyses can be orchestrated based on different strategies as described in Chap. 5 of this book [Hei+21]. If a composed analysis yields analysis results, these are typically contained in the analysis modules after execution of the analysis. Such results can be exploited in different forms, e.g., to calculate aggregated results or to serve as input for other analyses (cf. Chap. 7 of this book [Hei+21]). In our example, the results are collected from the modules and unified. Each above-mentioned analysis module can yield an integer number representing the number of syntax elements counted during analysis execution. A suitable technique for unifying the partial results in this analysis is to calculate the sum. Sometimes it is useful to exchange information between analysis modules, while the analysis is executed or to collect the analysis results in a common place. This can be realised by sharing a data structure between the analysis modules, e.g., by passing it to the analyses as argument. For instance, the model complexity analysis could store the syntax element counts by their abstract syntax type in a common map. The map could be passed to the analysis modules as argument.

Monolithic analyses can be reused for other analyses by means of delegator visitors as well. For instance, a new complexity analysis can use the analysis results both of the EffectiveStateDegrees analysis and of the ModelComplexityAnalysis. This new analysis can be realised as delegator visitor pointing to both analyses and combine their result. Sometimes, such analyses do not have to be composed at all: If the analyses do not depend on another, it is possible to execute these independently in sequence or parallel and combine their result (cf. Chaps. 4 and 7 of this book [Hei+21]).

10.4 Related Work

Research in software language engineering has produced a wealth of formalisms to define abstract and concrete syntaxes, well-formedness rules, and model transformations [Kle08, HRW18]. These include: (1) The grammar-based integrated syntax formalisms of Neverlang [VC15], Whole Platform [Erd+15], and Xtext [Bet16], as well as the abstract data types Spoofax [KV10] and the metamodels of GEMOC Studio [CBW17] and MPS [Voe11]. (2) Formalisms for the specification of well-formedness, such as OCL [Hei+10] or the Name-Binding Language [WKV14] of Spoofax. (3) Model transformation formalisms, such as ATL [Jou+06], the epsilon transformation language [KPP08], FreeMarker [HR17], or Xtend [Bet16]. Model transformations with ATL are explained in Chap. 12 of this book [Hei+21]. Language workbenches [Erd+15] combine multiple of such formalism to facilitate engineering the constituents of software languages. Yet, the compositionality of the related analyses is limited and rarely directly follows the composition of the syntaxes without severe manual implementation efforts.

For instance, in Neverlang [VC15], DSMLs are defined through language modules comprising grammars describing concrete and abstract syntax as well as through evaluation phases that realise well-formedness checking and syntheses. Extension points of grammars are used, but undefined, production names. While this enables to compose language modules along such extension points, there is no support for automatically composing the languages' analyses accordingly.

SugarJ [Erd+11] serves to specify syntactic extensions for Java that are contained in syntactic sugar libraries. By "desugaring", the extended syntax is transformed into the base syntax. SugarJ uses parsers that are capable of detecting ambiguities, for which they report an error. While it supports importing language modules into another, it does not automatically derive combined analyses from this integration.

The core of the ableC [Kam+17] language framework is an extensible variant of the C language. It uses attribute grammars to describe the syntax of independent language components and provides a composition mechanism for these that guarantees correct composition of the attribute grammars and, therefore, also of the related analyses. As the base language C, however, cannot be exchanged, this, of course, limits the application of ableC. The same holds for mbeddr [Voe+12], a projectional language workbench on top of a C base language.

SDF+FeatureHouse [LDA13] employs superimposition, weaving, and inheritance to compose language modules. While this supports powerful integration of syntaxes, the composition of the related analyses still demands significant effort.

10.5 Discussion

This chapter focuses on modularity in languages foreseen by language engineers; therefore, language components are built as individual units of reuse. In practice, however, this is rarely feasible and requires premature optimisation in identifying

such units of reuse. Instead, it occurs that parts of a language component's syntax are identified as units of reuse only once these parts are of use for another language, or if a similarity analysis between language components reveals a potential to extract a common part to a separate component. However, it is possible to modularise an existing MontiCore language with little effort by extracting the nonterminals that should be reused to a separate grammar. The original grammar then extends the new grammar, similar to the "pull-up attribute" refactoring in object-oriented programming.

MontiCore's support for engineering modular languages can be used to build product lines of languages [But+19] in which each feature uses a language component. These foster the reusability of language components for scenarios with a high complexity induced by the number and interrelations of available language components. Through the modular analysis infrastructure, analyses can be defined per feature and then are available for all products of the product line.

Reusing analyses as described in Sect. 10.3 has to be handled with care: If an analysis defined for a language is directly reused in a language that extends the original language, it might yield unintended results. Consider, for example, the automata language presented in Sect. 10.2 and a new HierarchicalAutomata language that extends this language and introduces decomposed states that themselves contain states and transitions. The model complexity analysis for the original automata language, as described in Sect. 10.3, can be applied to the language for hierarchical automata without modification. However, it depends on the realisation of hierarchical states whether these are taken into account or not.

If hierarchical states are introduced through overriding the production for states as depicted in Fig. 10.14a, neither hierarchical nor non-hierarchical states are visited by the visitor as the parser translates both into instances of the new ASTState class. Thus, both are not counted in the analysis. If, however, the hierarchical states are introduced by extending the state production (cf. Fig. 10.14b), non-hierarchical states are visited by the visitor and, thus, taken into account in the analysis.

The techniques described in this chapter can be used to realise both qualitative and quantitative, automated, static analyses as described in Chap. 4 of this book [Hei+21]. Given an AST of the input language as depicted in Fig. 10.1 enables realising analysis on the model/system structure, while the output AST, together with an understanding of the semantic domain, forms a basis for realising behavioural analyses.

MontiCore internally uses the modular analysis framework for each language: Context conditions are realised as Java classes and are checked against the AST using a visitor. Therefore, context conditions can be reused as part of reusing

(a)
```
State = "state" Name ( ("{" (State | Transition)* "}") | ";" );
```

(b)
```
HState extends State = "state" Name "{" (State | Transition)* "}";
```

Fig. 10.14 Adding hierarchical states via (**a**) overriding or (**b**) extending a grammar production

a language component. Similarly, the instantiation of the symbol table of a language [HR17] is performed by a symbol table creator realised as visitor.

10.6 Conclusion

We have presented an approach for compositional language engineering based on modular syntax definitions from which a modular, visitor-based infrastructure for model-based analyses and syntheses is derived. The presented approach relies on language extension and interface productions that can be extended in the extending languages. From this information, visitors for the participating languages are generated that automatically take care of model traversal. Hence, model-based analyses implemented through these visitors can be reused in other language combinations without modification.

The visitor-based infrastructure traverses the abstract syntax and can support realising a language's semantics. The applicability of the infrastructure for conceiving novel forms of generator compositions, however, has yet to be evaluated. This fosters not only the reuse of modelling languages and analysis tools but facilitates engineering truly domain-specific modelling languages to integrate experts of the different systems engineering domains more efficiently.

References

[Ada+17a] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Járôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*. Shaker, 2017. http://www.se-rwth.de/phdtheses/Modeling-Robotics-Tasks-for-Better-Separation-of-Concerns-Platform-Independence-and-Reuse.pdf.

[Ada+17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and AndreasWortmann. "Modeling Robotics Software Architectures with Modular Model Transformations". In: *Journal of Software Engineering for Robotics* 8.1 (2017), pp. 3–16. https://doi.org/10.1109/IRC.2017.16.

[Bet16] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[But+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. "Systematic Composition of Independent Language Features". In: *Journal of Systems and Software* 152 (2019), pp. 50–69.

[CBW17] Benoit Combemale, Olivier Barais, and Andreas Wortmann. "Language Engineering with the GEMOC Studio". In: *IEEE International Conference on Software Architecture Workshops*. 2017, pp. 189–191. https://doi.org/10.1109/ICSAW.2017.61.

[Dal+19] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, Louis Wachtmeister, and Andreas Wortmann. "Model-Driven Systems Engineering for Virtual Product Design". In: *First International Workshop on Multi-Paradigm Modelling for Cyber-Physical Systems, MPM4CPS*. Sept. 2019, pp. 430–435. https://doi.org/10.1109/MODELS-C.2019.00069.

[Dra+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. "SMArDT modeling for automotive software testing". In: *Software: Practice and Experience* 49.2 (2019), pp. 301–328. https://doi.org/10.1002/spe.2650.

[Eik+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. "Architecting Cloud Services for the Digital me in a Privacy-Aware Environment". In: *Software Architecture for Big Data and the Cloud*. 2017, pp. 207–226. https://doi.org/10.1016/B978-0-12-805467-3.00012-0.

[Erd+11] Sebastian Erdweg, Lennart CL Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. "Library-based Model-driven Software Development with SugarJ". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 2011, pp. 17–18. https://doi.org/10.1145/2048147.2048156.

[Erd+15] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47. https://doi.org/10.1016/j.cl.2015.08.007.

[FR07] Robert France and Bernhard Rumpe. "Model-driven Development of Complex Software: A Research Roadmap". In: *Future of Software Engineering* (May 2007), pp. 37–54. https://doi.org/10.1109/FOSE.2007.14.

[Hei+10] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. "Integrating OCL and textual modelling languages". In: *International Conference on Model Driven Engineering Languages and Systems*. 2010, pp. 349–363. https://doi.org/10.1007/978-3-642-21210-9_34.

[Hei+16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. "Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors". In: *Conference on Modelling Foundations and Applications*. 2016, pp. 67–82. https://doi.org/10.1007/978-3-319-42061-5_5.

[Hei+21] Robert Heinrich, Francisco Durán, Carolyn L. Talcott, and Steffen Zschaler (eds.) *Composing Model-Based Analysis Tools*. Springer, 2021. https://doi.org/10.1007/978-3-030-81915-6.

[HMR15] Katrin Hölldobler, Pedram Mir Seyed Nazari, and Bernhard Rumpe. "Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures". In: *Domain-Specific Modeling Workshop*. 2015, pp. 23–30. https://doi.org/10.1145/2846696.2846700.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Shaker, 2017. http://www.se-rwth.de/phdtheses/MontiCore-5-Language-Workbench-Edition-2017.pdf.

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. "Software Language Engineering in the Large: Towards Composing and Deriving Languages". In: *Computer Languages, Systems & Structures* 54 (2018), pp. 386–405.

[Jou+06] Frádáric Jouault, Freddy Allilaire, Jean Bázivin, Ivan Kurtev, and Patrick Valduriez. "ATL: a QVT-like transformation language". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006, pp. 719–720. https://doi.org/10.1145/1176617.1176691.

[Kam+17] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. "Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework". In: *Proceedings of the ACM on Programming Languages* 1 (Oct. 2017), 98:1–98:29. https://doi.org/10.1145/3138224.

[Kle08] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[KPP08] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. "The Epsilon transformation language". In: *International Conference on Theory and Practice of Model Transformations*. 2008, pp. 46–60. https://doi.org/10.1007/978-3-540-69927-9_4.

[KV10] Lennart C. L. Kats and Eelco Visser. "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*. 2010, pp. 444–463. https://doi.org/10.1145/1869459.1869497.

[LDA13] Jörg Liebig, Rolf Daniel, and Sven Apel. "Feature-oriented Language Families: A Case Study". In: *Seventh International Workshop on Variability Modelling of Software intensive Systems, VaMoS*. 2013, 11:1–11:8. https://doi.org/10.1145/2430502.2430518.

[MRR15] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. "Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table". In: *Domain-Specific Modeling Workshop*. 2015, pp. 37–42. https://doi.org/10.1145/2846696. 2846702.

[MRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. "An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information". In: *Modellierung 2016 Conference*. Vol. 254. Mar. 2016, pp. 133–140. https://doi.org/dl.gi.de/20.500.12116/819.

[Obj12] Object Management Group. *OMG Systems Modeling Language (OMG SysML), Version 1.3*. 2012. https://www.omg.org/spec/SysML/1.4/.

[Obj15] Object Management Group. *UML 2.5*. Tech. rep. formal/2015-03-01. Object Management Group, 2015.

[VC15] Edoardo Vacchi and Walter Cazzola. "Neverlang: A framework for feature-oriented language development". In: *Computer Languages, Systems & Structures* 43 (2015), pp. 1–40. https://doi.org/10.1016/j.cl.2015.02.001.

[Voe+12] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. "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*. 2012, pp. 121–140. https://doi.org/10.1145/2384716.2384767.

[Voe11] Markus Voelter. "Language and IDE Modularization and Composition with MPS". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. 2011, pp. 383–430. https://doi.org/10.1007/978-3-642-35992-7_11.

[WKV14] Guido H Wachsmuth, Gabriël D P Konat, and Eelco Visser. "Language Design with the Spoofax Language Workbench". In: *IEEE Software* 31.5 (2014), pp. 35–43. https://doi.org/10.1109/MS.2014.100.