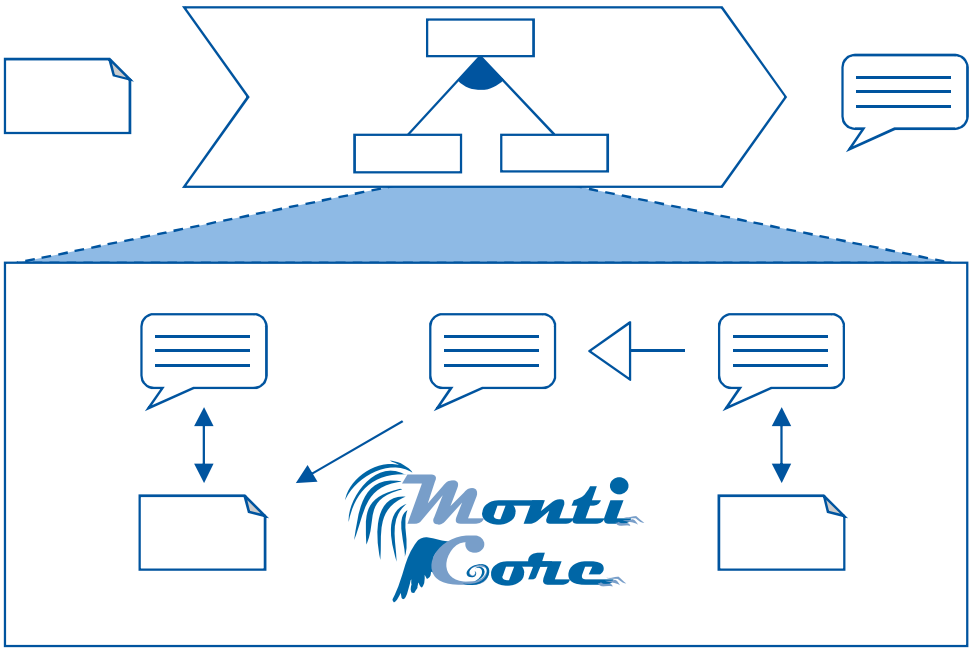
SE | RWTH AACHEN UNIVERSITY

Arvid Butting

Systematic Composition of Language Components in MontiCore



Aachener Informatik-Berichte,
Software Engineering
Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Systematic Composition of Language Components in MontiCore

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Arvid Butting, M.Sc. RWTH
aus Willich

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Professor Jeff Gray

Tag der mündlichen Prüfung: 12.07.2022

[But23] A. Butting:
Systematic Composition of Language Components in MontiCore.
Aachener Informatik-Berichte, Software Engineering, Band 53, ISBN 978-3-8440-8936-3, Shaker Verlag, Februar 2023.
www.se-rwth.de/publications/

Eidesstattliche Erklärung

I, Arvid Butting

erklärt hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden.

Hiermit erkläre ich an Eides statt

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand dieser Fakultät und Universität angefertigt;
2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;
3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;
4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;
5. Alle wesentlichen Quellen von Unterstützung wurden benannt;
6. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;
7. Teile dieser Arbeit wurden zuvor veröffentlicht und zwar in: [BEH$^+$20, BEK$^+$18a, BEK$^+$18b, BEK$^+$19, BMSN21, BW21]

Abstract

In model-driven development (MDD), models are central software engineering artifacts. MDD is applied to various domains such as avionics, law, mechanical engineering, or robotics, in which the domain engineers are not always software engineers. To this end, modelers should specify models in a notation close to the application domain, which is achieved by employing domain-specific modeling languages (DSMLs). In complex modern software applications, different aspects of an application are modeled with numerous integrated models. The models conform to heterogeneous, integrated DSMLs that can assure consistency between the models of an application.

Ad-hoc development of DSMLs is a time-consuming and error-prone process. Systematic and "off-the-shelf" black-box reuse of DSMLs or parts of it supports engineering DSMLs faster and more reliably. In black-box reuse, unlike reuse via clone-and-own, the reused parts remain unchanged and do not result in co-existing clones. Such reuse requires language engineers to be able to integrate DSMLs through different forms of language composition. Current approaches for engineering DSMLs often rely on generic language infrastructure, which complicates compatibility checks between the infrastructures of languages that are to be composed. Approaches for modularization of DSMLs typically focus on the conceptual parts of a language rather than on their realizations.

This thesis describes an approach for realizing modular language components that can be composed via their symbol tables to realize language product lines with the language workbench MontiCore. The proposed language components identify the entirety of source code artifacts that realize a DSML. The DSMLs rely on kind-typed symbol tables that assure language compatibility during language composition. Language composition via symbol tables is lightweight because language infrastructures are only loosely coupled. An approach for persisting symbol tables further decouples language infrastructures from another and increases the performance for type and consistency checking between models that conform to different DSMLs. With the approach for language product lines, language components can be composed systematically and undesired compositions can be avoided. Typed and persisted symbol tables, language components, and language product lines as presented in this thesis aim to realize DSML engineering in the large.

Kurzfassung

In der modellgetriebenen Softwareentwicklung (MDD) sind Modelle die zentralen Entwicklungsartefakte. MDD wird in verschiedenen Domänen wie Luftfahrt, Recht, Maschinenbau oder Robotik angewendet, in denen die Domänenexperten nicht immer auch Softwareentwickler sind. Daher sollten Modellierer die Modelle in einer Notation spezifizieren, die nah an der Anwendungsdomäne liegt. Dies wird durch die Nutzung von domänenspezifischen Modellierungssprachen (DSMLs) erreicht. In komplexen modernen Softwareanwendungen werden verschiedene Aspekte in zahlreichen integrierten Modellen dargestellt. Diese Modelle sind konform zu heterogenen, integrierten DSMLs, welche die Konsistenz der Modelle einer Applikation sicherstellen können.

Die Ad-hoc-Entwicklung von DSMLs ist ein zeitintensiver und fehleranfälliger Prozess. Systematische und standardmäßige Black-Box-Wiederverwendung von DSMLs oder Teilen von diesen unterstützt deren schnellere und zuverlässigere Entwicklung. Bei der Black-Box-Wiederverwendung werden im Gegensatz zur Wiederverwendung über Clone-and-Own die wiederverwendeten Anteile unverändert übernommen und führen nicht zu nebeneinander existierenden Klonen. Derartige Wiederverwendung setzt voraus, dass Sprachentwickler DSMLs durch verschiedene Formen der Sprachkomposition integrieren können. Bestehende Ansätze zur Entwicklung von DSMLs basieren oft auf generischer Sprachinfrastruktur, wodurch Kompatibilitätsprüfungen zwischen den Infrastrukturen von zu komponierenden Sprachen kompliziert sind. Ansätze für die Modularisierung von DSMLs fokussieren typischerweise die konzeptuellen Teile einer Sprache anstelle der Realisierung.

Diese Arbeit beschreibt einen Ansatz zur Realisierung von modularen Sprachkomponenten, die über ihre Symboltabellen komponiert werden können, um so Sprachproduktlinien in der Language Workbench MontiCore umsetzen zu können. Die vorgestellten Sprachkomponenten identifizieren die Gesamtheit der Quellcodeartefakte die eine DSML realisieren. Die DSMLs basieren auf durch Kinds getypte Symboltabellen, welche die Sprachkompatibilität während der Sprachkomposition sicherstellen. Die Komposition von Sprachen über Symboltabellen ist leichtgewichtig, weil die Sprachinfrastrukturen so nur lose gekoppelt werden. Ein Ansatz für die Persistenz von Symboltabellen entkoppelt die Sprachinfrastrukturen noch weiter voneinander und verbessert die Performanz für Typ- und Konsistenzprüfungen zwischen Modellen die zu unterschiedlichen DSMLs konform sind. Mit dem Ansatz für Sprachproduktlinien können Sprachkomponenten systematisch komponiert und unerwünschte Kompositionen vermieden werden. Die in dieser Arbeit vorgestellten getypten und persistierten Symboltabellen, Sprachkomponenten und Sprachproduktlinien zielen darauf ab, Sprachentwicklung im Großen umzusetzen.

Danksagung

Mein erster Dank gilt dem Erstgutachter dieser Arbeit, Prof. Dr. Bernhard Rumpe, welcher mich über meine gesamte Zeit am Lehrstuhl unterstützt und motiviert hat. Vielen Dank für das gute Feedback und die spannenden Diskussionen, sowie dafür dass ich für diese Arbeit – aber auch abseits dieser Arbeit – viele interessante Projekte und Themen bearbeiten durfte. Weiterhin bedanke ich mich bei Prof. Dr. Jeff Gray für die Übernahme der Rolle des Zweitgutachters sowie bei Prof. Dr. Erika Ábrahám und bei Prof. Dr. Ir. Dr. h.c. Joost-Pieter Katoen, welche die Prüfungskomission vervollständigen.

Für die stets gute Zusammenarbeit möchte ich mich bei meinen sämtlichen aktuellen und ehemaligen Kolleginnen und Kollegen bedanken. Ohne eure Unterstützung in den unterschiedlichen Phasen meiner Zeit am Lehrstuhl wäre mir das Erstellen dieser Arbeit deutlich schwerer gefallen und hätte mir deutlich weniger Freude bereitet. Besonders bedanken möchte ich mich bei Kai Adam, Daoud Ali, Vincent Bertram, Marita Breuer, Lennart Bucher, Joel Charles, Imke Nachmann, Niklas Dienstknecht, Robert Eikermann, Arkadii Gerasimov, Dr. Timo Greifenberg, Sylvia Gunder, Malte Heithoff, Steffen Hillemacher, Dr. Katrin Hölldobler, Nico Jansen, Dr. Oliver Kautz, Jörg Christian Kirchhof, Dr. Evgeny Kusmenko, Achim Lindt, Dr. Markus Look, Matthias Markthaler, Joshua Mingers, Sonja Müßigbrodt, Dr. Judith Michael, Dr. Pedram Mir Seyed Nazari, Lukas Netz, Jerome Pfeiffer, Mathias Pfeiffer, Nina Pichler, Manuel Pützer, Deni Raco, David Schmalzing, Dr. Christoph Schulze, Brian Sinkovec, Sebastian Stüber, Simon Varga, Galina Volkova, Louis Wachtmeister, Dr. Michael von Wenckstern und Jun.-Prof. Dr. Andreas Wortmann.

Für das Lesen von Abschnitten dieser Arbeit bedanke ich mich ganz herzlich bei Andreas, Christian, David, Evgeny, Imke, Jerome, Judith, Katrin, Lukas, Malte, Nico, Oliver, Sebastian, Simon und Steffen.

Ich bedanke mich bei meiner gesamten Familie für die Unterstützung während des Studiums, des Schreibens dieser Arbeit und der Prüfungsvorbereitung. Besonders bedanken möchte ich mich bei meinen Eltern Silke und Peer, die mir das Informatikstudium ermöglicht haben.

Zu guter Letzt bedanke ich mich ganz herzlich bei meiner Frau Sina: Du hast mich während der Anfertigung der Arbeit nicht nur durchweg unterstützt und motiviert, sondern auch verständnisvoll auf viel gemeinsame Zeit, insbesondere an den Wochenenden, verzichtet. Darüber hinaus bin ich überglücklich, dass du unseren Sohn Jano zur Welt gebracht hast.

Aachen, Juli 2022
Arvid Butting

Contents

Chapter 1

Introduction

With the digitalization of various documents and processes of different domains of daily life, more and more domain engineers are confronted with using and implementing software. As these domain engineers are rarely software engineers, digitalization raises the challenge of providing domain engineers with suitable software languages. The vocabulary used and understood by domain engineers differs from the notation of programming languages, which causes a conceptual gap between the problem and the implementation domains [FR07]. Enforcing domain engineers to learn programming languages can be avoided, *e.g.,* with *model-driven development* (MDD) [VSB+13] techniques. MDD uses models [Sta73] as central artifacts for problem descriptions and transforms such models into software implementations.

Dedicated domain-specific modeling languages (DSMLs) can support domain engineers in implementing software, as these languages typically have a reduced complexity compared to classical, general-purpose programming languages (GPLs) and make use of domain vocabulary [MHS05]. Domain-specific notations lead to more accuracy and better comprehensibility compared to GPLs [BGM10, KMC12]. Furthermore, using DSMLs instead of GPLs, such as Java or C++, reduces the accidental complexity [Bro87] for domain experts who are language users because these are liberated from learning complex concepts of GPLs. However, such domain-specific modeling [GNT+07] raises the challenge of building tailored DSMLs for different applications fast and reliably. This thesis combines approaches from software language engineering, component-based software engineering, and software product line engineering to meet this challenge.

Software language engineering investigates means and techniques for engineering software languages, such as DSMLs, with language workbenches [Fow05]. The complex applications of modern software and systems engineering require expertise from a variety of different domains. For example, the model-driven engineering of a service robotics application requires experts, among others, in human-robot interaction, robot kinematics, task planning, and software architectures. Experts of the different domains collaborate and ideally use DSMLs most suitable for their domains to achieve such complex endeavors. To ensure the internal correctness of models and conformance towards external requirements, the integration of the heterogeneous models of an application should be supported by the integration of the DSMLs to which the models conform. Such forms

of language composition in which the models remain individual artifacts but may refer to elements of other languages can be accomplished with symbol tables [MSN17].

Component-based software engineering helps increase the software's reuse and thereby also improves its quality and cost-efficiency. Software components [NR68] are loosely coupled to their environment, such as to other software components and, hence, can be individual units of reuse. The composition of components enables building software from tried-and-tested components. As "software languages are software, too" [FGLP10], the techniques of component-based software engineering can be applied to software languages. Recently, the modularization of software languages has been investigated in a wealth of different approaches [BvdBH+15, CKM+18, DCB+15, GGdL+19, GP15, KKCVW17, KVW13, LDC18, MGVB16]. However, the current approaches are often tied to underlying technologies. Furthermore, most approaches require in-depth knowledge about a language module to reuse it, limiting the approaches' scalability for larger constellations of languages.

Software product line engineering [ABKS13, CN02] investigates means to increase the reuse among similar software products in terms of software product lines. With such software product lines, the products can be engineered, analyzed, and evolved together. The means of software product line engineering can be applied to the engineering process of software languages, which yields language product lines [VCPC13, WHT+09]. Language product lines model families of similar software languages and increase the ability for language reuse.

Hoare noted that good programming language design requires "consolidation, not innovation" [Hoa73]. The reuse of tried-and-tested parts of language infrastructure and language syntax is not only beneficial for engineering high-quality languages but is also advantageous for language users who learn software languages faster if these rely on familiar vocabulary and concepts. An example of concepts that exist in many programming languages are for-loops. Curly brackets are a common notation in modern languages for enclosing blocks of statements. Besides better learnability, such common concepts and notations foster the understandability of models and programs by people who are not familiar with the syntax of a language.

This thesis describes means for software language engineering in the large by composing reusable language parts with the language workbench MontiCore [HKR21, KRV10]. Software languages can be encapsulated in language components to foster their reusability. Suitable forms of language composition based on kind-typed symbol table infrastructures enable composing language components to build novel languages reliably and with little effort. These forms of language composition integrate the software languages without creating a tight coupling between the individual languages. Moreover, the thesis explains an approach for realizing language product lines by the systematic composition of language components. This approach supports the controlled reuse of language components and prohibits undesired forms of language component compositions.

1.1 Research Question & Objectives

Engineering languages in the large bears numerous challenges to which this thesis can only contribute parts. The thesis focuses on techniques for engineering modular languages, enhanced forms of language composition suitable for being applied in larger contexts, and on engineering language product lines with MontiCore. The main research question of this thesis is:

> *How can modeling languages be composed via their symbol tables using reusable language components and how can these language components be arranged in variable language product lines, from which languages can be derived easily?*

To answer the main research question, the thesis investigates the following five partial research questions:

RQ1: *Can a symbol table infrastructure with typed symbol kinds support language composition?*
Symbol tables are central parts of a language infrastructure for realizing language composition. Type systems of programming languages enable constraining how typed language elements (*e.g.,* variables) are allowed to be used. A symbol table infrastructure whose symbols are typed with a symbol kind may support composing languages by constraining through a type system which elements of the languages are allowed to be used together. An answer to this research question should either propose a method for realizing a symbol table infrastructure that supports language composition through typed symbol kinds or argue that such symbol tables do not benefit language composition.

RQ2: *How can we reuse processed models of a language and of foreign languages?*
Reusing processed models of a language rather than unprocessed models increases the efficiency of language tools and, thus, enables agile development [Rum17]. Besides this, reusing processed models of a language with the tool of another language supports novel forms of language composition. An answer to this research question has to provide a means to persist processed models in a representation that can be loaded by language tools more efficiently than processing the models anew.

RQ3: *How to compose languages via kind-typed symbol tables?*
A language can only be separated into modules if suitable forms of language composition exist for composing the modules to obtain the complete language. An answer to this research question provides approaches for realizing different forms of language composition, such as language embedding, extension, or aggregation [HKR21] via typed symbol table infrastructures.

RQ4: *What constitutes a reusable language component?*
Technically, a software language is realized by software [FGLP10], comprising a set of interrelated source code artifacts. For reusing software languages, language engineers must be able to precisely identify which artifacts are part of the infrastructure of a language. An answer to this research question applies the concept of software components to languages for supporting language engineering in the large. This requires that language engineers can indicate which artifacts are part of a language component and which artifacts the language component uses.

RQ5: *How can a family/product line describing similar languages be modeled?*
In the same way as software engineering in general, software language engineering in the large can be supported by controlled reuse in product lines rather than cloning-and-owning [DRB+13] variants for different, similar applications. An answer to this research question proposes a concept for engineering language product lines that comprise individual language features, which can be enabled or disabled. Furthermore, a process for deriving languages as products from the product line has to be conceived.

The overall aim of this thesis is to increase the reusability of DSMLs and DSML parts. With kind-typed symbol tables, DSMLs have an interface for language composition that enables language composition without requiring in-depth knowledge about other parts of the language infrastructure. Through symbol table persistence, the tools of different languages can be decoupled from one another. With well-defined language components, the artifacts of a language can be identified and exchanged between language engineers. Furthermore, undesired relations between language components that could prevent reusing language components individually can be identified. Often, languages are only modularized based on a central artifact of the language definition, such as a context-free grammar or a metamodel. An objective of this thesis is to enable language engineers to modularize languages in terms of language components that contain all artifacts of a language definition.

Language components support the engineering of languages in the large. An objective of this thesis is to conceive a method for engineering language product lines that can support language engineers in providing tried-and-tested constellations of language components that can be used together. Moreover, language product lines should explicitly model the interrelations between language components to enable automatic derivation of languages from the product line. Furthermore, language product lines should model negative relations between languages, *i.e.,* to indicate that specific language components should not be used together for a particular application. Engineering of DSMLs typically has the challenge that language experts must be domain experts [MHS05]. Language product lines should separate concerns between language engineering experts who develop individual language components and domain experts who can compose languages by deriving languages from the product line.

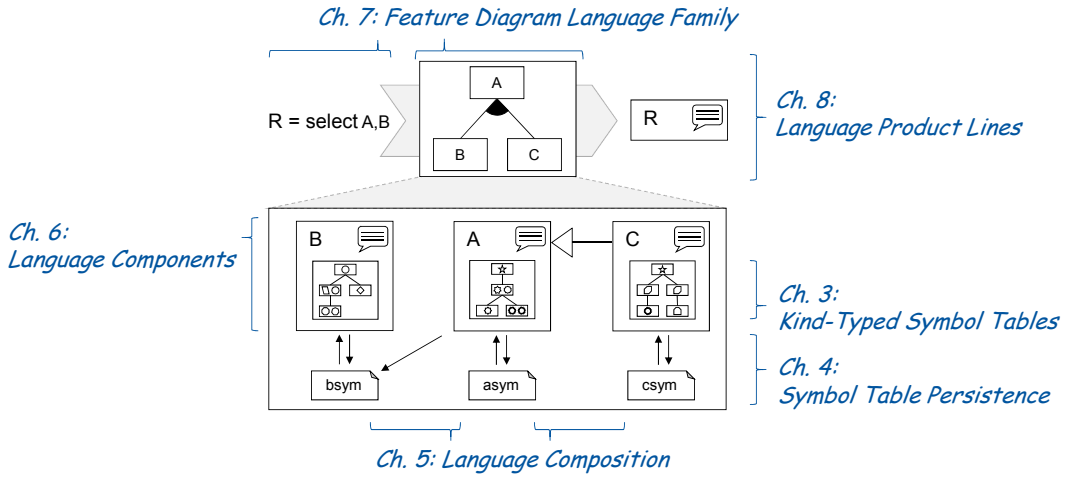


Figure 1.1: Structure of the thesis

1.2 Main Results and Structure of Thesis

The main results of this thesis are:

- The kind-typed symbol table infrastructure (STI) for MontiCore languages, for which large parts can be generated from MontiCore grammars and all generated parts can be customized with handwritten source code
- A customizable mechanism for loading and storing symbol tables in MontiCore that is integrated into the symbol resolution [HKR21]
- A tool for importing Java types to MontiCore symbol tables
- A modeling language for describing language components and a tool for identifying the artifacts of a language component
- An extensible feature diagram language for describing product lines that can be integrated with application domain languages through language composition
- An approach for realizing language product lines from language components and a semi-automated process for deriving languages from the product line by composing language components

Some research results presented in this thesis have already been published in a similar form before [BEH+20, BEK+18a, BEK+18b, BEK+19, BMSN21, BW21]. The thesis is structured in a bottom-up approach: it first explains results that support engineering a

single language and then iteratively expands the scope until it presents results that support language engineering in the large. Each chapter has individual sections discussing the chapter topics and relating these with similar work. An overview of the main contributions of this thesis and the chapters that describe these is given in Figure 1.1.

Chapter 2 explains the foundations and terminology upon which the consecutive chapters build. The foundations comprise topics from software language engineering and topics from software product line engineering. Furthermore, the chapter introduces the language workbench MontiCore.

Chapter 3 presents the overall method of systematic language composition for MontiCore language components that combines the individual approaches presented in this thesis.

Chapter 4 describes the STI that enables realizing type-safe cross-references of model elements within a single model and between models conforming to a single language.

Chapter 5 extends the STI described in Chapter 4 by the ability to load and store symbol tables to symbol table files. This increases the efficiency of language tools and prepares languages with a novel kind of interface for language composition.

Chapter 6 describes how different forms of language composition can be achieved with the STI and stored symbol table files.

Chapter 7 explains the MontiCore language component (MLC) language to describe which artifacts a language component comprises and a tool for investigating allowed and illegal relationships to other language components. Furthermore, the chapter introduces language component diagrams that enable visual representations of language components. Figure 1.1 uses the speech bubble icons that identify a language component.

Chapter 8 describes the MontiCore feature diagram language family and explains how its individual languages can be customized to realize different forms of product lines with the underlying language composition techniques of MontiCore with the STI.

Chapter 9 explains how the feature diagram language family and the MLC language can be combined to describe product lines of MontiCore languages. Language product lines support language engineering in the large, among other things, because undesired language compositions can be forbidden through the feature model and systematic reuse of language components in product lines reduces cloning-and-owning of common language parts.

Chapter 10 describes the evaluation of individual approaches presented in this thesis.

Chapter 11 summarizes the thesis results and describes how the results contribute answers to the research questions. The thesis concludes with an outlook on potential for future work.

Chapter 2

Foundations

This chapter defines basic terms, techniques, tools, and languages upon which the remaining chapters of the thesis build. Some of these notions belong to the field of software language engineering and are explained in Section 2.1. Large parts of the thesis use and extend the language workbench MontiCore, which is introduced in Section 2.2. Other notions relevant for the thesis belong to the field of software product line engineering and are introduced in Section 2.3.

2.1 Software Language Engineering

Software language engineering (SLE) is a sub-discipline of software engineering that aims at conceiving means and methods to support the development of software languages throughout their entire life cycle. This section explains basic properties of software languages and introduces essential features of the language workbench MontiCore [HKR21], which is a tool for building languages.

2.1.1 Software Languages

*Software language*s are artificial languages supposed to be both readable by developers and processable by machines. In MDD, the models typically conform to dedicated *modeling languages*, which form a particular group of software languages. SLE distinguishes *general-purpose languages* (GPLs) and *domain-specific languages* (DSLs) [Fow05]. GPLs enable language users to describe any computable problem in various application domains. Typical examples are programming languages [Mac99] such as Java or C. However, GPLs also exist within modeling languages. For instance, the unified modeling language (UML) [Rum16] is a general-purpose modeling language that can be employed for modeling aspects of various domains [BGM10].

DSLs, on the other hand, are specific to a particular domain and enable formulating problems of the domain with the vocabulary of this domain. Such specialized languages are typically limited in their expressiveness and are not necessarily Turing-complete. Instead, DSLs often have a concise syntax with limited complexity. Typical examples for DSLs are the hypertext markup language (HTML) in the domain of structured

documents or the structured query language (SQL) for the purpose of specifying queries for database management systems.

DSMLs are modeling languages for which the models are primarily intended to be used in certain domains only. Such models usually rely on terminology tied to specific domains rather than domain-agnostic formulations. As the remainder of this thesis mainly involves SLE in the context of DSMLs, we also refer to DSMLs as *languages*.

Definition of the Software Languages

Borrowed from formal language theory, a language can be defined as a "set of all linguistic utterances," where a linguistic utterance is, *e.g.,* a word, a sentence, or a conversation being part of the language [Kle08]. In the remainder of this thesis, we use the terms *word* and *sentence* interchangeably for linguistic utterances of a language. As the thesis lies in the context of MDD, it also uses the term *model* for a word of the language. For software language engineering, it is helpful to define a software language more precisely.

According to Harel & Rumpe [HR04], a software language definition consists of (1) the language's *syntax* or notation, which is a possibly infinite set of words that are legal in the language, (2) a *semantic domain*, and (3) a *semantic mapping* that gives each sentence of a language a meaning by mapping it to the semantic domain. The semantic domain is typically formally grounded and based on a mathematical theory such as, for example, FOCUS [BS01]. Together, semantic domain and semantic mapping form the *semantics* of a language.

An alternative definition for software languages by the GEMOC initiative [CBCR15] separates the syntax of a language into the *concrete syntax* and the *abstract syntax*. While the concrete syntax describes the appearance or presentation of a language, such as, in terms of the keywords or order of statements, the abstract syntax captures the essential parts of the language structure. According to the GEMOC definition, each language further has *static* and *dynamic semantics*. The "static semantics" [CBCR15] or context conditions in the form of Boolean predicates indicate the well-formedness of sentences of the language. According to the definition of Harel & Rumpe, the syntax of languages considers only well-formed models as legal words of the language. However, the definition by the GEMOC initiative allows ill-formed models as part of the language. Instead, the language's static semantics [CBCR15] checks the well-formedness of models is a separate part of the language according to the GEMOC initiative. The "dynamic semantics" of the GEMOC software language definition corresponds with the semantics in the definition of Harel & Rumpe. The latter definition, however, describes semantics more precisely as a tuple of semantic mapping and semantic domain. Moreover, the term *dynamic semantics* can be misleading for languages in which models cannot directly be executed. In the remainder of this thesis, we rely on the following definition that is based on the language definition by Harel & Rumpe but separates between concrete and abstract syntax according to the GEMOC definition:

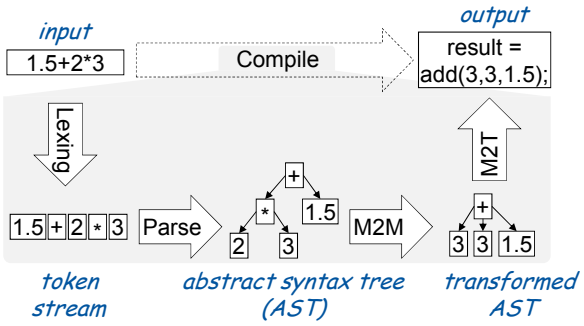


Figure 2.1: Example for the steps involved in compiling a model

Definition 1 (Software Language)**.** *A software language definition consists of*

- *the concrete syntax that can be, for instance, textual or graphical, and reflects the set of all sentences of the language,*
- *the abstract syntax, i.e., essential concepts and structure of the sentences,*
- *a semantic domain, and*
- *the semantic mapping that gives each sentence of a language a meaning by mapping it to the semantic domain.*

Software languages can be realized in a variety of technological spaces. The technological space determines whether the concrete syntax of a language is textual or graphical. Textual and graphical languages have individual advantages and disadvantages [GKR+07]. Textual languages usually employ context-free grammars for the description of the concrete syntax. The concrete syntax of graphical languages can be, *e.g.,* box-and-line based, tabular, or tree-based [CFJ+16]. Textual languages are usually editable with any text editor, while graphical languages are edited via dedicated model editors. Projectional editors [VSBK14] enable editing a model's abstract syntax through different forms of concrete syntaxes. The abstract syntax of a language can be specified through context-free grammars, such as in the language workbenches MontiCore [KRV07] or Xtext [Bet16]. Alternatively, the abstract syntax can be specified through a metamodel, which is a "model of a (user) model" [AK03]. EMF Ecore [SBMP08] is a common meta-language to define metamodels. The semantics [HR04] of a language is either denotational, operational, or axiomatic [CBCR15]. Operational realizations can be conceived through code generators, pretty printers, or interpreters.

Structure of a Compiler

Textual languages are typically processed with a compiler [ALSU07] that translates a source program/model into an executable representation (*i.e.,* machine code or a program of a programming language). In the following, we refer to the input of a compiler, which can be either in the form of a file or a String containing the content of a file, as a *model*. The process of compiling a model comprises several steps that are visualized by the example in Figure 2.1. The example input model is the expression `"1.5+2*3"`. In the first step, the model is split into a stream of tokens by a *lexer* in a process called *lexical analysis*.

A *token* is an abstract syntax element realized as a pair consisting of the token name that identifies the token and an optional value. The example contains tokens for (floating point) numbers and operators. The tokens for numbers have the name `number` and the respective value of the number as their value. For operators, the concrete operator can be either the value of a token with the name `operator`. Alternatively, the operators can be realized as tokens with different names (such as, a token `plus`) and without a variable value. A *parser* translates the token stream into an abstract syntax tree.

An abstract syntax tree (AST) is a data structure that reflects the structure of the model in terms of a tree. The AST contains all information required for performing further analysis and transformations of a model. Other information in a model, such as comments, white space characters, or syntactic sugar, may not be contained in the AST. Sometimes, the AST is also referred to as a *parse tree* because it results from parsing. In the remainder of the thesis, we often do not need to distinguish between lexing and parsing and use the term *parsing* for the entire process from input model to AST.

In a subsequent process, compilers realize transformations on the AST. We distinguish between model-to-model transformations (M2M) [MVG06] that have a model as the result of the transformation and model-to-text transformations (M2T) [KT08] that result in a sequence of characters (*i.e.,* text). M2M transformations can be used to simplify the AST structure, introduce additional relationships between AST nodes, or translate the nodes of an AST into an AST of another language.

The last step that a compiler performs while processing an input program is always a model-to-text transformation. M2T transformations can be realized with code generators or pretty printers. A code generator typically relies on code *templates* [CH03, Sch12]. A template is a document that contains free text elements and elements that are directives for a template processor. A template processor reads a template, replaces the directives with information conceived, *e.g.,* from the AST of a model, and prints the resulting text to a (generated) file. A pretty printer traverses the AST nodes one after another and translates each AST node into a piece of text. The result is a concatenation of these texts. As an alternative to an M2T transformation with a code generator, it is also possible to use an *interpreter* that assigns meaning to model/program parts instead of translating these into an executable representation. In this case, however, the language-

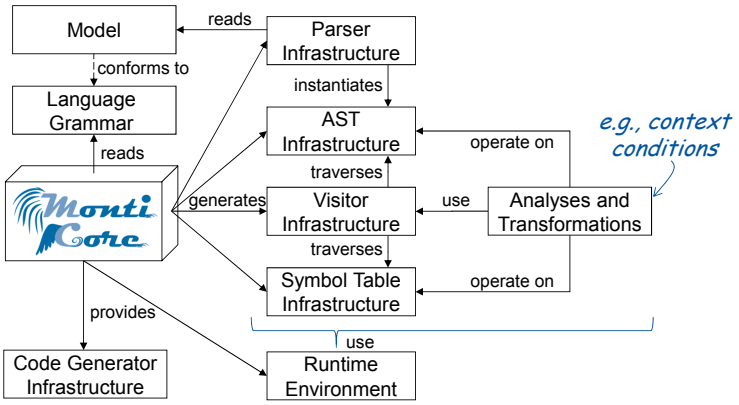


Figure 2.2: Overview of MontiCore's language engineering infrastructure

processing tool should not be referred to as a compiler since it does not translate the input program.

2.2 The MontiCore Language Workbench

Language workbenches [Fow05] are tools that support the development of software languages. MontiCore [GKR+08, HKR21] is a language workbench for the development of textual, external DSMLs. At its core, MontiCore uses MontiCore grammars (MCGs) that serve as integrated descriptions for a language's concrete and abstract syntax. MontiCore grammars are context-free grammars in a notation based on the extended Backus-Naur form (EBNF) [ALSU07] enriched with concepts from object-oriented programming. From such MontiCore grammars, MontiCore generates language processing tooling such as, a parser, an abstract syntax data structure, and a visitor infrastructure for traversing the abstract syntax data structure. The generation of the parser relies on ANTLR [Par13]. Figure 2.2 displays an overview of the core concepts of MontiCore. A parser contained in the generated parser infrastructure reads models that conform to the language and instantiates the AST that is part of the language's generated AST infrastructure. Besides the AST, the symbol table infrastructure is part of the abstract syntax and enables realizing symbolic links between name usages and name definitions. This is the basis for implementing type checks. MontiCore generates a visitor infrastructure that is capable of traversing both AST and symbol tables. All generated language infrastructure parts rely on a runtime environment that MontiCore provides for each language.

Context-free grammars are not powerful enough for certain restrictions, such as, the well-formedness of models. To check the well-formedness of processed models, MontiCore supports the definition of well-formedness rules. These are realized as Java classes

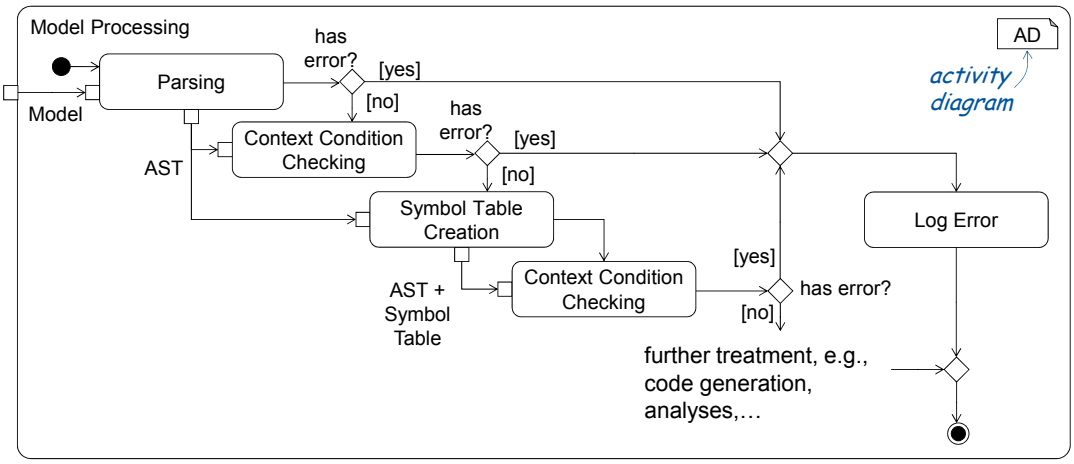


Figure 2.3: Example for the activities involved in processing a model with MontiCore

and referred to as context conditions (or short, CoCos). Each context condition is implemented against an element of the abstract syntax data structure. A context condition checker checks the context conditions with the visitor infrastructure that traverses the corresponding abstract syntax. Further analyses and transformations can be realized with the visitor infrastructure as well. MontiCore supports template-based code generation and contains a code generation framework based on the template engine FreeMarker [For13]. However, MontiCore supports transformations into other models instead of code generation as well [Höl18].

Processing a model with a language infrastructure generated by MontiCore involves several activities depicted as a UML activity diagram (AD) [www21e] in Figure 2.3. While most parts of the infrastructure for processing a model are generated from the language's grammar, the orchestration of the processes within a *language tool* that processes models has to be implemented manually. The first activity is parsing, which reads a model from a passed String or a passed file and instantiates the AST data structure. If the parsing process yields errors, these are logged, and any further steps of model processing are usually omitted. If the model is successfully parsed, the language tool can use the AST to realize context condition checks that operate on the AST and do not require the symbol table. For example, a context condition that checks certain naming conventions for a language element can be realized without the symbol table. If none of the context conditions is violated, the tool instantiates the symbol table from the AST. After the symbol table is instantiated, the context conditions that operate on the symbol table can be checked. Context conditions that rely on the symbol table typically check type correctness, referential integrity, or uniqueness of names. Since only some forms of context conditions can be checked without the symbol table, it is possible to check all

context conditions after the symbol table has been instantiated. However, symbol table instantiation for ill-formed models can be avoided if the models have been identified as ill-formed through context conditions checked before the symbol table is instantiated. Afterward, the abstract syntax can be the basis for analyses and transformations.

We sometimes refer to the infrastructure that realizes the parsing process and well-formedness checking as the language's *frontend*. Language users typically utilize the frontend of a MontiCore language through a language(-processing) tool that is realized as a command-line interface (CLI) tool. Optionally, additional tools for a language, such as tools integrated with a build system or tools that perform certain analyses or transformations, may co-exist with the language's CLI tool. The following sections describe the parts of the MontiCore infrastructure that are most relevant for the remainder of this thesis in more detail.

2.2.1 MontiCore Grammars

MontiCore grammars [HKR21] are context-free grammars [HMU01], *i.e.,* comprise nonterminals, terminals, a start nonterminal, and a set of grammar rules. For brevity, we refer to MontiCore grammars as *grammars* in the remainder of this thesis if the context is clear. The syntax of MontiCore grammars is based upon EBNF but also borrows elements from object-oriented programming. For instance, similar to Java types, MontiCore grammars may be located in a package indicated by a package declaration at the beginning of a grammar, and a grammar may import other grammar artifacts. Moreover, MontiCore grammars distinguish different forms of grammar rules that influence the parsing and the AST data structure. The main part of each MontiCore grammar begins with the keyword `grammar` followed by the name of the grammar. A grammar may extend one or more other grammars and, through this, reuse all grammar rules of the extended grammar(s). This is explained in more detail in Section 2.2.8. The body of the grammar follows the grammar's name and potential references to inherited grammars and is enclosed in curly brackets. The grammar body contains grammar rules that comprise a left-hand side and a right-hand side, separated by an equal sign `"="`. Similar to statements in Java, grammar rules in MontiCore end with a semicolon.

Nonterminal names used in grammar rules must start with an uppercase letter, and quotation marks enclose terminal names. The left-hand side of grammar rules in context-free grammars must contain a single nonterminal only. We denote this as the nonterminal *defined by* this grammar rule. On the right-hand side, grammars can contain both nonterminals and terminals. These can be either arranged as alternatives, indicated by `"|"`, or as concatenations, indicated by whitespace. Nonterminals and terminals can be arranged in blocks enclosed by round brackets. Each nonterminal, terminal, or block may be marked as optional with a consecutive question mark `"?"` or may be iterated. The grammar syntax indicates non-empty iterations with a consecutive plus `"+"` and other iterations through a Kleene star `"*"`.

```
01 grammar Automata extends de.monticore.MCBasics {
02
03   Automaton = "automaton" Name "{" (State | Transition)* "}" ;
04
05   State = [ "initial" ]? "state" Name ";" ;
06
07   Transition = from:Name "-" input:Name ">" to:Name ";" ;
08 }
```

MCG

MontiCore grammar

Figure 2.4: MontiCore grammar for an automata language

An example of a grammar is depicted in Figure 2.4. The example grammar describes the syntax for an automata language with states, transitions, and inputs that trigger the transitions. The grammar has the name `Automata` and extends a MontiCore grammar with the name `de.monticore.MCBasics` that is part of MontiCore's integrated language component library [BEH+20]. This grammar defines several tokens and nonterminals, such as the nonterminal `Name`. The `Name` nonterminal defines the syntax of identifiers in usual programming languages. A name must start with a letter, followed by any other letters, numbers, and separator characters such as underscores but no whitespace characters. In l. 3, the `Automata` grammar defines the nonterminal `Automaton` with a grammar rule. The concrete syntax of an automaton begins with the keyword `automaton`, followed by the name of an automaton and its body enclosed by curly brackets. The body may contain states and transitions in an arbitrary order but may also be empty.

By default, terminals in MontiCore are not part of the abstract syntax as their content does not provide any information relevant to the abstract syntax. However, terminals in MontiCore can be enclosed by square brackets. This causes the terminal to occur in the abstract syntax, which is helpful, for reflecting the presence or absence of an optional keyword in the AST. For instance, a language that employs modifiers as part of a nonterminal for variable declarations can realize the modifiers as terminals relevant to the abstract syntax. In the abstract syntax, the information which modifiers are present in a model is relevant for further steps such as code generation. In the automata grammar in Figure 2.4, states are individual nonterminals defined through the rule in l. 5. A state may be marked as an initial state with the optional keyword `initial` that is marked relevant to the abstract syntax. Each state further has the keyword `state`, followed by the name of the state and a semicolon.

To distinguish different occurrences of the same nonterminal on the right-hand side of a grammar rule, nonterminals may optionally be given a distinctive name. We refer to such nonterminals as *named nonterminals*. A named nonterminal begins with the distinctive name that usually starts with a lowercase letter, followed by a double colon and the nonterminal name. In the automata grammar in Figure 2.4, transitions are represented via an individual nonterminal defined by the rule in l. 7. Transitions have

```
01 grammar AltAutomata extends de.monticore.MCBasics {                    MCG
02
03   Automaton = "automaton" Name "{" AutElement* "}" ;
04
05   interface AutElement;
06
07   State implements AutElement = [ "initial" ]? "state" Name ";" ;
08
09   Transition implements AutElement = from:Name "-" input:Name ">" to:Name ";" ;
10 }
```

Figure 2.5: MontiCore grammar for an automata language with interface nonterminals

three names: the name that syntactically appears first is the name of the source state and, hence, is named `from`. The consecutive name refers to the input of the transition and the third name identifies the target state of the transition.

Grammar rules can extend other grammar rules. If a grammar rule `A` extends a grammar rule `B`, the right-hand side of `B` is automatically considered an alternative to the right-hand side of `A` by the parser. Any usage of the nonterminal `B` on the right-hand side of another grammar rule may be derived by the parser with the nonterminal `A`. The grammar rules described above are the usual rules of context-free grammars. They translate into class types in the AST, which is explained in more detail in the section about the AST data structure. These rules, hence, are also referred to as *class rules*. Besides class rules, MontiCore grammars may use several kinds of special grammar rules.

An *interface rule* defines an interface nonterminal. It begins with the keyword `interface`, followed by the name of the nonterminal. Instead of a class rule that defines a class type in the abstract syntax, an interface rule is translated into an interface type in the abstract syntax. Class rules can implement one or more interface rules, and interface rules can extend other interface rules. An interface nonterminal may be used on the right-hand side of any class or interface rule. The effect of using an interface nonterminal on the right-hand side of a grammar rule for the parser can be described by a transformation that would remove the interface rule and accepts the same concrete syntax: an interface nonterminal can be replaced by a class nonterminal with an alternative over all nonterminals that extend or implement the interface nonterminal on its right-hand side. This would, however, result in a different AST data structure. An interface rule may omit a right-hand side because interface rules per se do not provide syntax elements relevant to the generated parser.

However, an interface rule may also have a right-hand side that contains (cardinalized) nonterminals. The effect of this is that the interface rule prescribes the presence of these nonterminals in the right-hand side of each rule that implements the interface nonterminal. The order of the nonterminals may change in implementing rules, but the cardinality has to match. Furthermore, implementing rules may use any nonterminals

and terminals beyond the ones prescribed by the interface rule. With this behavior, interface rules can act as extension points of a grammar that prescribe certain elements of their implementations and, thus, can prevent undesired forms of implementations. At the same time, the information about elements that every implementation contains can be leveraged to check some well-formedness properties of the nonterminal without being aware of the exact implementations. Upon a violation of the prescribed elements, the MontiCore generator produces an error.

An alternative variant of the grammar in Figure 2.4 that accepts the same concrete syntax is depicted in Figure 2.5. This variant employs an interface nonterminal (*cf.* l. 5) `AutElement` for the elements of an automaton (l. 3) instead of an alternative of the `State` and `Transition` nonterminals. The interface nonterminal can be used on the right-hand side of the rule for the nonterminal `Automaton` in the same way as any class nonterminal. The nonterminals `State` and `Transition` implement the `AutElement` interface nonterminal as depicted in ll. 7-9.

A MontiCore grammar may define at most one *start rule* that indicates the start nonterminal of a grammar. The start nonterminal, however, must be defined in a separate grammar rule. A start rule begins with the keyword `start`, followed by the name of the start nonterminal and is terminated with a semicolon. A start rule must not have a right-hand side. If a grammar does not have an explicit start rule, the nonterminal defined by the first rule of a grammar is used as start nonterminal.

MontiCore grammars may further contain *AST rules* that begin with the keyword `astrule`. An AST rule is specific to a nonterminal and defines members of the AST node type generated from the nonterminal. To do so, the right-hand side of an AST rule may contain nonterminals that are translated into attributes in the corresponding AST node type and method declarations that are translated into methods of the AST node type. AST rules do not have any effect on the parser.

MontiCore enables language engineers to use further kinds of grammar rules that are, however, not directly relevant for the remainder of this thesis. These include grammar rules of the following kinds:

Lexer rules begin with the keyword `token` and contain regular expressions for defining token classes. The generated lexer and parser internally use these token classes. Further lexer rules can define parts of a token with the keyword `fragment token`. These can be used on the right-hand side of token rules and, among other things, increase the readability of complex regular expressions.

Abstract rules begin with the keyword `abstract`. Like interface rules, abstract rules can be used to indicate an extension point. The parser does not distinguish abstract and interface nonterminals, but interface nonterminals are translated into interface types in the abstract syntax, and the nonterminals defined by abstract rules are translated into abstract AST types. The allowed forms of inheritance for abstract rules follow the allowed forms of inheritance for abstract Java classes. For instance, a class rule in

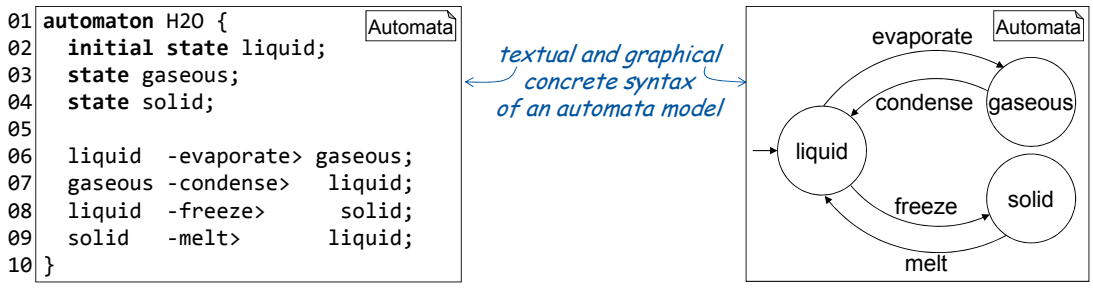


Figure 2.6: Example model conforming to the automata language

MontiCore may extend at most one abstract rule, and an interface rule must not extend an abstract rule.

External rules begin with the keyword `external` and indicate a mandatory extension point that other grammars must fill. Other grammars may provide an implementation for the external rule by extending the external nonterminal. Contrary to abstract and interface rules, external rules must not contain a right-hand side. Hence, they cannot prescribe elements of the right-hand side of rules that provide an implementation.

Enumeration rules begin with the keyword `enum` and describe enumerations as an alternative over terminals on the right-hand side of the grammar rule. In the abstract syntax, enumeration rules are translated to enumeration types.

Besides different kinds of grammar rules, the body of a MontiCore grammar may contain *grammar directives* and *grammar concepts*. Grammar directives and concepts configure the generation of parser and lexer and, hence, the parsing process for certain special cases. For instance, the grammar directive `nokeyword` enables using a keyword as a variable name without causing clashes in the underlying token classes [HKR21].

An example of a model `H2O` that conforms to the automata language is depicted on the left side of Figure 2.6. As the grammars for the automata language depicted in Figure 2.4 and Figure 2.5 define the same concrete syntax, the `H2O` model conforms to both languages. The model describes the aggregation states of water and the most relevant transition between these. The right side of the figure displays a graphical representation of the textual model depicted on the left side of the figure. The textual model begins with the keyword `automaton` followed by the name of the automaton and an opening curly bracket to indicate the beginning of the automaton's body (l. 1). In ll. 2-4, the model contains definitions of the three states `liquid`, `gaseous`, and `solid` where `liquid` is marked as the initial state. The automaton has four transitions in ll. 6-9 that connect the three states. Each transition begins with the name of the source state, followed by a textual representation of an arrow containing the name of the transition input and the name of the target state of the transition.

```
01 component grammar BaseADL extends de.monticore.types.MCBasicTypes {
02
03   ComponentDef = "component" Name "{" CmpElement* "}" ;
04
05   interface CmpElement;
06
07   Port implements CmpElement = "port" dir:["in"|"out"] portType:Name Name ";" ;
08
09   SubComponent implements CmpElement = "component" type:Name Name? ";" ;
10
11   Connector implements CmpElement = "connect" src:MCQualifiedName
12                                     "->"     trgt:MCQualifiedName ";" ;
13
14   interface IBehavior extends CmpElement;
15 }
```

MCG

Figure 2.7: Component grammar for a basic architecture description language

A grammar may be left incomplete intentionally. A grammar is incomplete if it contains interface, abstract, or external rules for which no realization is available in the grammar or any inherited grammars. For instance, an interface nonterminal has to be implemented by at least one class rule. An incomplete grammar is also referred to as a *component grammar* and is indicated with the keyword `component` before the keyword `grammar`. For component grammars, MontiCore does not generate a parser. If a grammar is incomplete but not marked as component grammar, MontiCore produces an error. A component grammar may be completed by another grammar that extends the component grammar and provides all required implementations and realizations for interface, abstract, and external rules.

The `BaseADL` grammar depicted in Figure 2.7 is an example of a component grammar. The grammar describes the syntax of a simple component & connector architecture description language (ADL) [MT00]. The syntax is strongly inspired by the syntax of the MontiArc ADL [BKRW17]. It is marked with the keyword `component` and reuses the grammar `de.monticore.types.MCBasicTypes` from MontiCore's integrated language component library [BEH+20] (l. 1). Each model of the `BaseADL` grammar is a component with a name and a body enclosed by curly brackets (l. 3). The body contains different elements that are realized by the interface nonterminal `CmpElement`. Ports (l. 7) are component elements that represent directed and typed endpoints of message-based communication between different components.

Moreover, components may be composed of subcomponents (l. 9), which are instances of other components with a type and, optionally, a given name. Connectors (ll. 11-12) may also be part of a component and realize communication channels between a source and a target port. To identify these ports, the grammar uses qualified names, *i.e.,* names that comprise dot-separated name parts. The syntax for such qualified names is reused from the grammar `MCBasicTypes`. The interface nonterminal `IBehavior` describes

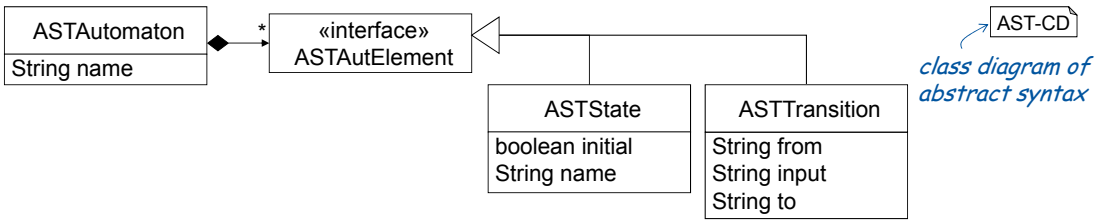


Figure 2.8: AST data structure of the automata language

the internal component behavior and is also a `CmpElement`. The `BaseADL` does not provide an implementation for the interface nonterminal and, hence, must be marked as component grammar. Other languages can provide syntax for describing the behavior of a component [BHH+17]. For further information about the MontiCore grammar language, we refer to the technical report of MontiCore [HKR21].

2.2.2 Abstract Syntax Tree Data Structure

The AST data structure in MontiCore is defined through a language's grammar. For each nonterminal defined in a grammar, MontiCore generates a corresponding element of the AST data structure that equals the nonterminal's name with the prefix `"AST"`. The host language of MontiCore languages is Java and, hence, the AST data structure is generated in terms of Java types.

MontiCore generates AST classes for each class nonterminal, AST interfaces for each interface nonterminal, and AST enumerations for each enumeration nonterminal. The nonterminals on the right-hand side of a class rule are translated into attributes of the AST. The type of such an attribute is the AST type of the corresponding nonterminal and the name of the attribute equals the name of a named nonterminal. If a nonterminal has no distinctive name, MontiCore uses the nonterminal name by default, starting with a lowercase letter. Terminals marked as relevant for the abstract syntax through enclosed square brackets are translated into attributes of the type `boolean`. The attribute name is derived from the terminal.

Depending on the (non)terminal cardinalities, the type of the attributes is generated differently: instead of the original attribute type as described before, optional (non)terminals are translated into attributes of the Java type `Optional` with the generic type argument of the original type. Similarly, iterated (non)terminals are translated into attributes of the Java type `List` with the generic type argument of the original type. If a nonterminal is optional because it appears only in some out of different alternatives, it is handled the same way as nonterminals marked with an optional cardinality. If a nonterminal occurs multiple times in the right-hand side of a grammar rule without a distinctive name or with an equal distinctive name, it is handled in the same way as an iterated nonterminal.

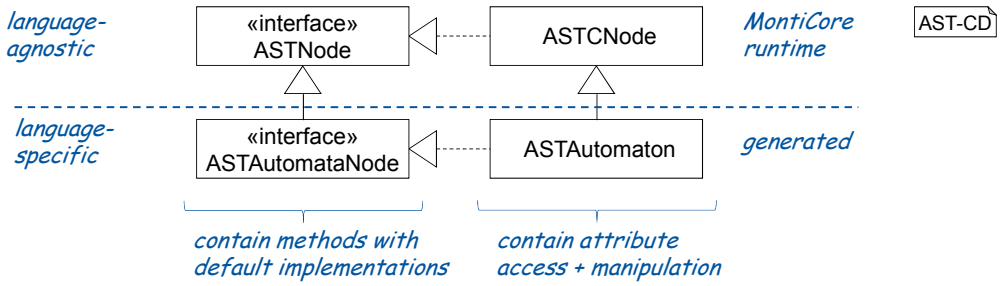


Figure 2.9: Relationships between AST node types

Some nonterminals are translated into built-in data types of Java. For instance, the nonterminal `Name` is translated into an attribute of the Java type `String`. For each attribute of an AST type, MontiCore generates accessor and mutator methods, including getter and setter methods. For optional and iterated types, methods that delegate to common methods of the Java types `List` and `Optional` are generated. These include a method for adding an element to a list attribute and a method for checking the presence of an optional attribute. If a grammar rule extends or implements other rules, the relationship is translated into corresponding inheritance relationships between the abstract syntax types.

An example of the AST data structure of the automata language as defined by the grammar in Figure 2.5 is depicted as a UML class diagram (CD) [Rum16, www21e] in Figure 2.8. The AST class `ASTAutomaton` has an attribute for the name of the automaton and a list of elements of the interface type `ASTAutElement`. The classes `ASTState` and `ASTTransition` implement the interface `ASTAutElement`. The class `ASTState` has an attribute of the type `boolean` that indicates whether the name is an initial state and a `String` attribute for the state's name. The class `ASTTransition` has three attributes of the type `String` that indicate the name of the source state, the transition's input, and the target state.

In addition to the AST types generated for each nonterminal, MontiCore generates an interface that all AST types of a language implement. This interface supports the traversal of the abstract syntax with visitors. MontiCore has a runtime environment that contains common Java types that all languages can use. This runtime environment contains an interface `ASTNode` and a class `ASTCNode`. The interface `ASTNode` contains the signatures of methods that are common to all AST node types. The class `ASTCNode` implements the interface `ASTNode` and provides method implementations that are not specific to any type of AST node, such as the source position of an AST node within a model. Each AST interface generated for a MontiCore language implements the interface `ASTNode`, and each generated AST class extends the class `ASTCNode`. The relationships between these types are depicted in Figure 2.9. Furthermore, MontiCore generates a

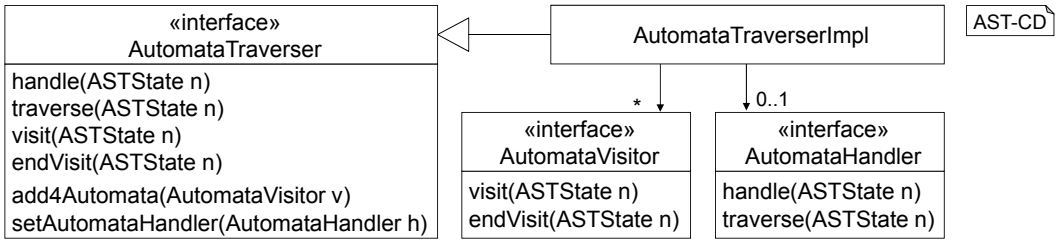


Figure 2.10: Interfaces and classes that form the visitor infrastructure for traversing the abstract syntax of the automata language

builder class [GHJV95] for each AST node class. The generated parser uses the builders to instantiate the AST node classes while parsing a model.

2.2.3 Traversing the Abstract Syntax

The AST data structure of a language can be traversed with the visitor infrastructure that MontiCore generates for each language. The visitor infrastructure is based on the visitor pattern [GHJV95] and realizes a depth-first traversal of ASTs. The visitor infrastructure is integrated with the AST data structure through double dispatching, where each AST class or interface has an `accept` method that calls a `handle` method in the visitor infrastructure. The `handle` methods trigger the traversal of the AST structure. MontiCore supports both pre-order and post-order traversals [ALSU07], where the pre-order traversal is realized through `visit` methods of a visitor, and the post-order traversal is realized through `endVisit` methods. The `traverse` methods carry out the traversal of the contained substructures of an AST node.

Other than usual for implementations of tree structures, the AST nodes of MontiCore do not have `getChildren` methods that offer uniform access to all children of the node in the tree at once. Instead, each type of the contained structure can be addressed by its concrete type because the AST node types are generated from the grammar. In the example of the automata language, an automaton has two lists: one for contained states and another one for contained transitions. Both are traversed with the `traverse` method for the type `ASTAutomaton`.

The interfaces and classes of the visitor infrastructure for the automata language are depicted as a class diagram in Figure 2.10. The infrastructure comprises a *visitor interface*, a *handler interface*, a *traverser interface*, and a *traverser implementation* class. The visitor interface, such as `AutomataVisitor`, contains `visit` and `endVisit` methods for each AST class or interface defined in a grammar. For brevity, the example depicts only the methods for the type `ASTState`. The methods are hook point methods that have an empty body. Language engineers can implement the visitor interface and realize analyses or transformations that operate on the abstract syntax, such as context con-

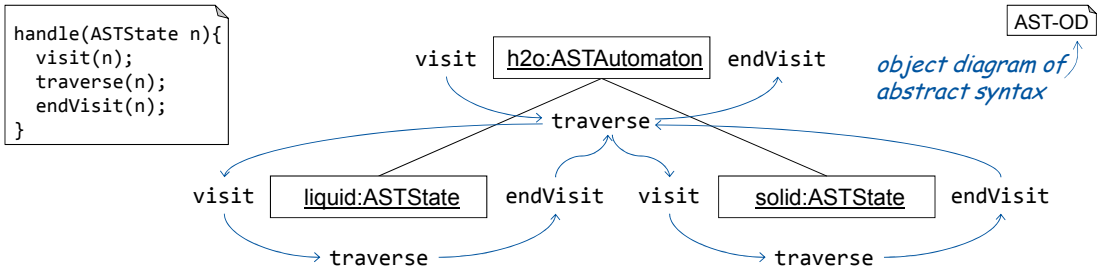


Figure 2.11: Example for the order of traversal of an AST enacted by visitor

ditions or pretty printers. These can provide implementations for the `visit` method, the `endVisit` method, or both. A visitor implementation can operate on a single or multiple AST types of the languages.

The traverser of a language is the entry point for the traversal of the abstract syntax with the visitor infrastructure. It manages a list of visitor interfaces and realizes MontiCore's default traversal of the AST. Upon each occurrence of a particular AST node type in the AST, the corresponding `visit` and `endVisit` methods of visitor implementations are called. To do so, the `visit` and `endVisit` methods of the traverser delegate to the respective `visit` and `endVisit` methods of the visitor implementations.

The traverser of a language consists of the traverser interface, which contains methods with default implementations for the `visit`, `endVisit`, `handle`, and `traverse` methods. Furthermore, it defines the signatures of methods that manipulate the visitors and handlers of a traverser. The actual realization of the attributes must be located in the traverser class because Java does not allow interfaces to have attributes. The traverser is split into an interface and a class to reuse the traverser interface during language composition.

The default traversal strategy of a visitor can be adjusted by implementing the handler interface of a language and overriding the contained `handle` or `traverse` methods. The default traversal strategy is depicted by the example of an excerpt of the automata model `H2O` in Figure 2.11. The left side of the figure displays a simplified form of the implementation of the `handle` method by the sample of the type `ASTState`. The implementation calls the `visit` method first, proceeds to traverse any substructures with the method `traverse`, and finally calls the `endVisit` method. The right side of the figure displays a UML object diagram (OD) [Rum16, www21e] with an excerpt of the AST for the model. The traversal of the model's AST is initiated by invoking the method `h2o.accept(v)`, where `v` is the visitor. First, the `visit` method of the automaton is called, followed by the traversal of its internal structure. The traversal initially encounters the state `liquid`, for which the `visit` method for the type `ASTState` is called. As states do not have any contained structures, the traversal for states does not perform any actions by default. Afterward, the `endVisit` method for the state `liquid` is called,

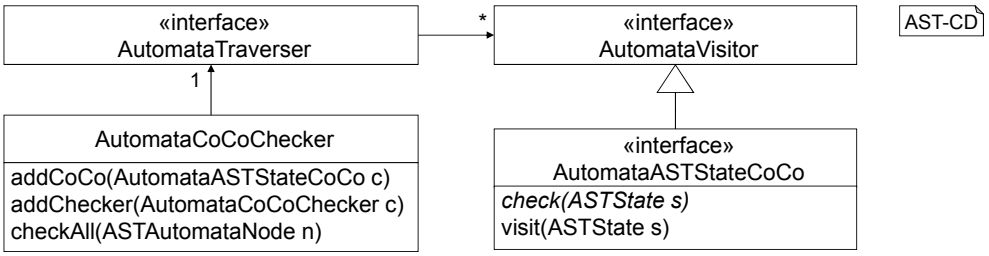


Figure 2.12: Generated language infrastructure for context conditions

and the traversal of the automaton continues with the traversal of further states, such as the state `solid`. After all parts of the automaton have been visited, the `endVisit` method of the automaton is called and the traversal terminates. Since the AST is a tree, the traversal always terminates.

2.2.4 Context Conditions

MontiCore languages can investigate the well-formedness of models with the help of context condition checks implemented as Java classes. To support the engineering of context conditions, MontiCore generates a context condition checker for each language. The context condition checker manages context condition classes and evaluates all of these against the AST by utilizing the language's generated visitor infrastructure. Furthermore, MontiCore generates a context condition interface for each class or interface nonterminal of the language. Context condition classes are realized as handwritten classes that implement this interface. By implementing the interface, a context condition can be checked with the generated context condition checker.

Figure 2.12 depicts the generated classes that are most relevant for the realization of context conditions by the example of the automata language. The generated context condition classes begin with the name of the language's grammar, followed by the AST type of the nonterminal for which the context condition is realized and the suffix `CoCo`. For the nonterminal `State`, the generated context condition class is called `AutomataASTStateCoCo`. Each context condition interface defines the signature of a `check` method with the AST type of the context condition as a method argument. Context condition classes can implement the context condition interface and provide an implementation for this method to realize the context condition check. The context condition interface extends the visitor interface of the language to be able to traverse the AST. It overrides the `visit` method of the corresponding AST type with a default implementation that delegates to the `check` method.

The context condition checker class has the name `AutomataCoCoChecker` and contains `addCoCo` methods for adding context condition classes that implement any of

```java
public class HasInitialState implements AutomataASTAutomatonCoCo {

  public void check(ASTAutomaton n) {
    boolean hasInitialState = n.getStateList().stream()
        .filter(state -> state.isInitial())
        .findAny().isPresent();

    if (!hasInitialState) {
      Log.error("The automaton '" + n.getName() + "' has no initial state!");
    }
  }
}
```

Figure 2.13: Context condition checking that an automaton has at least one initial state

the generated context condition interfaces. Furthermore, the checker has a method `addChecker` for adding all context conditions of a foreign context condition checker at once. The method `checkAll` can be invoked with a passed AST node to check all context conditions added to the checker. To enable checking arbitrary parts of an AST of a model, the method has the language-specific AST node interface as argument. Internally, the checker class uses the language's traverser to manage the context conditions as visitors and enact the traversal of the AST.

Context conditions check well-formedness constraints not reflected by the grammar. The reasons for checking a property via a context condition can be manifold: context-free grammars, in general, cannot express certain forms of well-formedness, such as the uniqueness of names. Language engineers may also decide to check a property with a context condition instead of checking the property through a refined grammar to provide dedicated, detailed error messages if the context condition check fails. Other properties might be captured through the grammar but are instead realized as context condition to simplify the grammar structure or the AST data structure generated from the grammar. In the example of the automata language, initial states and other states are realized by the same nonterminal. Automata may contain an arbitrary number of states. Hence, a context condition has to ensure that at least one state is marked as an initial state. The context condition class `HasInitialState` depicted in Figure 2.13 implements the context condition interface for the `Automaton` nonterminal of the automata language described by the grammar in Figure 2.4. It further implements the `check` method defined in the interface. The implementation first checks whether the list of states of the automaton contains at least one state marked as an initial state and stores the result to a Boolean variable. If the variable's value is $false$, the method logs an error with a helpful error message using MontiCore's `Log` class [HKR21].

2.2.5 Identifying Artifacts in the File System

The symbol management infrastructure of MontiCore [MSN17] is capable of loading models from the content of (model) files with a model loader. To do so, all directories of an application that may contain models have to be indicated in the *model path*. The concept of a model path is similar to the class path in Java [www21b]. A model path contains model path *entries* where each entry identifies a directory that contains models. Alike the package structure of Java classes, models in MontiCore can optionally be located within packages where a package is reified by a corresponding folder structure in the file system. Both the class path entries in Java and the entries of a model path are locations excluding the folder structures that reify the packages. The Java class `ModelPath` in the runtime environment of MontiCore realizes the model path.

Model paths are used for resolving references between model elements that are located in different models. The behavior of loading models is replaced by the results of this thesis with the behavior of loading symbol tables as described in Chapter 5. Despite the name that implies that the path entries identify directories containing models, the technical realization is not limited to model artifacts. Hence, the model path can be reused for identifying path entries that contain stored symbol tables.

2.2.6 Instantiating the Language Infrastructure

MontiCore uses builders [GHJV95] for instantiating the AST nodes. Therefore, the MontiCore generator synthesizes individual builder classes, *e.g.,* for each AST class in addition to the AST class itself. To instantiate the builder classes, MontiCore employs a *language mill*. A language mill is a Java class generated for each MontiCore language. The mill has methods for obtaining all builders of a language, including the builders of the AST nodes. Furthermore, language mills hold instances of several classes of the language infrastructure that are realized as singletons. Language mills are a central place for *reconfiguring* the language infrastructure. To this effect, language mills have methods to exchange the singleton instances and replace each individual builder of a language. Such reconfigurations are utilized for realizing language composition.

2.2.7 Integration of Handwritten Code

Handwritten source code artifacts and artifacts generated by a code generator can be integrated with different mechanisms [GHK+15]. MontiCore employs a unique mechanism for customizing generated artifacts with handwritten ones, which is the *TOP mechanism* [HKR21].

MontiCore distinguishes directories that contain handwritten artifacts from other directories that contain generated artifacts. Although the actual directories are freely configurable, we refer to a directory containing handwritten artifacts as a *source* directory and a directory containing generated artifacts as a *target* directory. Any generated

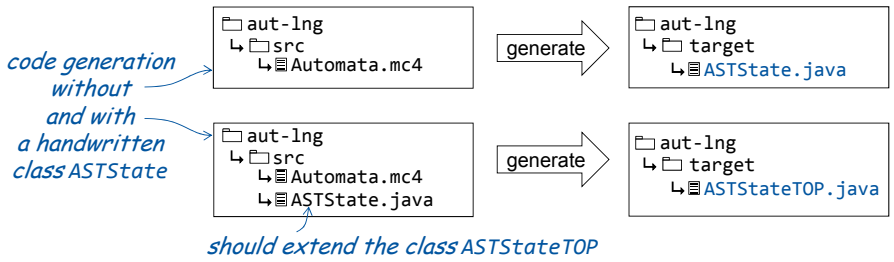


Figure 2.14: The TOP mechanism for integrating handwritten source code artifacts with generated artifacts

class or interface in a target directory can be extended with a handwritten class or interface that customizes the generated type. To utilize the TOP mechanism, a handwritten artifact has to have the same name as the generated artifact that it customizes. Furthermore, the handwritten artifact must be located at the same relative location from a source directory as the generated artifact from a target directory.

If such a handwritten artifact exists, the generator that is parametrized with the location of the source and target directories generates an artifact with the suffix `TOP`. The name of the Java type located in this artifact also is suffixed with `TOP`. Afterward, the Java type defined in the handwritten artifact can extend the type defined in the generated artifact. Through this, any methods may be overridden and redefined.

Figure 2.14 depicts an example of the TOP mechanism. The top left box depicts the artifact of the grammar for the automata language, which is located in a source directory `src`. After executing the code generator, MontiCore produces, among others, an artifact containing the Java class `ASTState`, which is located in the `target` directory. This is depicted in the top right box. The bottom left box depicts the source directory with a handwritten class `ASTState`. If the code generator is executed again, it produces a class `ASTStateTOP`. The handwritten class `ASTState` can now extend the generated class `ASTStateTOP`.

A significant advantage of the TOP mechanism is that generated code, which refers to the artifact that is adjusted with handwritten code does not have to be modified. The name of the initially generated artifact and the artifact with the handwritten adjustments are equal. Hence, the generated code that initially referred to the generated artifact refers to the handwritten artifact after applying the TOP mechanism. The majority of classes and interfaces of the language infrastructure generated by MontiCore are adjustable with the TOP mechanism.

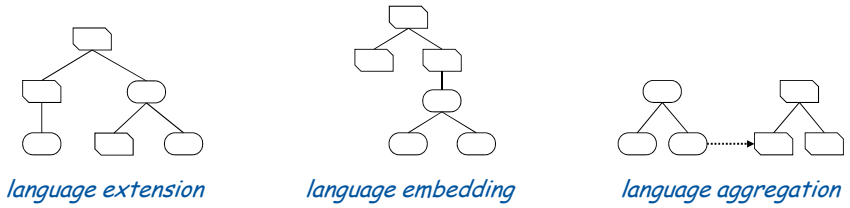


Figure 2.15: ASTs in the different forms of language composition

2.2.8 Language Composition

In software engineering, the decomposition of software into components or modules increases the software reusability. Composing languages [EGR12] is quintessential for enabling efficient reuse of software languages or parts of their infrastructures. With language composition techniques, languages can be made extensible and customizable for various applications [BHH+17].

The techniques to compose languages largely depend on the technological space in which the languages are realized. For instance, textual languages that employ context-free grammars for the definition of syntax require different forms of language composition than languages that employ metamodels for the definition of abstract syntax.

The composition of textual languages should cover both the integration of the language grammars and the integration of the entire language infrastructure such as AST nodes, well-formedness checks, and code generators. This especially includes any handwritten adjustments to generated parts of the language infrastructure. MontiCore supports four different kinds of language composition mechanisms that are presented in the following. The four kinds are *language inheritance*, *language extension*, *language embedding*, and *language aggregation*. All forms except for language aggregation produce a composed language for which the models have an integrated syntax as well. These three forms rely on grammar inheritance in MontiCore and require executing the MontiCore generator to synthesize integrated language tooling. Language aggregation is a loose coupling for which the models remain in individual artifacts. From the language infrastructure, only the symbol tables are composed, so executing the MontiCore generator for language aggregation is unnecessary.

Language Inheritance The concept of language inheritance transfers the concepts of inheritance relationships from object-oriented programming to MontiCore languages. Language inheritance in MontiCore is indicated by inheritance between grammars. A grammar can extend one or more other grammars with the keyword `extends` after the name of the grammar. A MontiCore grammar that inherits from another grammar reuses all its nonterminals and terminals and may, optionally, extend or override some of the nonterminals. A nonterminal is overridden by assigning an inherited nonterminal a

new right-hand side. A nonterminal may extend or implement nonterminals of inherited languages in the same way as nonterminal extension and implementation are realized within a single grammar, which is described above. Multiple inheritance between languages is explicitly allowed and is the basis for different kinds of language composition, such as language embedding.

A grammar can reuse the start nonterminal from an inherited language by explicating the nonterminal with a start rule. The starting nonterminal of an inherited language is not reused implicitly, *i.e.,* without a start rule, the nonterminal defined by the first rule of the inheriting language becomes the start nonterminal.

Beyond the grammar, most other parts of a language are integrated. The AST nodes of inherited languages are integrated with the AST nodes of the inheriting language. For example, overridden nonterminals cause the language infrastructure to instantiate the AST types of the overridden nonterminal instead of the AST types of the original nonterminal. This is achieved by reconfiguration of the mills for inherited languages [HKR21]. The visitor infrastructures of languages reuse the visitor infrastructure of inherited languages through composition via the traverser. If a language inherits from one or more other languages, visitors and handlers for inherited languages can be added to the traverser via dedicated methods. Through the visitor infrastructure, the context condition checker is integrated with inherited languages as well. Any context conditions implemented against context condition interfaces of inherited languages can also be added to the language.

In the following, we sometimes distinguish between the fact that a language *directly* inherits from another language and that a language inherits from other languages *transitively.* If this information is omitted, we refer to both forms of inheritance.

Language Extension Language extension is a particular form of language inheritance in which a language `A` extends another language `B`. In MontiCore, this is indicated through grammar inheritance. Hence, the integration of the remaining parts of language infrastructure is carried out in the same form as described for language inheritance.

Typically, language extension reuses the start nonterminal of the extended language and adds additional alternatives to any nonterminal of the extended language. We say that `B` is a conservative extension [HKR21] of `A` if any model that conforms to `A` also conforms to `B`. The AST of language extension is schematically depicted on the left side of Figure 2.15. The different shapes of the tree nodes refer to the different languages. Hexagonal-shaped nodes refer to the extended language that provides, among other things, the root node of the AST. Round-shaped nodes refer to nodes of the extending language. The nodes of both languages can be interwoven, *i.e.,* a child node of each node can be either a node of the inherited or the inheriting language.

The hierarchical automata language is an example of a language that extends the automata language described by the grammar in Figure 2.4. The hierarchical automata

```
01 grammar HierarchicalAutomata extends Automata {                    MCG
02
03   start Automaton;
04
05   HState extends State = ["initial"]? "state" Name "{" (State | Transition)* "}";
06
07 }
```

Figure 2.16: The hierarchical automata grammar extends the automata grammar

language adds hierarchical states that can define nested states and transitions to the automaton body. The grammar of the hierarchical automata language is depicted in Figure 2.16. The grammar `HierarchicalAutomata` extends the grammar `Automata` (l. 1) and reuses the start rule `Automaton` of the automata language (l. 3). The new nonterminal `HState` describes the syntax of hierarchical states as an alternative to the syntax of states from the automata language. Hence, the grammar rule for `HState` extends the grammar rule `State` (l. 5). By extending this grammar rule, a `HState` may be parsed as an alternative to a `State` in the body of an automaton. The right-hand side of the grammar rule introduces the syntax of hierarchical states that comprise a body containing states and transitions. As the nonterminal `HState` extends `State`, a hierarchical state may contain further hierarchical states.

Language Embedding In contrast to language extension, which reuses a single language, language embedding reuses at least two languages: one language is reused as *host* language, and another language is reused as *embedded* language. It is also allowed that multiple languages are embedded into the same host language. The language embedding describes how the embedded language is embedded into the host language. Typically, the two languages interact at a single spot, which is the *extension point* of the syntax in the host language and the *extension* of the embedded language. We say that the extension realizes or implements the extension point.

In MontiCore, language embedding is realized through multiple inheritance of the involved languages. The composed language extends both the host language and the embedded language and may introduce new syntax for the integration "glue". The multiple inheritance of the languages is described by inheritance in the grammar, where the composed grammar extends both the grammars of the host and the embedded languages. The composed grammar reuses the start nonterminal of the host language. The extension point and the extension of a language are usually integrated with a novel grammar rule in the composed grammar. However, the form of this grammar rule can differ depending on the grammar rule kinds of the extension point and the extension. For example, if the extension point rule is an interface rule with the nonterminal `I` and the extension is a class rule with a nonterminal `E`, the integrating grammar rule can be realized by `C extends E implements I;`. Through this, the new nonterminal `C` is

added as an implementation of the extension point `I`, which is relevant for the parser. Furthermore, `C` reuses the syntax of `E` by extending it. Extending the nonterminal of the extension instead of overriding it yields the advantage that `E` can be used as an extension for different extension points.

The AST of language embedding is schematically depicted in the center of Figure 2.15. The root node of the AST is typically part of the host language. The figure visualizes AST nodes of the host language with hexagonal shapes. A specific node, the extension point, is the only node of the host language that has a child node of the embedded language, which is the "extension" node. All nodes contained in the sub-tree induced by the extension node are nodes of the embedded language. All nodes in the combined AST that are not contained in the sub-tree induced by the extension node are nodes of the host language.

Language Aggregation Language aggregation is a loose coupling between languages whose models remain in separate model artifacts. However, the models may refer to elements of models conforming to aggregated languages via names. To ensure specific properties, such as, type correctness of referenced elements, the language infrastructures of aggregated languages are integrated through their symbol tables. This is described in more detail in the remainder of this thesis, which explains the symbol table infrastructure in Chapter 4 and Chapter 5 and its effect on language composition in Chapter 6.

A central benefit of language aggregation over other forms of language composition is that all involved language tools can be reused entirely, and no new language infrastructure has to be generated. All parts of the language infrastructure that realize the language aggregation can be added to the language tools through (re)configuration of the language infrastructure.

The AST of language aggregation is schematically depicted on the right side of Figure 2.15. The ASTs of two aggregated languages remain individual trees. The only relation between the two ASTs is a symbolic link from the referring model element to the defining model element.

2.3 Software Product Line Engineering

Product lines have been investigated in classical engineering disciplines since the early days of mass production. Henry Ford once stated that "Any customer can have a car painted any colour that he wants so long as it is black" [For22] and by that, proposed to prefer efficiency in the production process over variability [PH04]. Nowadays, cars are offered to customers with a plethora of customization options, bearing grand challenges to manage the variability in engineering and production processes. For example, due to the large number of different bodies, power trains, colors, and optional features, Mercedes offered approximately 10^{24} variations of the E-Class in Europe in 2002 [PH04].

Therefore, today there are "almost no two equal cars rolling from the assembly line the same day" [CE99][1]. Product lines [ABKS13, CN02] help to systematically develop a portfolio or family of similar products that have common parts among them as well as particular options. In times of mass customization of produced goods, it is crucial to manage the product variability to ensure that all configured products are valid in the sense that they fulfill the intended set of requirements. Furthermore, identifying cross-relations between optional features of the variants in the product line is essential. This holds especially for safety-critical systems such as airplanes or cars, as a failure of such systems can lead to significant problems.

Product line engineering is a field that develops means for engineering families of similar products. One of the main drivers behind engineering product lines is reusing parts common to multiple products of the product line. These parts are also referred to as *commonalities*, and they can be effectively reused for all products of a product line. A motivation for this is that the effort to develop and test new parts is usually larger than the effort to reuse these parts. Product line engineering furthermore develops means to manage the variability in the products of the product line, among other things, to gain an overview of all products, to analyze constraints such as those imposed by product requirements, and to be able to perform controlled evolution of product lines.

Software product line engineering (SPLE) [ABKS13, CN02] is a sub-discipline of software engineering that investigates methods and techniques to manage variability in software systematically. Software products are similar to products of classical engineering disciplines but differ from these because their reproduction cost is negligible once the software has been designed. This increments the motivation for reusing software parts across different software variants rather than re-engineering these from scratch.

Software product lines (SPLs) have been successfully applied in different domains such as consumer electronics, avionics, and automotive[2].

2.3.1 Variability in Software and Software Product Lines

Variability is omnipresent in today's software. During the software design phase, variability can be built into the software in terms of explicit *underspecification* to realize open design decisions. Typically, such forms of underspecification are removed from the software in an iterative process via refinement of the software throughout its design phase. Variability further occurs during the evolution of software, where the different versions of a software are its variants. Hence, every new release of a software is a new variation. Variability is built into software products, for instance, to increase the reusability of software parts for similar applications or to address contradictory requirements by different stakeholders. Towards users, variability of a software product can reflect as

[1] In practice, this may be violated, inter alia, due to the production of vehicle fleets.

[2] The Software Product Line Conference (SPLC) has collected several success stories of applying product line techniques: `http://splc.net/fame/`

preference menus in graphical user interfaces of software applications, as parameters in command-line applications, as settings files, or in the form of plug-ins.

Internally, variability in software can be realized in numerous forms [SVGB05] that, among other things, depend on the means of the underlying software languages. The parts of the software that realize the variability are called *variation points*. In C++, variation points can be realized in the form of `#ifdef` directives checked by the preprocessor. A variation point can also be a parameter for an executed program.

The *features* of an SPL are the user-experienceable characteristics that distinguish the products of the SPL. Features come in various shapes and guises. For instance, a software development platform can have a feature that enables project-specific burndown charts, or a model editor can have the feature to export a model to a PDF document.

In the remainder of the thesis, we distinguish different forms of variability:

Open variability enables an unbound number of variants that can be added to a variation point. This is typically the case for extension points of a plug-in infrastructure or customization points that are not aware of all potential extensions/customizations.

Closed variability – in contrast to open variability – is an alternative over a fixed number of known variants. For example, a Boolean flag realizes closed variability as it can be set either to `true` or `false` and, hence, yields two variants.

Positive variability describes that a feature adds software parts or functionality to a product. For example, a merge operator that adds extensions to a base variant realizes positive variability.

Negative variability – in contrast to positive variability – describes that a feature removes software parts or functionality from a product. For example, this can be achieved by applying delta operations that remove parts from a base variant [SBB+10].

With an annotative approach [TAK+14], some SPLs realize all products in a single artifact. This artifact is sometimes referred to as a *150% model*. The 150% model is used as the base variant, and further products can be derived from the SPL by employing negative variability to remove undesired parts. 150% models have certain advantages, such as simplified analyses on the level of product lines. However, product lines typically comprise numerous features, resulting in complex 150% models that are hardly maintainable.

The development process of SPLs distinguishes *domain engineering* from *application engineering* [ABKS13], as depicted by the rows in Figure 2.17. Domain engineering focuses on the reusability aspect of the product line. This includes making artifacts reusable across different variants of the product line and identifying commonalities between variants. Application engineering, on the contrary, focuses on engineering a specific application as a product of the product line. This includes analyzing the requirements of an application and identifying artifacts that can be reused to realize these

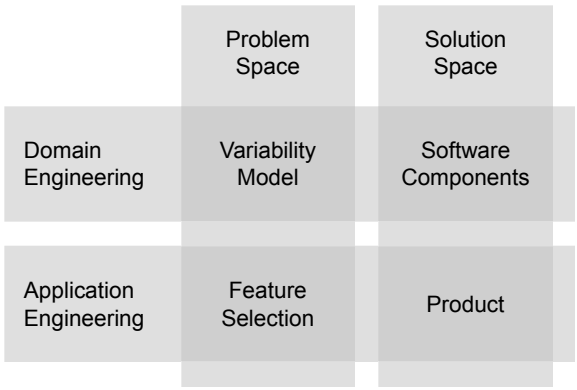


Figure 2.17: Overview of engineering feature-oriented SPLs (inspired by [ABKS13])

requirements. Domain engineering can be referred to as "development for reuse", and application engineering is "development with reuse" [ABKS13].

The development process of SPLs further distinguishes the *problem space* and the *solution space* perspectives, depicted as columns in Figure 2.17. The problem space is a perspective that abstracts from the actual realization of the product line in the software and focuses on requirements, variation points, and customization options of the product line. On the other hand, the solution space perspective includes the realization of the variability in the software. Distinguishing the two perspectives enables a separation of concerns: the problem space models the variability in the SPL without the implementation details of the solution space and, hence, is a helpful representation for the different stakeholders that are involved in the process of planning, conceiving, and maintaining the product line.

In the context of feature-oriented SPLs, the intersections between the two engineering processes and the two perspectives contain specific artifacts. The variability model, which can be a feature model (*cf.* Section 2.3.3), is in the problem space of the domain engineering. The software components that realize the product line are contained in the solution space of the domain engineering. The feature selection manifests, for example, in the form of a feature configuration model, is in the problem space of the application engineering, and the actual software product for the selection is in the solution space of the application engineering.

2.3.2 Software Reuse

Software reuse is "the use of existing software artifacts or knowledge to create new software" [FT96] with the aim of increasing software quality while decreasing the cost and effort of implementing new software [FK05]. We distinguish *black-box reuse* [FT96] of software from reuse via *clone-and-own* [DRB+13].

	Sport	Family	City XL
Body Design	Coupe	SUV	SUV
Fuel Type	Petrol	Hybrid	Electro
All Wheel Drive		X	
Park Assistant		X	X
Traffic Sign Recognition	X	X	X
Lane Change Assistant	X		X

Figure 2.18: Example for car variants in terms of features, depicted as a feature table

Black-box reuse is a form of software reuse that prohibits modifications of the original software for its reuse. This form of reuse is fostered by component-based software engineering [NR68], where a component can be reused "off-the-shelf" without modifications. The term *black-box* indicates that no knowledge about component internals is required for reusing a component. An advantage of black-box reuse is that the reused parts are not cloned and are, thus, typically less prone to co-evolution between clones. Modern build tools such as Maven [MVM10] or Gradle [Mus14], for instance, enable the reuse of software modules by indicating these as dependencies. In this, the reused modules are addressed with dedicated coordinates that include a unique identifier and a version number. Indicating a dependency to the source module of a reused artifact avoids cloning it into a new application. This form of reuse is black-box reuse because it does not allow modification of the reused artifacts.

In contrast, reuse via clone-and-own typically relies on copying a piece of software and modifying it according to its new application. While this form of reuse is more straightforward in terms of ad-hoc reuse [FT96] of individual artifacts, its appliance for systematic reuse of larger pieces of software within software product lines is often discouraged [DRB+13]. Whenever we use the term *reuse* in the remainder of this thesis, we refer to black-box reuse rather than reuse via clone-and-own.

2.3.3 Feature Diagrams

A variety of notations and techniques for modeling commonalities and variation points of software product lines have been conceived [BSL+13]. Among these, a notation commonly used today are *feature diagrams* [CE00, KCH+90], which model common and variable parts of a software product line in terms of user-experienceable *features*. Beyond feature diagrams, there are other variability modeling languages [BSL+13, CGR+12], such as decision models or the common variability language (CVL). Originally developed in the context of the feature-oriented domain analysis (FODA) [KCH+90], feature diagrams have been adapted and extended in various contexts [CE00]. There is a variety of feature diagram modeling tools, such as DarwinSPL [NES17], Familiar [ACLF13], FeatureIDE [MTS+17], FeaturePlugin [AC04], pure::variants [Beu12], and SPLOT [MBC09].

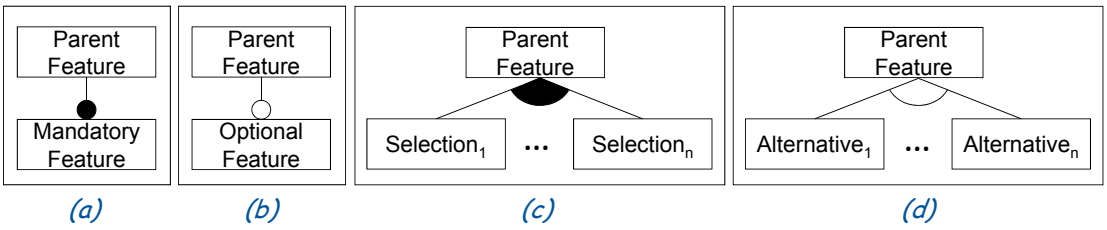


Figure 2.19: Basic elements of the feature diagram notation: (a) mandatory feature (b) optional feature (c) selection group (d) alternative group

In feature diagrams, the products of a product line are separated along their commonalities into features. For example, Figure 2.18 depicts a table with features of a car platform as rows and three variants of the platform as columns. Some features, such as the *Traffic Sign Recognition*, are contained in all three variants, some features, such as the *Park Assistant*, are contained in two variants, and other features, *e.g.,* the *All-Wheel Drive*, are contained in a single variant only. A disadvantage of representing features and variants in a table is that it becomes complex if the number of features or the number of variants increases. For larger product lines, feature diagrams offer a more compact notation.

We refer to models of the feature diagram language as *feature models*. A single feature model describes the features of all products of an entire product line. A *configuration* of a feature model is a set of selected features and a set of unselected features. If a feature model contains at least one feature that is contained in neither of the two sets of a configuration, the configuration is called a *partial configuration*. If each feature of a feature model has been either selected or unselected in a configuration, the configuration is called a *full configuration* or *product* of the product line. A feature model induces constraints on its configurations, rendering a configuration valid or invalid. The semantics of a feature model is defined by all possible configurations that are valid.

Despite the variability in concrete notations and optional language extensions, there are certain "core" feature model notation elements that most feature diagram languages use. We introduce a notation that is mainly based on [CE00]. Each feature of the feature model is characterized by a unique name, and the features of a feature model form a tree. In graphical feature model syntaxes, features are typically represented by boxes containing the feature name. Each feature model has a *root feature* that is part of every valid feature configuration. As features are arranged as a tree, every feature may have several child features, and every child feature has exactly one parent feature. Usually, the meaning of a parent-child relationship in feature models is that the child feature *refines* the parent feature. The children of a feature, also referred to as subfeatures, can be or have to be part of a configuration depending on the kind of relation to their parent feature. It is, however, not possible that a feature is part of a valid configuration that

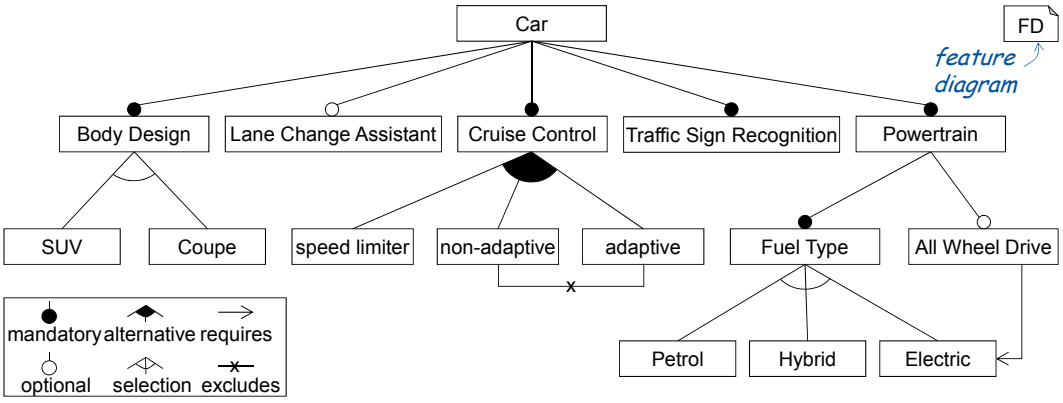


Figure 2.20: Feature model for a product line of cars

does not contain its parent feature. There are two kinds of relations between a single feature and its parent feature: mandatory features and optional features.

Mandatory features are contained in every valid configuration that contains their parent feature. However, if the parent feature is not part of a configuration, its mandatory subfeatures must not be included either. Mandatory features are graphically represented by edges between parent and subfeature with a filled circle at the end towards the subfeature (*cf.* Figure 2.19 (a)).

Optional features (*cf.* Figure 2.19 (b) can be – but are not necessarily – part of a valid configuration if their parent feature is part of this configuration. In graphical notations, optional features are represented by an edge between parent and subfeature with an empty circle at the end towards the subfeature.

Besides optional and mandatory features that connect a single feature with their parent feature, there are relations between a group of sibling features and their (common) parent feature. These are called *feature groups* and are subdivided into *selection groups* (*cf.* Figure 2.19 (c)) and *alternative groups* (*cf.* Figure 2.19 (d)).

Graphical feature diagram notations represent selection groups with a filled arc crossing the lines connecting the parent feature and the subfeatures that are members of the group. Each valid configuration of a feature diagram that includes the parent of a selection group must select at least one and at most all features of the group. Other literature refers to selection groups of feature diagrams as *or-features* [CE00] or *some-out-of-many choices* [ABKS13].

Alternative groups are graphically represented by an empty arc crossing the lines connecting the parent feature and the subfeatures that are part of the group. To form a valid configuration that includes the parent of an alternative group, exactly one feature of the group has to be selected as well. Alternative groups are also referred to as *xor-features* [CE00] or *one-out-of-many choices* [ABKS13].

The above associations between features only constraint features that are in a parent-child or in a sibling relationship to each other. To make assumptions about any features in the tree of a feature model, *cross-tree constraints* can be added to the feature model. Cross-tree constraints are Boolean constraints that involve arbitrary features. A valid configuration of the feature model must satisfy all cross-tree constraints. Beyond the usual Boolean operators (`and`, `or`, `xor`, `not`), feature diagram languages typically allow `requires` and `excludes` constraints. A feature may require the selection of another feature `B`. If a feature `A` requires the feature `B`, every valid configuration that contains `A` must also contain `B`. Otherwise, the configuration is invalid. If a feature `A` and a feature `B` exclude each other, no valid configuration contains both `A` and `B`.

An example of a feature model is depicted in Figure 2.20. The feature model expresses a product line of a car platform that includes, among others, the features and configurations depicted in the feature table of Figure 2.18. While the feature table describes three variants, the feature diagram visualizes 80 variants in an integrated form. Moreover, the feature diagram can group features through the tree representation. For instance, *Fuel Type* and *All-Wheel Drive* are grouped via the feature *Powertrain*.

Feature Diagram Analyses

A significant benefit of modeling a product line in terms of a feature model is the formal grounding of the semantics of feature models and feature configurations. This bears the advantage that different analyses and transformations can be performed on feature models [BSRC10]. A fundamental analysis is the *valid product* analysis that determines whether a given full configuration satisfies the constraints induced by a given feature model, thereby yielding a valid product of the product line. The *valid configuration* analysis indicates whether a partial configuration can be completed to a valid full configuration by selecting further features. Beyond these analyses against a feature diagram and a configuration, different analyses detect inconsistencies, anomalies, and redundancies in a single feature model [vdML04].

Inconsistencies indicate contradictory information in the feature diagram such that typically no valid products are in the semantics of the feature model. For example, a feature model whose semantics is an empty set of valid configurations is called a *void feature model*. Such an inconsistency can be caused by an `excludes` cross-tree constraint between any mandatory feature and the root feature.

Anomalies , similar to inconsistencies, represent contradictory information in the model but prevent only some configurations from being in the semantics of the feature model. Anomalies do not necessarily render errors but are typical sources of behavior that is modeled unintentionally. An anomaly can be caused by a mandatory feature requiring an optional sibling feature. The latter is referred to as a *false-optional feature* because the feature model's semantics would be the same if the feature were mandatory.

Redundancies typically do not influence the semantics of a feature diagram but indicate spots that can lead to inconsistencies in evolved feature models. A redundancy, for instance, is caused by an `excludes` cross-tree constraint between two features in an alternative group. Removing the excludes relationship does not influence the semantics of the feature diagram.

Variability Resolution Mechanisms

Depending on the relation between features and their realizations, there are different forms for resolving the variability in a product line. In the remainder of the thesis, we also call this *deriving a product* or *deriving a variant* from the SPL. If the feature realizations are modules, a product can be derived from the SPL by composing the modules of selected features with a suitable composition operator. If a feature is realized via models and the SPL is aware of the languages to which the models conform, the composition can be carried out by language-specific merge operators that consider the model structure. Such merge operators are also referred to as superimposition [AKL09]. For example, if each feature is realized as a class diagram model, a dedicated merge operator for class diagrams can merge the members of classes with the same name regardless of their order within the class diagram models. As an alternative to language-specific merge operators, general-purpose merge operators, for instance, with line-based differencing can be employed to derive products from the SPL. If a feature is realized through switches in the source code (*e.g.,* with `#ifdef` directives), the feature selection activates or deactivates all associated switches. Depending on the realization of the variant derivation, deactivated switches are either contained in the product but remain unused or are removed from the product. If a feature is realized through delta operations instead, the delta operations are applied to the base product during derivation of the variant [SBB+10].

A variant can be derived from the product line by selecting all features of a configuration at once or in a process called *staged configuration* [CHE05]. In a staged configuration, sets of features can be selected in a stepwise process. The result of each step is a feature model in which parts of the variability have been resolved.

Chapter 3

Method for the Systematic Composition of Language Components in MontiCore

This chapter gives an overview of the method for the systematic composition of language components in MontiCore. The method comprises individual development steps that support language engineering in the large. Although some of these steps build upon other steps, it is not necessary to use all steps together in the proposed overall method that this chapter describes. Instead, the steps can also be applied independently. For instance, the language components can be used without the purpose of engineering language product lines, and the kind-typed symbol tables can be used without using language component models.

The method for the systematic composition of language components relies on reusing language components in product lines. We refer to this as the *language component product line* (LCPL) approach. The central goal of LCPL is to produce tailored DSMLs for different applications or application domains. In this sense, a product line is a language family where each variant of the product line is a language targeted to an application or application domain.

The LCPL approach for engineering language product lines is feature-oriented. Therefore the variability of the product line is represented as a feature diagram (*cf.* Section 2.3.3). At the base, LCPL uses a customized form of textual feature diagrams that contains dedicated rules connecting the features to their realization in terms of language components. The language for this form of feature diagrams is realized on top of the extensible feature diagram language described in Chapter 8 of this thesis.

Languages can generally be derived from a language product line either by removing parts of a 150% language [WHT+09] or by composing languages from language modules [VCPC13]. 150% languages enable a compact notation of product lines that are not scattered across numerous artifacts. However, they have the disadvantage that the language definitions, which are complex for individual languages already, become even more complex for 150% language constituents. This reduces the maintainability of the language as well as the potential for extending and evolving the language product line. Therefore, LCPL relies on the composition of language components to derive products from the product line.

Language components are the consequence of applying component-based software engineering to software languages. Bundling languages into components is a prerequisite for reusing languages in the large, such as in the LCPL. Only if a software language can be identified as a unit of reuse, it can be composed with other languages without in-depth knowledge of the language components' internals. Our notion of language components is based on the software artifacts that realize the language infrastructure. Language components in MontiCore are described with MontiCore language component (MLC) models that identify all artifacts of the language's infrastructure via artifact-based tooling. The notion of language components in general and the MontiCore language component language and tooling are described in more detail in Chapter 7.

Each feature model in LCPL represents a set of *language features*. In analogy to software features in general, a language feature is a user-experienceable part of the DSML. Thus, a language feature covers syntax, semantics, and other language infrastructure of this user-experienceable part. A language feature can realize, for example, hierarchical states of an automata language, generic types in a class diagram language, or SI unit data types in an action language. This is in contrast to some other approaches for language product lines [MAGD+16] in which the variability dimension of a language feature is crosscutting to the dimension of a language feature in LCPL. In such approaches, a language feature describes, *e.g.,* a specific concrete syntax or a transformation for a language instead of a language component. While such notions of language features support realizing presentational [CGR09] or semantic [MRR11] variability, they complicate representing variability in terms of the language itself as should be addressed by LCPL.

In LCPL, the feature tree and cross-tree constraints restrict which selection of language features form a valid feature configuration and which feature constellations are forbidden in the product line. As usual in feature modeling, the parent-child relationship between features should express that the child feature refines the parent feature. For instance, a feature realizing expressions of a language could have a subfeature realizing assignment expressions. The realization of this and the coherence with the refinement relation is not mandatory but lies in the responsibility of the language engineers. The refinement relation can be realized via conservative extension [BEK+19] of the respective language components.

The feature models in LCPL reify the variability in the problem space. However, a feature model alone does not describe how a feature is realized in the solution space and how the individual feature realizations relate to each other. A language feature in LCPL is realized through a MontiCore language component as described in Chapter 7. *Language component mapping rules* in the customized feature diagram notation of LCPL map a feature of the feature model to a language component. If a feature is not mapped to a language component, it is an abstract feature [TKES11], and its only purpose is to group a set of related features as its child features. Leaf features of the tree are not allowed to be abstract. Multiple mapping rules of a product line can point to the same language component. For example, a feature for preconditions of a transition can map

to an expression language, and another feature for postconditions is allowed to map to the same language component. Through this mapping, a language component can be used multiple times within a single language product line but for different purposes. Conceptually, the language component is the realization of a particular (part of a) language such as, syntax and evaluation for expressions, while the language feature is the *purpose* of a language component in the context of the language product line, such as the precondition of a transition of an automata language.

The set of language features of a feature model, thus, maps to a set of language components. However, through the set of language components, it is not specified how the languages are composed, *i.e.,* how the concrete and the abstract syntax of the languages are integrated. As language components can use other language components, parts of the syntax may be already integrated in a single language feature. This form of composition is *implicit* in the product line. However, to foster the reusability of language components without modifying these, our concept enables implementing integration "glue" for the language by modeling how languages are composed as an *explicit* part of a language product line.

A prerequisite for successful language composition in the large is to reuse languages reliably. Language reuse can be simplified if composed languages are only loosely coupled to each other in terms of their language infrastructures. Especially, it should be avoided that any parts of language infrastructure from reused languages have to be re-compiled or even re-generated. The language composition mechanisms in MontiCore (*cf.* Section 2.2.8) generally avoid this. Chapter 6 of this thesis presents language composition mechanisms via symbol tables requiring little or no knowledge about foreign languages, for instance, via symbol adapters or exchanging stored symbol table files.

Beyond this, loading and storing symbol tables of models serves several other purposes. Stored symbol tables can be exchanged between developers or languages to communicate the central information contained in models without the need to communicate the entire model. This improves the performance of loading models, *e.g.,* in the context of implementing type checks. Furthermore, stored symbol tables can be exchanged between language tools during language composition to decouple the language infrastructures from another completely. The purposes of persisting symbol tables are explained in more detail in Chapter 5 together with the concept and implementation of loading and storing symbol tables in MontiCore.

Binding rules (in contrast to mapping rules) in LCPL relate an extension point of a language component to an extension of another language component. There are different kinds of binding rules, extension points, and extensions for different forms of language composition and the different constituents of a language component. For example, for realizing language embedding between grammars, both extension points and extensions must be nonterminals. For realizing language aggregation, extension points and extensions are usually symbols kinds. Realizing language embedding in the abstract syntax requires integrating AST classes of the individual languages to a joint AST data struc-

ture. For the symbol table infrastructure, the scope of the composed language must be able to resolve symbols of both individual languages. This is explained in greater detail in Chapter 6. MontiCore generates large parts of the language infrastructure from the grammar and, therefore, the language embedding for many language infrastructure constituents can follow the embedding of the grammars. To this end, it suffices for realizing binding rules to indicate language embedding by stating how the grammars are integrated. For generated parts of the language infrastructure (such as AST classes, symbol table classes, and visitors), a code generator can produce the infrastructure required for integrating the languages.

The aggregation of languages via symbol tables is supported by kind-typed symbol tables presented in Chapter 4 of this thesis. A *symbol kind* distinguishes different kinds of symbols. With symbol kinds, the name definitions of different language elements can be distinguished. For instance, a name that is part of a method definition in Java is associated with a different symbol kind than a name that defines a Java class attribute. Therefore, Java can distinguish method names from attribute names and allow a method and an attribute of a Java class to have the same name.

A symbol kind is a conceptual part of the symbol table. In the kind-typed symbol table infrastructure, each symbol kind is realized as an individual Java class. To this end, the kind-typed symbol table infrastructure can check the compatibility of symbol kinds through the type compatibility of the corresponding Java classes. This fosters language composition, as no custom compatibility checks have to be implemented.

Via the mapping from feature model to language components, the product line determines which combinations of binding rules can be applied to language components. Thus, the feature model ultimately reduces the combinations of language components that are technically realizable with all binding rules and all language components to those that are intended and allowed to be realized. Based on a feature configuration, the LCPL tooling can derive a language from the product line by applying the binding rules to all language components mapped from selected features. The process of deriving a language from a product line as well as the involved roles and tools are explained in more detail in Chapter 9 of this thesis.

Chapter 4

Generating Kind-Typed Symbol Table Infrastructures

In compiler construction [ALSU07], symbol tables are a means to realize the connection between the usage of an identifier, like a variable, to the definition of the identifier. In the engineering of DSMLs, symbol tables can be used not only to look up identifiers but also to bridge the gap between identifiers across different models and across different languages [MSN17, Völ11]. Furthermore, symbol tables can be a suitable data structure for realizing context conditions or code generators. This section motivates why symbol tables are part of language-processing infrastructures in MontiCore and introduces the notions behind central parts and tasks of the symbol table infrastructure. These notions build upon and extend previous work [HKR21, MSN17, Völ11].

In software languages, a common practice is that elements are identifiable via their name. With such identifiers, a model can refer to other model elements by using their names. Therefore, software languages employ names as part of the definition of a model element and for using a model element. In the following, we refer to this as *name definitions* and *name usages*.

The connection between a name usage and a name definition is the basis for checking, *e.g.,* whether a variable used within an expression refers to a variable declaration. Similarly, the type of a variable refers to the definition of this type. This is the foundation for realizing type systems, where name definition and name usage may be located in different artifacts.

Definition 2 (Symbol & Symbol Kind (based on [MSN17]))**.** *A symbol contains all essential information about a model element that is identifiable by a (usually unique) name. Each symbol has a symbol kind that depends on the model element and determines which information a symbol provides.*

Name definitions that occur in a software language usually have the purpose that the name can be used in another part of the same or of another model of the language. We refer to this as *symbol definition* and *symbol usage*. Other sources [MSN17] refer to symbol usage as symbol reference, but we avoid this due to the proliferation of the different meanings of the term "reference". An example of a model element that defines

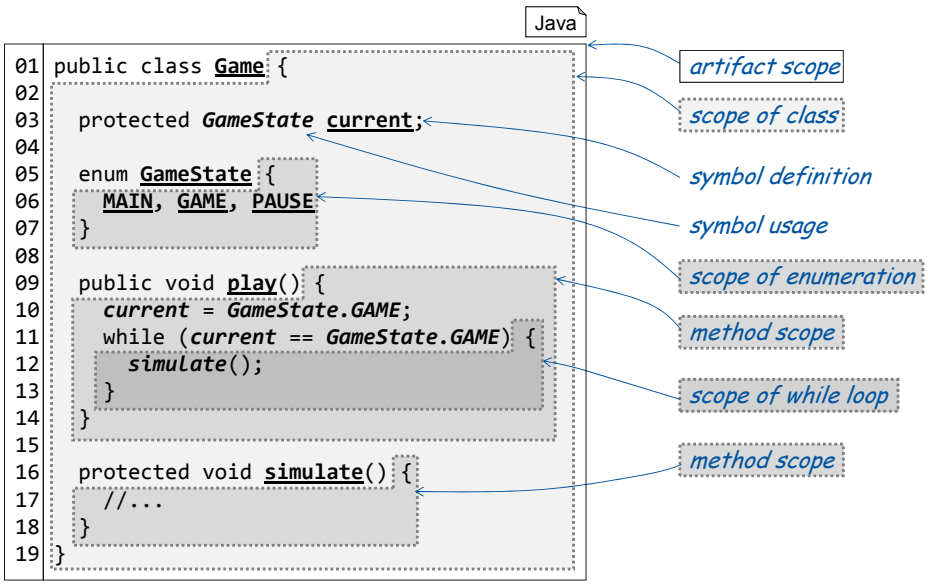


Figure 4.1: Example for scopes (areas with dotted boundaries), symbol definitions (bold and underlined), and symbol usages (bold and italic) in a Java class `Game`

a symbol is a variable declaration `"static int x;"` in a programming language. A variable is a viable candidate for being a symbol since it is identifiable by the variable's name. In this example, the symbol is of the kind `variable`. The kind determines that variables have a name, a type, and a set of modifiers. An expression `"x < 3"` that compares the value of the variable with a constant value is an example that uses the symbol `x` by utilizing the name of the symbol.

While programming languages usually have a limited number of similar symbol kinds, modeling languages have a wide variety of symbol kinds. For example, architecture description languages can have symbol kinds for component types, port definitions, and typed messages [BHH+17], while state machine languages could have symbol kinds for state machines and states [HKR21]. Other modeling languages may define symbol kinds such as activities, features, association roles, or pins accordingly. To this end, a concept for defining and distinguishing individual symbol kinds is much more relevant for modeling languages than for programming languages.

In software languages, a variable definition is typically visible only in certain *scopes*. Scopes [MSN17] are parts of the abstract syntax of a language that support realizing visibility concepts for symbols. Sometimes, scopes are also called blocks [ALSU07, GJR79] or namespaces [Völ11].

Definition 3 (Scope (based on [MSN17]))**.** *A scope is a concept of the syntax that encloses symbols and impacts their visibility.*

In MontiCore, each symbol is contained in a scope. A scope determines the visibility of the symbols it contains and usually corresponds to a subtree of the AST. Each scope can have individual properties that impact the symbol visibility. For instance, scopes can export symbols such that these symbols become visible in other scopes. This is explained in further detail in Section 4.1.3. Scopes can be nested and form a tree-shaped scope graph. For each model artifact, an *artifact scope* in the abstract syntax captures symbols defined in the artifact. The root of this tree shape is always a *global scope* that subsumes all known artifact scopes. In the following, a scope that is the child of another scope in the scope tree is said to be contained in the parent scope or to be the *subscope* of the parent scope. We further refer to the parent scope as the scope's *enclosing scope*.

An example of symbol definitions, symbol usages, and scopes is the Java class `Game` depicted in Figure 4.1. The class is contained in an individual artifact that spans an artifact scope. Therefore, all symbols and scopes defined in this class are (transitively) contained in the artifact scope enclosing the class. The class defines a symbol of the kind `JTypeSymbol` with the name `Game` (l. 1). The symbol spans a scope that, in the concrete syntax of Java, is enclosed by curly brackets. This scope is a subscope of the artifact scope and itself contains symbol definitions of the class members, which in this example are a `JFieldSymbol` for the class attribute `current` (l. 3), two `JMethodSymbols` for the two methods (ll. 9-14 and ll. 16-18), and a `JTypeSymbol` for the contained enumeration `GameState` (l. 5-7). The symbols for inner types and methods span a scope that, again, can contain symbol definitions and subscopes. The scope spanned by the `GameState` symbol contains symbol definitions for the enumeration values (l. 6) and the scope of `play` contains a subscope for the `while` loop body (ll. 11-13).

Classical compiler construction [ALSU07] considers a symbol table as a data structure that has entries (or records) for each definition (*e.g.,* variable definition). In addition to the identifier, an entry contains further attributes of the definition. One of the primary purposes of symbol tables in compilers is to search these for an identifier and find suitable entries. In MontiCore, there is no dedicated concept realizing a symbol table. However, all symbols are contained in scopes and, hence, the scope graph contains all symbols. Furthermore, the scope graph can be traversed to search for suitable symbols of a given kind for a given name. This process is referred to as *symbol resolution*, and it is explained in Section 4.1.8. In the following, we denote the conceptual elements for scopes and symbols as well as their management in terms of instantiation, traversal, modification, and persistence as the *constituents of the symbol table infrastructure*. We call the implementation of these constituents the *symbol table infrastructure*.

The proper analysis of a set of interrelated models requires the models to be integrated [DCB+15]. Symbol tables are data structures that enable integrating different models of the same language. The symbols of each model are contained in a separate artifact scope, and all known artifact scopes are contained in the global scope. Furthermore, symbol tables are the interface of a language for realizing a composition of the language with other languages. A global scope can be aware of artifact scopes of different

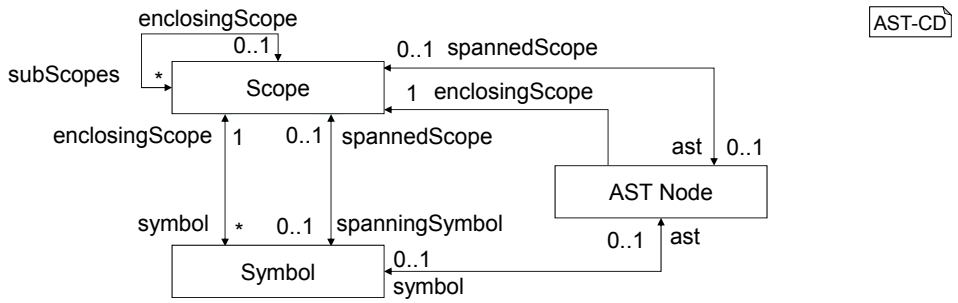


Figure 4.2: Associations between symbols, scopes, and AST nodes in general

languages, which is the basis for language aggregation. Language extension, inheritance, and embedding result in integrated artifacts and, hence, these forms of language composition require integrated artifact scopes. The effects of language composition on symbol table infrastructures are explained in more detail in Chapter 6.

This chapter presents the typed symbol table infrastructure (STI) for MontiCore, in which large parts of the infrastructure can be generated from a language's grammar. In this infrastructure, the Java classes that realize symbol table constituents such as symbols and scopes have language-specific types rather than generic ones for all languages. This supports language composition because the composition of the symbol table infrastructures can reuse the type compatibility checks for the infrastructure constituents from the host language of the language workbench, *i.e.,* Java.

In the following, Section 4.1 presents the concepts behind the STI. Section 4.2 explains concepts for controlling the generation of symbol table infrastructure through annotation of MontiCore grammars with symbol table information. Section 4.3 describes central aspects of the implementation of the symbol table infrastructure and, through examples, how the symbol table of a language can be customized to meet the language engineer's intents. Section 4.4 discusses central design decisions and Section 4.5 compares the STI with related approaches.

4.1 Concept of Kind-Typed Symbol Tables

The STI is a language-specific symbol table infrastructure. With the application of a code generator, large parts of the infrastructure can be generated from a language's grammar to alleviate language engineers from manually implementing the infrastructure for each language individually. In the STI, there is a generated class for each symbol kind of a language and several generated classes and interfaces for each scope. Together with supporting classes and interfaces, such as for the instantiation of the symbol table, these classes are generated based on information in a language's grammar. This section explains the concepts behind the constituents of the STI.

4.1.1 Relationships between Symbols, Scopes, and AST Nodes

The parts of the infrastructure that realize symbols and scopes are associated with each other through various interrelations. Furthermore, as the symbol table infrastructure is part of a language's abstract syntax data structure, symbols and scopes are associated with the nodes of the AST. An overview of the relationships between these three kinds of abstract syntax concepts is depicted in a language-agnostic form in Figure 4.2. Each scope contains a set of symbols that are defined in the scope. To improve the navigability of symbol tables, the association is bidirectional and each symbol has an enclosing scope. It is not allowed that a symbol has no enclosing scope or is directly contained in more than one scope.

Scopes are usually arranged in a scope graph. However, we follow a common restriction to this and consider trees of scopes only. To this effect, each scope but the scope that forms the root of the scope tree has an enclosing scope. In the reverse direction, each scope has a set of subscopes that may be empty.

Some nonterminals define symbols. The AST nodes of such nonterminals have an association with the corresponding symbols. If a nonterminal defines a symbol, the association from the AST node to the symbol has a cardinality of 1. In the reverse direction, the association from symbol to AST node has a cardinality of 0..1. This is because the AST of a symbol is only available if the symbol table has been created from an AST. After loading a stored symbol (*cf.* Chapter 5), the AST is not available. If a nonterminal does not define a symbol, the AST node for this nonterminal does not have such an association.

Each AST node is associated with an enclosing scope, regardless of whether the nonterminal defines a symbol or spans a scope. This association is unidirectional because there is currently no use case in which the reverse direction is relevant. For nonterminals that span a scope, the scope is associated with the AST node of the nonterminal. As with nonterminals that define symbols, the AST node of a nonterminal that spans a scope is associated from the scope with the cardinality 0..1 to take into account that scopes can be loaded from stored symbol tables. In the reverse direction, the AST node of a nonterminal that spans a scope is always associated with the spanned scope.

Some scopes are spanned by a nonterminal that also defines a symbol. In this case, there is a relationship between the spanned scope and the defined symbol. A symbol that spans a scope is always associated with the scope (*i.e.,* has a cardinality of 1). In general, however, not all symbols span a scope and, thus, the association's cardinality is 0..1. As not all scope instances are spanned by a symbol, the cardinality in the reverse direction is 0..1 as well.

Section 4.2 explains language-specific examples for the relationships between AST nodes, symbols, and scopes as well as the generated classes and interfaces that realize these in the context of modeling symbol table information within MontiCore grammars.

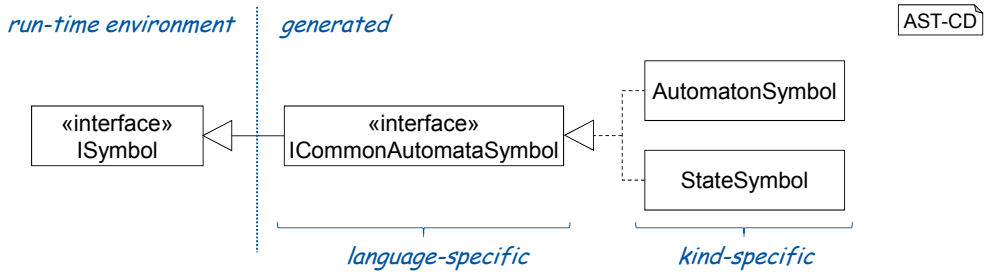


Figure 4.3: Interfaces and classes for symbols

4.1.2 Defining Names via Symbols

According to Definition 2, each symbol must be identifiable by a name. Consequently, there are no anonymous symbols. In MontiCore, both name definitions and name usages are represented by the nonterminal `Name`. To this end, not every occurrence of the nonterminal `Name` is a name definition. Furthermore, not every name definition is required to define a symbol. Only name definitions to which other model elements may refer via their name should define symbols. This can be indicated by annotating the nonterminals of a grammar with the keyword `symbol` as described in Section 4.2. In the following, we use the terms *symbol-defining model element* and *symbol-defining nonterminal* interchangeably. *Symbol kinds* associate additional information with a symbol-defining nonterminal beyond the symbol name. Symbols in MontiCore further have an access modifier [MSN17] that indicates which access rights other symbols require to be able to access the symbol.

The classes and interfaces for symbols in the STI are depicted in Figure 4.3 by the example of the automata language. In the STI, method signatures of common methods are contained in an interface `ISymbol` that is part of the MontiCore runtime environment. For each language, MontiCore generates a language-specific interface that declares methods signatures common to all symbols of the language, such as methods that use language-specific types for parameters or return types. The language-specific symbol interface, in this example the interface `ICommonAutomataSymbol`, extends the interface `ISymbol`. For each symbol-defining nonterminal, MontiCore generates a symbol class representing the symbol kind. For each AST object of the symbol-defining nonterminal in a parsed model, a symbol object of the corresponding symbol class is created during the instantiation of the symbol table. All symbol classes implement the language-specific symbol interface. The separation between symbol class and interface enables better support for traversal with visitors.

For example, the `Automata` language defines two kinds of model elements that are identifiable through names: `Automaton` and `State`. Transitions, on the other hand, are not directly identifiable via explicit names. It is useful to distinguish the names

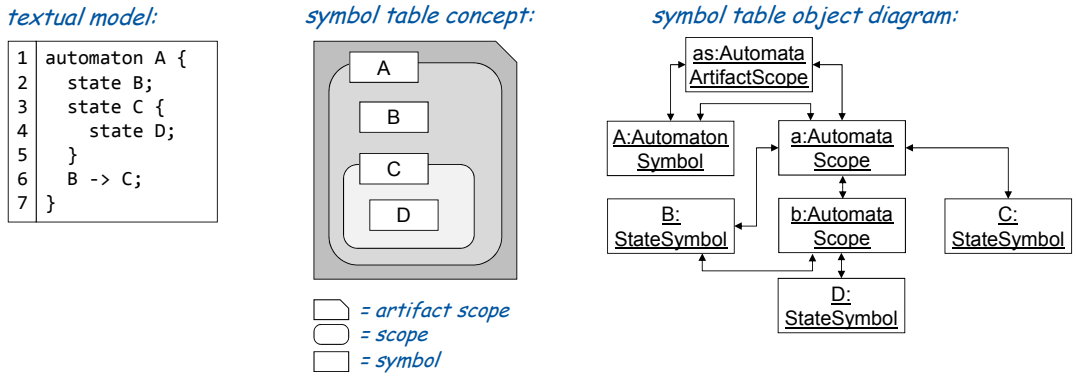


Figure 4.4: Interfaces and classes for scopes

that define automata from the names that define states to enable modelers to use the same name for a state and for an automaton without causing ambiguities. Therefore, the language defines two symbol kinds `AutomatonSymbol` and `StateSymbol`. While automata symbols have no additional attributes besides their name, state symbols can have a Boolean property indicating whether the state is an initial state. The `H2O` automata model defines an automaton symbol with the name `H2O` and three state symbols `Solid`, `Liquid`, and `Gaseous`.

Besides the simple name that typically identifies a symbol uniquely within a single scope, each symbol has a (derived) qualified name that identifies the symbol uniquely from the global scope. However, the qualified name may be used internally only, and if a symbol is not exported to the global scope, it might not be used at all. Qualified names are conceptually interwoven with scopes and are explained in more detail in Section 4.1.6.

A symbol kind k can optionally extend another symbol kind h. The effect of this is that symbols with the symbol kind k have all the essential information that symbols with the kind h have. Symbols of kind k may provide additional information to the information provided by symbols of kind h but may not omit (*i.e.,* remove or hide) any information of h. A symbol kind may not directly extend more than one other symbol kind, *i.e.,* multiple inheritance on symbol kinds is not allowed. Multiple inheritance on symbol kinds would require conceiving novel concepts for realizing symbol resolution and for handling potential conflicts in inherited symbol information. Section 4.1.8 explains the effects of symbol kind hierarchies for symbol resolution.

4.1.3 Capturing Name Visibility with Scopes

The STI uses scopes to realize the visibility of symbols within a single model but also across different models. The scopes from programming languages such as Java, as depicted in Figure 4.1, exist in a similar form in other software languages as well. This

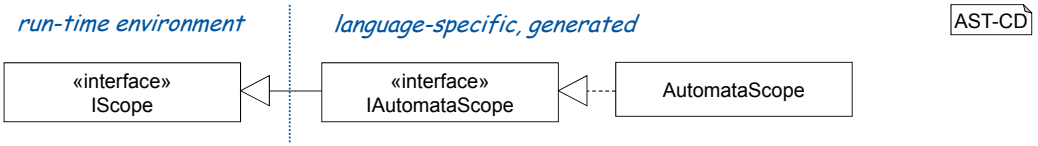


Figure 4.5: Interfaces and classes for scopes

especially includes modeling languages such as automata. An example of the scopes of a model of the automata language is depicted in Figure 4.4. In the concrete syntax of Java-style, textual languages, scopes are typically indicated by curly brackets as depicted on the left side. The body of automaton `A` is enclosed by curly brackets and spans a scope containing the state symbols `B` and `C`. The state symbol `C` spans a scope that contains a state symbol `D`. The center part of Figure 4.4 depicts the concept for scopes and their contained symbols and subscopes. The right part demonstrates an OD that represents the Java objects that constitute the STI of the automata language with the interrelations between symbols and scopes as described in Section 4.1.1.

Similar to the interface `ISymbol` for symbols, the runtime environment of MontiCore contains an interface `IScope` that all scopes implement. This interface defines language-agnostic method signatures. For each language, there is a generated scope interface defining language-specific method signatures, some of which contain default implementations. Figure 4.5 depicts the scope classes and interfaces by the example of the automata language. In this example, the scope interface is `IAutomataScope`. The attributes of a scope are managed by a language-specific scope class that implements the scope interface. In this example, the scope class is `AutomataScope`. Scope classes and interfaces are separated to support language composition (*cf.* Chapter 6).

A scope has to manage all local symbols that are defined within the scope. As in the STI, an individual Java type realizes each symbol kind, scopes have to handle each of these Java types individually. Managing all symbols in a single collection would be possible, as all symbols implement the interface `ISymbol`, but would result in a loss of type information. Hence, scopes have individual collections of local symbols per symbol kind. For each symbol kind, the scope class manages a multimap that maps a symbol name to a list of symbols that have this name and are defined in the current scope. Usually, a map with a single symbol entry per name would suffice. However, sometimes languages allow multiple symbols in a scope to have the same name if these are distinguishable with further criteria. For instance, the scope of the body of a Java class allows multiple method symbols with the same name if these are distinguishable by their arguments. The STI does not distinguish different types of scopes of a single language. Some languages, for instance, Java, use different kinds of scopes for methods, classes, and other language elements [Fla05]. In the STI, these different kinds of scopes are realized by the configuration of the language's scope via scope properties.

There are different properties that each scope (object) of a language can assume. The STI provides eight variants of scopes that can be controlled by setting the value of three Boolean properties. Additional scope properties can be realized by applying the TOP mechanism [HKR21] to scope interfaces and classes.

Export of symbols A scope may either export all its local symbols or it may not export these. If a scope exports symbols, it exports symbols of all kinds. More fine-grained control over exporting symbols of specific kinds has to be realized manually. A scope that exports symbols typically makes these available for symbol resolution from other scopes. Scopes that do not export symbols enclose symbols that are not visible from other scopes. For example, the scope of a Java class exports local symbols such as fields and methods to make these visible to other classes and subscopes of the class. The scope of a Java method, on the other hand, does not export any symbols defined within the body of the method.

Order of symbols Some scopes are sensitive for the order in which symbols are defined while other scopes are agnostic of any order. In an ordered scope, symbol resolution can only find symbols whose definition is syntactically located before the usage, *i.e.,* if the source position of the definition appears in the model before the source position of the usage. In Java, for example, the body of methods is an ordered scope. A variable name within a method body may only be used after it has been defined. The body of Java classes, however, is a non-ordered scope. An inner class can be used as an attribute type before the class is defined.

Shadowing of symbols Scopes may optionally shadow symbols that are defined in other scopes. Non-shadowing scopes are also referred to as visibility scopes [MSN17]. If a symbol with a certain kind and name is defined in the local scope, but another scope exports a symbol with the same name and kind, a shadowing scope prefers the local symbol over the foreign symbol(s) during resolution. Any foreign symbols with the same name and kind are "shadowed", *i.e.,* ignored. A non-shadowing scope, on the other hand, yields an ambiguity error in the same situation. By default, each artifact scope in the STI is a shadowing scope. In Java, the body of a method is a shadowing scope. Any variables introduced in the method shadow class attributes with the same name. The body of a for loop, on the other hand, is a visibility scope. Defining a variable within the for loop with the same name as a variable in the enclosing method causes a compilation error.

All scopes have an optional name. By default, the name is either empty if a scope is not spanned by a symbol or if it is derived from the name of a symbol that spans a scope. However, scopes can be customized to set the name of a scope differently. This is relevant in combination with symbol resolution, as explained in Section 4.1.8.

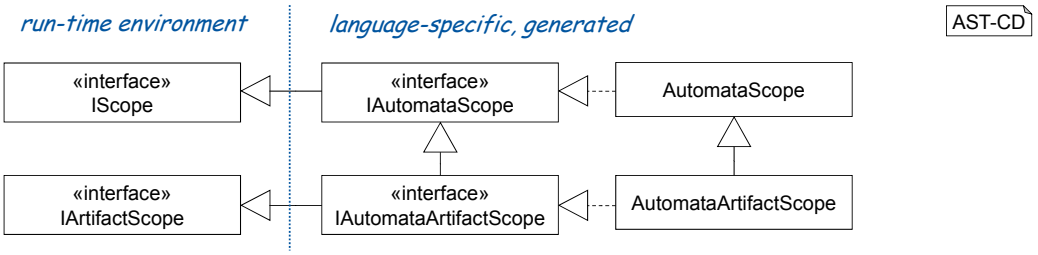


Figure 4.6: Interfaces and classes for artifact scopes

4.1.4 Providing Access to a Model's Symbol Table with Artifact Scopes

Artifact scopes are special scopes that enclose the symbols and scopes of a model artifact. Typically, the artifact scope has one or more symbols that are directly contained in the artifact scope. We refer to such symbols as *top-level symbols*. Top-level symbols can span scopes that contain further symbols. In Figure 4.4, the artifact scope for the textual model of the automaton contains a single top-level symbol `A` that spans a scope, which contains other symbols and subscopes.

Artifact scopes have all attributes of other scopes but have an additional package name. Java-style languages organize artifacts in folder structures and indicate these folder structures within the artifacts as qualified package names. Although, obviously, not all languages use packages, and the concrete syntax of package declarations may differ in different languages, the STI has a built-in mechanism to handle Java-style packages that can optionally be used for symbol resolution. If a language does not use packages, the package attribute in artifact scopes of this language can be left empty. The scope name, which is an optional attribute of other scopes, is by default always present in artifact scopes and should be equal to the name of the model. With the model's name being present in the symbol table, symbol resolution can rely on the model names without requiring that the model file is known. This fosters symbol resolution in combination with loading persisted symbol tables.

As depicted in Figure 4.6, the MontiCore runtime environment contains an interface `IArtifactScope` that all artifact scopes implement. For each language, MontiCore generates a language-specific artifact scope interface and an artifact scope class. Similar to other scopes, the artifact scope class implements the language-specific artifact scope interface, and the latter implements the runtime environment interface. As artifact scopes are special kinds of scopes, the language-specific artifact scope interfaces extend the language-specific scope interfaces, and the same holds for the (artifact) scope classes. The interface `IArtifactScope` does not extend the interface `IScope` to avoid introducing unnecessary diamond inheritance.

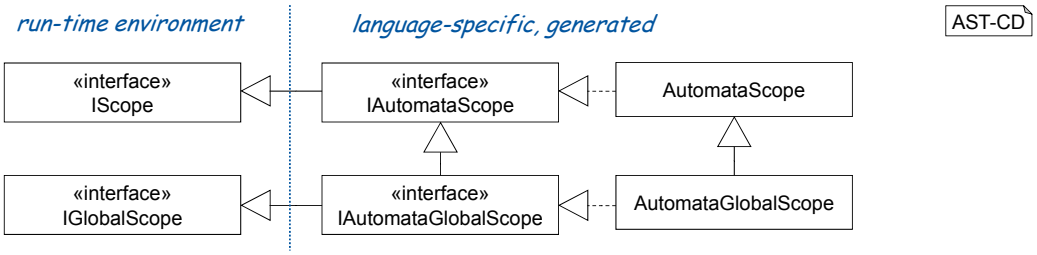


Figure 4.7: Interfaces and classes for global scopes

4.1.5 Bridging the Gap Between Models with Global Scopes

Global scopes are the root of each scope tree in MontiCore and as such, have artifact scopes as their direct subscopes. With global scopes, the symbol tables of different models are integrated into a common data structure. Global scopes are realized as singletons [GHJV95], and the singleton instance can be obtained from a language mill. Mills are relevant for language composition and are explained in more detail in Section 4.3.1. As singletons, global scopes are central for the (re)configuration of a language's symbol table. Hence, global scopes manage different configurable attributes as well as accessor and mutator methods for these attributes. The attributes of global scopes are utilized for different purposes:

Inter-model symbol resolution. Inter-model symbol resolution (*cf.* Section 4.1.8) is realized in global scopes and, hence, global scopes realize the class attributes required for this. Besides the symbols that global scopes could contain as any other scope, the inter-model resolution requires a map with names of artifact scopes that are already loaded to avoid loading these multiple times.

Symbol table persistence. Loading of symbol tables as presented in Chapter 5 is realized in the inter-model symbol resolution that is part of the global scope. For loading and storing symbol tables, symbol DeSers (de)serialize symbols of a specific kind, scope DeSers (de)serialize scopes, and a `Symbols2Json` class traverses the symbol table for its serialization. All of these are explained in more detail in Chapter 5. Global scopes have an attribute of the `Symbols2Json` class (*cf.* Section 5.4.4) for configuring the loading and storing of symbol tables. Furthermore, they manage a map of symbol DeSers (*cf.* Section 5.4.5) that can be reconfigured by adding or modifying entries. The global scope manages the DeSer of the language's scope (*cf.* Section 5.4.6) to use it during the deserialization of symbol tables. For efficient loading of symbol table files, the set of files considered for loading is constrained by a regular expression indicating potential file extensions. The model path of a global scope holds entries that are folders of archive files containing symbol table files.

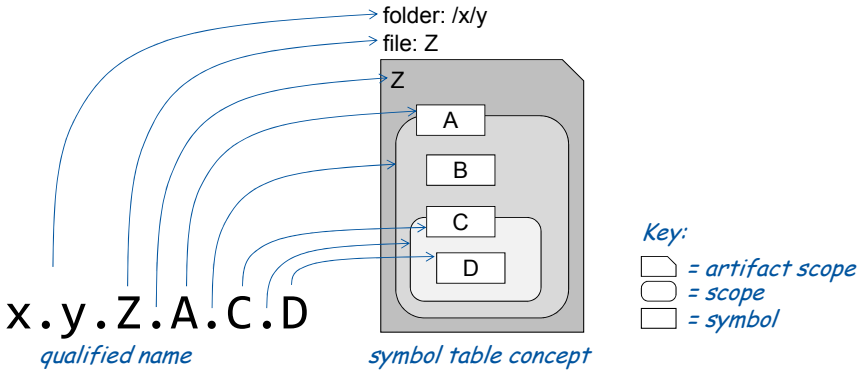


Figure 4.8: Interfaces and classes for global scopes

Language aggregation. Aggregation of languages (*cf.* Section 6.4) is realized in the inter-model symbol resolution that is part of the global scope. Therefore, global scopes manage sets of resolver interfaces for each symbol kind. By (re)configuring these, the inter-model resolution searches for symbols of known or unknown symbol kinds in global scopes of foreign languages.

As depicted in Figure 4.7, the MontiCore runtime environment contains an interface `IGlobalScope` that is extended by generated, language-specific global scope interfaces. Language-specific global scope classes implement these interfaces. Global scopes are special kinds of scopes, and hence, the language-specific global scope interfaces extend the language-specific scope interfaces. The same holds for the (global) scope classes. The interface `IGlobalScope` does not extend the interface `IScope` to avoid introducing unnecessary diamond inheritance. The types of global scopes and artifact scopes are not in a mutual relationship.

4.1.6 Using Model Elements through Names

A fundamental purpose of symbol tables is to find a suitable symbol definition or even the definition of the corresponding named model element from a given name usage. A name usage refers to a name definition that is part of a symbol definition located in the same model or in a different model. A name usage, hence, is itself a name. Names, in MontiCore, are sequences of numbers, letters, and separating characters, such as underscores. Typically, a name must not begin with a number and must not contain any whitespace characters.

A name can be either a *qualified name* or an unqualified name, which we refer to as *simple name*. A simple name is a name that does not contain any dots. A qualified name comprises dot-separated parts, each of which is a simple name. Qualified names create a hierarchical namespace that enables identifying names across different scopes or

even globally unique. Thus, a language tool has to be able to address each symbol that is exported beyond the artifact scope by a qualified name. A name does not necessarily have to be stated as a qualified name in the model but instead can be qualified, *e.g.,* via import statements.

The beginning of a qualified name identifies the artifact scope by a simple name or a qualified name that begins with the package of the model. Each consecutive name part of a qualified name identifies a symbol. If a dot follows a name part, the name part is expected to identify a symbol of a symbol kind that spans a scope. The dot identifies the spanned scope, and the consecutive name part must identify a symbol in this scope.

An example of qualified names that identify symbols globally is depicted in Figure 4.8. In Java-style languages, the folder structure of a model file corresponds with the package of the model. In this example, the package `x.y` corresponds with the folders `/x/y` that are relative to a starting root directory. In Java and MontiCore, such a root directory is typically a class path entry. The model file in this example is `Z` and, hence, the qualified name `x.y.Z` identifies the model artifact. To access the artifact scope of the model, the qualified name has an appended dot. A simple name that follows this dot must identify a symbol in the artifact scope. In this example, the qualified name has the simple name `A` at the described position and, hence, identifies the symbol `A` in the artifact scope. A dot that follows the symbol name in a qualified name identifies the scope spanned by the symbol. In this example, the qualified name part `x.y.Z.A.` identifies the scope spanned by the symbol `A`. The next name part in the qualified name is `C`, which equals the name of the symbol `C` in the expected scope. The same holds for the consecutive dot, and the name part `D` that equals the symbol name `D` in the scope spanned by the symbol `C`.

The simple name of each symbol can be *qualified*. The qualifier is calculated by setting the name parts from the symbols that span the enclosing scopes iteratively. For example, in the automata language, a state with the name `liquid` can be qualified for global identification via the name of the automata `H20` that contains the state symbol. The resulting qualified name is `H20.liquid`. In a language with hierarchical states, the state `liquid` could span a scope that contains a state symbol `inMotion`. The qualified name of this state would be `H20.liquid.inMotion`. A qualified name can be *unqualified* by separating it into the qualifier and the simple name at the last occurrence of a dot. The unqualified/simple name of the state `H20.liquid.inMotion` would be the name `inMotion`.

There are different patterns for implementing symbol usages [MSN17]. A symbol usage can be realized (1) via *delegation* from the usage to the definition, (2) with the *proxy pattern* in which the usage can be replaced with the definition, or (3) in the same class as the symbol definition. All three patterns have individual advantages and disadvantages. Similar to the SMI [MSN17], the STI applies the pattern with a delegation from the symbol usage to the symbol definition. However, contrary to the SMI, the STI does not rely on explicit `SymbolReference` classes. Instead, the delegation is realized

from the AST class of the usage to the symbol definition, thus reducing the number of involved classes and their objects. Our experiences have shown that languages do not have information specific to the symbol usage, rendering any specifically generated classes useless. If a language requires usage-specific information, it is often connected to the type system of a language. For realizing symbol usages of symbols that are part of a type system, MontiCore has a library of `SymTypeExpressions` [HKR21] as part of its built-in type system framework [BEH+20]. `SymTypeExpressions` are handwritten Java classes that represent usages of type symbols. Type symbols and their usages are explained in more detail in Section 4.1.7. We recommend two options for language engineers who intend to represent symbol-usage-specific information in a language that does not use MontiCore's built-in type system framework:

If the symbol-usage-specific information is visible only in the model in which it is contained, the AST of the model is always present. Hence, the AST of the symbol usage can be adjusted with the TOP mechanism to represent the additional symbol-usage-specific information. An example of this scenario lies in the automata language, in which transitions refer to the symbols of source and target states. If any information about the source and target states should be kept in the transitions, it is viable to realize such information in the AST class of the transition instead of the symbol table. This option can be applied especially if the symbol-usage-specific information is not directly related to the definition of another symbol.

If the symbol-usage-specific information must be visible for other models, the symbol table infrastructure of the language should not rely on the AST to be present as the AST of foreign models may not be available. Instead, the symbol table has to contain the symbol-usage-specific information. This is usually the case in type systems of object-oriented languages, where a type definition has usage-specific information about supertypes. To realize this, the information should be extracted to a new class, which is set as an attribute of the symbol that manages the usage-specific information. The `SymTypeExpression` classes can serve as a guideline for this option.

4.1.7 Type Definitions and Type Expressions

To support engineering type systems [ALSU07] for DSMLs created with MontiCore, the language component library [BEH+20, HKR21] contains five language components described below that introduce abstract and concrete syntax for types. This includes symbol kinds for type definitions in object-oriented languages. *Type symbols* are not to be confused with *typed symbol classes* in the kind-typed symbol table infrastructure. The former are symbols with a specific kind, such as `OOTypeSymbols` that contain information about types, while the latter refer to the types of the Java classes that realize symbol kinds in general.

This section explains the fundamentals of the support for engineering type systems with the MontiCore language component library. New MontiCore languages can option-

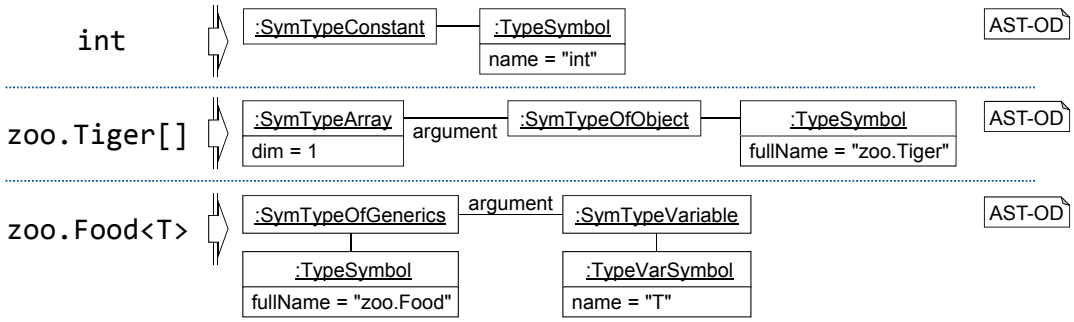


Figure 4.9: Object structures of exemplary `SymTypeExpressions` for a type constant, an array, and a generic type.

ally reuse (and extend) this type system or engineer an entirely new type system with the means of the STI. We say that type symbols are defined within *type definitions*, and type symbols are used within *type expressions* [ALSU07]. In this, a type expression is a piece of syntax that reflects in a model as a type. A type expression in an object-oriented language can be a type declaration including the statement of an associated supertype or the syntactic part of a variable declaration that declares its type. The language component library contains five language components that introduce syntax for type expressions:

MCBasicTypes introduces the central interface nonterminal `MCType` that all other nonterminals for type expressions implement. Moreover, it contains the syntax for common built-in types of programming languages (such as, `int`) and qualified names (*e.g.,* `example.Person`) as type expressions.

MCArrayTypes extends the `MCBasicTypes` and provides the syntax for array types.

MCCollectionTypes extends the `MCBasicTypes` and provides the syntax for four common collection types `List`, `Set`, `Map`, and `Optional`, that have one or two type arguments.

MCSimpleGenericTypes extends the `MCCollectionTypes` and introduces the syntax for expressions over generic types such as `Person<S,T>` and `Person<S<T>>`, where the constructor can be defined freely. Wildcards and restrictions on the generic type arguments are not allowed.

MCFullGenericTypes extends the `MCSimpleGenericTypes` and introduces the syntax for wildcards such as `Person<?>`, and restrictions on the generic type arguments like in `Person<S extends T>`.

The AST of type expressions instantiated via the parser can be unhandy for complex type expressions. To better support using type expressions, for instance, in the

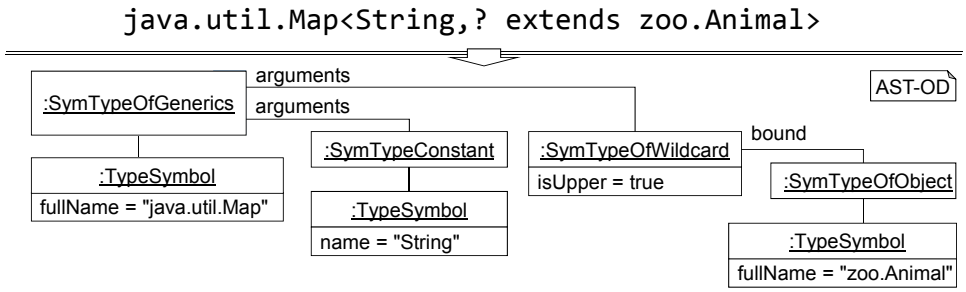


Figure 4.10: Object structure of a complex exemplary `SymTypeExpression`.

context of persisting the symbol table of a model (*cf.* Chapter 5), the language components for type expressions rely on handwritten abstract syntax classes in the form of `SymTypeExpressions`. `SymTypeExpression` is realized as an abstract Java class that manages a link from the type expression to a type definition in terms of an attribute of the type `TypeSymbol`. Specific kinds of `SymTypeExpressions` are realized as classes that extend the abstract class and add further, more specific attributes and behavior. Currently, the built-in type checker of MontiCore supports eight different kinds of `SymTypeExpressions`. The different kinds of `SymTypeExpressions` are:

SymTypeConstant represents a type expression of a constant type. For example, this includes the typical built-in primitive types of (programming) languages, like `int`, `float`, and `boolean`. The symbols for these built-in types can be added to the global scope of a language before any models are processed. The type expressions of constant types refer to these symbols. The top of Figure 4.9 depicts an example for a type expression of `int`.

SymTypeArray represents a type expression for an array type, such as `Tiger[]`. Internally, the class has an integer attribute that represents the dimension of the array. In the example, the dimension is 1. Another attribute of the class `SymTypeArray` is a type expression that refers to the argument of the array, in this case `Tiger`. This type expression can refer to the argument also via a qualified name. The center of Figure 4.9 depicts an example of a type expression of an array.

SymTypeOfObject represents a type expression of an object type. An object type refers to a type definition via the qualified or simple name of the type. This kind of `SymTypeExpression` does not introduce additional particular attributes. The center of Figure 4.9 depicts an example of a type expression of an object that is used as an argument for the array type.

SymTypeOfGenerics represents a type expression of a generic type. Generic types directly refer to a type symbol via a simple or a qualified name. Generic types have one or

more type arguments that are realized as a list attribute over `SymTypeExpressions`. The bottom of Figure 4.9 depicts an example for a type expression of a generic type that has a single type argument. Figure 4.10 demonstrates a more complex example of a type expression, in which the generic type `java.util.Map` has two type arguments, out of which one uses a wildcard.

SymTypeVariable represents a type expression for a type variable. A type variable is a variable that is part of a type expression whose values are, again, type expressions [ALSU07]. Type variables are used, for instance, for generic type arguments. The bottom of Figure 4.9 depicts an example of a type expression of a type argument with the name `T` in the context of the generic type `zoo.Food`.

SymTypeOfWildcard represents a type expression for a wildcard in the context of a generic type. In Java, the concrete syntax of a wildcard is realized via a question mark. Optionally, a wildcard can have an upper or a lower bound that restricts the types that the wildcard may assume. In Java, a lower bound for the wildcard is realized via the keyword `super`, and an upper bound with the keyword `extends`. The bound of a wildcard is a type expression. The type expression in Figure 4.10 depicts a wildcard with the upper bound `zoo.Animal` that is a type expression of an object type.

SymTypeVoid represents a type expression for the particular type `void` that may be used only as a return type of functions or methods. Similar to `SymTypeConstants`, this kind of `SymTypeExpression` refers to a dedicated type symbol.

SymTypeOfNull represents a type expression for `null`. This kind of type expression exists for reasons of completeness and compatibility with common programming languages. Utilizing `null` types in type systems for languages that are designed anew, in general, should be avoided [Hoa09].

For each of the five language components for type expressions, there is a visitor-based `Synthesize` class like the Java class `SynthesizeSymTypeFromMCBasicTypes` that translates instances of the AST classes of the language component to instances of the `SymTypeExpressions`. Moreover, the language component library contains two language components `BasicSymbols` and `OOSymbols` that include syntax for type definitions. While the concrete syntax of these definitions comprises only a name, the abstract syntax contains type expressions that are directly related to a type definition. The language component `BasicSymbols` introduces the symbol-defining and scope-spanning interface nonterminal `Type`. Each type has a list of type expressions that indicate the supertypes of the type. In the MontiCore abstract syntax, this is realized by the `TypeSymbol` that has a list of `SymTypeExpressions`. For each super type of the type, the list contains an entry with a `SymTypeExpression` that points to the type definition of the super type. Other elements of a type, such as type attributes, are realized as individual symbols located in the scope spanned by a `TypeSymbol`.

In addition to the `Type`, `MCBasicSymbols` introduces syntax for `TypeVariables`, `Variables`, and `Functions`. These are only an abstraction of the notions of types in object-oriented languages and are also usable for functional and specification languages. The language component `OOSymbols` extends the `BasicSymbols` and introduces further syntax for object-oriented type definitions, including `OOTypes`, `Methods`, and `Fields`. `OOTypes` are divided into classes, interfaces, and enumerations. Types, methods, and fields have the usual visibility concepts (*i.e.,* `public`, `protected`, `private`, or none of these). Additional modifiers indicating, *e.g.,* that a method can be `static`, are realized as well.

A *type check* can evaluate properties against given type expressions as well as other expressions and literals of a language. Such expressions and literals can be reused from the language component library [BEH+20] that contains five language components for expressions and three language components for literals. The type check is realized as a Java class with two attributes. One attribute is of the type `Synthesize` and is used to instantiate a type expression from a given AST. The other attribute is of the type `Derive` and determines type expressions from expressions and literals. The `Synthesize` and the `Derive` attributes can be configured with specific implementations to use the type check for a particular constellation of types, expressions, and literals.

Type expressions hold symbol-usage-specific information for type symbols that the type check of a type system requires. Therefore, a central capability of the type check is to test whether two `SymTypExpressions` are compatible. For instance, a class diagram language uses type symbols for the definition of classes. A class with the name `List` defines a `TypeSymbol` with this name. A type constructor can contain further information about the type definition, such as a generic type variable `T` of the type `List`. If the type `List<T>` is used in a type expression, a generic type argument is assumed for the generic type variable. For instance, the type of a class attribute `items` is `List<String>`, where `String` is the type argument. For the type check, this usage-specific information kept in `SymTypeExpressions` is important to indicate that `items.add("Ball")` is typed correctly whereas `items.add(1.0)` is not.

4.1.8 Symbol Resolution

Symbol resolution is the search from a symbol usage to a symbol definition considering the visibility concepts realized via scopes. MontiCore's concept for symbol resolution relies on trees of lexical scopes as introduced in Section 4.1.3 that have a global scope as their root. The global scope typically has artifact scopes as direct subscopes. These bridge the gap between a set of known model artifacts. We distinguish three phases of symbol resolution that are executed sequentially during the search for a symbol definition. If the execution of a phase does not yield a symbol definition as a result, the resolution continues with the execution of the next phase. Otherwise, it terminates and returns the symbol definition.

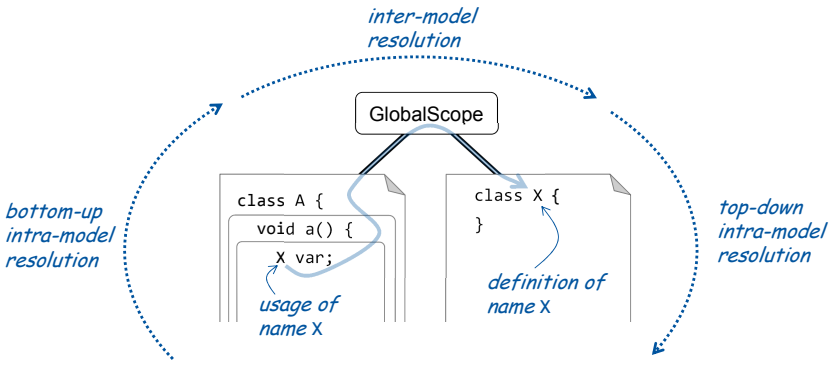
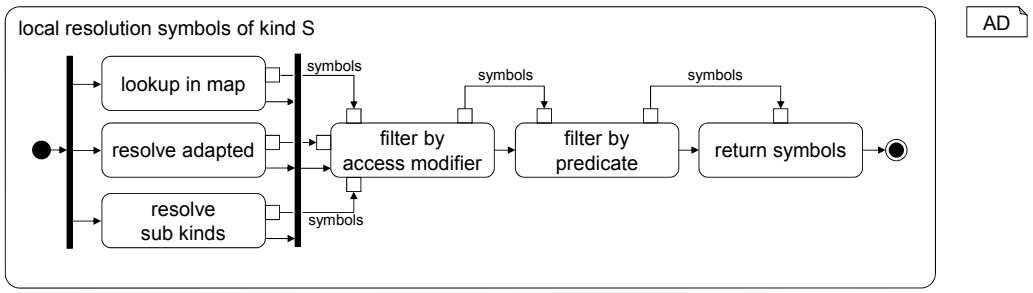


Figure 4.11: Overview of resolution of symbols by an example of resolving the type `X` of the variable `var`

An exemplary scenario for symbol resolution in the context of two Java classes `A` and `X` is depicted in Figure 4.11. In this scenario, the class `A` spans a scope for the method `a` that, again, spans a scope containing the variable with the name `var`. The type of the variable is `X`, which is a usage of a name that is defined somewhere in the same or in another artifact. In this example, the definition of the name is contained in another artifact in which `X` is the name of a Java class.

Bottom-up intra-model resolution resolves for a symbol in the local scope and from there proceeds resolving in the enclosing scopes iteratively until the artifact scope is reached. In the example, the resolution begins in the scope spanned by the method *a*. This scope does not contain a definition of the name `X` and the resolution, therefore, continues to search in the enclosing scope, which is the scope spanned by the class `A`. As this scope does not contain a definition of `X` as well, the resolution continues with the enclosing scope, which is the artifact scope of the class `A`. This scope also does not contain a definition of `X` and the resolution, thus, proceeds with inter-model resolution. *Inter-model resolution* resolves for artifact scopes that may contain definitions of the searched symbol. If candidates for such artifact scopes are found, the resolution proceeds with top-down intra-model resolution. In the example, the artifact scope of the class `X` has been found as a candidate that may contain a definition of the name `X`. *Top-down intra-model resolution* resolves for a symbol in the local scope and from there, proceeds resolving in subscopes. In the example, the artifact scope contains a definition of the name `X` in the form of a declaration of the class `X`. Resolution terminates and returns the symbol.

Symbol resolution has to handle hierarchies of symbol kinds. If a symbol of kind `k` is resolved, all symbols that are of kinds that specialize the kind `k` should be found, too. As the Liskov substitution principle [LW94] applies to symbol kinds, a symbol of a subkind can replace a symbol of the original kind in any program. However, this does



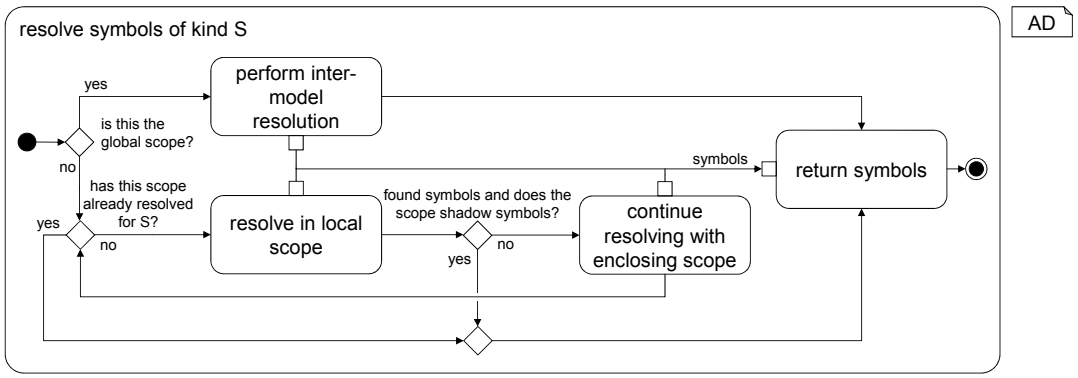
Method	Description
`resolveSLocally(..)`	performs local symbol resolution for a single symbol of kind `S`
`resolveSLocallyMany(..)`	performs local symbol resolution for symbols of kind `S`
`resolveAdaptedSLocallyMany(..)`	hook point for resolving adapted symbols in the local scope

Figure 4.12: Activities involved in resolution in a local scope (top) and methods realizing these (bottom)

not hold for symbols of super kinds, which therefore, should not be considered during symbol resolution. To integrate resolving for subkinds into the overall process of symbol resolution, the algorithm searches for symbols of subkinds in the local symbols contained in each scope individually. The following sections explain the three phases of symbol resolution in more detail.

Local Symbol Resolution

During both bottom-up and top-down intra-model resolution, the scopes of a model are traversed to search for symbols. Both resolution algorithms rely on the resolution of symbols that are locally contained in a scope. The local symbol resolution is depicted as an activity diagram in the top of Figure 4.12. The bottom of the figure contains a table that lists central methods that realize the local symbol resolution in the implementation of the STI. These methods are explained in more detail in Section 4.3.2. Resolving local symbols in a scope begins with three activities that can be carried out in an arbitrary order. Symbols contained in a scope are organized in individual maps per symbol kind. Each entry maps a symbol name to a list of symbols that have this name and that are contained in the local scope. A lookup in this map with the name and kind of the symbol resolution yields a list of symbols that are passed to the next activity. Besides the lookup for the symbol kind S that is resolved for, the local resolution resolves for all symbol kinds that are direct subkinds of S. This integrates hierarchical symbol kinds



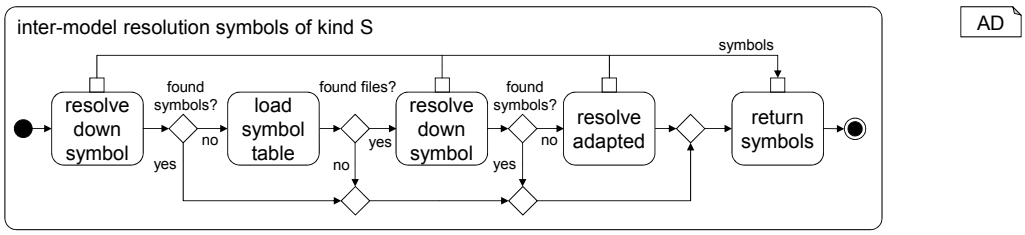
Method	Description
`resolveS(..)`	performs bottom-up intra-model resolution for a single symbol of kind `S`
`resolveSMany(..)`	performs bottom-up intra-model resolution for symbols of kind `S`

Figure 4.13: Activities involved in bottom-up intra-model resolution (top) and methods realizing these (bottom)

into the resolution process. Transitive subkinds do not have to be considered since the resolution for their direct super kinds considers these. Multiple inheritance is currently not allowed for symbol kinds, and, hence, diamond inheritance cannot resolve for symbols of the same kind multiple times. However, as the resulting symbols are collected in a set, finding the same symbol more than once is inefficient but does not produce errors. Usually, it is problematic if a type has to be aware of its subtype. The symbol resolution infrastructure has to assume to be aware of the entire universe of symbol kinds. Symbol kinds that the infrastructure is not aware of cannot be integrated into the resolution for hierarchical kinds. This is challenging for the realization of language composition. The implementation of the STI, however, handles this as described in Section 4.3.2.

The local symbol resolution algorithm further has a hook point that can be configured to integrate resolution for adapted symbols (*cf.* Section 6.2). The default implementation of this hook point is empty, but it can be implemented by languages, which is especially useful in the context of language composition. The usage of the hook point is explained in more detail in Section 6.2.

All symbols that the preceding three activities have found are filtered for the access modifier in a subsequent activity. This activity filters all symbols that have an access modifier with insufficient access rights. Afterward, the remaining symbols are filtered by a predicate that can be passed to the resolution algorithm. Only if a symbol satisfies the predicate, it is included in the set of symbols that is returned as a result in the last



Method	Description
`resolveS(..)`	performs inter-model resolution for a single symbol of kind `S` in the global scope
`resolveSMany(..)`	performs inter-model resolution for symbols of kind `S` in the global scope
`loadS(..)`	loads the symbol table for a symbol of kind `S`
`resolveAdaptedS(..)`	hook point for resolving adapted symbols through symbol resolvers in the global scope

Figure 4.14: Activities involved in inter-model resolution (top) and methods realizing these (bottom)

activity. Access modifiers and predicates are inspired from the respective concepts in the symbol management infrastructure [MSN17].

Bottom-up Intra-model Resolution

The activities involved in bottom-up intra-model resolution are depicted by means of an activity diagram in the top of Figure 4.13. The bottom of the figure contains a table that lists central methods that realize the bottom-up symbol resolution in the implementation of the STI. These methods are explained in more detail in Section 4.3.2. The bottom-up resolution of a symbol in the same model usually is the first step in symbol resolution. Bottom-up resolution can only be performed in an artifact scope or a subscope of an artifact scope but not in a global scope. Hence, the bottom-up intra-model resolution algorithm terminates if the resolution is performed on the global scope.

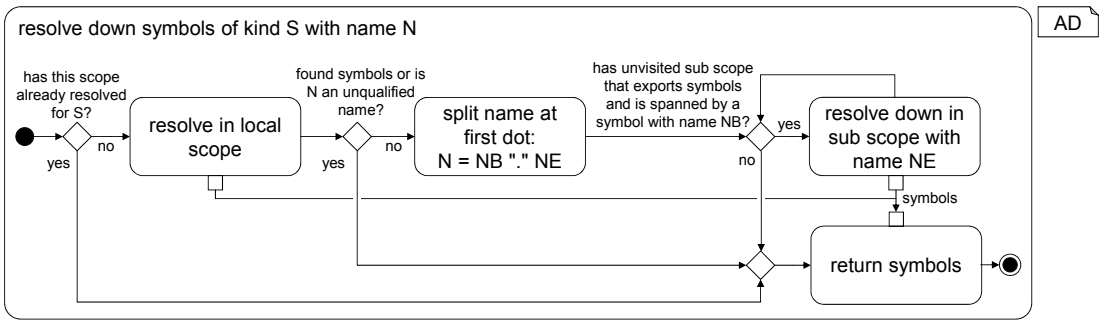
The first step in this phase of symbol resolution is to check the termination condition. If the current scope is the global scope, the overall symbol resolution continues with inter-model resolution. Otherwise, the resolution algorithm checks if it has already resolved for the symbol with the passed name and kind in the current scope. This is part of a mechanism for avoiding cyclic symbol resolution in the presence of cyclic symbol

adapters. The mechanism is explained in more detail in Section 6.2.2. If the symbol has already been resolved for, the algorithm returns the symbols that have been found and performs no further resolution. Otherwise, the algorithm resolves in the local scope. If one or more local symbols have been resolved in the scope and the scope is a shadowing scope (*cf.* Section 4.1.3), the algorithm returns the symbols and terminates. If either no symbols have been found or the scope is not a shadowing scope, the bottom-up resolution continues with the resolution in the enclosing scope of the current scope. In this case, the result of the resolution in the enclosing scope is set as the result of the resolution in the current scope.

Inter-model Resolution

The inter-model resolution algorithm is executed if the bottom-up intra-model resolution has reached the global scope. The main activities in this algorithm are visualized as an activity diagram in the top of Figure 4.14. The bottom of the figure contains a table that lists central methods that realize the inter-model symbol resolution in the implementation of the STI. These methods are explained in more detail in Section 4.3.2. The algorithm begins with executing the top-down intra-model resolution in all artifact scopes that are available as subscopes of the global scope. If this yields a non-empty result, the inter-model resolution returns the symbols and terminates. Otherwise, the resolution calculates candidates for names of symbol table files that contain artifact scopes in which the resolved symbol may be contained. The candidates for symbol table files are iterated. If a candidate points to an existing symbol table file, the contained symbol table is loaded. Afterward, the top-down intra-model resolution algorithm is executed on the loaded artifact scopes. If no suitable artifact scopes could be loaded from symbol table files, the resolution returns an empty set of symbols. If one or more symbols are found as a result of the top-down intra-model symbol resolution in the loaded artifact scopes, the symbols are returned as a result of the inter-model resolution, and the algorithm terminates. If no symbol is found, the algorithm resolves for adapted symbols with all resolvers (*cf.* Section 6.2.2) that are configured for the symbol kind in the global scope. If this yields any symbols as a result, these symbols are returned as a result, and otherwise, an empty set of symbols is returned.

Contrary to the symbol management infrastructure [MSN17], the STI does not qualify names as part of the inter-model resolution. Instead, the qualification of names, *e.g.,* with package names and applicable import statements, has to be performed a priori of the resolution as part of the symbol table instantiation that is explained in Section 4.1.10. An advantage of this is that a name is qualified only once instead of during each symbol resolution.



Method	Description
`resolveSDown(..)`	performs top-down intra-model resolution for a single symbol of kind `S`
`resolveSDownMany(..)`	performs top-down intra-model resolution for symbols of kind `S`

Figure 4.15: Activities involved in top-down intra-model resolution (top) and methods realizing these (bottom)

Top-down Intra-model Resolution

The activities involved in top-down intra-model resolution are depicted by means of an activity diagram in the top of Figure 4.15. The bottom of the figure contains a table that lists central methods that realize the top-down symbol resolution in the implementation of the STI. These methods are explained in more detail in Section 4.3.2. The resolution algorithm begins by checking whether resolution for the symbol with the passed name and kind has already been executed in the current scope. This is part of a mechanism for avoiding cyclic symbol resolution in the presence of cyclic symbol adapters that is explained in more detail in Section 6.2.2. If this is the case, the resolution returns an empty set of symbols. Otherwise, the resolution proceeds with enacting the local symbol resolution activity. If the local resolution has found at least one matching symbol, the symbol(s) are returned. If no symbols were found but the name of the symbol that the algorithm resolves for is not a qualified name (*i.e.,* it does not contain a `"."`), the resolution returns an empty set of symbols as well. This is due to the assumption that each scope that is considered for resolution is identified via the name of the symbol that spans the scope. If a resolved name is not qualified, it must be contained in the local scope or an enclosing scope but not in a subscope. As stated before, this is the part of the default resolving algorithm and can be customized for individual languages. If no symbol was found in the local scope and the name is qualified, the top-down resolution proceeds with splitting up the qualified name. The name is split on the first occurrence of a dot. As visualized in Figure 4.15, a name `N` is split such that it equals a concatenation of three

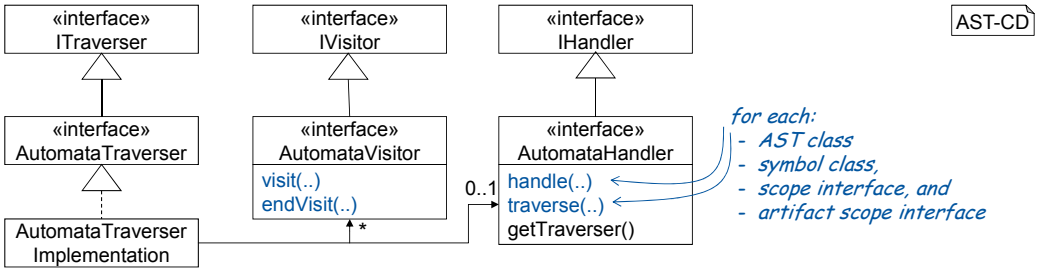


Figure 4.16: Generated visitor infrastructure for AST, scopes, and symbols

Strings `NB`, `"."`, and `NE`. If the resolved symbol is contained in a subscope of the current scope, the name part `NB` must be the name of a symbol located in a direct subscope of the current scope that spans this scope. Furthermore, if such a subscope exists, it must export the symbols, and it must not have been visited in the current execution of the resolution algorithm. If any of these conditions is violated, the algorithm returns an empty set of symbols. Otherwise, if all conditions hold, the top-down resolution continues with top-down resolution in the subscope for a symbol with the name `NE`. In other words, top-down resolution removes the qualifiers of a qualified name iteratively if these match suitable scopes. The set of symbols that results from this resolution is returned as a result of the current resolution.

The current realization of top-down intra-model resolution has one exception to this behavior. In Java-like languages, it is common practice that the name of a model/program artifact equals the name of a symbol that is directly contained ("top-level") in the artifact scope. For instance, a Java class `Person` must be contained in a file `Person.java`. If such a symbol exists, the top-down intra-model resolution in an artifact scope continues resolving in subscopes not only with the name `NB` but also with the name `N`. The resulting sets of symbols are merged. This behavior enables resolution for symbols in the presence of this Java-like convention without requiring to qualify a name both with the name of the artifact and the (same) name of the top-level symbol.

The realization of all symbol resolution algorithms is implemented in `resolve` methods that are part of generated scope interfaces, artifact scope interfaces, and global scope interfaces. Their implementation is described in Section 4.3. All parts of the symbol resolution algorithm, especially the top-down intra-model resolution, may be conceptually different in each language and therefore, it is essential that the generated code realizing the algorithm is customizable.

4.1.9 Symbol Table Traversal

As introduced in Section 2.1, MontiCore generates a language-specific visitor infrastructure for traversing the abstract syntax of a language. This infrastructure contains a

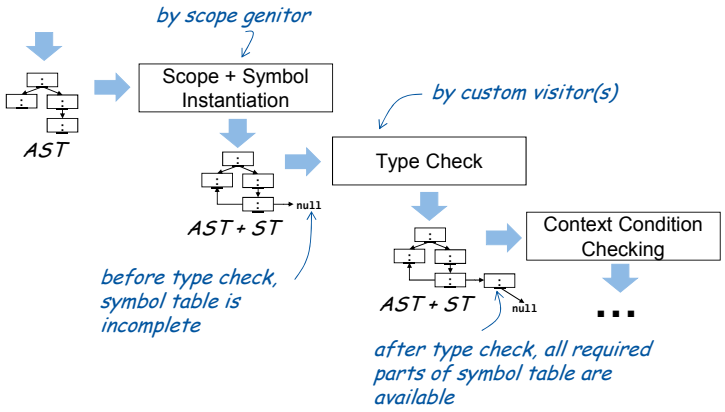


Figure 4.17: Instantiation of symbol tables in the context of processing models

visitor, a traverser, and a handler that are all realized as Java interfaces. The visitor infrastructure described in Section 2.1 explains only the parts of the visitor that traverses the AST. This section explains how the traversal is extended to consider symbol tables as well.

The classes and interfaces of the generated visitor infrastructure are depicted at the bottom of Figure 4.16. The top of the figure shows the interfaces of the MontiCore runtime that the language-specific interfaces extend. Visitor interfaces, such as the interface `AutomataVisitor`, contain `visit` and `endVisit` methods for each AST node, each symbol class, and the scope interface and artifact scope interface of a language. All of these methods are realized as empty default methods of the interface that are hook points for realizing specific behavior in implementations of the visitor interface.

Handler interfaces, such as the interface `AutomataHandler`, contain `handle` and `traverse` methods for each AST node, each symbol class, and the scope interface and artifact scope interface of a language. The `handle` methods first call the `visit`, then the `traverse`, and finally the `endVisit` method of an AST node, scope, or symbol. The `handle` method realizes the double dispatching together with the corresponding `accept` method in the AST node, scope, or symbol. The traverse methods for AST nodes are explained in Section 2.1. Traverse methods of a scope visit the contained symbols via double dispatching. Traverse methods of symbols, even if the symbol kinds span a scope, are empty by default. However, the traversal strategy can be adjusted by applying the TOP mechanism.

4.1.10 Symbol Table Instantiation

While processing models, a language tool typically creates the symbol table from a given AST directly after the AST has been conceived as the result of parsing the model.

As depicted in Figure 4.17, a scope genitor creates the symbol and scope objects that constitute the symbol table of a model from an AST.

The instantiation of symbol tables in the STI largely depends on the types of symbols and scopes and, hence, all parts are generated or have to be implemented manually, but nothing is contained in the MontiCore runtime environment. The symbol table instantiation is divided into multiple phases. In the first phase, the objects for all symbol definitions are instantiated, and the symbol objects are linked with the corresponding AST node objects. Furthermore, for symbols that span scopes, the scope objects are instantiated and linked with their environment as depicted in Figure 4.2. This task is realized by a *scope genitor* that is generated for each language. The scope genitor is a class that implements the language's visitor interface and uses a traverser to traverse the AST. Scope genitors, however, only initialize the "skeleton" of the symbol table of a model. This explicitly excludes additional symbol or scope attributes that have been either added via symbol rules or scope rules (*cf.* Section 4.2) or by manually applying the TOP mechanism to symbol and scope interfaces or classes. The initialization of symbol and scope attributes cannot be generated because it may depend on parts of the symbol table that may or may not have been initialized already. To this end, the initialization of symbol or scope attributes can be realized by applying the TOP mechanism to the scope genitor class only if the initialization does not rely on other parts of the symbol table. This does not hold if a symbol has a direct association with another symbol. In this case, the scope genitor cannot distinguish the case that the referenced symbol does not exist from the case that it has not been instantiated yet.

If a symbol or scope attribute requires information about other parts of the symbol table, the initialization of this attribute has to be carried out in a separate phase of symbol table creation. This is achieved by manually implementing a class that implements a language's visitor interface and traverses either the AST or the symbol table to add the missing information. The traversal of this visitor has to be executed after the scope genitor has created the symbol table skeleton. In complex cases with transitive dependencies between symbol table parts, adding further visitors and executing their traversal consecutively might be necessary. Due to the tremendous variety in the requirements for such visitors, they have to be conceived for each language individually by hand, and MontiCore does not generate any additional infrastructure beyond the visitors. The phase of establishing connections between usages of symbols and their definition is carried out with visitors and is called the *type check*. After the type check, the symbol table of a model is established entirely and can be the basis for further well-formedness checks realized in context conditions that are checked after the type check.

Internally, the scope genitor manages a stack of created scopes to set the enclosing scope of each symbol correctly. Initially, the scope stack contains only the global scope. Scope genitors have a method `createFromAST` that initiates the symbol table instantiation for a given AST object. As a first step, this method creates a new artifact scope with the language mill and adds it on top of the stack. Then, the method begins with

```
01 grammar Automata3 extends de.monticore.MCBasics {                    MCG
02
03   symbol scope Automaton = "automaton" Name "{" AutElement* "}" ;
04
05   interface AutElement;
06
07   symbol State implements AutElement = [ "initial" ]? "state" Name ";" ;
08
09   Transition implements AutElement =
10                               from:Name@State "-" input:Name ">" to:Name@State ";" ;
11
12 }
```

Figure 4.18: Examples for symbol table information in the `Automata` grammar

the traversal of the AST. For every AST node that the visitor visits, the scope genitor sets the scope that is on top of the scope stack as the enclosing scope of the AST object in the implementation of the respective `visit` method. If the traversal encounters an AST object of a symbol-defining nonterminal, the scope genitor further creates a new symbol instance with the mill and integrates it with the remaining symbol table in the `visit` method. As a part of this, the scope genitor sets the scope that is on top of the scope stack as the enclosing scope of the symbol.

If the traversal encounters an AST object that spans a scope, the implementation of the respective `visit` method of the scope genitor creates a scope instance with the mill, integrates it with the environment, and adds it on top of the scope stack. If a nonterminal defines a symbol and spans a scope, the created symbol and scope objects are associated with each other as well. In this case, the enclosing scope of the symbol is the enclosing scope of the spanned scope, *i.e.,* the symbol is not contained in the spanned scope. In the `endVisit` method of a scope-spanning nonterminal, the scope object on top of the scope stack is removed. Consequently, the following nonterminals that the traverser encounters are added to the enclosing scope. If symbols should be assigned with different scopes, the scope genitor can be customized with the TOP mechanism.

For reconfiguration that is required, inter alia, for language composition, MontiCore further generates a *scope genitor delegator* class. This class manages a traverser with genitors for the individual languages that constitute a composed language. With this, the scope genitor delegator takes into account manual changes to the individual scope genitors. To foster reconfiguration through inheriting languages, a language tool should always use the scope genitor delegator rather than directly employing the scope genitor. If a language inherits from one or more other languages, only the scope genitor delegator creates symbols and sets the enclosing scopes for nonterminals of the inherited languages.

```
MCG
symbol A = Name;       // defines a symbol of kind A
symbol AO = Name?;     // produces a symbol of kind A if name is present
symbol AO2 = ( Name | "constant" ); // same as for nonterminal AO
symbol AN = id:Name;     // produces abstract symbol class
symbol AN2 = "no name";  // produces abstract symbol class

symbol AL = Name*;     // not allowed
symbol AM = Name Name; // not allowed
```

Figure 4.19: Examples for allowed and forbidden uses of the `symbol` keyword

4.2 Annotating Grammars with Symbol Table Information

Most constituents of the STI can be generated with a code generator. To control the generation of symbol- and scope-specific information, the grammar model of a language can be enriched with symbol table information. This section describes when and how symbol table information can be indicated in a language's grammar and the effect of such annotations on the generated symbol table infrastructure of a language.

An example of symbol table information in a language's grammar is depicted for the example of the automata language in Figure 4.18. Nonterminals can be annotated with the keywords `symbol` and `scope`, and the `Name` nonterminal on the right-hand side of a grammar rule can be annotated with an `"@"` and the name of a symbol kind.

4.2.1 Indicate that a Nonterminal Defines a Symbol Kind

The keyword `symbol` can be added at the beginning of a class or an interface grammar rule. This keyword indicates that the nonterminal realizes a model element that is identifiable through a (unique) name and that is intended to define a symbol kind. Hence, each occurrence of such a nonterminal in the AST of a model produces a new symbol of this kind. If the `symbol` keyword is added to an interface nonterminal, all nonterminals that implement this interface nonterminal define a symbol of this kind as well. Furthermore, if a nonterminal extends or overrides a symbol-defining nonterminal, it defines a symbol of the same kind.

However, a nonterminal that extends another symbol-defining nonterminal can also define a novel symbol kind. To do so, the grammar rule of the nonterminal must be annotated with the symbol keyword individually. The novel symbol kind then inherits from the symbol kind of the inherited symbol-defining nonterminal.

Figure 4.19 depicts several allowed and forbidden usages of the symbol keyword for different grammar rules by example. As depicted by the rule for the nonterminal `A`, the

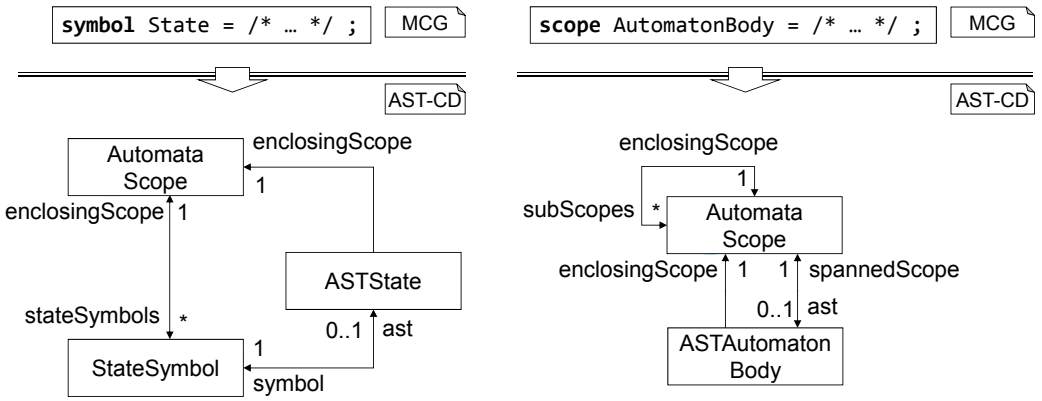


Figure 4.20: Example of the influence of the symbol keyword (depicted left) and the scope keyword (depicted right) on the associations between symbols, scopes, and AST nodes

symbol keyword can be added before the nonterminal name on the left-hand side of the rule if the right-hand side contains an (unnamed) nonterminal `Name`. Sometimes, rules have an optional name on their right-hand side either as an optional nonterminal (*cf.* `AO`) or as an alternative (*cf.* `AO2`). In such cases, the nonterminal defines a symbol as well, but symbols are only instantiated for instances in the AST that contain the name.

The symbol keyword can also be added to rules that either have a named `Name` nonterminal (*cf.* AN) or no `Name` nonterminal at all on their right-hand side. The first case may occur if a nonterminal has multiple names and no nonterminal should remain unnamed. The second case may occur if the identifying name of a symbol is not realized with the nonterminal `Name` in the grammar but with a different nonterminal instead. However, these cases require manual adjustment of the symbol table infrastructure to identify the name correctly and should be used carefully. The symbol classes generated for such symbol-defining nonterminals are generated as abstract classes to indicate the requirement for manual activity.

Iterations of names such as in direct iterations (*cf.* `AL`) or through multiple usages of the nonterminal `Name` with the same name (*cf.* `AM`) are not allowed, as these would create multiple symbols for a single nonterminal. This prevents a direct association between a symbol and an AST node, as described in Section 4.1.1. If a language engineer intends this, the grammar has to be restructured to use different nonterminals for the different symbol definitions or different instances of the same nonterminal. The same regulations for adding the symbol keyword to class nonterminals also hold for adding the symbol keyword to interface nonterminals.

The left side of Figure 4.20 depicts the effect of the symbol keyword for the nonterminal `State` on the code generation of the corresponding abstract syntax. Besides the class

`ASTState` that represents the AST node for states, the code generator produces a class `StateSymbol` for the nonterminal. The classes `ASTState` and `StateSymbol` are in a bidirectional association. The `AutomataScope` that is generated for the language is also in a bidirectional association with the `StateSymbol`.

4.2.2 Indicate that a Nonterminal Spans a Scope

The keyword `scope` can be added to a nonterminal to indicate that the nonterminal spans a scope. In the following, we refer to such nonterminals as *scope-spanning* nonterminals. The effect of this keyword is that all occurrences of the nonterminal in the parse tree create a new instance of the language's scope. All (transitive) children in the parse tree are contained within this scope (or subscopes of this scope). This holds especially for symbols produced by instances of symbol-defining nonterminals.

In Java, for example, a nonterminal for the body of a for-loop spans a scope that impacts the visibility of any contained symbols. Another example of a scope-spanning nonterminal and its effect on the code generation of the abstract syntax data structure of the automata language is depicted on the right side of Figure 4.20. The keyword `scope` is added before the name of the nonterminal `AutomatonBody` on the left-hand side of the grammar rule. In the generated abstract syntax, there is an additional association between the AST node of the nonterminal and the language's scope class. In contrast to the `symbol` keyword that produces a new symbol class per nonterminal, all nonterminals of a language share the same scope class. The scope class can be configured with scope properties (*cf.* Section 4.1.3). To this end, the grammar language has keywords for customizing the scope properties for each scope-spanning nonterminal individually, in case the value of the properties should deviate from the default. These keywords are indicated in brackets after the `scope` keyword of a grammar rule. If multiple keywords are indicated, these have to be separated by whitespace.

`non_exporting` marks the produced scope instance as a non-exporting scope, *i.e.,* a scope that does not export any locally defined symbols. By default, scopes export all local symbols.

`non_shadowing` marks the produced scope instance as non-shadowing scope, *i.e.,* a scope in which local symbols do not shadow symbols of other scopes that have the same name. By default, scopes shadow symbols from enclosing scopes.

`ordered` marks the produced scope instance as an ordered scope, *i.e.,* a scope in which the source position of a symbol must occur before the source position of any usages of the symbol. By default, scopes are not ordered.

In many block-based languages, symbol definitions can be strongly related with scopes that are spanned. In the language's grammar, this is typically realized by a nonterminal

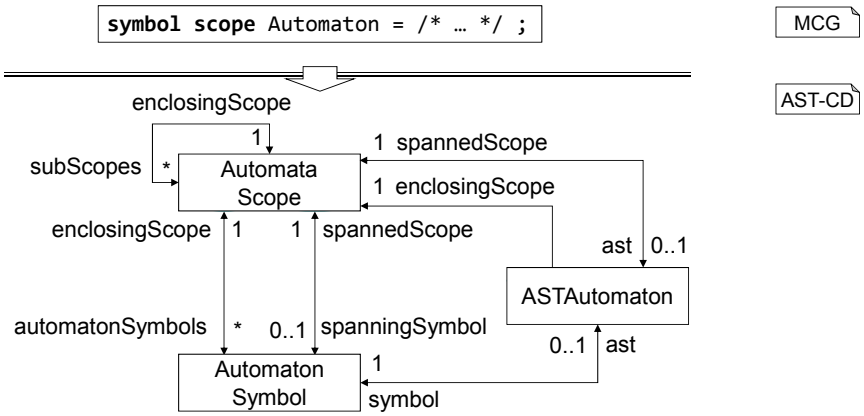


Figure 4.21: Example of associations between symbols, scopes, and AST nodes if a nonterminal spans a scope and defines a symbol

that both defines a symbol and spans a scope. In Java, for instance, a Java class defines a symbol, and the body of the Java class spans a scope. Similarly, a method definition defines a symbol, and the body of the method definition spans a scope. For scopes that export symbols, the connection to the symbol that spans the scope is relevant for calculating qualified names. For instance, a Java class `Foo` defines a symbol `Foo` and spans a scope with the body of the class. Any static members of the class can be identified with the name of the class as a qualifier.

In the automata language, the nonterminal `Automaton` defines a symbol and spans a scope. The effect of this is depicted in Figure 4.21. Besides the combined effects of the individual keywords `symbol` and `scope`, the combination of both keywords produces a bidirectional association between the symbol class and the scope class.

4.2.3 Indicate that a Nonterminal Uses the Name of a Symbol

In addition to marking nonterminals as symbol-defining and scope-spanning, nonterminals can be enriched with information about symbol usages. A symbol is always used via the symbol name, *i.e.,* by using the nonterminal `Name` on the right-hand side of a grammar rule. To indicate that a name on the right-hand side of a grammar rule is a symbol usage of a symbol of certain kind `K`, the `Name` nonterminal can be suffixed with an `@K`, *i.e.,* `Name@K`. To distinguish multiple symbol usages in the right-hand side of the same nonterminal, the annotation can be added to named nonterminals as well, such as in `n:Name@K`.

The effect of a symbol usage in the grammar for the code generator is that the corresponding AST class of the nonterminal obtains an additional attribute with the name of the nonterminal and the suffix `Definition`. The type of the attribute equals

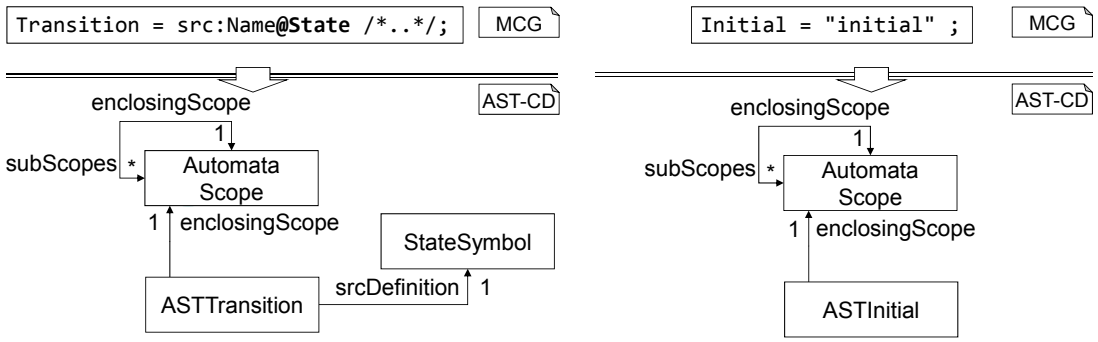


Figure 4.22: Example of associations between symbols, scopes, and AST nodes if a nonterminal uses a symbol (depicted left) or if a nonterminal does neither define a symbol nor span a scope (depicted right)

the type of the symbol class. This additional relationship enables navigation from the AST class of the symbol usage to the class of the symbol definition. Other than the `SymbolReferences` in the SMI [MSN17], no dedicated infrastructure is generated for symbol usages, and, hence, the annotation with a symbol usage has no further influence on the code generator. An example of a symbol usage of the nonterminal `Transition` in the automata language is depicted on the left side of Figure 4.22. The `Name` nonterminal with the name `src` is indicated to refer to the name of a `State` symbol. In the generated code, this reflects in the directed association from the class `ASTTransition` to the class `StateSymbol`.

For completeness of this description, the right side of Figure 4.22 depicts an example of the abstract syntax in case no symbol table annotation is added to a nonterminal with the name `Initial`.

4.2.4 Providing Symbol Kind Attributes

With the `symbol` keyword, a new symbol kind can be introduced through the grammar. With this, the code generator produces a symbol kind with attributes common to all symbol kinds, such as the connections to the symbol environment (*e.g.,* AST and enclosing scope) and a String attribute for the symbol name. However, additional attributes cannot be indicated through the grammar rule of the symbol-defining nonterminal.

Theoretically, it would be possible to add additional information from the parts of the right-hand side of the grammar rule that is reflected in the abstract syntax. However, this pollutes the symbol table infrastructure with additional information that is not required in most languages. Moreover, this would duplicate information already contained in the AST. However, together with loading and storing of symbol tables as presented in Chapter 5, the AST is not always available from a given symbol. Therefore, the

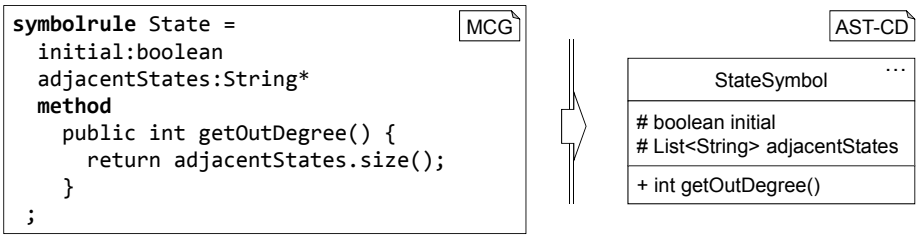


Figure 4.23: Example of a symbol rule (left) and its effect on the generated symbol table infrastructure (right)

STI has a dedicated mechanism for providing additional attributes for each symbol kind through *symbol rules*, which are a specific kind of grammar rules.

Similar to AST rules presented in Section 2.1, symbol rules can contain named and typed attributes as well as methods. A symbol rule begins with the `symbolrule` keyword followed by the name of the symbol-defining nonterminal. The right-hand side of a symbol rule contains attributes by indicating the attribute name, followed by a double colon and the attribute type. Each attribute may be marked as optional (with `"?"`) or may be iterated (with `"*"`). Each attribute is translated into an attribute of the generated symbol class.

Methods begin with the keyword `method` followed by the method signature and body in the same syntax as the method would have in Java. Each method of the symbol rule becomes a method in the generated symbol class.

As an alternative to symbol rules, attributes and methods can be added to a symbol kind by applying the TOP mechanism to the symbol class. An advantage of adding attributes to a symbol kind via a symbol rule is that more symbol-kind-specific language infrastructure can be generated from the grammar. MontiCore generates infrastructure for loading and storing additional symbol kind attributes in the context of symbol table persistence (*cf.* Chapter 5). For symbol rule methods, MontiCore does not generate additional language infrastructure. On the contrary, the body of methods in symbol rules may be lengthy and, hence, "pollute" the grammar with implementation details. Therefore, we recommend adding methods to symbol kinds with the TOP mechanism.

For each symbol-defining nonterminal, a grammar may contain at most one symbol rule. An example of a symbol rule for the nonterminal `State` is depicted on the left side of Figure 4.23. The symbol rule introduces an attribute `initial` of the type `boolean` to transport the information whether the state is an initial state through the language's symbol table. Furthermore, the symbol rule defines a second attribute `adjacentState` that is of the iterated type `StateSymbol` and lists all states to which the current state is connected via a transition. Non-primitive types have to be indicated via their qualified names, *i.e.,* including the package. As this symbol rule associates `StateSymbol` objects with other `StateSymbol` objects, the symbol table instantiation has to be carried out

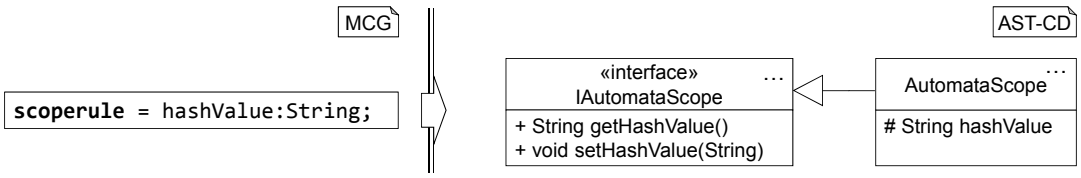


Figure 4.24: Example of a scope rule (left) and its effect on the generated symbol table infrastructure (right)

in two phases, and the `adjacentStates` attribute has to be set in the second phase. If set in the first phase, the symbol table instantiation could only set all adjacent states from transitions visited before, which may omit transitions that are visited afterward. The symbol rule further defines the method `getOutDegree()` that returns the size of the list of adjacent states.

An excerpt of the effect of this symbol rule on the generated class `StateSymbol` is depicted on the right side of Figure 4.23. Symbol rule attributes are translated to attributes of the symbol class, iterated attribute types are realized as lists. The code generator produces access and manipulation methods for the attributes. These are omitted in the figure for presentational reasons. The method of the symbol rule is translated into a Java method of the symbol class.

4.2.5 Providing Scope Attributes

Similar to symbol rules, *scope rules* enable adding additional attributes and methods to the scope of a language. A scope rule begins with the keyword `scoperule`. Since languages do not distinguish different scope types, a scope rule does not mention a nonterminal name on the left-hand side of the rule. The right-hand side of the rule, however, may contain the same elements as symbol rules and AST rules. There must not be more than a single scope rule per grammar.

An example of a scope rule is depicted in Figure 4.24. The scope rule has a single attribute `hashValue` of the type `String`. The intention of the attribute in this example is to manage a hash value calculated on the scope that can be used to check whether the scope has been modified. The generated scope interface contains abstract access and manipulation methods for the scope rule attribute. The attribute itself and the implementation of the abstract methods is contained in the generated scope class, which can be customized with the TOP mechanism to calculate the hash value.

4.3 Implementation of the Typed Symbol Table Infrastructure

The implementation of the STI follows the concepts described in Section 4.1 and is integrated into MontiCore in version 7. As such, the implementation is described in detail in the technical report on MontiCore [HKR21], especially in the chapter about symbol tables [BMSN21]. Furthermore, MontiCore is open-source and, hence, the implementation of the STI is publicly available[1]. The implementation of the STI follows a set of central principles:

Generate language-specific types To enable customization through extension with subtypes, each constituent of the symbol table infrastructure is generated as a language-specific type. To further foster extensibility, no inner types are used in generated code.

Enable customization of generated types The TOP pattern and a language mill are employed to integrate handwritten customizations with generated code [HKR21]. Every generated artifact is extensible with the TOP pattern. All artifacts that realize compositional constituents of the symbol table infrastructure (*cf.* Chapter 6) are instantiated through a language mill and, hence, can be exchanged with subtypes through reconfiguration of the mill.

Generate default realizations In different software languages, type systems, visibility concepts, and namespaces can rely on various concepts and, hence, large parts of the implementation of the symbol table infrastructure can vary. The STI contains a generated default realization for such concepts that can be adjusted via customization of the corresponding Java types.

Reduce generating boilerplate code The implementation of the symbol table infrastructure for language-agnostic parts contains boilerplate code. To reduce manually written boilerplate code, reusable units of language-agnostic symbol table infrastructure are contained in default method implementations of interfaces that the generated types implement. Therewith, such parts can be customized by overriding the corresponding methods and extending the generated types with the TOP mechanism. If boilerplate methods instead were extracted to static methods in "helper" classes, the customization would be more complex in general.

The following explains the generated parts of the STI using the automata language as an example. To reduce redundant explanations, source code generated for each symbol kind is often explained for state symbols only, and the analogous explanation for automaton symbol kind is omitted.

[1] MontiCore on GitHub: `monticore.github.io`

4.3.1 Implementation of Language Mills in MontiCore

Language mills [HKR21] provide builders for objects of language infrastructure classes and enable reconfiguration of language infrastructure for language composition. They are needed for flexible instantiation of objects from symbol table infrastructure types for composed languages (*cf.* Chapter 6). Mills are realized as singleton Java classes, and MontiCore generates a single mill class for each language. Each mill has static access methods to obtain the objects of the language infrastructure from the singleton mill instance. Furthermore, the singleton instance of the mill can be set with a method `initMe(..)` to reconfigure the mill with a subtype of the mill that must be passed to the method as an argument. The `init()` method initializes the current mill and the mills of inherited languages. This is explained in more detail in the context of language composition in Section 6.1.4. With the mill's `reset()` method, any reconfigurations of a mill can be set to their initial configuration.

As a singleton, a mill class manages the single mill instance as a class attribute. The class contains several static methods that delegate to corresponding non-static methods of the mill instance. The non-static methods implement the actual behavior and have the same method signature as their static equivalences, except that the method names have an appended underscore.

While a language tool processes a model, multiple instances of AST nodes and symbol classes have to be created. MontiCore contains builders for these classes, and the language mill creates the builders. The mill, hence, is a "builder for the builders". For all other parts of the language infrastructure that have to be reconfigured during language composition, the mill provides individual access methods. This includes the following parts of the STI:

- Instances of the scope class are created via the method `scope()`
- Artifact scope class instances are created via the method `artifactScope()`
- The singleton instance of the global scope class can be obtained through the method `globalScope()`
- The scope genitor class can be accessed via the method `scopesGenitor()`
- The scope genitor delegator class can be accessed via the `scopesGenitorDelegator()` method

The names of the methods do not include the language name and the usual syntax of getter methods to keep their names short and readable. In the usual naming conventions for Java methods, the method `scope()` would be named, *e.g.,* `getAutomataScope()`. Apart from the constituents of the STI described above, mills contain methods for obtaining other parts of language infrastructure, such as the parser and the traverser of a language.

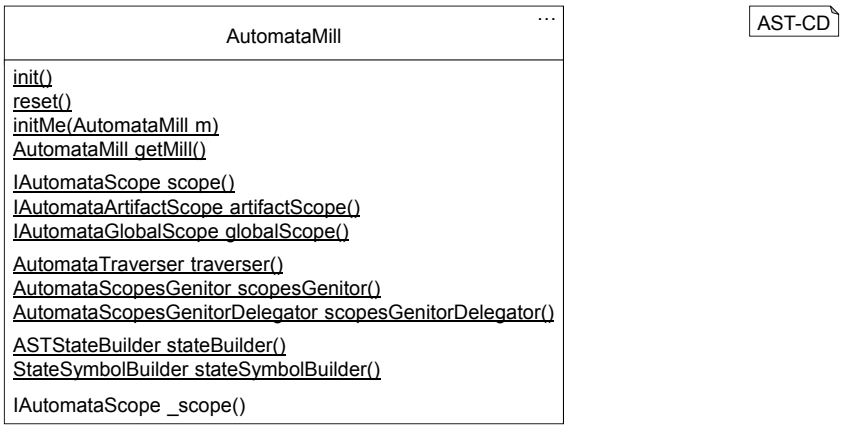


Figure 4.25: Static methods of the language mill for the automata language that delegate to non-static methods of the singleton instance

Figure 4.25 depicts the static methods of a mill by the example of the automata language. The `init()`, `reset()`, and `initMe()` methods are part of every mill. In addition to these methods, the mill contains static accessor methods for each language-specific type of the STI described above, such as the global scope, the traverser, and the scope genitor. The implementation of these methods uses an instance of the mill and a corresponding non-static method to implement the getter method and enable proper reconfiguration. By convention, the names of the non-static methods match the names of the static methods but begin with an underscore. For each symbol-defining nonterminal, the mill contains two static builder methods, one for the AST node and one for the symbol class. The figure depicts these by the example of the `State` nonterminal. Other methods such as all non-static methods and methods for types that are not directly related to the STI, such as the language's parser or its inheritance traverser, are omitted.

4.3.2 Implementation of Scopes in MontiCore

Scopes contain and manage local symbols, realize the visibility concepts for symbols, and are capable of performing symbol resolution. The concept for scopes is explained in Section 4.1.3. MontiCore generates a single dedicated scope type for the STI of each language. Hence, all nonterminals of a language that span a scope rely on the same scope type. To support multiple inheritance of scopes for language composition, the implementation of scopes is split into a scope interface and a scope class implementing this interface. This is explained in more detail in Chapter 6. Furthermore, the artifact scopes of a language have a particular behavior that is realized in a separate artifact scope type. Again, to support multiple inheritance of languages, artifact scopes are split into artifact scope interfaces and classes.

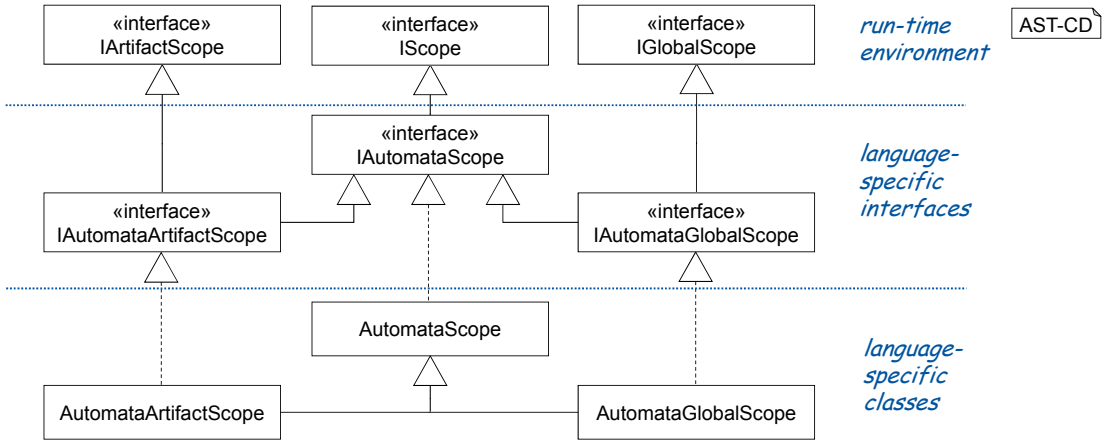


Figure 4.26: Classes and interfaces realizing scopes in the STI

The behavior of artifact scopes deviates from other scopes of a language only in some parts. Thus, the artifact scope interface extends the scope interface to reuse most of its methods directly and override only some methods. Global scopes support resolving symbols between different artifacts. A language-specific global scope class and a global scope interface are generated for each language. Similar to artifact scopes, the types of global scopes require only parts of dedicated behavior that are different from other language scope types. To reuse the other parts of the behavior encoded into methods, global scope interfaces extend the scope interfaces.

Figure 4.26 presents an overview of the classes involved in realizing scopes within the STI based on the example of the automata language. At the top row of the figure are the runtime interfaces that realize default implementations of language-independent methods to reduce boilerplate code in the generated scope classes and interfaces. These default methods can be customized by applying the TOP mechanism pattern [HKR21] to a generated scope class or interface to override the methods. The middle row of the figure shows the scope interfaces, which realize language-specific functionality such as the symbol resolution algorithm. The bottom row comprises the scope classes that manage scope attributes and direct access to the attributes. Artifact scopes are depicted in the left column, global scopes in the right column, and the center column presents the general scopes. The inheritance relationship between scopes, artifact scopes, and global scopes is realized on the level of scope interfaces and scope classes. An inheritance relationship among runtime environment scope interfaces is not required as it does not provide any benefits but instead produces additional diamond inheritance relationships. The following introduces the scope interface and scope class in Figure 4.26 in more detail.

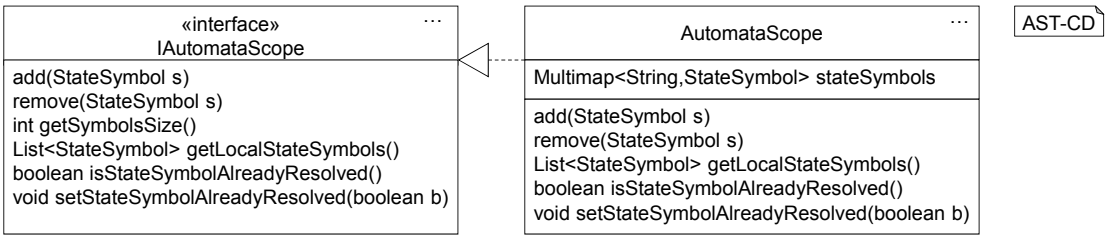


Figure 4.27: Managing local symbols within scopes

Managing Symbols

Scopes in the STI manage their local symbols separately for each kind. For each symbol kind defined in a language, the scope class contains a multimap as depicted in Figure 4.27 that maps the symbol name to a list of symbols with this name. In many languages, a list mapping a symbol name to a single local symbol would suffice, but some languages allow multiple symbols with the same name to be present in a scope. In such cases, the symbols must be differentiable based on further information. For example, the scope of a Java class can have multiple method symbols of the same kind that have the same name but are differentiable based on the method arguments. The STI uses a multimap as data structure to enable languages defining such cases. However, this requires language engineers to define custom predicates passed to the symbol resolution as arguments [MSN17].

All method signatures for managing symbols are defined in the scope interface `IAutomataScope`. The implementation of methods that directly access or modify the symbol map attributes, such as getter and setter methods, are contained in the scope class. Some methods, such as `getSymbolsSize()`, delegate in their implementation to the getter and setter methods of the class and, hence, can be located as default implementation in the scope interfaces. The following shortly explains the purpose of the most relevant methods for managing symbols in scope:

`add(..)` is a method that a scope class realizes once per symbol kind of the language. It adds a passed symbol of a certain kind to the respective multimap attribute. The method is contained as an abstract method in the scope interface and implemented in the scope class.

`remove(..)` is a method for removing a symbol from the current scope. It is contained as an abstract method in the scope interface and implemented in the scope class for each symbol kind.

`getSymbolsSize()` calculates the number of all local symbols in the current scope. This method is generated because it requires calculating the sum of the multimap sizes

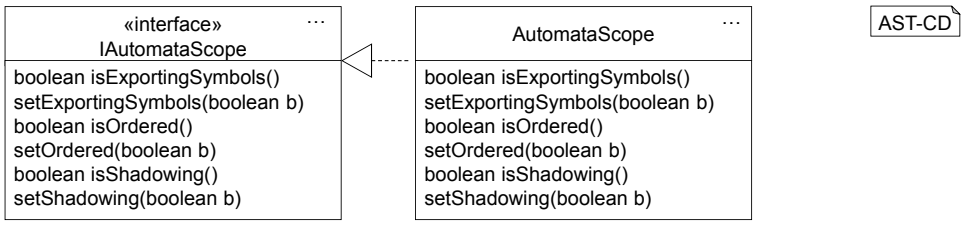


Figure 4.28: Access to scope properties

of all symbol kinds. As the implementation relies on other scope methods instead of direct access to the multimap attributes, it can be realized as a default method in the scope interface.

`getLocalStateSymbols()` is a method that returns a list of all local symbols of a specific kind, in this example, all local `StateSymbols`. It is generated once per symbol kind of each language and is contained as an abstract method in the scope interface and implemented in the scope class.

`isStateSymbolAlreadyResolved()` is a method that returns *true* if during an execution of the symbol resolution algorithm, resolution of the symbol has already been attempted with the current kind in this scope instance. The method is required to prevent cycles in the resolution algorithm that could occur due to cyclic symbol adapters. Although the method is not intended to be used outside of the scope, it is contained in the scope interface to be used within the resolution algorithm encoded into default implementations of the `resolve` methods in the scope interface.

`setStateSymbolAlreadyResolved(..)` is a method used together with the method `isStateSymbolAlreadyResolved()` to avoid cycles and, similarly, should not be used outside of the scope. It sets a local Boolean attribute to the passed value.

Scope Properties

Each instance of a scope type can be adjusted with three properties, as explained in Section 4.1.3. To control these properties, scopes in the STI define the methods depicted in Figure 4.28.

`isExportingSymbols()` and `setExportingSymbols(..)` are methods to access and control whether the scope instance is set to export its symbols.

`isOrdered()` and `setOrdered(..)` are methods to access and modify whether a scope instance considers the order of the locally defined symbols.

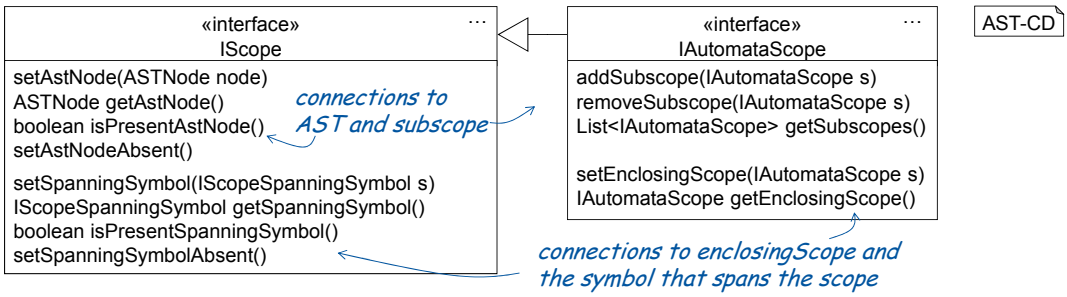


Figure 4.29: Integration of the scope with its environment

`isShadowing()` and `setShadowing(..)` are methods for obtaining and setting the scope property that determines whether local symbols shadow symbols with the same name and kind of other scopes.

Integration with Environment

Scopes are typically arranged within scope trees, contain symbols, and can be connected to the AST, as explained in Section 4.1.1. To this end, scopes have methods for managing the connections to their direct environment, *i.e.,* the technical realization of the scope's associations to other constituents of the symbol table infrastructure and the AST. An overview of these methods is depicted in Figure 4.29, and the following explains the methods individually.

`addSubscope(..)`, `removeSubscope(..)`, and `getSubscopes()` manage the attribute `subscope` of a scope. Each scope has a list of subscopes that can be of the same scope type or a subtype. The abstract methods are contained in the scope interface, and the implementation of these methods is contained in the scope class.

`getEnclosingScope()` and `setEnclosingScope(..)` are methods to access and modify the enclosing scope of the current scope. As scope instances are usually arranged as a tree, each scope – except the scope that is the root of the scope tree – has an enclosing scope. As the root of a scope tree is always the global scope, these methods are overridden and handled differently there. The type of the enclosing scope is always the same type or a subtype of the type of the current scope.

`getAstNode()` and `setAstNode(..)` access and modify the AST node that can be associated with the scope. The type of the AST node is the interface `ASTNode`, as it can be of various concrete types. During model processing, each AST node instance is associated with an enclosing scope after the symbol table has been created. A scope is associated with an AST node if the scope has been created during model processing, *i.e.,* as a processing step after parsing a model. If a scope is created as part of loading a

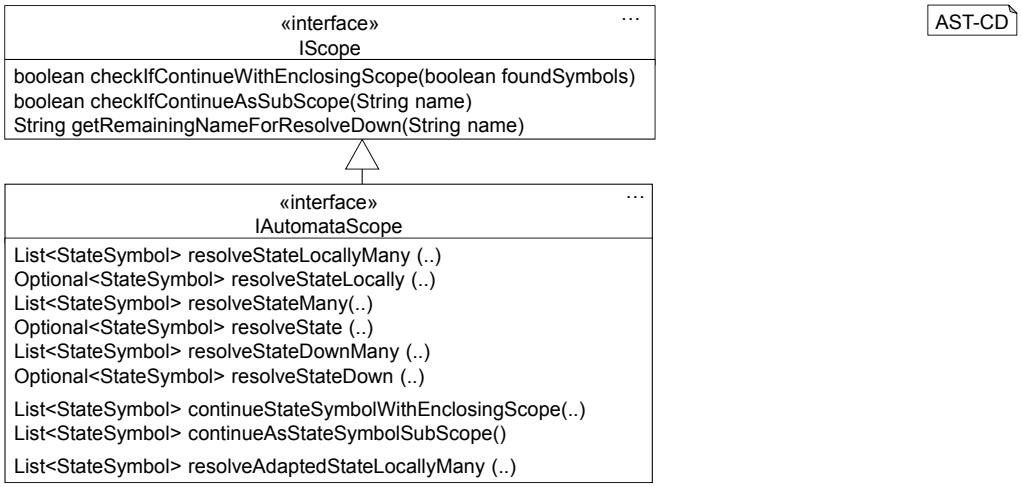


Figure 4.30: Methods realizing the symbol resolution by the example of state symbols

symbol table from a stored symbol table file, no AST node is associated with it. Thus, the AST node is realized as an optional attribute of a scope.

`isPresentAstNode()` and `setAstNodeAbsent()` delegate to the respective methods of the optional attribute.

`getSpanningSymbol()` and `setSpanningSymbol(..)` access and modify the symbol that spans the scope if the scope is spanned by a symbol. Whether a scope is spanned by a symbol depends on the language element that spans the scope. As the scopes spanned by different elements of a language are realized through the same Java type, the spanning symbol is realized as an optional attribute and is not typed with a specific symbol class.

`isPresentSpanningSymbol()` and `setSpanningSymbolAbsent()` delegate to corresponding methods of the optional attribute.

`getName()` and `setName(..)` access and modify the scope's name if the scope has a name. An artifact scope typically has the name of the model artifact, and scopes that are spanned by a symbol have the same name as their spanning symbol. Other scopes typically do not have a name and, thus, the name is realized as an optional attribute.

`isPresentName()` and `setNameAbsent()` delegate to the respective methods of the optional attribute.

Symbol Resolution

The implementation of the symbol resolution algorithm is contained in the `resolve` methods of the scope interfaces. The algorithm is explained in Section 4.1.8 in detail, and this section gives an overview of its technical realization. All resolve methods are specific to the kind of symbols that these resolve in the symbol table. To indicate this, the name of the symbol kind is encoded in the name of the methods. An overview of the resolve methods for state symbols is depicted in Figure 4.30. Each resolve method exists in variants of overloaded methods that have different arguments. All resolve methods exist in variants with and variants without the argument of a `Predicate` over the symbol. This predicate enables further restrictions of the symbols that are searched. It can be utilized, for instance, for realizing the resolution of method symbols with a particular list of arguments among multiple available method symbols with the same name. By default, all symbols satisfy the predicate. This is equal to using the predicate `s -> true`.

Furthermore, some variants of the resolve methods offer to pass an `AccessModifier` as an argument. Language engineers can use access modifiers to realize sophisticated visibility concepts. In general, scopes can use the `exportSymbols` property to globally restrict access to symbols from all scopes except the scope in which the symbol is defined. Sophisticated visibility concepts such as the package visibility in Java, however, require granting access to the symbol from some scopes but prohibiting access from others. To realize this, the visibility is passed as an access modifier argument to the `resolve` methods. The following gives an overview of the resolve methods contained in a scope interface by the example of the symbol kind `StateSymbol`:

`resolveStateLocallyMany(..)` realizes the local symbol resolution. The method implementation searches for symbols of a specific kind (in this example, the kind `StateSymbol`) that are directly contained in the local scope and have the passed symbol name. Furthermore, the method searches for symbols of subkinds of the kind. It returns the resolved symbols as a list.

`resolveStateLocally(..)` realizes the local symbol resolution but expects to find a single matching symbol only. If no such symbol exists, it returns an empty `Optional` value. If multiple symbols exist, the method yields an ambiguity error listing the ambiguous symbols.

`resolveStateMany(..)` realizes the bottom-up intra-model resolution. The resolution begins in the local scope and continues with the resolution in enclosing scopes until the global scope is reached. The subsequent inter-model resolution is carried out by the method `resolveStateMany(..)` in global scope interfaces. The consecutive top-down resolution in foreign artifacts is executed by the `resolveStateDownMany(..)` method. Internally, the method uses the `continueStateSymbolWithEnclosing-`

`Scope(..)` method to execute the resolution in enclosing scopes. The `resolveStateMany(..)` method returns the resolved symbols as a list.

`resolveState(..)` resolves for a single symbol with the passed name. If no such symbol exists, it returns an empty `Optional` value. If multiple symbols exist, the method yields an ambiguity error that lists the ambiguous symbols.

`resolveStateDownMany(..)` realizes the top-down intra-model resolution and returns the resulting set of symbols as a list. The method internally uses the method `continueAsStateSymbolSubScope(..)` to perform the evaluation of qualified names and to continue resolution in subscopes.

`resolveStateDown(..)` performs the top-down intra-model resolving but expects to find a single matching symbol only. If no such symbol exists, it returns an empty `Optional` value. If multiple symbols exist, the method yields an ambiguity error that lists the ambiguous symbols.

`continueStateSymbolWithEnclosingScope(..)` supports the realization of the bottom-up symbol resolution. The method realizes the part of the resolution that changes from the resolution in the current scope with the resolution in the enclosing scope. This is carried out in a dedicated method to foster customization through the TOP mechanism and by overriding the method. Internally, the method first checks whether the intra-model resolution may continue in the enclosing scope with the method `checkIfContinueWithEnclosingScope(..)`. If this is the case, the method invokes `resolveStateMany(..)` on the enclosing scope of the current scope.

`checkIfContinueWithEnclosingScope(..)` is a method that supports the realization of the bottom-up symbol resolution by checking whether the resolution has to continue searching for symbols in the enclosing scope. If a scope shadows its enclosing scope and the current execution of the resolution algorithm has found at least a single symbol, there is no need to continue resolution in the enclosing scope, and the method returns the Boolean value $false$. Otherwise, it returns *true*. This check is carried out in a dedicated method to foster its customization.

`continueAsStateSymbolSubScope(..)` supports the realization of the top-down symbol resolution. This is carried out in a dedicated method to foster its customization. The method first checks whether the resolution should be continued in any subscope with the method `checkIfContinueAsSubScope(..)`. If this is the case, the method iterates over name candidates for the resolution in subscopes that are calculated by employing the method `getRemainingNameForResolveDown(..)`. For each of these candidates, the resolution is performed with the method `resolveStateDownMany(..)`, and the resulting sets of symbols are merged.

`checkIfContinueAsSubScope(..)` supports the realization of the top-down symbol resolution. The default implementation of this method is contained in the interface `IScope` and is carried out in a dedicated method to foster its customization. The method returns a Boolean value that is *true* if the current scope exports symbols, the name to resolve for is a qualified name, and the first part of the qualifier equals the name of the symbol that spans the scope. Otherwise, the method returns $false$.

`getRemainingNameForResolveDown(..)` supports the realization of the top-down symbol resolution. The default implementation of this method is contained in the interface `IScope` and is carried out in a dedicated method to foster its customization through the TOP mechanism and method overriding. The default implementation removes the first part of the qualifier if the name is a qualified name and returns the remainder of the qualified name. Otherwise, it returns the full name.

`resolveAdaptedStateLocallyMany(..)` is a method that acts as a hook point for resolving adapted symbols in the current scope. By default, the implementation of this method returns an empty list. It can be overridden by applying the TOP mechanism to the scope interface and instantiating symbol adapters. This is explained in more detail in Section 6.2.

Global scopes override some of these methods and provide a different behavior for realizing the inter-model symbol resolution. These methods are described in Section 4.3.4.

Support for Traversal with Visitors

Scopes can be traversed with the visitor infrastructure that MontiCore generates for each language. The default traversal strategy that the generated traverser of the visitor infrastructure implements visits all local symbols of a scope [HKR21]. Subscopes of a scope are not traversed. Each scope contains an `accept(..)` method that enables accepting a passed traverser for traversing the scope. `accept(..)` methods of scopes realize double dispatching in collaboration with the `handle(..)` methods of traversers. Artifact scopes have an individual implementation of the `accept(..)` method to enable handling the traversal of artifact scopes and the traversal of other scopes individually. Global scopes do not have an `accept(..)` method, as these are currently never traversed completely.

4.3.3 Implementation of Artifact Scopes in MontiCore

The implementation of artifact scopes follows the concept described in Section 4.1.4. Similar to other scopes, the language-agnostic method signatures are contained in the runtime-environment interface `IArtifactScope`. The generated language-specific artifact scope interface, such as the interface `IAutomataArtifactScope` for the automata language, contains method signatures with language-specific arguments or return

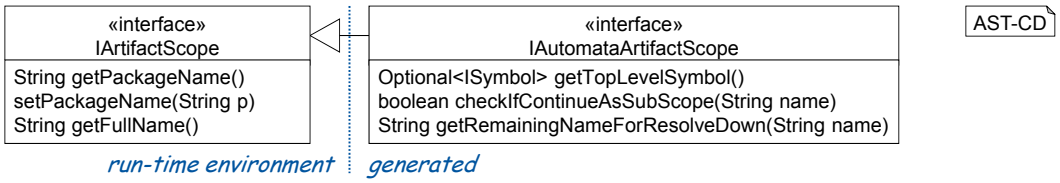


Figure 4.31: Artifact scope methods by the example of the automata language

types. Methods that do not require direct access to any attributes are realized as default methods in interfaces. The artifact scope class, *e.g.,* `AutomataArtifactScope`, is also generated for each language and contains an attribute for the language's package declaration. Other scope attributes and method implementations are reused from the scope class that the artifact scope class extends (*cf.* Figure 4.26). Contrary to the name of usual scopes, the name of an artifact scope object is not derived based on the symbol that spans the scope. Instead, the name of an artifact scope should match the name of the model artifact. In Java-based languages, the name of an artifact scope is usually equal to the name of the top-level symbol if a unique top-level symbol exists. The STI supports this behavior by default. To realize this, the scope class overrides the usual `getName()` method of scopes and calculates the name accordingly if no name has been set explicitly.

Artifact scope interfaces contain other methods to preserve potential customizations of the methods for language composition. These methods are depicted in Figure 4.31 and shortly introduced in the following:

`getPackageName()` and `setPackageName(..)` are methods for obtaining and setting the package of the artifact scope as a String. The signatures of the methods are defined in the interface `IArtifactScope`, and the implementations are located in the artifact scope class since these require direct access to the corresponding attribute.

`getFullName()` is a method with default implementation in the `IArtifactScope`. It calculates the full name of the artifact scope from the package declaration if it is non-empty and the simple name of the artifact scope.

`getTopLevelSymbol()` is a method with a default implementation in the language-specific artifact scope interface. The method obtains the top-level symbol if an artifact scope contains a unique top-level symbol. Otherwise, the method returns an empty `Optional`. As symbols of different kinds can be top-level symbols, the return type is `ISymbol`.

`checkIfContinueAsSubScope(..)` overrides the method of the language-specific scope interface to realize artifact scope-specific behavior. The method returns *true* if

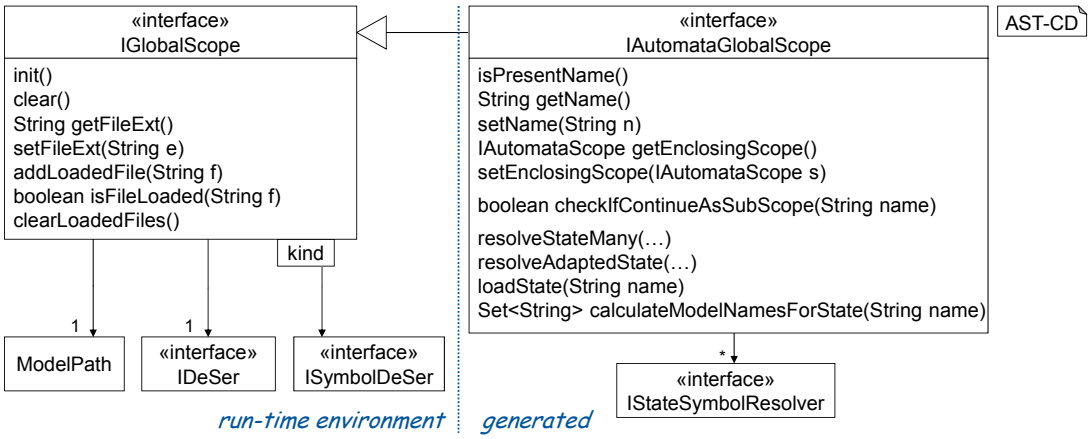


Figure 4.32: Global scope methods by the example of the automata language

the artifact scope exports symbols and the beginning of the resolved name matches the package of the artifact scope. Otherwise, it returns $false$.

`getRemainingNameForResolveDown(..)` overrides the method of the language-specific scope interface to realize artifact scope-specific behavior. The method performs the separation of name parts during top-down intra-model resolution as described in Section 4.1.8. If the package of an artifact scope is not empty, the package is cut off from the resolved name for the remaining symbol resolution. Artifact scopes handle the special behavior of Java-style languages in which an artifact typically has the same name as the top-level symbol. To this end, the `getRemainingNameForResolveDown(..)` method returns two candidates for remaining names for which to resolve. One candidate includes the name of the artifact scope at the beginning of the resolved name to resolve for symbols that have this name in the current scope. The other candidate cuts the first name part off of the resolved name to include the case that no such symbol exists, and the name part identifies the artifact scope.

4.3.4 Implementation of Global Scopes in MontiCore

The implementation of global scopes follows the concept described in Section 4.1.5. Similar to artifact scopes, the realization of global scopes is divided into a language-agnostic runtime-environment interface `IGlobalScope`, a generated, language-specific global scope interface and a generated, language-specific global scope class (*cf.* Figure 4.26). Global scopes are singletons whose instance is managed through the language's mill.

The interfaces of global scopes contain the signatures for access and manipulation methods of attributes to enable their use within default implementations of other interface methods. This includes such methods for the attribute `modelPath`, the language's

deSer, and the map of symbolDeSers. The map maps symbol kinds in the form of Strings to implementations of the interface ISymbolDeSer. Global scopes contain the realization of inter-model resolution, including attributes and methods for realizing loading and storing of symbol tables. These parts of the global scope, such as the language's DeSer and the symbol DeSers, are explained in more detail in Chapter 5.

Language-specific global scope interfaces further offer access and manipulation methods for all individual resolver interfaces (*cf.* Section 4.3.5) of a language. Again, the actual attribute and the implementation of the methods are located in the global scope class. The methods of the global scope interfaces are depicted in Figure 4.32 and explained in the following:

init() is a method for initializing the DeSer and the symbol DeSers of a global scope with default values. This method is called by the global scope constructor and as part of clearing a global scope. The global scope class contains the implementation of the method.

clear() is a method for resetting all reconfigured attributes to default values. This includes that the method empties all collection attributes, such as the resolvers, the model path entries, the loaded files, and any local symbols. The method implementation is contained in the global scope class.

getFileExt() and setFileExt(..) are methods for accessing and modifying the fileExt attribute that realizes the regular expression for file extensions of files to consider for loading stored symbol tables (*cf.* Section 4.1.5). The method implementations are contained in the global scope class.

addLoadedFile(..), isFileLoaded(..), and clearLoadedFiles() are methods for accessing and manipulating the value of the list of files that have been considered for loading symbol tables (*cf.* Chapter 5). The files are represented as Strings. The method implementations are contained in the global scope class.

getName(), isPresentName(), and setName() are methods for accessing and manipulating the value of the optional name of a scope. The global scope overrides these methods inherited from scopes, as global scopes must not have a name. Hence, the getter and setter methods yield an error, and the isPresentName() method always returns $false$. The method implementations are contained in the language-specific global scope interface. They do not actually access the attributes and, hence, do not need to be defined in the global scope class.

getEnclosingScope() and setEnclosingScope(..) are methods inherited from scope interfaces with the purpose of accessing and manipulating the enclosing scope. As global scopes must not have an enclosing scope, these methods are overridden and yield

an error if they are called. The methods do not access any attributes and, hence, are defined in the language-specific global scope interfaces.

`checkIfContinueAsSubScope(..)` is a method inherited from the scope that usually is employed during top-down intra-model resolution. As global scopes are not subscopes of any other scopes, the implementation of this method in the language-specific global scope interface always returns $false$.

`resolveStateMany(..)` inherits from the scope interface with a default implementation in the language-specific global scope interface. The global scope method overrides the behavior of scopes that realizes bottom-up intra-model resolution for state symbols in this method and instead performs inter-model resolution as described in Section 4.1.8.

`resolveAdaptedState(..)` has a default implementation in the language-specific global scope interface. It searches for adapted state symbols with the `IStateSymbolResolver` implementations configured to be used by the global scope.

`loadState(..)` is a method with a signature defined in language-specific global scope interfaces and an implementation contained in global scope classes. It is used in the realization of the inter-model resolution. The method attempts to load an artifact scope that contains a state symbol with the qualified name passed as an argument. Internally, the method relies on the `calculateModelNamesForState(..)` method and the deserialization infrastructure of the STI.

`calculateModelNamesForState(..)` is a method with a default implementation in the language-specific global scope interface that calculates candidates for model names that may contain a state symbol. To do so, the method obtains a qualified name as an argument. Any qualifiers of the qualified name that may identify model names should be returned as candidates by this method. As this can vary for each symbol kind in each language, the implementation of the method, by default, considers only the entire qualifier of a passed qualified name as a candidate for identifying the model name. Consequently, only symbols that are directly located in the artifact scope can be found during inter-model resolution. To change this, the method has to be overridden, as explained in Section 4.3.6.

4.3.5 Implementation of Symbol Resolvers in MontiCore

Symbol resolvers enable integrating resolution for foreign symbol kinds into the inter-model resolution. To prepare the language infrastructure for reconfiguration, the STI contains a resolver interface for each symbol kind of a language. The global scope manages a list of each resolver interface to integrate these into the actual resolution for the corresponding symbol kind. Language engineers can realize a class that implements a

generated symbol resolver interface. Resolvers are mainly relevant for language composition and, hence, their purpose is explained in more detail in Section 6.2.2.

The generated symbol resolver interfaces contain a single method signature for integrating the adaptation for the symbol resolution strategy. For state symbols, the resolver interface has the name `IStateSymbolResolver`, and the abstract method is called `resolveAdaptedStateSymbol(..)`. The method returns a list of state symbols.

4.3.6 Customizing Symbol Resolution

The symbol resolution algorithms described in Section 4.1.8 provide default solutions for searching for the (symbol) definition of a given name and a given kind. These solutions are encoded in the method implementations of generated scope interfaces as described in Section 4.3.2. The default solutions are suitable for various languages, but in general, languages may handle symbol resolution with strategies that deviate from the default solutions. To this end, the generated resolve algorithms are completely customizable. All methods that constitute parts of the resolution algorithm can be overridden by applying the TOP mechanism to the surrounding scope interfaces.

The generated inter-model symbol resolution is based on the assumption that symbols located in foreign artifacts are located in one of two expected locations. Either they must be located directly in the artifact scope, *i.e.,* it is assumed that a foreign artifact scope does not have subscopes that export symbols. Alternatively, the symbols must be located in a scope spanned by a symbol in the artifact scope that has the same name as the artifact scope. The latter is a common case in Java-style languages. The resolution algorithm uses only the complete qualifier and the entire name that is resolved for as candidates for the name of a model that contains the symbols. The method `calculateModelNamesForState(..)` of the automata language, for instance, calculates names of automaton models that may contain a `StateSymbol`. As the method is located in an interface, its behavior is realized as a default implementation. For a given qualified name of a state `material.H2O.liquid`, the calculated candidates for qualified model names are `material.H2O.liquid` and `material.H2O`. In this example, `material` refers to the package of the automaton model `H2O` and `liquid` is the state's name. Symbol resolution, hence, would find the symbol in the artifact scope for the model `material.H2O`.

A language engineer realizing an automata language with hierarchical states may want to export state symbols with a hierarchical namespace. In this example, a qualified name `material.H2O.liquid.compressed` may refer to an automaton model `H2O` located in a package `material` that defines a hierarchical state `liquid` with a substate `compressed`. The default symbol resolution would not be able to resolve this symbol through inter-model resolution, as the model name is neither equal to the qualified name that is resolved for nor equal to its entire qualifier. Instead, `material.H2O` identifies the model containing the searched symbol. Language en-

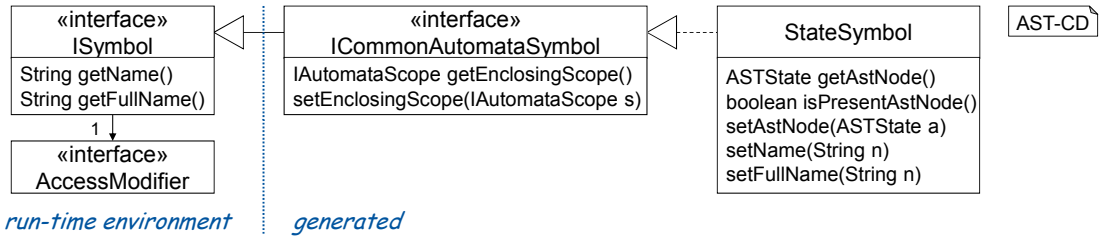


Figure 4.33: Methods of symbol classes by the example of the class `StateSymbol`

gineers can extend the interface `IAutomataGlobalScope` to override the method `calculateModelNamesForState(..)` and adjust its behavior such that it calculates candidates for model names differently. One solution for adjusting the behavior such that the searched state symbols in hierarchical namespaces are found during inter-model resolution is to consider all qualifier prefixes, *i.e.,* `material.H2O.liquid`, `material.H2O.liquid`, `material.H2O`, and `material`.

4.3.7 Realization of Symbols in `Symbol` Classes

As described in Section 4.1.2, the STI provides a generated symbol class per symbol kind of a language and a common, generated symbol interface for each language. Additionally, the MontiCore runtime-environment contains an interface `ISymbol` that provides signatures for language-agnostic methods and enables basic handling of symbols without being aware of their concrete kind, *i.e.,* the type of the corresponding Java class. The `ISymbol` interface has getter and setter methods for an access modifier of a symbol whose implementations are contained in the generated symbol classes. The access modifier of symbols corresponds with the access modifiers of scopes to realize sophisticated visibility concepts. The realization of this reuses the implementation of access modifiers of the SMI [MSN17] and hence, is not detailed in this thesis. Figure 4.33 depicts the symbol interfaces and classes with an excerpt of relevant methods that are explained in the following:

`getName()` and `setName(..)` are accessor and mutator methods for the (simple) name of a symbol that is internally represented as a String attribute in the symbol class. The method signature of the getter method is defined in the interface `ISymbol` to enable accessing a symbol's name without the requirement of being aware of its concrete kind (*i.e.,* symbol class). There is no signature for the setter method in the interface to prohibit altering the name of a symbol without being aware of its concrete kind.

`getFullName()` and `setFullName(..)` are methods for accessing and modifying the full name of a method. The full name of a symbol is derived from the enclosing scopes of the symbol as described in Section 4.1.6. Similar to the methods for the `name`

attribute, the signature of the getter method is contained in the interface `ISymbol`, and both method implementations are located in the symbol class.

`getEnclosingScope()` and `setEnclosingScope(..)` are methods for accessing and modifying the scope that encloses a symbol. The enclosing scope of a symbol is always present if the symbol table has been completely instantiated. The methods are located in the language-specific symbol interface in which the type of the language-specific scope interface is available.

`getAstNode()`, `setAstNode(..)`, and `isPresentAstNode()` access and modify the AST node that corresponds to the symbol. The AST node of a symbol may not be set if the symbol table has been loaded from a stored symbol table rather than created from the AST. Hence, the method `isPresentAstNode()` checks whether the AST node is available. If the AST node is unavailable, the method `getAstNode()` yields an error. The enclosing scope of a symbol is always present if the symbol table has been completely instantiated.

Symbol classes further have an `accept(..)` method to be traversable with the visitor infrastructure. However, the default traversal strategy does not visit anything other than the symbol itself, even if the symbol spans a scope. The default traversal does not visit spanned scopes, as it is intended to be based on the AST. Traversing the AST and the scope tree simultaneously can lead to multiple visits of the same abstract syntax concepts. This can be adjusted by applying the TOP mechanism to the class of the traverser implementation. By overriding the `traverse(..)` methods, language engineers can realize different traversal strategies. Example scenarios are the traversal of a scope tree along the association between a scope and its subscopes or along the association between symbols and spanned scopes.

4.3.8 Instantiating Symbol Tables with Scopes Genitors

The scope genitor is responsible for instantiating symbol and scope objects and populates the initial symbol table structure. It follows the concept described in Section 4.1.10. Scope genitors implement the language's visitor to implement `visit(..)` methods for AST nodes and set their enclosing scopes. If the corresponding nonterminal defines a symbol kind, the implementation of the visit method controls the instantiation of symbol objects and the linking between symbol, AST node, and enclosing scope. If the corresponding nonterminal spans a scope, a new scope object is instantiated, linked with the environment, and added to the stack of scopes managed by the scope genitor.

Scope genitors further implement the handler interface of a language to enable language engineers to override the `handle(..)` and `traverse(..)` methods if the scope genitor is customized with the TOP mechanism. To carry out the actual traversal of the AST, scope genitors employ an object of the language's traverser.

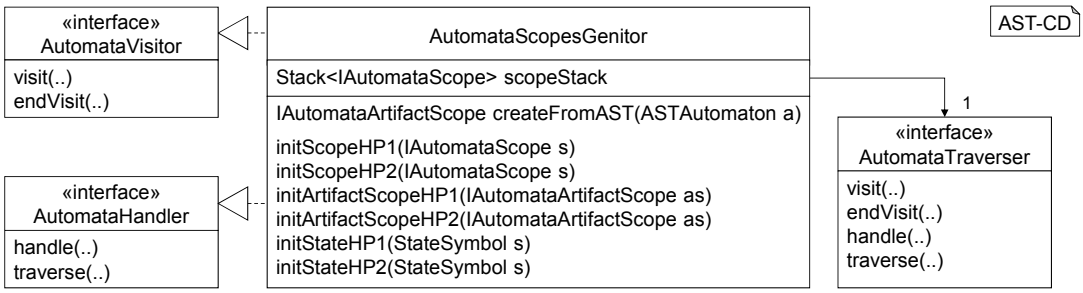


Figure 4.34: Methods of a scope genitor by the example of the class `AutomataScopesGenitor`

Language engineers typically utilize the scope genitor in a language tool via the method `createFromAST(..)`. The method uses the language's traverser to traverse the AST and instantiate and interconnect the objects of the symbol table. This results in an instance of the language's artifact scope that is returned by the method. Afterward, the instantiation of the symbol table is not necessarily complete, as potential subsequent phases of the instantiation may follow. The scope genitor contains several hook point methods whose default implementation is empty to enable handwritten additions for the first phase of symbol table creation. The following shortly explains the central methods of the scope genitor that are depicted in Figure 4.34:

`createFromAST(..)` instantiates the symbol and scope objects based on the AST node passed as an argument and returns an instance of the language's artifact scope. The type of the AST node in the argument is the AST node of the start rule of the corresponding grammar.

`initScopeHP1(..)` is a hook point method for initializing any scope attributes before the content of the scope is traversed. It is called as the last operation of the `visit(..)` methods of scope-spanning nonterminals.

`initScopeHP2(..)` is a hook point method for initializing any scope attributes after the content of the scope is traversed. It is called in the `endVisit(..)` methods of scope-spanning nonterminals.

`initArtifactScopeHP1(..)` is a hook point method for initializing any artifact scope attributes before the content of the scope is traversed. It is called in the `createFromAST(..)` method.

`initArtifactScopeHP2(..)` is a hook point method for initializing any artifact scope attributes after the content of the scope is traversed. It is called at the end of the `createFromAST(..)` method.

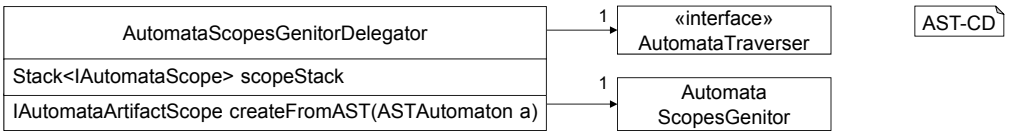


Figure 4.35: Methods of a scope genitor delegator by the example of the class `AutomataScopesGenitorDelegator`

`initStateHP1(..)` is a hook point method for initializing any attributes of a state symbol. It is called at the end of the `visit(..)` methods of nonterminals that define state symbols.

`initStateHP2(..)` is a further hook point method for initializing any attributes of a state symbol. It is called in the `endVisit(..)` methods of nonterminals that define state symbols. For symbol kinds that span a scope, the `HP2` methods are called after the spanned scope has been instantiated.

4.3.9 Instantiating Symbol Tables of Composed Languages with Scopes Genitor Delegators

A scope genitor delegator class is generated for each language. For example, the scope genitor delegator class for the automata language, the `AutomataScopesGenitor-Delegator`, is depicted in Figure 4.35. The class is mainly used for instantiating the symbol tables of composed languages in which a single AST may contain nodes from several languages. The scope genitor delegator delegates the symbol table creation of each node to the scope genitor of the corresponding language. Language engineers can use the scope genitor delegator via the `createFromAST(..)` method with the same signature as the method in the scope genitor class. Internally, the scope genitor delegator uses a traverser configured to use the individual scope genitors. As the nodes of different languages may be interwoven, all scope genitors share a mutual stack of scopes.

4.4 Discussion

Instead of realizing kind hierarchies by integrating the symbol resolution for subkinds into the local symbol resolution algorithm as described in Section 4.1.8, it would have been possible to manage hierarchical symbol kinds by adding the same symbol object to different symbol maps in a scope. For instance, if a symbol kind i extends a symbol kind j that extends a symbol kind k, a symbol object of the kind i could be added to the individual maps of symbols of kind j and k. This, however, could lead to inconsistencies if manipulation of the maps was not applied consequently. As scopes, in general, should be fully customizable, such inconsistencies could not be prevented. Furthermore, loading

and storing the symbol tables would have to consider the object identity of the same symbols in different maps.

The top-down intra-model symbol resolution that follows the inter-model resolution can be omitted if symbols are loaded directly into the global scope. This is different from the current solution for symbol table persistence (*cf.* Chapter 5) that re-establishes scope objects while loading an artifact scope. While loading symbols directly into the global scope could be more efficient, the symbol table infrastructure of the symbols would be incomplete as the associations with scope objects would not be present. These associations, however, are crucial for proper symbol resolution in some languages. For example, resolving the symbol of a method in an object-oriented programming language requires information about method arguments and return types, which are usually located within the scope of a method. Furthermore, re-establishing the scope structure enables language engineers to customize the symbol table, *e.g.,* by integrating transformation operations on the symbol table into the activities of processing a model.

Typically, languages define symbol kinds, and models that conform to the language define symbols that conform to the symbol kinds of the language. Sometimes, languages may provide symbols themselves. For example, symbols for built-in types, such as the symbol `int` in Java, are symbols that should be available in every model of the language. For such symbols, it is useful if the language itself defines the symbols such that these are not required to be defined in each model. We recommend realizing this by adding such built-in symbols directly to the global scope. The case that models define new symbol kinds is rare. However, languages can use annotations or stereotypes that make the language extensible with novel language concepts at the level of models [BHH+17]. These, however, are limited in the freedom of syntax design, and new language concepts must be reducible to the language elements that are annotated.

The relationships between AST, symbols, and scopes, as presented in Section 4.1.1, consider that the AST may not be available since symbols and scopes can be loaded from persisted symbol tables instead of being created from the AST. This is reflected in the cardinality of the associations from scopes and symbols to the AST that is 0..1. However, the fact that the symbol table is not yet created from the AST and only the AST is present is not reflected in the relationships. Hence, associations from AST to enclosing scopes have a cardinality of 1. This is not reflected in the cardinalities of the relationships because symbol tables should always be created in the next or in a consecutive step after parsing and before the actual use of an AST. Most operations (type checks, well-formedness rules, code generation, etc.) are executed after the symbol table for a model has been created.

It is a deliberate decision that symbol-defining language elements have to be identifiable via a name that is indicated explicitly in the model. There are possibilities that language elements may be identifiable via derived names or by the position in the AST, such as transitions of automata. For example, a transition from the state `solid` to the state `liquid` with the stimulus `melt` could be identified within the automaton by

the derived name `solid_melt_liquid`. However, such derived names rarely identify definitions that are used via their name by other language elements. In language design, it is a good practice to make modelers decide for identifier names they use in other parts of the same or a different model. Derived names, on the other hand, can be unintuitive or unnecessarily complex. Nevertheless, it is possible (but not recommended) to add derived names as symbols in symbol tables.

4.5 Related Work

The design of the kind-typed symbol table infrastructure as presented in this thesis relies on the symbol management infrastructure [MSN17] (SMI). The STI borrows several concepts from the SMI, such as access modifiers and predicates during symbol resolution or the presence and purposes of artifact scopes and global scopes. A significant difference between the two approaches is that the SMI in large parts uses language-agnostic infrastructure. For instance, the STI contains language-specific scope classes and interfaces, while the SMI uses the class `CommonScope` for all languages. This makes language-specific customization more challenging to apply, as the TOP mechanism cannot be employed individually.

The symbol resolution algorithm of the SMI obtains the symbol kinds as an argument, while the resolution strategy presented in this thesis uses different resolution algorithms for each symbol kind, as the symbol kind is encoded into the infrastructure realizing the algorithms. While both the SMI and our approach distinguish bottom-up and top-down *intra* model resolution, the SMI further distinguishes between bottom-up and top-down *inter* model resolution. The reason for this is that the SMI has a built-in strategy for qualifying symbol names during bottom-up inter-model resolution.

In the SMI, names are qualified during each symbol resolution. In our approach, names are not qualified during the resolution but instead have to be qualified as part of the type check (*cf.* Section 4.1.10). The benefit of this is that it usually suffices to perform the qualification once instead of each time a symbol is resolved. In the STI, the type check can be executed as part of the symbol table instantiation. It connects a symbol usage to the correct symbol definition by considering that an unqualified name of the symbol usage may need to be qualified with import statements or the package declaration. This can require manual effort to qualify symbols correctly. The SMI uses symbol references for name usages that resolve for a suitable symbol definition on demand. Realizing type checks in the SMI typically relies on context conditions to check the success of the resolution for all symbol references of a model.

The symbol table creation of the STI uses scope genitors and a phased symbol table creation process. The symbol table creators of the SMI are similar to scope genitors, as both are realized as classes implementing the visitor interface to traverse the AST of a model. However, the SMI instantiates a `SymbolReference` for every usage of a symbol.

As the type checks in the SMI are typically carried out within context conditions that are checked after the symbol table is created, the lazy loading of symbol definitions in symbol references yields correct results. However, symbol references have the disadvantage that an explicit check for the existence of the symbol is either performed multiple times (*i.e.,* on every query for obtaining the symbol definition) or may lead to inconsistencies if the symbol definition is cached locally. Furthermore, most symbol usages do not require usage-specific information leading to an unnecessarily large number of objects for symbol references.

The SMI does not foresee the loading and storing symbol tables. Hence, symbol resolution uses model loaders to load symbol tables of other models by parsing these, and creating the symbol table of the resulting AST. Under the (realistic) assumption that instantiating the symbol table from persisted symbol table files requires less time than parsing the model and instantiating the symbol table from the AST afterward, model loaders are less efficient.

Another approach for realizing symbol tables for MontiCore languages [Völ11] is based on namespaces instead of scopes. A namespace in this approach may refer to different symbol tables that contain symbol table entries. The latter correspond to symbols in the SMI and the STI. While the SMI and the STI rely on access modifiers to constrain the visibility of symbols, the symbol table entries in the approach described in [Völ11] can be contained in different symbol tables that are part of the same namespace. A namespace may import, export, forward, and encapsulate symbol tables. Furthermore, a symbol table entry can be contained in more than a single symbol table. Implementing and using the symbol table infrastructure with the approach in [Völ11] is more complex than with the STI. In the former, language engineers have to handle different symbol tables for a model, and the data structure in the STI is less intricate.

The Spoofax [KV10] language workbench uses a name binding language [KKWV12] for describing type systems realizing type checks. A name binding specification comprises declarative rules, where each rule starts with a pattern that is matched against the abstract syntax of a model. The name binding languages distinguishes different kinds of names via different namespaces. In contrast to the STI, which distinguishes different kinds of names via different symbol kinds, the concept of namespaces is optional. If no explicit namespaces are declared, all name definitions are located in the same namespace [www21d]. Rules in the name binding language can also introduce scopes that influence the visibility of name definitions. A name binding assigns a unique qualified identifier to the definition of a name. Name usages are assigned a qualified identifier as well. All name bindings of a project are contained in a single hash table, in which each name binding may contain additional information. This is in contrast to the STI, in which each scope manages the locally defined symbols individually.

When resolving for name definitions, the Spoofax approach relies on three phases. In the first phase, name definitions and references are gathered and assigned (preliminary) qualified identifiers. In the second phase, name definitions are analyzed and assigned

further information. The third phase completes the name references by potentially updating their qualified names to point to the correct name definition.

Chapter 5

Infrastructure for Loading and Storing Symbol Tables

As stated in Chapter 4, the symbol table of a model can be seen as a description of its interface that can be used for correctness checking and for including a model in a greater model composition. In other words, the symbol table of a model contains all information that other models require to check their correct usage. Therefore, language-processing infrastructure has to be able to access the symbol tables of other models efficiently. Instead of parsing these models again, modern programming languages, such as Java rely on symbol tables that are persisted in symbol table files. In the case of Java, the symbol information is serialized and persisted as part of the `class` files [www21c]. This enables

- efficient model processing, as the information required for correctness checking is only a part of the information contained in a model.
- persisting the symbol table in a form that can be loaded faster than loading the entire model.
- efficient type checking because the information for type checking in the symbol table has itself undergone a type-checking phase.

In Java, the location of class files and the qualified names of imported classes correlate, which fosters efficient identification of the storage location of a symbol table. However, as the level of detail of type information depends on the purpose for which it is required, the suitable form of abstraction is often language-specific. In fact, for correctness checking between model artifacts, it suffices to store symbol table information of an artifact that is visible to other artifacts. For different purposes, however, it may be viable for languages to store symbol tables with more information than is directly necessary for correctness checking.

In MDD [VSB+13], models are central software engineering artifacts and, thus, contain (large parts of) the intellectual property of the software. However, for integrating a model with other models or other software in general, it is usually not required to be aware of all intricacies of a foreign model. Instead, it suffices to communicate an interface of

the model that describes its abilities for integration and documents the model without internal details. Internal parts of the model, on the other hand, are not required for this integration and can be hidden to keep the intellectual properties private. As symbol tables contain the information that other models require to check their type correctness, stored symbol tables together with a documentation of the model's interface can provide an interface for accessing a model or model library that does not contain the model internals.

Persisted symbol tables foster efficient language aggregation under the assumption that adapting between stored symbol tables is more efficient than adapting between symbol tables on the level of Java objects. Consequently, the adaption between symbol kinds can be carried out by translating persisted symbol table files. Adaptations via persisted symbol tables are efficient if symbol tables are persisted in a notation that can be processed quickly. Various suggestions for realizing suitable symbol adapters are described in Section 6.2.

With persisted symbols tables, the infrastructure that stores symbol tables can be decoupled from the infrastructure that loads the symbol tables. This enables realizing an exchange format for sharing information about models between different variants or different versions of a language under the assumption that the exchanged data format remains compatible.

In language engineering, the symbol table of a model, among other things, has the purpose of acting as its interface for language composition. Persisting symbol tables can support language extension in two scenarios: if a symbol table of a model conforming to a language L is stored, it should be possible to load it as a symbol table of a language M that extends L. On the other hand, it should be possible to load a symbol table stored in the symbol table data structure of language M with the symbol table infrastructure of language L. The abilities described by the two scenarios foster the reusability of persisted symbol tables and avoid unnecessary translations of symbol tables. In summary, persisting symbol tables enables the

- efficient correctness checking between models,
- encapsulation of model internals,
- realization of efficient symbol adapters for language aggregation, and
- decoupling of language tools.

As the information captured in symbol tables is language-specific, the loading and storing of symbol tables has to be realized for each language individually. However, most parts of the infrastructure for loading and storing symbol tables can be synthesized with a code generator.

In the following, Section 5.1 introduces terminology and concepts for realizing serialization and deserialization in general. Section 5.2 explains the concept for realizing the

(de)serialization of MontiCore symbol tables. Section 5.3 describes a JavaScript object notation (JSON) infrastructure [www17, www20c] for realizing serialization and deserialization before Section 5.4 explains the implementation of symbol table persistence in MontiCore centered around `DeSer` classes, and Section 5.5 gives examples for possible customizations of the implementation. Section 5.6 discusses central design decisions, and Section 5.7 explains related approaches.

5.1 Serialization in General

The concept for realizing type-safe serialization and deserialization of MontiCore symbol tables relies on central definitions that are explained in the following.

5.1.1 Serialization and Deserialization

Various applications require that objects (in the context of object-oriented programming) are encoded in a persistable or transmittable representation. For example, distributed systems such as web applications require encoding of objects to communicate these between physically distributed systems. Database management systems for object-oriented programming require encoding of objects that can be stored in persistable memory. A typical agreement for *persistable or transmittable representations* is that these are realized as a sequence of characters. For brevity, we use the term *String* for such character sequences in the following interchangeably. The process of encoding an object into a String is referred to as *serialization*, and *deserialization* is the process of decoding a serialized object.

Definition 4 (Serialization and Deserialization)**.** *Serialization is the process of translating an object structure into a storable sequence of characters. Deserialization is the inverse process of translating a serialized object structure, i.e., a sequence of characters, into the original object structure.*

An object can be serialized in a variety of different forms. For example, a Java object representing a blue color that is an instance of a Java class `Color` can be serialized as the String with the symbolic color name `"blue"`, as a String with the tuple of RGB values, such as `"(0,0,204)"`, or as a String containing the hexadecimal number `"#0000CC"`. Deserialization of the respective representation produces an instance of the color type with the same properties as the original object.

The following describes properties of serialization and deserialization in which, for reasons of simplicity, we assume that two operations `serialize` and `deserialize` exist that realize the serialization and the deserialization.

Often, serialization and deserialization are symmetric with regard to the persisted information. Deserializing the result of serializing an object `o`, hence, produces an object with the same visible properties as the original object `o`, which means that `o ==`

`deserialize(serialize(o))`. This property can be leveraged to realize correctness tests for serialization and deserialization strategies that are unaware of the serialized representation. Sometimes, this property is violated on purpose to enable compactness in the serialized representation. For example, empty collections can be omitted during serialization, which renders it impossible to distinguish these from an explicit `null` reference during deserialization. However, this can be feasible as such a distinction is often not required or even undesired. Further, `null` must generally be avoided in good software engineering practice.

Deserialization of links of associations between objects requires that the associations are established correctly. This entails that all association cycles in an object structure must prevail after serialization and deserialization. For instance, a bidirectional association between two objects `a` and `b` may assume that `a == a.b.a`. This must still hold after serialization and deserialization of `a1 = deserialize(serialize(a))`, *i.e.,* `a1.b.a == a1`).

5.1.2 Serialization Strategies

To operationalize the functionality for serialization and deserialization of object structures, procedures for translating between object structures of a particular type and Strings are required.

Definition 5 (Serialization Strategy)**.** *A serialization strategy is an algorithm that realizes serialization and deserialization for objects of a particular type.*

For example, a serialization strategy for the Java type `Color` comprises an algorithm for translating color objects into Strings and another algorithm for translating Strings with an encoded serialized color into instances of the Java type `Color`.

Serialization strategies have to ensure that serialization of bidirectional associations between objects does not produce cyclic nesting of objects. On the other hand, deserialization must establish the bidirectionality of these associations such that each bidirectional association connects precisely two objects. For avoiding cycles in object structures, serialization strategies can use indirection to decouple objects from each other or make use of knowledge about the underlying data structure. Decoupling associated objects through indirection, for example, can be achieved by storing only an identifier of each associated object instead of the object as a whole. Knowledge about the data structure behind object structures can be leveraged to serialize only one direction of bidirectional associations and establish the opposite direction during deserialization. While the solution with indirection is applicable even without knowledge about the data structure, the produced serialized character sequences are longer.

Serialization strategies for attributes comprise either *rigorous* or *robust* deserialization algorithms for the attributes. Robust deserialization ignores serialized data that cannot be processed, while rigorous deserialization produces errors on unrecognizable serialized

data. Robust deserialization can also assume default values for information that a serialized String lacks but is required to build an object. Evolution of the data structure causes fewer problems if the deserialization is robust because additional serialized information does not yield errors. Rigorous deserialization, on the other hand, is more suitable for safety-critical systems, as it is more restrictive and enables better detection of erroneous program behavior.

Whether a realization of deserialization is rigorous or robust affects the serialization strategies for optional members of an object. The serialization strategies for optional members have to consider symmetric serialization and deserialization for the cases that a value is present and that it is absent. Different solutions for serialization strategies of optional attributes exist. For example, the fact that an attribute is optional can be explicated in the serialized character sequence. This enables distinguishing the case that an optional attribute is absent on purpose from the case that an optional attribute is absent due to, *e.g.,* faulty serialization behavior.

Robust deserialization fosters realizing serialization strategies for inheritance in serialized attribute types. Serialization strategies must be able to handle the inheritance of the serialized types. Inheritance of the types can be either handled implicitly or explicitly. If inheritance is handled implicitly, a serialization strategy holds for a type and, implicitly, for all of its subtypes. This is feasible due to the Liskov substitution principle [LW94] in inheritance between types. Explicit handling of inheritance prohibits serialization and deserialization of subtypes of the original type of the serialization strategy.

Persistence of Type Information

We distinguish three alternative solutions for handling type information of objects during serialization and deserialization. The type information is the type to which an object conforms. In Java, the type information of an object is the corresponding class or interface.

Persisted type information. The type of an object can be serialized as explicit information that is part of the serialized String. This type information remains prevalent throughout serialization and deserialization, even if these are executed by different tools, in different contexts, or on different machines. However, the type information can obstruct realizations of rigorous serialization with implicit handling of type inheritance.

Typeless persistence. The type information can be omitted during serialization, and deserialization can reconstruct the type based on the remaining information that is available for the serialized object. Moreover, the type reconstruction can rely on knowledge encoded in a language-specific or domain-specific deserializer. In combination with robust deserialization, as much information of a serialized type as possible can be retrieved. However, this is less suitable for safety-critical systems as the loss of type information during deserialization easily leads to misinterpretation of the type.

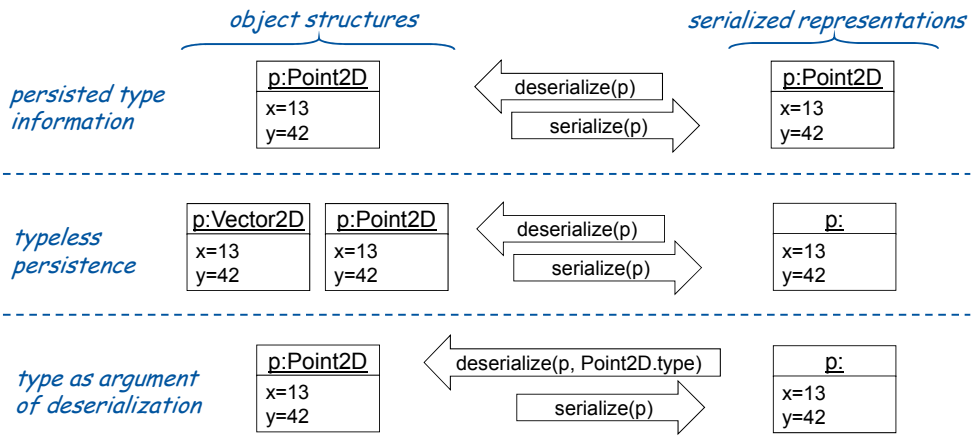


Figure 5.1: Alternative solutions for serializing type information

Type as argument of deserialization. In combination with omitting the type information of a serialized object, the type information can be an argument for deserialization instead of reconstruction of the type from serialized information. This prevents undesired type recognition errors but requires checking of type information during runtime, *i.e.,* the application of reflection. In addition, it prevents the use of sub types in composed or extended languages.

Examples of the three solutions are depicted in Figure 5.1. An object with the name `p` of the type `Point2D` contains two integer members, one with the name `x` and the value `13` and the other with the name `y` and the value `42`. When the type information is persisted, all information is also contained in the serialized form of the object. With typeless persistence, only the name of the object and the members are persisted, but not information about the type. During deserialization, the object can be constructed not only with a serialization strategy for `Point2D` but also with other serialization strategies, such as for the type `Vector2D`, that has members of the same type and name. While this enables flexible reusability of persisted information, it can yield unintended results. This can be prevented by using the desired type, such as the type of `Point2D`[1], as an argument of the deserialization.

For the serialization strategies of MontiCore's symbol table, the proposed approach persists the *kind* of a symbol as an explicit attribute of the persisted symbol object. For scopes, the approach does not persist any type information. During deserialization, the choice of the concrete DeSer determines which symbol or scope types are instantiated. Therefore, the type can be considered an argument of the (configurable) deserialization.

[1] In Java, the type of a class `Point2D` is `Point2D.class`

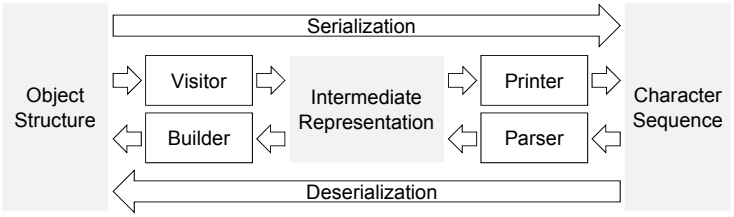


Figure 5.2: Overview of serialization strategy with an intermediate representation

Omission of Default Values

Serialization of sparse object structures, *i.e.,* object structures for which a large part of object members are empty lists or absent optional values, can yield numerous redundant sections. To improve the readability and reduce the complexity of such serialized structures, default values can be omitted during serialization. We distinguish two different approaches for realizing default values for serialization.

Type-specific default values are default values specific to the type of the serialized value. Typical examples are empty Strings, the Boolean value `false`, or empty lists. Default values do not only exist for primitive and built-in types but can also be defined for any other data types. For example, the default for a `Color` object can be the black color. In combination with collections, it is essential that values that are entries in a collection are not omitted in case their value equals the default. Omitting these would lead to a loss of information. Depending on the context, type-specific default values can already capture large parts of redundant characters within a serialized String.

Member-specific default values further improve type-specific default values as the default value of a type can be defined for each object member individually. For example, the default color of an attribute `backgroundColor` can be white, while the default for an attribute `fontColor` is black. This is especially helpful if a type has numerous Boolean flags for which some, by default, should be activated while others are not. However, member-specific default values have to be handled with care in combination with typeless persistence, as the type information determines the default values of the object. Furthermore, member-specific default values harm the readability of serialized objects as external context information about the type is essential.

5.1.3 Serialization with Intermediate Structure

Conceiving a serialization strategy requires realizing both serialization and deserialization. By employing an intermediate representation between the object structure and the serialized String, the knowledge about the serialization format can be decoupled from the knowledge about the data structure behind the object structure. This reduces

the complexity of the serialization infrastructure that is specific to a data structure as the remaining parts, *e.g.,* the printer and parser of the intermediate representation, can be reused independently. Serialization with intermediate structures can rely on well-understood design patterns of object-oriented software engineering [GHJV95], such as builders and visitors. Figure 5.2 depicts the constituents for realizing a serialization strategy with an intermediate structure. During serialization, an object structure (depicted on the left) is translated into an intermediate representation (depicted in the center) by employing a visitor. The visitor traverses the object structure. For each element that should be serialized, it uses a printer to translate the element to a character sequence (depicted on the right). In this, the visitor requires only knowledge about the structure of the intermediate representation and not about the concrete syntax of the format in which the object structure is serialized. The printer, on the other hand, translates elements of the intermediate structure into character sequences without knowledge of the data structure of the underlying serialization strategy. During deserialization, a parser processes the character sequence and instantiates the intermediate representation. Builders for the data types of the serialization strategy can be employed to instantiate objects systematically. Again, parsing into the intermediate structure does not require knowledge about the data structure, and building up the object structure from the intermediate representation does not require knowledge about the serialization format.

5.2 Concept for Symbol Table Persistence

The concept for persisting symbol tables comprises a mechanism for loading and storing symbol tables that has to be integrated into the process of processing of models. Besides this, it requires a concept for organizing serialized symbol tables in terms of artifacts of the file system. At the core of loading and storing symbol tables is the serialization and deserialization of symbol tables. The concept further requires means to adjust the serialization infrastructure through language-specific customization.

5.2.1 Overview of Symbol Table Persistence

In MontiCore, models are the central development artifacts of a language and are individual units of reuse in the context of model-driven software engineering. In the symbol table infrastructure of MontiCore, each model is associated with a corresponding instance of an artifact scope (*cf.* Chapter 4). When persisting symbol tables, it is helpful to realize the modularity of files containing stored symbol tables in the same way as the modularity induced by the model artifacts. To this effect, a symbol table file contains the symbol table information for precisely one model artifact, *i.e.,* it contains a serialized artifact scope including its internal structure in terms of subscopes and contained symbols. A central global scope manages a list of all known artifact scopes that may include artifact scopes that have been established as subscopes of the global scope by

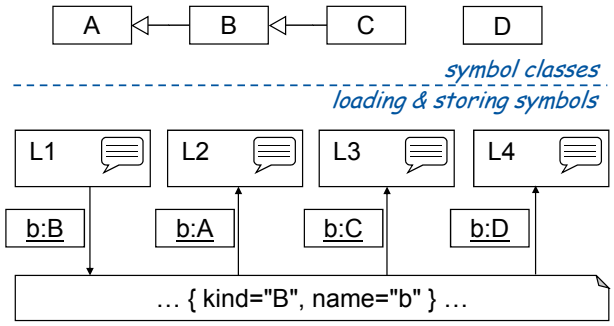


Figure 5.3: Reusing a symbol stored as kind `B` by loading it as a symbol of a super kind `A`, a subkind `C`, or an unrelated kind `D`.

loading a file that contains a stored symbol table. Global scopes, consequently, are not persisted.

Loading stored symbol tables is integrated into the inter-model symbol resolution algorithm (*cf.* Section 4.1.8), for which MontiCore generates the infrastructure. Storing symbol tables, however, has to be integrated into the tool that orchestrates model processing, which is handcrafted. Undeniably, storing symbol tables requires that these have been instantiated before. However, it is not useful to store ill-formed symbol tables. To this effect, the context conditions ensuring the well-formedness of a model should be executed before an artifact scope is stored.

Methods for loading and storing symbol tables are realized within the `Symbols2Json` classes described in Section 5.2.3. However, these methods can only load the content of a file as a String and print a String to a file. The content of the String is calculated and processed by `DeSers` that realize serialization strategies and are explained in Section 5.2.3.

A central decision in the concept for symbol table persistence is that after loading a stored symbol table, the object structure of the artifact scope shall be instantiated again. Alternatively, it would have been possible to store only the symbols and omit storing scopes. This is a feasible alternative because under the assumption that a loaded artifact scope must not be modified (which would render it potentially ill-formed), the scopes are not required. Instead, the visibility of symbols can be pre-calculated before symbol tables are persisted since the visibility is always considered only from viewpoints that are external with respect to the artifact scope. Nevertheless, the concept presented in this thesis deliberately stores the object structure of scopes for several reasons:

- Reconstructing scope trees from loaded symbol tables integrates the scope structure better with the remaining abstract syntax infrastructure. Since scopes and symbols in MontiCore are associated with each other in various forms, re-establishing the associations between these yields a more complete abstract syntax object structure.

In an incomplete object structure, analyses may be required to distinguish between associations that have not been loaded from associations that do not exist.

- Persisting the scope structure yields the option to use the information about scopes and the symbols that span these scopes, inter alia, for symbol adaptation during language composition. If, instead, only symbols are stored, the information about scopes is lost. This includes the information which symbols are contained in scopes spanned by other symbols.
- Loading symbols as part of reconstructed artifact scopes enables a less invasive integration of symbol table persistence into the symbol resolution algorithm.

The serialization of MontiCore symbol tables is realized as a robust serialization (*cf.* Section 5.1) to allow for reusability of persisted symbol tables via language composition. This form of reusability is depicted by example in Figure 5.3. The example uses four symbol kinds `A`, `B`, `C`, and `D` where `A` is a super type of `B` and `B` is a super type of `C`. The symbol kind `D` is not related to the other symbol kinds through any type hierarchy. The tool of a language `L1` stores a symbol `b` with the symbol kind `B`. Other languages, such as the language `L2`, should be able to load the symbol `b` as a super kind `A` of the original kind `B`. Attributes of `b` that are specific to the kind `B` and not available in symbols of kind `A` are, thus, simply omitted during deserialization. Furthermore, it should be possible for other languages to deserialize the symbol `b` in a limited form as the subkind `C`. In this case, the deserialization has to assume default values for attributes required for instantiating a symbol of kind `C` that are not contained in the symbol stored as kind `B`. In addition to that, the symbol `b` should be deserializable as a symbol of kind `D` that is neither sub nor super kind of `B`. In this case, attributes are deserialized whenever available, redundant attributes in the serialized symbol are ignored, and missing attributes have to be deserialized with default values. All three forms can be achieved by reconfiguring the deserialization through customization of the global scope as described in Section 5.2.3.

5.2.2 Organization of Persisted Files

The absolute path of each persisted symbol file comprises four parts. Only all four parts make a symbol file uniquely identifiable. The organization of symbol files is inspired by the organization of class files in Java [www21b].

Symbol path: A symbol path is the path from the root of the file system to the directory in which the symbols are located. This is conceptually similar to entries of the classpath for Java class files and entries of the model path for models of MontiCore languages. From the viewpoint of a language tool that stores symbol tables, the symbol (output) path is the output directory for stored symbols. From the viewpoint of a tool that

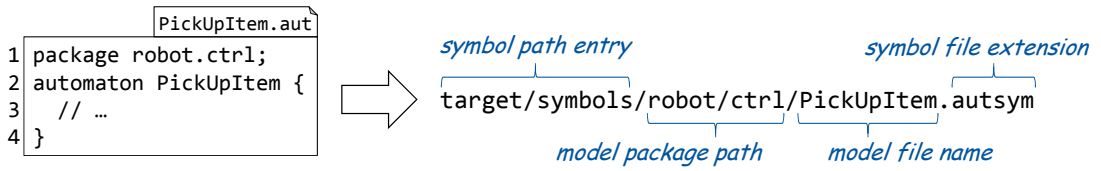


Figure 5.4: Organization of symbol files

loads stored symbol tables, a symbol (input) path is the location that is searched for stored symbols. Internally, the symbol path may be stated relative to a project root (*cf.* Chapter 7) to enable its machine-independent specification. Whether this is the case or not does not influence the concept for persisting symbol tables.

Package qualifier: The package qualifier is a folder structure that reflects the name parts of a qualified package path in the same way as Java. The package statements in Java-style languages are typically qualified names where a dot separates the name parts. The package qualifier path is derived from the qualified name by replacing each dot with a file separator. The purpose of package qualifiers is to realize a hierarchical namespace for symbol table files that can be efficiently searched through a traversal of the file system.

Filename: The name of a symbol file is equal to the name of the artifact scope that it contains, which in turn should be equal to the name of the model for which this artifact scope has been created. In Java-style languages, the name of the artifact scope typically is also equal to the name of a top-level symbol within the model.

File extension: The file extension of a symbol file indicates the language of the stored symbol table. With language-specific file extensions, the file system can distinguish symbol tables for models with the same name that conform to different languages. In MDD, this is a common case as different aspects of a system are modeled with different modeling techniques, *i.e.,* with different modeling languages. For example, the software architecture of a car can be modeled in an architecture model `Car.arc`, whereas the variability is modeled in a feature diagram `Car.fd`. For MontiCore symbol files, the file extension is composed of the language-specific file extension of the models of the language with the suffix `sym` to indicate that the file contains symbol tables.

An example of the four parts of a symbol file path is depicted by example in Figure 5.4. The example demonstrates the absolute file path for the automaton model `PickUpItem` depicted on the left. The model is located in the package `robot.ctrl`. To this end, the model is contained in a model file `PickUpItem.aut` that is located in the file path `robot/ctrl` as described by the package. This path, again, is relative to an absolute location in the file system that contains the models of the example application.

The persisted symbol table file, in this example, is contained in a symbol path entry `target/symbols` that is relative to some absolute location in the file system that is of no further importance for this scenario. Relative to the symbol path entry, the folder structure induced by the package is re-created. In this location, the symbol table is stored in a file with the same filename as the model and the file extension of the language with the suffix `sym`, *i.e.,* to the file `PickUpItem.autsym`.

When loading MontiCore symbol tables, a distinction between symbol paths and model paths often is not required. For efficiency, MontiCore languages typically load symbol tables of other models during symbol resolution rather than parsing these and creating the symbols from the ASTs. However, loading the model instead of the symbol table can be useful for some languages, *e.g.,* if more information about the loaded models than contained in the symbol table is required. However, enabling a language to load both symbol tables and models should usually be avoided to reduce the risk of inconsistencies. If both models and symbol tables should be considered for loading, separating model path and symbol path improves the performance of loading with the cost of additional management of individual paths in language tools.

To this end, the paths in which a language searches for files to load contains either models or symbols. Due to historical reasons, MontiCore languages make use of a model path. With the introduction of persisted symbol tables, we avoided introducing novel infrastructure for symbol paths because either model path or symbol path would be empty. Instead, the model path is reused to bundle entries that contain symbol files. In the following, we use the term model path interchangeably with the term symbol path. This is in line with the Java compiler [www21b] that uses the classpath mainly for `class` files but, by default, also for searching `java` source files.

5.2.3 Concept for Symbol Table Serialization and Deserialization

The concept for serializing and deserializing symbol tables is centered around a mechanism for serialization with an intermediate structure (*cf.* Section 5.1.3). The serialized symbol tables should be compact, machine-processable, and human-readable. The realization of serializing symbol tables in MontiCore relies on JSON [www20c] as serialization format, a standardized notation widely used for various kinds of applications. To decouple the DSML-specific serialization infrastructure from the JSON infrastructure explained in Section 5.3, a model of the abstract syntax of JSON is used for an intermediate representation.

Serializing and deserializing symbol tables requires conceiving serialization strategies for all parts of the symbol table infrastructure that should be persisted. This includes serialization strategies for symbols and (artifact) scopes. In the concept, each serialization strategy is realized through an individual type of *DeSer*. The traversal of a language's symbol tables with the purpose of their serialization is realized through *Symbols2Json* types. MontiCore generates the Symbols2Json classes and the DeSers of all symbols and

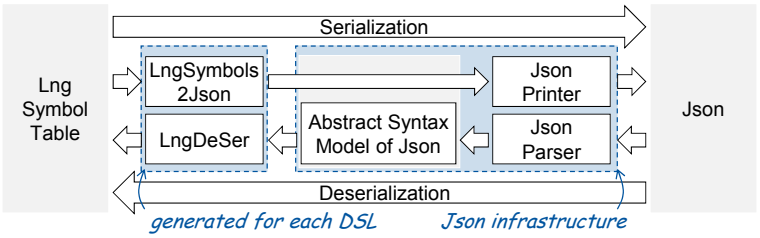


Figure 5.5: Overview of the classes realizing the serialization strategy for MontiCore symbol tables

scopes of a language as part of the STI. As for other artifacts generated by MontiCore, the TOP mechanism [HKR21] can be applied to customize generated types.

An overview of the concept for (de)serializing symbol tables is depicted in Figure 5.5. The object structure of a symbol table of the language `Lng` (depicted left) is serialized as JSON-encoded String (depicted right). During serialization of the symbol tables, the data structure is traversed with a language-specific `LngSymbols2Json` object that reuses the visitor for the language. The serialization of each object of the symbol table is delegated to the `DeSer` that realizes the serialization strategy for this type. DeSers employ a `JsonPrinter` to translate objects into a String encoded in JSON.

During serialization, no explicit intermediate representation in terms of an abstract syntax model of JSON is instantiated to increase the performance of serialization. Nevertheless, the JSON printer encapsulates the concrete syntax of JSON from the type-specific serialization strategies, which yields the opportunity to exchange the serialization format to other textual representations of object structures. s During deserialization, an abstract syntax model of JSON is instantiated by parsing the serialized symbol table with a `JsonParser`. The outermost object of a serialized symbol table is always a serialized artifact scope. Therefore, the deserialization can always begin with the DeSer for scopes. The scope DeSer delegates the deserialization of contained symbols and scopes to the DeSers for these. Internally, the DeSers use the builders for scopes and symbols that are generated by MontiCore [HKR21].

DeSer Classes

A DeSer class realizes a serialization strategy for a concrete type of object that should be serialized and deserialized. MontiCore generates a `DeSer` class for each symbol kind defined within a language. As handling artifact scopes and scopes of a language is similar, a single `DeSer` class realizes the serialization strategies for both scopes and artifact scopes of a language. To distinguish these in the following, we denote DeSers that serialize symbols as `SymbolDeSers` and DeSers of (artifact) scopes as `ScopeDeSers`. Serialization strategies have to be realized in a way that all information that should be

persisted is serialized. Furthermore, serialization strategies should omit the serialization of irrelevant information. To satisfy the purpose of symbol tables being interfaces for language composition, serialized symbol tables must be deserializable by tools that may not be aware of all contained serialized information.

DeSers realize explicit handling of inheritance (*cf.* Section 5.1). A DeSer, thus, is only responsible for serializing and deserializing objects of a single type and none of its subtypes or supertypes. Instead, the global scope manages the DeSers used for each type of scope or symbol individually (*cf.* Section 5.2.3). To shorten the produced Strings, DeSers realize type-specific default serialization (*cf.* Section 5.1) for object members of built-in types. DeSers omit the serialization of:

- empty Strings, *i.e.,* Strings with the value `""`
- numeric types with the value `0`, `0L`, `0.0`, etc.
- Booleans with the value `false`
- empty lists
- optional types with an absent value

As a further means to compactify Strings, DeSers omit producing unnecessary whitespace characters in the produced JSON Strings by default. Instead, developers can use external tools to beautify/format the JSON-encoded Strings[2]. Besides methods for realizing the serialization and deserialization of the type that the DeSer is responsible for, DeSers have other methods that enable individual customization through overriding methods via implementing a subclass of the DeSer. A typical scenario for customization is the adjustment of the serialization of attributes that have been added through symbol rule or scope rule attributes (*cf.* Section 4.2). Therefore, the serialization and deserialization of such attributes is encapsulated in individual methods. Furthermore, DeSers have hook point methods that enable (de)serialization of additional members. By default, these methods have an empty implementation.

Serialization usually includes the persistence of the intermediate processing steps of objects. When serializing data structures such as symbol tables, however, not all attributes need to be persisted. Any state information of symbols or scopes, such as the Boolean flags for avoiding circular resolving (*cf.* Section 6.2.2), are omitted in the serialization and set to a default value after deserialization.

Besides built-in types such as `boolean` or `int`, symbol rules and scope rules can use any other Java type, such as `java.awt.Color`. While the serialization strategies for built-in types as well as optionals and iterations thereof can be generated, the serialization strategies for arbitrary Java types cannot be generated without runtime knowledge

[2]For example, the MontiCore Json tool offers such functionality: `https://github.com/MontiCore/json`

about the fields of the Java types. Since this would require utilizing the Java reflection API [McC98], our approach does not provide serialization strategies for arbitrary Java types. Instead, if a symbol has a symbol rule attribute of a type for which no built-in serialization strategy is available, the serialization and deserialization of such attributes is realized as an abstract method. The same holds for scope rule attributes and scope DeSers. If any abstract method is contained in a generated DeSer class, the class is generated as an abstract class, and language engineers have to apply the TOP mechanism to implement the abstract methods and, thereby, complete the missing (parts of) serialization strategies.

Concept for Serialization Strategies of Scopes

The serialization strategy for scopes comprises individual serialization strategies for artifact scopes and for scopes that are no artifact scopes because these yield different properties. However, most parts are common to both strategies.

In general, scopes are serialized with typeless persistence (*cf.* Section 5.1). This is due to the fact that in many languages, scopes are only containers for the symbols that are defined within the scope. Therefore, scopes rarely transport language-specific information, and thus, the type of the scope harms the reusability of persisted symbol tables. By default, MontiCore supports a single scope type for each language. All scopes within an artifact scope, therefore, are of the same scope type and persisting scope types is a piece of redundant information as the language is already determined by the file extension of the symbol table files.

Serialization and deserialization of scopes are not realized as a fully symmetric operation. Instead, scopes are only persisted if these contribute information that is useful when a symbol table is loaded. To this end, a scope is only serialized if it is not empty and if it exports symbols visible beyond the artifact scope. The scope property `isExporting`, therefore, does not have to be serialized as an explicit attribute of a serialized scope or artifact scope. The property `isOrdered` is not serialized as well since its value does no affect the symbol resolution from beyond an artifact scope. The property `isShadowing` is only serialized for inner scopes and not for artifact scopes because the latter are always shadowing scopes. A non-shadowing artifact scope would prohibit all local symbols that have the same name as any other symbol of this kind that is exported by any other known model artifact. Artifact scopes are serialized as JSON objects with the following members:

- name if present
- package if present
- scope rule attributes
- list of contained symbols if it is not empty

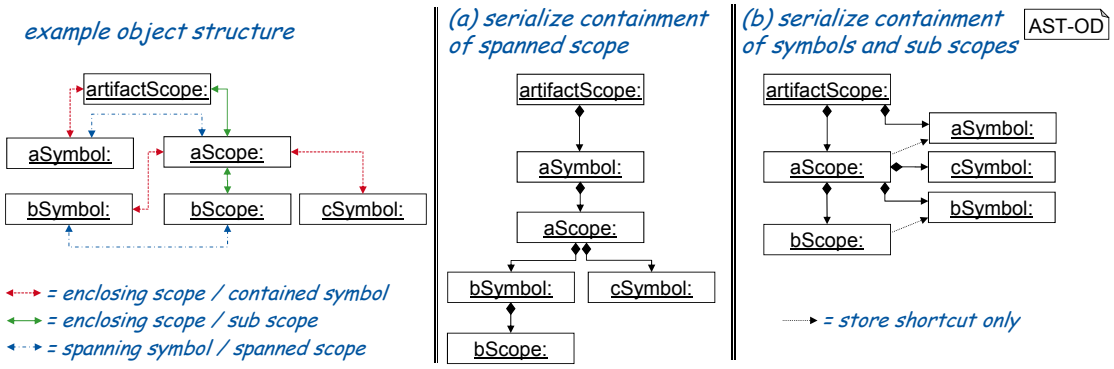


Figure 5.6: Example for two alternative approaches for serializing object structures of scopes and symbols

The list of symbols includes all symbols that are directly contained in the scope regardless of their kinds. The reason for this is that a separation into individual lists per symbol kind would be less compact.

For non-artifact scopes, the name of the scope can be derived from the symbol that spans the scope. Scopes that are not spanned by symbols are not serialized, as these can never be addressed from beyond an artifact scope. Inner scopes are serialized as JSON objects with the following members:

- property `isShadowing`
- scope rule attributes
- list of contained symbols, if not empty

Scopes are associated with other scopes and symbols through different bidirectional associations. As depicted in Figure 5.6, there are different approaches for serializing scopes and symbols. The left side visualizes an example of an artifact scope that contains a symbol `aSymbol`, which spans the scope `aScope`. This scope contains two symbols `bSymbol` and `cSymbol`, where the symbol `bSymbol` spans a scope `bScope`. In this, any scope and its enclosing scope, any scope and the contained symbols, as well as any symbol and its spanned scope, are connected with each other via a bidirectional association. During serialization with JSON, the object graph is transformed into a tree, and thus, bidirectional associations and other cycles have to be removed. This can be achieved by serializing the association between a symbol and a spanned scope as a composition, as depicted in Figure 5.6 (a). During deserialization, the association between scopes and enclosing scopes can be re-established. This solution would require a special handling of scopes that are not spanned by a symbol, which in general is allowed in MontiCore. However, these scopes are not serialized since all contained symbols

cannot be addressed via names. A disadvantage of this approach is that the kinds of the symbols that span scopes must be known in the language that loads a stored symbol table to deserialize the contained scope. An alternative solution is depicted in Figure 5.6 (b), where both, a list of contained symbols and a list of subscopes, are realized as members of a scope. However, this form of serialization requires information about the symbols that span a scope to establish the same object structure when an artifact scope is loaded. Therefore, a shortcut from the scope to the symbol that spans the scope has to be serialized as well. For this shortcut, it suffices to indicate the name and the kind of the symbol as the symbol must be contained in the scope that encloses that current scope. The MontiCore implementation of DeSers realizes the alternative (a) since all persisted scopes are spanned by a symbol and for this, (a) yields a more compact serialization.

The serialization of an artifact scope begins with instantiating a `JsonPrinter` object that captures the current state of the JSON-encoded String. During serialization, a DeSer for scopes begins with serializing the (artifact) scope members as described above before it leaves the traversal of the local symbols in the scope open for the `Symbols2Json` class. After performing the traversal of the scope that delegates serialization of all individual symbols to the respective symbol DeSers, the scope DeSer invokes the hook point for additional scope attributes. Afterward, it returns the content of the JSON printer that contains the serialized scope.

During deserialization, a scope DeSer parses the JSON-encoded String and instantiates an artifact scope with the object members of the persisted artifact scope. For the deserialization of local symbols, the scope DeSer reads the symbol kind of a persisted symbol and uses the global scope to obtain the DeSer configured to deserialize symbols of this kind. The deserialization of the symbol persisted as a JSON object is delegated to the responsible DeSer, and the result is added as a symbol to the (artifact) scope. DeSers of symbols that span a scope also instantiate the spanned scope. After all symbols of an (artifact) scope are deserialized, the hook point for additional scope attributes is invoked, and the resulting (artifact) scope is returned. All scopes are instantiated through the language mills.

Concept for Serialization Strategies of Symbols

Contrary to the serialization strategy for scopes, the serialization strategy for symbols relies on type information that is persisted in each symbol explicitly (*cf.* Section 5.1). A symbol is serialized as a JSON object, and the symbol kind is realized as a String member of this object. As in the STI, the symbol kind is directly related to the type of the symbol class, the type information is persisted explicitly. However, the kind can be re-interpreted during deserialization by reconfiguring the global scope (*cf.* Section 5.2.3). Whenever a symbol is deserialized, the symbol kind can be obtained by scope DeSers without requiring more knowledge about a concrete symbol kind's serialization strategy. Instead, it can delegate further deserialization of the symbol to a suitable symbol DeSer.

Serialized symbols are JSON objects with the following members:

- kind as a String
- name as a String
- symbol rule attributes
- spanned scope, only if the symbol kind spans a scope, the scope exports symbols, and the scope is not empty

The serialization of a symbol begins with instantiating a `JsonPrinter` for capturing the serialized String. During serialization of a symbol, the kind and name of the symbol are printed as first members, followed by all symbol rule attributes that are defined for the symbol kind. If the symbol kind spans a scope, the DeSer uses the Symbols2Json class of the language to traverse the spanned scope and serialize it if it exports symbols and is not empty. Finally, the symbol DeSer invokes the hook point for additional symbol attributes and returns the content of the JSON printer containing the printed symbol.

During deserialization, a symbol DeSer parses the JSON-encoded String and instantiates a symbol with the persisted name. The symbol DeSer does not check whether it is capable of deserializing the symbol kind. Instead, scope DeSers perform this check during deserialization of a scope. If the symbol kind spans a scope, the deserialization proceeds with deserializing the scope by obtaining the scope DeSer from the global scope. The deserialization of the scope is delegated to this DeSer, and the resulting scope object is connected with its environment. If a symbol kind spans a scope, but no scope is contained in the serialized String, the scope either does not export symbols or is empty. To avoid deserializing an incomplete object structure in such scenarios, symbol DeSers create and connect empty scope objects with the environment. Afterward, the hook point for additional symbol attributes is invoked, and the resulting symbol is returned. All symbols and scopes are instantiated through the language mills.

GlobalScopes

Global scopes realize the inter-model symbol resolution and, as such, trigger the loading of symbol table files. Beyond this, global scopes have other tasks in symbol persistence.

Global scopes manage a regular expression for symbol file extensions that can address symbol files of multiple languages at once. All files with file extensions that satisfy the regular expression and are found in the symbol path are considered for loading artifact scopes during symbol resolution. Each global scope further has a map of symbol DeSers. The keys of the map are Strings with symbol kinds, and the values are instances of symbol DeSers. Each entry maps a symbol kind to the DeSer that should be employed to serialize and deserialize symbols of this kind. This map can be used as a central spot for the reconfiguration of DeSers since the singleton global scope instance is statically

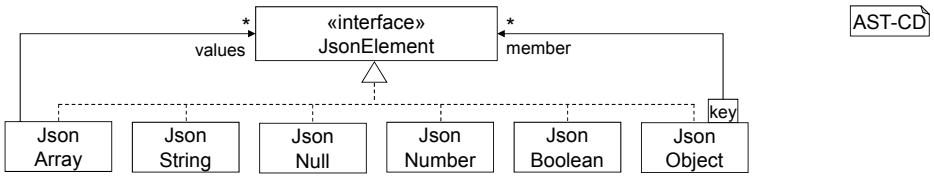


Figure 5.7: Overview of the classes of the JSON abstract syntax model

available via the language mill. Instead of using the DeSer for the exact symbol kind by default, language engineers can configure the map in the global scope to use a DeSer for a type compatible symbol kind instead (*cf.* Section 5.5.6 and Section 5.5.7). We say that symbol kinds are compatible if they are in a mutual inheritance relationship. By default, the map contains entries for all symbol kinds that are defined in a language and all symbol kinds defined in any (transitively) inherited language. Besides the map of symbol DeSers, global scopes manage the scope DeSer. Each global scope uses, by default, a single scope DeSer instance to serialize and deserialize all scopes of symbols tables that are loaded and stored. This includes scopes that are instantiated by symbol DeSers of symbols defined in foreign languages. To this end, symbol DeSers obtain the scope DeSer that is employed for deserializing a spanned scope from the global scope.

Furthermore, the global scope manages a Symbols2Json object that carries out the loading of symbol tables and can be used by a language tool for storing symbol tables.

Symbols2Json

The `Symbols2Json` classes generated by MontiCore realize both the traversal of the symbol table data structure with the purpose of its serialization and offer methods for loading and storing symbol tables. To realize the traversal, a Symbols2Json class implements the visitor interface of a language. Symbols2Json classes do not have a custom traversal that deviates from the traversal strategies implemented by the visitors. For language composition, the class uses a traverser that employs the Symbols2Json classes of inherited languages to serialize types defined in these languages. The traversal of a scope or an artifact scope covers the local symbols of this scope. The traversal of symbols is empty, *i.e.,* for all symbols, including symbols that span a scope, the traversal does not cover any contained structures. Upon visiting a symbol, a scope, or an artifact scope, the Symbols2Json class delegates the serialization to the responsible DeSers obtained from the global scope.

5.3 JSON Infrastructure

JSON [Mar17, www20c] is a compact, yet human-readable notation for object structures. The JSON syntax is standardized [www17] and, due to its compactness, a common for-

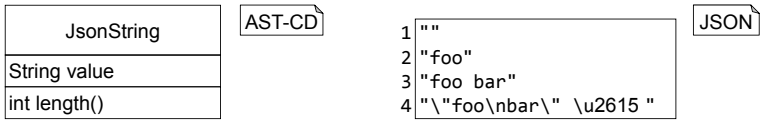


Figure 5.8: The `JsonString` class (left) and example JSON representations (right)

mat in web-based software systems [Mar17] for data exchange or the syntax of configuration files[3]. Although the foundations of JSON are based on JavaScript, the notation is language-independent, and any object structure can be represented in JSON.

Unlike documents of markup languages, such as the extensible markup language (XML) [PSMB+06], JSON documents typically do not conform to an explicit schema. Certain properties of JSON enable efficient lexing and parsing of JSON models. For example, the JSON language requires only a few different token classes and the first character of each token uniquely identifies the token class. For instance, whenever a JSON lexer encounters the letter `t` at the beginning of a lexed character sequence, the only valid token is the Boolean `true`, and all other token classes can be discarded.

The following presents an infrastructure that we conceived for serializing and deserializing Java object structures in JSON. The motivation for this implementation is to provide a JSON infrastructure that can support serialization strategies with JSON as an intermediate representation as depicted in Figure 5.2 and avoids using the Java reflection API [McC98]. Beyond this, the infrastructure should be customizable for serializing MontiCore symbol tables. A further aim is to provide fast serialization and deserialization of JSON, focusing on the latter. The remainder of this section describes the abstract syntax model of the JSON infrastructure in Section 5.3.1 before explaining the infrastructure for serializing JSON in Section 5.3.2 and deserializing JSON in Section 5.3.3.

5.3.1 JSON Abstract Syntax Model

An abstract syntax model of JSON can serve as an intermediate representation for serialization strategies, as depicted in Figure 5.5. The abstract syntax of a JSON document consists of a single (main) JSON element. A JSON element is either an object, an array, a String, a number, a Boolean value, or null, as depicted in Figure 5.7. Arrays and objects are elements decomposed from other JSON elements. The following presents for each of these abstract syntax classes the concrete syntax and the realization of the abstract syntax classes in our JSON infrastructure.

A JSON String is a character sequence surrounded by quotation marks. JSON supports the usual escape sequences (*e.g.,* `\n` to indicate a line break) that begin with a backslash character. The class `JsonString` of our JSON infrastructure realizes JSON Strings and is depicted on the left side of Figure 5.8. This class wraps the data type

[3]Angular uses JSON for configuration files: `https://angular.io/guide/build`

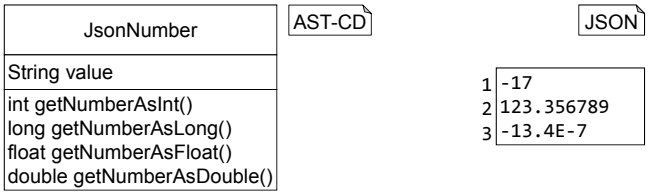


Figure 5.9: The `JsonNumber` class (left) and example JSON representations (right)

Figure 5.10: The `JsonBoolean` class (left) and example JSON representations (right)

`java.lang.String` through a class attribute `value` that can be accessed and modified through `getValue` and `setValue` methods. `JsonString` further has a method delegating to the `length()` method of the Java String. The right side of the figure presents examples for the concrete syntax JSON Strings. Each JSON String begins and ends with quotation marks. A String can be empty (l. 1) or contain characters (l. 2) including whitespace characters (l. 3) and escape sequences (l. 4) such as `\"`, `\n`, or a Unicode escape sequence `\u2615`.

A JSON Number is any signed decimal integer or floating-point number with an optional exponent. Contrary to the type systems in typical programming languages, JSON does not further distinguish different types of numbers, such as `int`, `float`, `double`, `short`, or `long`. To transport such type information beyond serialization and deserialization, it has to be either encoded explicitly or through a symmetric form of serialization and deserialization. The class `JsonNumber` of our JSON infrastructure realizes JSON numbers and is depicted on the left side of Figure 5.9. Internally, this class manages the numeric value in the form of a String attribute to be independent of a numeric data type in Java. Users of the JSON infrastructure can decide on a numeric data type through methods that convert the String representation to common built-in Java data types. These methods internally use parsers provided by the JDK to ensure compatibility of the number formats. For instance, the method `getNumberAsInt()` internally employs `java.lang.Integer.parseInt(String)`. Three examples for the concrete syntax of JSON numbers are depicted on the right side of Figure 5.9: an integer number (l. 1), a floating-point number (l. 2), and a number using the exponent notation (l. 3).

A JSON Boolean is either *true* and $false$, *i.e.,* it forms the basis for a bivalent and not a ternary logic. However, through the absence of a Boolean value, ternary logic values can be realized as well. The class `JsonBoolean` of our JSON infrastructure

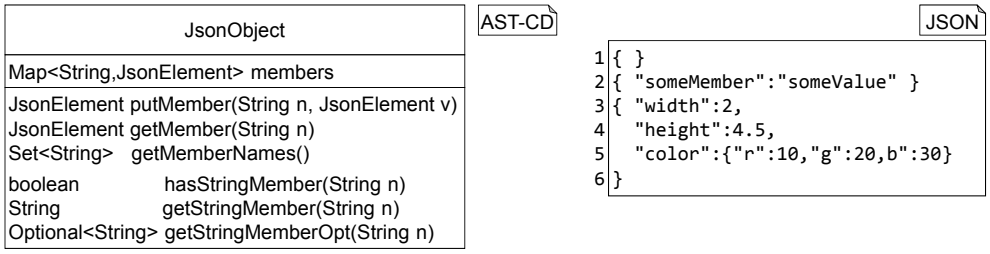


Figure 5.11: The `JsonObject` class (left) and example JSON representations (right)

realizes JSON Booleans and is depicted on the left side of Figure 5.10. This class wraps the primitive data type `boolean` through a class attribute `value`. The concrete syntax of the two different JSON Boolean values is depicted on the right side of Figure 5.10.

A JSON Null is the character sequence `null` and represents the absence of another value, thereby realizing the infamous null pointer that exists in many modern programming languages [Hoa09]. We recommend avoiding JSON nulls and instead omitting serialization of members or elements that would be null and handling this properly during deserialization. The class `JsonNull` of our JSON infrastructure realizes JSON nulls. As JSON nulls should not be used, this class exists only for reasons of completeness.

A JSON Object contains an unordered set of members. Each member has a member name and a value, which is an instance of a JSON element. Therefore, JSON objects can also be decomposed from other JSON objects. A common convention is to assume that member names are unique within a JSON object. This simplifies the handling of JSON objects and is reasonable as JSON object members typically reflect serialized class attribute members, whose names have to be unique per class in many object-oriented programming languages. Contrary to other object notations such as UML object diagrams, there is no explicit notation for types to which JSON objects conform. However, if desired, the type information of an object can be encoded into any JSON object member. The class `JsonObject` of our JSON infrastructure realizes JSON objects and is depicted on the left side of Figure 5.11. Members of an object are realized as a map attribute that maps member names to member values in the form of JSON elements. To this end, member names are unique Strings that form the key set of the map. The class `JsonObject` contains methods to modify and access the members of an object, which delegate to methods of the underlying map. Besides these, the `JsonObject` contains methods for checking whether a member with a given name and an expected kind of JSON element exists. Direct access methods (*e.g.,* `getStringMember(..)`) yield an error if no such member exists, and other methods (*e.g.,* `getStringMemberOpt(..)`) return an empty `Optional` value if the member does not exist. The latter can be used to handle the deserialization of optional attributes by avoiding the usage of `JsonNulls`. The right side of Figure 5.11 presents the concrete syntax of three exemplary JSON ob-

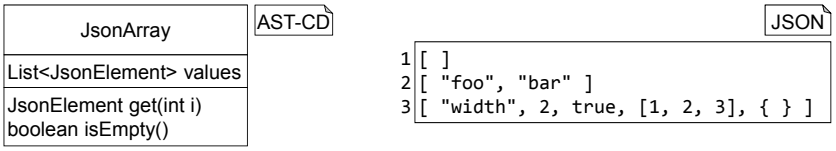


Figure 5.12: The `JsonArray` class (left) and example JSON representations (right)

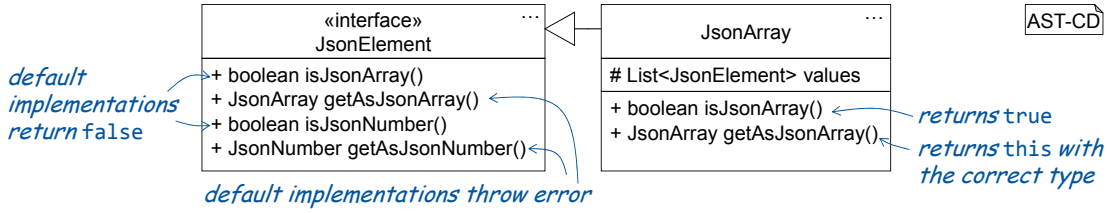


Figure 5.13: Avoidance of down casts for the JSON infrastructure

jects. Each object is surrounded by curly braces. An object can be empty (l. 1) or contain one (l. 2) or more (l. 3-6) comma-separated members. A member begins with the member name surrounded by quotation marks followed by a colon and the member value. The member value can be of any JSON kind such as a String (l. 2), a number (ll. 3-4), or an object (l. 5).

A JSON Array is an ordered list of JSON elements. As a JSON array can contain values that are arrays, it is a recursive data structure. A single JSON array may contain elements of different kinds, such as a String, a Boolean value, and an object. The `JsonArray` class of our JSON infrastructure realizes JSON arrays and is depicted on the left side of Figure 5.12. It manages a list of JSON elements that contain the array's values and defines methods that delegate to methods of the list. Each JSON array is surrounded by square brackets and can be empty (l. 1) or contain one or more comma-separated values of the same (l. 2) or different kinds (l. 3).

The JSON data structure adheres to the JSON standard, *i.e.,* it is unlikely to be extended with novel JSON element kinds. However, the classes realizing JSON element kinds can be extended. For example, a class `MemberTracingJsonObject` could extend the class `JsonObject` and log read and write access on JSON object members. To this end, the `JsonElement` interface is aware of all kinds of distinguishable JSON elements, which are realized by classes that implement the `JsonElement` interface. This can be leveraged to provide methods for each distinguishable kind of element. The `JsonElement` provides methods for checking that the element is of a specific kind, such as the method `boolean isJsonArray()` for the element kind `JsonArray`, as depicted in Figure 5.13. These methods exist for each distinguishable kind of JSON element, and the default implementation in the `JsonElement` returns *false* for all of

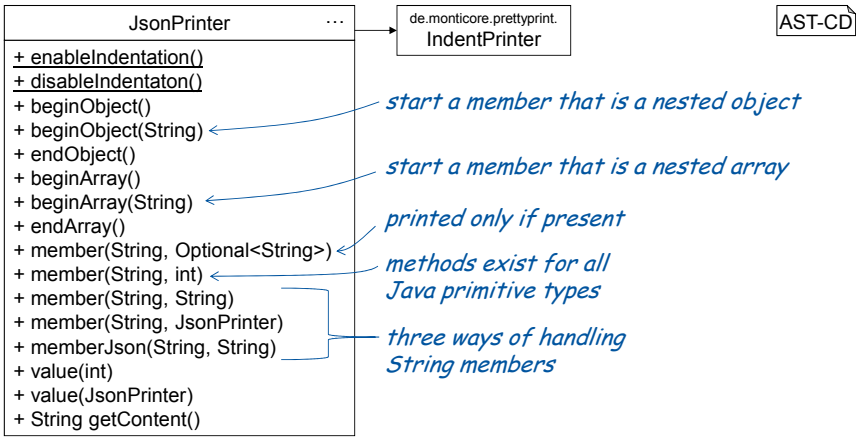


Figure 5.14: Overview of the JSON printer

these methods. The individual kinds override the methods for checking that a JSON element is of their own kind and return *true*. For example, the method `boolean isJsonArray()` in the class `JsonArray` overrides the method from the interface and returns *true* instead.

Similarly, the `JsonElement` interface provides methods for returning itself as an instance of each concrete kind. The default implementation in the interface throws an error if the method is called. The classes implementing the interface override the methods of their own kind and return the current instance. For example, the class `JsonArray` overrides the method `getAsJsonArray()` and returns the current object (*i.e.*, `this`) without a type cast. Through these methods, the subtypes of each class can override the methods for checking the element kind individually. This mechanism, thus, offers more flexibility for extensions through subclasses than `instanceof` checks and down-casting.

5.3.2 Serialization Infrastructure

A challenge in serializing Java object structures is to apply type-specific serialization strategies. Some JSON libraries make use of the Java reflection API to handle serialization strategies for each type [www20a]. The realization of the symbol table serialization in MontiCore employs visitors for traversing the object structures that are serialized. With a JSON abstract syntax model as an intermediate representation (*cf.* Figure 5.5), the printing of the concrete JSON-encoded Strings can be decoupled from the traversal of the data to be serialized. Thus, the JSON serialization infrastructure does not include any explicit handling of the serialized types.

For building JSON-encoded Strings from Java, the JSON infrastructure contains the `JsonPrinter` class. Figure 5.14 depicts an excerpt of the methods of the `JsonPrinter` class. This class realizes an API including `value(..)` methods for printing values of

common built-in Java data types such as `int`, `double`, or `boolean`. For printing decomposed JSON arrays, the printer offers `beginArray()` and `endArray()` methods. Values of the array can be printed by using the `value(..)` methods. For printing JSON objects, the printer offers `beginObject()` and `endObject()` methods.

The `JsonPrinter` is a facade for the concrete syntax of JSON as its methods reflect mainly the abstract JSON syntax. In the realization of the concept for serialization strategies (*cf.* Figure 5.5), no explicit instance of the JSON abstract syntax has to be created during serialization. Instead, it suffices to use the `JsonPrinter` in the symbol table printers. As an alternative to the `JsonPrinter`, it would have been possible to instantiate the JSON abstract syntax model when serializing objects. Then, a pretty printer could traverse this structure by employing a visitor and print the JSON abstract syntax to concrete syntax. However, this would require creating additional objects and performing additional traversal. Hence, it would be less efficient both in terms of memory and time consumption compared to using the `JsonPrinter`.

To support being used in combination with MontiCore visitors [HKR21] that traverse the objects to be serialized, the `JsonPrinter` separates printing the begin, content, and end of objects and arrays. This enables to invoke a `beginObject()` method in the `visit(..)` method of an object, then traversing the object using the `traverse(..)` method of the visitor, and afterward invoking the `endObject()` method in the `endVisit(..)` method of the object.

The JSON printer does not perform any formatting such as indentation by default. Instead, only the least necessary characters are printed to reduce the size of JSON documents for efficient storage. This contradicts the fact that JSON is often the format of choice, inter alia, because it is character-based and, thus, should be human-readable. However, as JSON is a common data format, many editors support auto-formatting of JSON and, thus, JSON documents can be formatted with little effort. Nevertheless, the `JsonPrinter` can be set to print formatted JSON documents if manual inspection of the produced documents is intended. This is realized in the implementation as a property that is controlled statically for all `JsonPrinter` instances through the methods `enableIndentation()` and `disableIndentation()`.

To encode a Java String in JSON, it has to be surrounded by (escaped) quotation marks, and all escape characters that have to be escaped both in JSON and in Java have to be escaped twice. Without the double escaping, the JSON printer would yield vulnerabilities to the security of the serialization. If, for instance, quotation marks are not escaped properly, it would be possible to inject any JSON code through a String member or value. Printing String members or values in the context of printing JSON, therefore, has to be treated with care. Whereas in some cases, the String that should be printed is not encoded in JSON, in other cases, it might be intended that the String is already encoded in JSON. The latter is the case if the String that should be printed is the result produced by another JSON printer. To foster this, the `JsonPrinter` provides three alternatives for printing Strings:

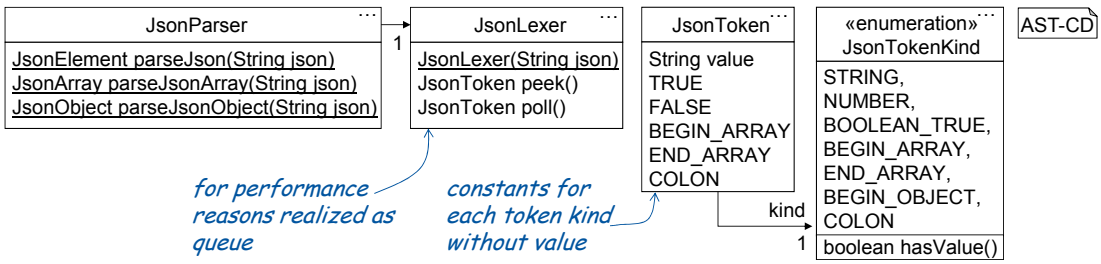


Figure 5.15: Overview of the JSON parser

- The methods `value(String)` and `member(String,String)` escape the passed String and should be used for serializing any non-JSON-encoded Strings.
- The methods `valueJson(String)` and `memberJson(String,String)` do not escape the passed String and should be used for serializing any JSON-encoded Strings.
- The methods `value(JsonPrinter)` and `member(String,JsonPrinter)` use the result of the passed `JsonPrinter` for serialization.

To avoid vulnerabilities due to malformed JSON, the methods `valueJson(..)` and `memberJson(..)` should be handled with care. Usually, it is not necessary to use these methods.

5.3.3 Deserialization Infrastructure

Deserializing a JSON document to an object structure requires information about the data structure of the objects. As stated before, this information can be passed to the JSON infrastructure in Java via the Java reflection API. For example, Google GSON [www20a] uses an argument of type `java.lang.Class` to indicate the type of data in a serialized String. However, our JSON infrastructure has the ambition to avoid the reflection API and thus, uses individual builders for each data type to be deserialized. Similar to the serialization explained in Section 5.3.2, we separate deserialization into two phases. In the first phase, the serialized String is parsed to an intermediate data structure using a JSON parser. During the second phase, the instances of the intermediate JSON data structure are used as the information basis for systematically building up object structures.

As depicted in Figure 5.15, the JSON Parser offers a static method for parsing a passed String encoded in JSON into an instance of `JsonElement`. Other methods exist to parse a String directly into concrete kinds of JSON elements such as into a `JsonArray`. Internally, the parser uses the `JsonLexer` to transform the JSON-encoded String into tokens before further processing. As JSON documents can be large, it is not feasible

for the JSON lexer to transform the entire JSON String into a list of tokens at once. This would cause problems, such as a significant memory overhead and the risk of an overflow of indices. Instead, the JSON lexer is realized as a queue that reads only one token at a time from the input String. The JSON parser uses a JSON lexer to obtain the next token from the input document through the common queue operations `peek()` and `poll()`.

The JSON lexer distinguishes twelve kinds of tokens in the `JsonTokenKind` enumeration. Some token kinds (such as numbers) have a variable value, while others, such as a colon, have a constant value. For each token with a constant value, the class `JsonToken` defines a constant attribute. For these tokens, the lexer always uses the same instance. Using these constants prohibits providing instance-specific information with each occurrence of a token, such as the source position within the input document. However, this increases the lexing performance, and for the use case of loading and storing symbol tables, no instance-specific information for these tokens is required. For tokens with variable values, individual instances of `JsonToken` are created during lexing. Regular automata are employed for identifying the kind of these tokens. Given that the first character of a token excludes all but (at most) one token kind, lexing does not require parallel execution of these automata. The following lists all different kinds of tokens that the JSON lexer distinguishes:

Begin object: denotes the start of an object and has the constant value '`{`'.

End object: denotes the end of an object and has the constant value '`}`'.

Begin array: denotes the start of an array and has the constant value '`[`'.

End array: denotes the end of an array and has the constant value '`]`'.

Boolean true: denotes the Boolean *true* and has the constant value '`true`'.

Boolean false: denotes the Boolean $false$ and has the constant value '`false`'.

Colon: separates the name and the value of an object member. It has the constant value '`:`'.

Comma: separates the values of an array or the members of an object. It has the constant value '`,`'.

Null: denotes the JSON null pointer and has the constant value '`null`'.

String: denotes a String that can be both member name or value in a syntax as described in Section 5.3.1. The variable value of the token contains the content of the String, excluding the surrounding quotation marks.

Number: denotes a floating-point or integer number in a syntax as described in Section 5.3.1. To avoid a concrete numeric data type, the token's variable value is managed as a String whose value can be transformed into a numeric data type in a later stage.

Figure 5.16: Example of a serialized symbol table visualized as object diagram (left) and encoded in JSON (right)

Whitespace: denotes a sequence of whitespace characters with a constant value of ‘ ’. It is a deliberate decision to ignore any formatting of the processed JSON document, as this is of little importance in further processing of the JSON document and – assuming that the whitespace follows strict formatting rules – can be reproduced by pretty printing with a formatter. The ignored whitespace characters include line breaks.

5.4 Realization of Loading and Storing of Symbol Tables in MontiCore

This section presents the realization of the loading and storing of symbol tables in MontiCore following the concepts explained in Section 5.2. The implementation of the concepts forms the serialization infrastructure of languages and relies on the JSON infrastructure presented in Section 5.3.

The right side of Figure 5.16 depicts an example of an artifact scope encoded in JSON, and the left side of the figure depicts the corresponding object diagram. The name of the model that contains the exemplary artifact scope is `GameFile` and is persisted as the first member of the JSON object containing the serialized artifact scope. Types of scopes are not persisted. The second member of the serialized artifact scope describes the symbols contained in the scope as JSON array. In this example, the array contains only a single symbol, which is serialized as a JSON object. The symbol is of the kind

Figure 5.17: Common interface for all symbol DeSers

`AutomatonSymbol` and has the name `Game`. Both attributes are serialized as members of the type `JsonString`. The symbol kind `AutomatonSymbol` spans a scope, and therefore, the serialized symbol contains a member for the spanned scope that is serialized as a JSON object. This scope contains three symbols that are serialized within a JSON array. Each individual symbol is serialized as a JSON object that is a value of the array. All three symbols are of the kind `StateSymbol` and have individual names that are, again, serialized as object members.

The realization of the serialization infrastructure for MontiCore symbol tables comprises parts belonging to the MontiCore runtime and parts generated for each language individually. The following sections introduce central constituents of the implementation of symbol table persistence in MontiCore to realize the serialization described by the example above.

5.4.1 Commonalities of Symbol DeSers in the `ISymbolDeSer` Interface

The interface `ISymbolDeSer` is part of the MontiCore runtime and is implemented by all generated symbol DeSer classes. The interface has two generic type arguments that support indicating language-specific method return types and method parameters from the language-agnostic MontiCore runtime. The first generic type argument indicates the concrete symbol class that a symbol DeSer is able to (de)serialize, and the second argument indicates the language-specific Symbols2Json class. The `ISymbolDeSer` interface is depicted in Figure 5.17 and defines three methods.

serialize(S extends ISymbol, S2J) translates a symbol object passed as method argument into a String representation that is returned by the method. For printing the serialized characters, the implementation of the method in the individual symbol DeSer classes employs the `JsonPrinter` that is a member of the passed Symbols2Json object. If the symbol kind of the DeSer spans a scope, the passed Symbols2Json is further used for traversing the scope. This is realized by accepting the traverser of the Symbols2Json class from the spanned scope.

deserialize(String) translates a serialized symbol encoded in JSON into a newly instantiated symbol object. To do so, the String is parsed with the `JsonParser` first, which yields an instance of a `JsonObject`. Otherwise, the deserialization terminates with an

error. The remaining deserialization is delegated to the `deserialize(..)` method with the `JsonObject` as an argument. For this method, the `ISymbolDeSer` interface provides a default implementation.

deserialize(JsonObject) translates a serialized symbol in the JSON object passed as method argument into a newly instantiated symbol object that the method returns. Internally, the method implementation contained in the individual symbol DeSer classes deserializes the JSON object member containing the symbol name and uses the symbol builder to instantiate a novel symbol object with this name. Furthermore, if the symbol kind has any attributes defined via symbol rules, the values for the attributes are also set. The deserialization of each of these attributes is carried out in a separate deserialize method to foster customization of their serialization strategies. If the symbol kind spans a scope, the scope is deserialized as well. To realize this, the symbol DeSer obtains the scope DeSer object from the global scope and delegates the deserialization of the scope to this DeSer. It is essential that the scope DeSer is obtained through the global scope, as in the presence of language composition, the type of the scope and hence, the scope DeSer to employ may have been reconfigured. Afterward, the hook point method for additional symbol attributes is called.

While the primary intention for using the `deserialize(..)` method with a String parameter is testing the (de)serialization of individual symbols, the generated implementation of symbol table persistence uses the `deserialize(..)` method with the JSON object as an argument. The reason for this is that once the JSON of a scope is parsed, the scope DeSers delegate the deserialization of contained symbols to the symbol DeSers. If these used the deserialization with a String as input, the parsed JSON would have to be printed and parsed again, which is inefficient. The `ISymbolDeSer` interface does not introduce method signatures for the hook point methods that can provide additional attributes to be considered for the (de)serialization. This is due to the fact that the method signatures are only used within the respective DeSer class and, thus, have a `protected` visibility.

5.4.2 Commonalities of Scope DeSers in the `IDeSer` Interface

The interface `IDeSer` is part of the MontiCore runtime. It is an interface that all generated DeSer classes for scopes implement. The interface defines eight methods that are implemented by each DeSer and one method with a default implementation. The type `IDeSer` is generic with three type arguments. The first generic type argument is the scope type for which the DeSer realizes the serialization strategy, and the second argument is the type of the corresponding artifact scope. The third argument is the type of the language-specific Symbols2Json class required for traversing the symbol table for its serialization. All methods that are involved in the serialization have a parameter of the language-specific Symbols2Json type. This serves two purposes. For one, the

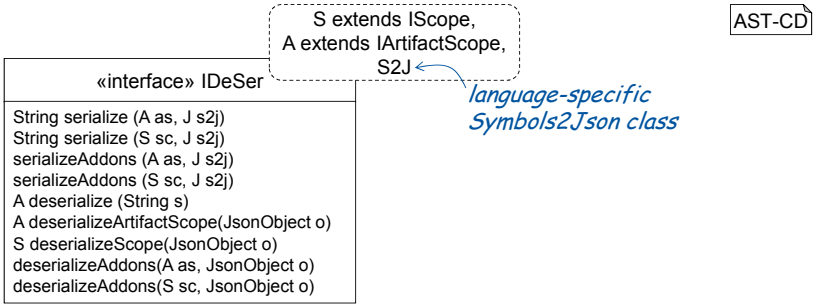


Figure 5.18: Common interface for all scope DeSers

class has an attribute of the JsonPrinter object containing the String serialized so far. Furthermore, the class implements a visitor and is employed for traversing the scope or artifact scope.

For the methods realizing the serialization and the deserialization, DeSers for scopes distinguish the deserialization of scopes and artifact scopes. This is due to the fact that artifact scopes yield different attributes than other scopes. In the methods realizing the hook points for additional attributes to be (de)serialized, the DeSer distinguishes artifact scopes and other scopes as well. This simplifies realizing hooks for one of these only. Other than for the `ISymbolDeSer`, the hook point methods for (artifact) scopes are contained in the `IDeSer` interface. The reason is that these are called from the Symbols2Json class that is not aware of the concrete scope DeSer class. The `IDeSer` interface is depicted in Figure 5.18 and defines nine methods explained in the following:

serialize(A extends IArtifactScope, S2J) translates an artifact scope object passed as method argument into a JSON-encoded String that the method returns. The method requires a Symbols2Json object as a second argument. The `IDeSer` interface declares only the signature of this method, but concrete DeSer classes implement this method and realize the serialization. For artifact scopes, the name and the package are printed as JSON String members if these are present. The serialization of symbols is delegated by employing the Symbols2Json object. Afterward, the hook point for serializing additional artifact scope attributes is invoked. This method only prints the members of a JSON object. The Symbols2Json class (*cf.* Section 5.4.4) prints the beginning and the end of the object.

serialize(S extends ISymbol, S2J) translates a scope object passed as method argument into a JSON-encoded String that the method returns. Like the `serialize(..)` method for artifact scopes, the method requires a Symbols2Json object as a second argument for traversing the scope. The Boolean property `isShadowing` is only printed if its value is *true*. The name of a scope is not serialized as it can be derived from the symbol that spans the scope. Again, this method only prints the members of the JSON

object, the Symbols2Json class (*cf.* Section 5.4.4) prints the beginning and the end of the object.

serializeAddons(A extends IArtifactScope, S2J) is a hook point for realizing serialization of additional members of an artifact scope. The method has two arguments: the artifact scope object and the Symbols2Json object of the language. The latter provides an attribute of the `JsonPrinter` that has to be employed for printing the attribute to JSON. As the Symbols2Json is also a visitor, it can optionally be used to traverse the symbol table further. The default implementation of this method is empty.

serializeAddons(S extends IScope, S2J) is a hook point for realizing serialization of additional members of a scope. The method has two arguments: the scope object and the Symbols2Json object of the language. The default implementation of this method is empty.

deserialize(String) translates a serialized artifact scope encoded in JSON into a newly instantiated artifact scope object. For this method, the `IDeSer` interface provides a default implementation. In this implementation, the String is parsed with the `JsonParser` first, which yields an instance of a `JsonObject`. The actual deserialization is delegated to the `deserializeArtifactScope(..)` method with `JsonObject` as an argument.

deserializeArtifactScope(JsonObject) translates a serialized artifact scope passed as method argument into a newly created artifact scope object that is returned by the method. The `IDeSer` interface defines the signature of the method, and concrete scope DeSers provide the implementation. The first step in the realization is to instantiate an artifact scope object with the artifact scope builder obtained from the language's mill. If the JSON object contains a member containing the name of the artifact scope, the value of this member is set as the name of the artifact scope. Afterward, the hook point method for deserialization of additional attributes is called. Finally, the list of symbols is deserialized. Each element of the list is a `JsonObject` containing a serialized symbol. Therefore, each of these objects has a member with the persisted symbol kind. The scope DeSer deserializes the symbol kind member and obtains the symbol DeSer for this kind from the DeSer map in the global scope. The deserialization of the JSON object containing the symbol is delegated to the `deserialize(..)` method of the symbol DeSer. The resulting symbol object is added to the artifact scope object.

deserializeScope(JsonObject) translates a serialized scope passed as method argument into a newly created scope object that is returned by the method. Similar to the method `deserializeArtifactScope(..)`, the `IDeSer` interface defines the signature of the method, and concrete scope DeSers provide the implementation. The scope is instantiated via the builder obtained from the mill, and the `isShadowing` property is

deserialized. Afterward, the hook point for scopes is called, and the deserialization of symbols is performed in the same way as it is for artifact scopes.

deserializeAddons(A extends IArtifactScope, JsonObject) is a hook point for realizing deserialization of additional members of an artifact scope. The method has two arguments: one for the artifact scope object deserialized so far and another argument with the JSON object of the entire artifact scope. The default implementation of this method is empty.

deserializeAddons(S extends IScope, JsonObject) is a hook point for realizing deserialization of additional members of a scope. The method has two arguments: one for the scope object deserialized so far and another argument with the JSON object of the entire artifact scope. The default implementation of this method is empty.

5.4.3 The `JsonDeSers` Class

The class `JsonDeSers` is contained in the MontiCore runtime. It defines constants and static methods that support realizing language-specific DeSers. While many parts of the typed symbol table infrastructure are specific to the different types, other parts are common to all scopes or all symbols. The class `JsonDeSers` manages constant Strings for commonly used member names of symbols and scopes. Using String constants for member names avoids inconsistencies between the member names that are used for serialization and the member names expected during deserialization. These constants include the member names for symbol kinds, the spanned scope of a symbol, the `isShadowing` property of scopes, and the list of symbols in scopes.

The static methods of the `JsonDeSers` class are used by the generated symbol DeSer and scope DeSer classes and include handling of the serialization and deserialization of members that are common to all (artifact) scopes. These include the deserialization of an artifact scope's package that evaluates to the empty package `""` if the member is not contained in a serialized artifact scope. On the contrary, the deserialization of symbol kinds yields an error if a serialized symbol does not contain this member.

5.4.4 `Symbols2Json` Classes for Traversing Symbol Tables

A `Symbols2Json` class is generated for each MontiCore language. The name of the class is the name of the grammar with the suffix `Symbols2Json`. Symbols2Json classes provide the load and store methods to interact with the serialization infrastructure. Other than for DeSers, the MontiCore runtime does not contain an interface for Symbols2Json classes. The reason is that such an interface has no benefits for the current implementation. Loading symbol tables is integrated into the symbol resolution algorithm and therefore, usually controlled by generated code only. The symbol resolution algorithm uses universal resource locators (URLs) to identify a file that contains a persisted symbol

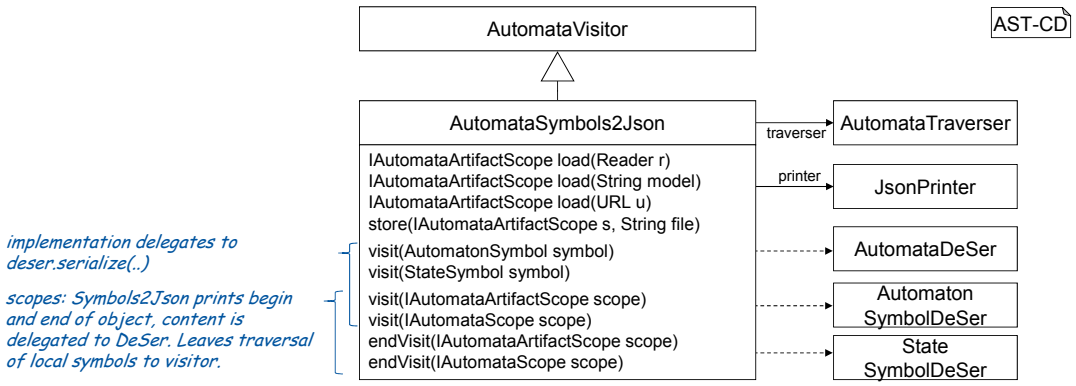


Figure 5.19: Methods of the Symbols2Json class of the automata language

table. For manual purposes, *e.g.,* testing, the Symbols2Json classes also offer methods for identifying files via a String with the filename or a `Reader`. The result of all methods is an instance of the artifact scope interface of the language.

For storing symbol tables, which is currently always controlled through handwritten code, only a single method is available. The method has an argument of the artifact scope interface of the language and a String with the file path comprising all four parts (*cf.* Section 5.2.2).

Conceptually, Symbols2Json classes are not able to perform the actual (de)serialization for any type. Instead, the (de)serialization is delegated to the DeSers responsible for an object that the traversal encounters. However, the Symbols2Json classes have no direct association to any concrete DeSer. Instead, the DeSers for each type are obtained from the global scope to enable their reconfiguration during the instantiation of the Symbols2Json object. To increase performance, the DeSers for symbols and scopes are obtained from the global scope during the `init()` method and are cached for quick access. An example of the `AutomataSymbols2Json` class of the automata language is depicted in Figure 5.19. The class contains methods that are explained in the following.

load(Reader) loads a persisted artifact scope from a file with the `java.io.Reader` object passed as method argument. The result of the method is a new instance of the `IAutomataArtifactScope` interface. The method first reads the content of the file into a String variable, which then is deserialized with the `AutomataDeSer` instance obtained from the global scope.

load(String) loads a persisted artifact scope from a file at the location passed as String method argument. The result of the method is a new instance of the `IAutomata-ArtifactScope` interface. The method first reads the content of the file into a String variable, which then is deserialized with the `AutomataDeSer`. The String argument of this method must not be confounded with a String representing the content of the

serialized file. Such a method is not contained in the Symbols2Json class because its implementation would equal the implementation of the `deserialize(..)` method of scope DeSers.

load(URL) loads a persisted artifact scope from a file at the location passed as URL method argument. The result of the method is a new instance of the `IAutomataArtifactScope` interface. The method first reads the content of the file into a String variable, which then is deserialized with the `AutomataDeSer`. While the `load(..)` methods with a String of the file location and a reader for the file as argument are for manual testing purposes, the `load(..)` method with the URL is used within the resolving algorithm. This is because MontiCore internally uses `URLs` for identifying files, among other things, since these enable accessing files in compressed archives.

store(IAutomataArtifactScope, String) stores the artifact scope object that is passed as a method argument to a new file created at the location passed to the method as String argument. The passed artifact scope object first is serialized with the corresponding DeSer. Afterward, the String is printed to a file at the given location. If a file at this location exists, it is overridden. The file location argument should follow the guidelines explained in Section 5.2.2. If these guidelines are violated, the symbol file will not be considered for loading by the default symbol resolution algorithm.

visit(AutomatonSymbol) delegates to the `serialize(..)` method of the corresponding DeSer, *e.g.,* the `AutomatonSymbolDeSer`. Handling of the spanned scope of an automaton symbol is contained in the symbol DeSer, the traversal of MontiCore's visitors does not traverse scopes spanned by symbols.

visit(StateSymbol) delegates to the `serialize(..)` method of the corresponding DeSer, in this example the `StateSymbolDeSer`.

visit(IAutomataArtifactScope) is part of the serialization for artifact scopes. While most parts of the serialization are delegated to the scope DeSer, an empty JSON object is printed by the Symbols2Json class. The reason for this is that the traversal of symbols that are contained in the scope, which is carried out by the traverser in the Symbols2Json class, requires splitting the serialization of scopes into a part before the traversal of the scope and a part after the traversal.

visit(IAutomataScope) is part of the serialization for scopes. The method uses the `JsonPrinter` to print the beginning of an object (`{`). If the passed scope object is not the first object being printed with the JsonPrinter, the beginning of an object is printed as part of an object member (`"symbols" :{`). This way, the Symbols2Json class distinguishes scopes that are printed as spanned scopes and scopes that are printed as outermost objects. Although usually, only the artifact scope is persisted as outermost

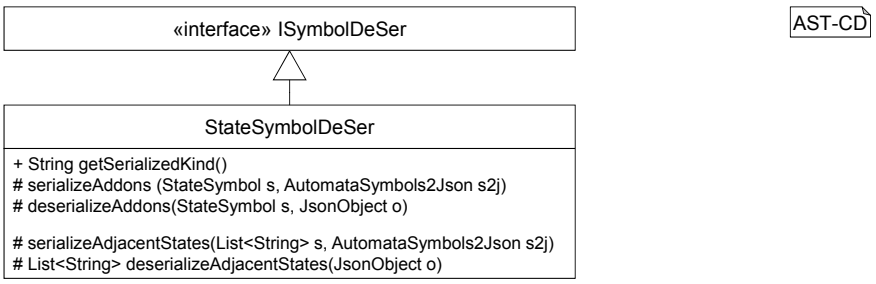


Figure 5.20: Methods of the `StateSymbolDeSer`

scope, for other purposes, such a unit testing, language engineers may serialize a non-artifact scope as outermost scope.

endVisit(IAutomataArtifactScope) prints the parts of a serialized artifact scope after the traversal and serialization of symbols contained in the scope. First, the method prints the end of the JSON array of symbols. Afterward, the hook point method for additional serialization of artifact scopes is called, followed by the end of a JSON object closing the serialized artifact scope.

endVisit(IAutomataScope) prints the parts of a serialized scope after the traversal and serialization of symbols contained in the scope. First, the method prints the end of the JSON array of symbols. Afterward, the hook point method for additional serialization of scopes is called, followed by the end of a JSON object closing the serialized scope.

5.4.5 `SymbolDeSer` Classes with Serialization Strategies for Symbols

For each symbol kind defined by a language via the grammar, MontiCore generates a symbol class and a corresponding symbol DeSer class. The name of the symbol DeSer class is the name of the symbol-defining nonterminal with the suffix `SymbolDeSer`. Each symbol DeSer class implements the interface `ISymbolDeSer` that is described in Section 5.4.1. If a symbol kind inherits from another symbol kind, the symbol DeSer classes are not in an inheritance relationship. An inheritance relationship between symbol DeSers would introduce unnecessary complexity since each symbol kind may have a different form of representing serialized objects, and only a few parts could be reused. The signatures for central methods of symbol DeSers are provided by the `ISymbolDeSer` interface and are implemented with type-specific implementations in the symbol DeSer classes. The implementation of these methods is described in Section 5.4.1. Other methods are only defined in the symbol DeSer classes and are not contained in the interface. These methods are depicted by the example of the `StateSymbolDeSer` in Figure 5.20. For each symbol rule attribute, the symbol DeSer class has a method for the serialization of the attribute and a method for its deserialization.

getSerializedKind() is a method that returns the symbol kind that the symbol DeSer (de)serializes in the form of a String. This method is used to print the symbol kind into serialized Strings and, by default, returns the fully qualified name of the symbol class. Language engineers can override this method to serialize a different kind. This can be useful if the generated symbol kind is unnecessarily long, *e.g.,* in case the simple name of a symbol class suffices as unique identifier of the symbol kind.

serializeAddons(..) is a hook point for the serialization of additional symbol attributes. The generated body of this method is empty, but language engineers can apply the TOP mechanism to the symbol DeSer class to provide implementations of the hook point.

deserializeAddons(..) is a hook point for the deserialization of additional symbol attributes. The generated body of this method is empty, but language engineers can apply the TOP mechanism to the symbol DeSer class to provide implementations of the hook point.

serializeAdjacentStates(..) is a method for the serialization of the symbol rule attribute with the name `adjacentStates` that symbols of the state symbol kind have. The name of the method, hence, is the name of the attribute with the prefix `serialize`. The method has two arguments: a `List<String>` argument that passes the value of the attribute to the method and an `AutomataSymbols2Json` object for additional traversal of the symbol table and for obtaining the `JsonPrinter` to print the serialized String. MontiCore has a built-in serialization strategy for lists of Strings, which are serialized as a JSON array of JSON Strings. Thus, the implementation of the method `serializeAdjacentStates(..)` realizes this serialization strategy.

deserializeAdjacentStates(..) is a method for the deserialization of the symbol rule attribute with the name `adjacentStates` of state symbols. The name of the method, hence, is the name of the attribute with the prefix `deserialize`. The method has a single argument, which is the JSON object of the entire symbol. The reason that this is not only the serialized value of the attribute is that the serialization of the attribute may be realized as one or more members of the JSON object of the symbol with arbitrary member names. The method returns the result of the deserialization of the attribute.

5.4.6 `ScopeDeSer` Classes with Serialization Strategies for Scopes

From each grammar, MontiCore generates one (scope) DeSer class for the artifact scopes and scopes of the language. The name of the DeSer class equals the name of the grammar with the suffix `DeSer`. All DeSer classes implement the interface `IDeSer` that is explained in Section 5.4.2. The signatures for central methods of symbol DeSers are provided by the `IDeSer` interface and are implemented with type-specific implementations in the DeSer classes. The implementation of these methods is described in Section 5.4.2.

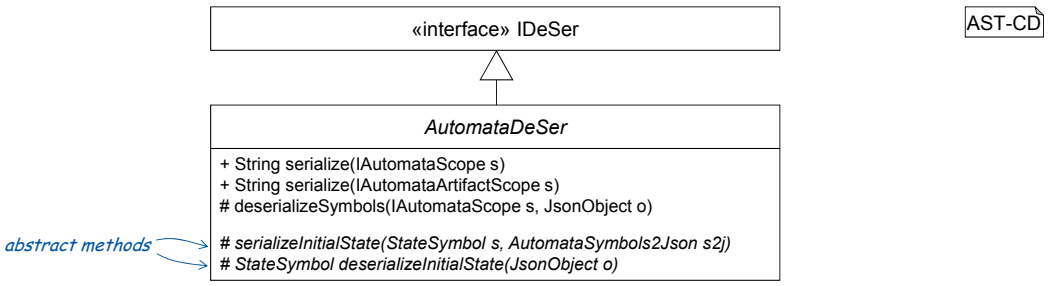


Figure 5.21: Methods of the `AutomataDeSer`

For other methods, the signature is not contained in the interface `IDeSer`. These methods are explained in the following by the example of the `AutomataDeSer` that is depicted in Figure 5.21.

For each scope rule attribute, the scope DeSer class has a method for the serialization and a method for the deserialization of the attribute. If no built-in serialization strategy is available for the type of the attribute, both methods are generated as abstract methods, and the entire DeSer class is generated as abstract class. In the example of the `AutomataDeSer`, there is a scope rule attribute `initialState` of the type `StateSymbol`, for which no serialization strategy is available. Therefore, the two methods are generated as abstract methods, and the DeSer class is generated as an abstract Java class.

serialize(IAutomataScope) is a method that can be employed for manual serialization of a scope of the `Automata` language. Internally, the method implementation instantiates a new object of the Symbols2Json class, accepts the traverser of the Symbols2Json object from the scope object passed to the method as an argument, and returns the content of the `JsonPrinter` contained in the Symbols2Json as the result of the method.

serialize(IAutomataArtifactScope) is a method that can be employed for manual serialization of an artifact scope of the `Automata` language. The method is realized in the same way as the equivalent method for scopes.

deserializeSymbols(IAutomataScope, JsonObject) is a method that is internally used during the deserialization of a scope. The task of the method is to iterate over the serialized list of symbols, find a suitable DeSer for each symbol from the symbol map in the global scope, and employ the DeSer to deserialize the symbol and add it to the respective list of symbols contained in the scope object. If the deserialized symbol spans a scope, this method adds the deserialized spanned scope as a subscope of the current scope, which establishes the bidirectional association between enclosing scopes and subscopes. If the method encounters a serialized symbol that is of a kind for which

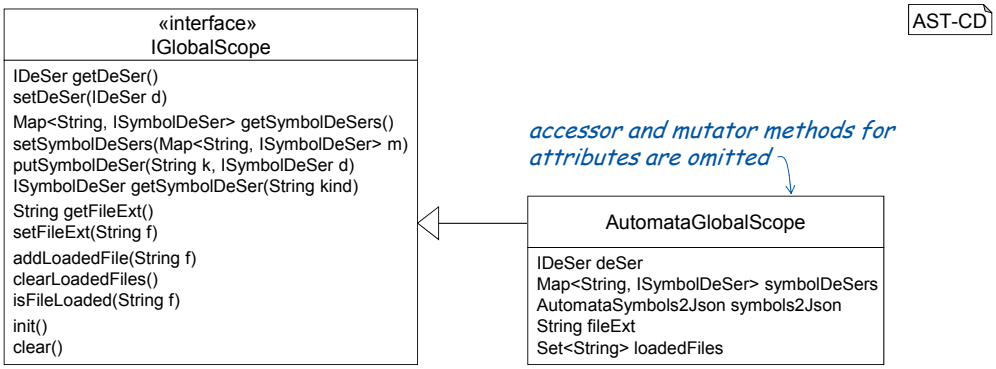


Figure 5.22: Methods of global scope interfaces and attributes of the global scope class that are relevant for symbol table persistence

no symbol DeSer is contained in the symbol map of the global scope, the method omits the further deserialization of this symbol and yields a warning.

serializeInitialState(StateSymbol, AutomataSymbols2Json) is a method for the serialization of the scope rule attribute with the name `initialState`. The name of the method, hence, is the name of the attribute with the prefix `serialize`. The method has two arguments: a `StateSymbol` argument that passes the value of the attribute to the method and an `AutomataSymbols2Json` object for additional traversal of the symbol table and for obtaining the `JsonPrinter` to print the serialized String. MontiCore has no built-in serialization strategy for `StateSymbols`, and thus, the method is generated as an abstract method.

deserializeInitialState(JsonObject) is a method for the deserialization of the scope rule attribute with the name `initialState`. The name of the method, hence, is the name of the attribute with the prefix `deserialize`. The method has a single argument, which is the JSON object of the entire scope. The reason that this is not only the serialized value of the attribute is that the serialization of the attribute may be realized as one or more members of the JSON object of the scope with arbitrary member names. The method returns the result of the deserialization of the attribute. However, in this concrete example, the method is generated as an abstract method since MontiCore has no built-in serialization strategy for `StateSymbols`.

5.4.7 Loading and Storing Symbol Tables via the Global Scope

In MontiCore, the global scope is realized as a singleton, for which the instance is centrally available via the language mill (*cf.* Section 4.3.4). Therefore, the symbol table persistence infrastructure uses the global scope for (re)configuration of the infrastruc-

ture. To do this, *e.g.,* in the presence of language composition, the instances of symbols DeSers, the scope DeSer, and the Symbols2Json class are managed by the global scope. Furthermore, the regular expression for file extensions of symbol table files enables adjustments for loading symbol tables from files in the file system. The initial configuration of the global scope including the map containing the symbol DeSers, is initialized in the `init()` method. All reconfigurations can be set to the initial configuration by invoking the `clear()` method of the global scope.

Accessor and mutator methods for the attributes of the global scope class are generated following the usual MontiCore schema for such methods of the respective types. The signatures of methods for language-agnostic attributes of the global scope class that are used from generated code are defined in the MontiCore runtime class `IGlobalScope`. Figure 5.22 depicts the methods and attributes for symbol table persistence in global scopes by the example of the automata language. All method signatures depicted in the interface `IGlobalScope` are implemented by methods in the class `AutomataGlobalScope`, which are omitted in the class diagram for reasons of compactness. The attributes for symbol table persistence that global scope classes manage are described in the following:

deSer is the DeSer object employed for the (de)serialization of scopes and artifact scopes. The type of this attribute is the general MontiCore runtime interface `IDeSer` and not the generated, language-specific scope class. This enables reconfiguration of the scope, *e.g.,* for language composition in which the scope DeSers are not in a hierarchical relationship with each other that follows the inheritance of grammars.

symbolDeSers is a map containing symbol DeSer objects configured for all known symbol kinds. The map entries have a String of the symbol kind as key and a symbol DeSer object as their value. This map can be employed for reconfiguration of symbol DeSers for individual symbol kinds as described in Section 5.5. The Symbols2Json class obtains the symbol DeSers from the global scope for the serialization, and the scope DeSers obtain the symbol DeSers from the global scope for deserialization.

symbols2Json is an object of the language-specific Symbols2Json class generated for each language. Global scopes use this to load symbol table files, to traverse the symbol table, and to provide access to the `JsonPrinter` object. Symbols2Json classes do not implement a common interface, and it is not required for these to be reconfigurable for language composition. Instead, the traverser contained as an attribute in the Symbols2Json classes is internally reconfigured for language composition. Hence, the accessor and mutator methods for these attributes are not contained in the global scope interface.

fileExt is a String containing the regular expression for file extensions to consider for loading symbol tables from files. The mechanism is explained in more detail in Section 5.4.8.

loadedFiles contains a set of filenames that have been either successfully loaded or for which an attempt to load these has failed. This is part of the integration between the symbol table persistence and the symbol resolution algorithm and avoids loading the same artifact scope from a persisted symbol table multiple times.

5.4.8 Integrating Loading of Symbol Tables into Symbol Resolution

Loading artifact scope objects from symbol table files is integrated into the process of inter-model symbol resolution explained in Section 4.1.8. At first, a set of candidates for model names is calculated based on the (qualified) name of the symbol that is resolved for. Typically, the model name is a prefix of the name to resolve. The calculation of candidates for model names is realized specific to each symbol kind. For example, the calculation of model names for symbols of the kind `StateSymbol` in the automata language is realized in the method `calculateModelNamesForState(..)` of the `IAutomataGlobalScope` interface. For each calculated model name candidate, the global scope attempts to load a symbol table file. For state symbols, this is realized in the global scope method `loadState(..)`.

Global scopes contain a regular expression for file extensions of symbol table files. Fixing a single file extension or an immutable list of file extensions in a global scope limits the reusability of the language as it cannot be extended with stored symbol tables of foreign languages that provide symbol definitions of known symbol kinds without explicit reconfiguration. On the other hand, including every file extension into the process of loading symbols can yield unintended side effects if numerous heterogeneous models are employed. If symbol table files are located in the same directory as other files (*e.g.,* documentation, explanatory videos), traversing files with all extensions can be inefficient. To this end, the default regular extension for file extensions is `".*sym"`, which includes all symbol table files that are named according to the guidelines described in Section 5.2.2. The intention behind this is to maximize the extensibility with novel languages providing symbols while minimizing the consideration of files that are unlikely to contain symbol tables. This default expression can be altered by reconfiguring the global scope if it causes undesired side effects (*cf.* Section 5.5.5).

For each model name candidate, the symbol resolution searches in all model path entries for files with the calculated model name candidate. Only files that satisfy the regular expression for symbol table file extensions are considered. If such a file exists for the given model name candidate, the file is loaded by employing the `load` method of the Symbols2Json object of the global scope. Furthermore, the fully qualified path of the symbol table file is added to the set of loaded files to avoid loading it more than once. If a file contains a serialized symbol table and the deserialization is successful, the resulting artifact scope object is added as a subscope of the global scope. The inter-model symbol resolution algorithm then proceeds the attempt to resolve for the symbol in all known artifact scopes, including those that have been loaded from symbol table files.

```java
public class AutomataTool {

  AutomataSymbols2Json s2j = new AutomataSymbols2Json();

  public void run(String model) {
    ASTAutomaton ast = parse(model);
    IAutomataArtifactScope symtab = createSymbolTable(ast);
    checkCoCos(ast);
    storeSymbolTable(symtab);
    // pretty printing, code generation,…
  }

  public void storeSymbolTable(IAutomataArtifactScope symtab) {
    s2j.store(symtab, "target/" + model + "sym");
  }

  // further methods omitted
}
```

Figure 5.23: Integrating persistence of symbol tables into a language tool by the example of the automata language

5.4.9 Supporting Storing of Symbol Tables for Model Processing

Storing symbols is manually integrated into the process of processing a model. This is typically carried out by a language tool that, in MontiCore, currently must be implemented by hand. The language-specific Symbols2Json class generated for each language contains a method for storing symbol tables (*cf.* Section 5.4.4).

Figure 5.23 depicts an example of the integration of symbol table persistence into the `AutomataTool` that processes models of the automata language in the method `run`. As a first step, the model is parsed, which yields either an AST or the tool terminates with an error. From this AST, the symbol table can be instantiated, which results in an instance of the `IAutomataArtifactScope`. Afterward, the well-formedness of the model can be checked by applying context conditions to the integrated structure of the AST and symbol table. Only if a model is considered well-formed should the symbol table be persisted. Therefore, a violation of a context condition produces an error that terminates the processing of the model. In this example, the storage of artifact scopes is extracted to a method `storeSymbolTable(..)`.

The persistence requires calculating the name of the symbol table file according to the guidelines described in Section 5.2.2. The example sets the output path for stored symbols to the folder `target`. The automata language does not use packages, but languages that use packages have to transform the package of the model into a corresponding folder structure. The name of the symbol table file equals the name of the model file, including its file extension, suffixed with `"sym"`.

Java

```java
public class StateSymbolDeSer extends StateSymbolDeSerTOP {

  protected void serializeColor(Color color, AutomataSymbols2Json s2j) {
    JsonPrinter printer = s2j.getJsonPrinter();
    printer.beginArray("color");
    printer.value(color.getRed());
    printer.value(color.getBlue());
    printer.value(color.getGreen());
    printer.endArray();
  }

  protected Color deserializeColor(JsonObject symbolJson) {
    List<JsonElement> cArray = symbolJson.getArrayMember("color");
    int r = cArray.get(0).getAsJsonNumber().getNumberAsInteger();
    int g = cArray.get(1).getAsJsonNumber().getNumberAsInteger();
    int b = cArray.get(2).getAsJsonNumber().getNumberAsInteger();
    return new Color(r, g, b);
  }
}
```

JSON

```json
{ "kind": "automata._symboltable.StateSymbol",
  "name": "main",
  "color": [115, 172, 87]
}
```

Figure 5.24: Implementation (top) and example (bottom) of a custom serialization strategy for the color attribute of state symbols in the automata language

5.5 Customizing the Persistence of Symbol Tables in MontiCore

In general, the TOP mechanism [HKR21] can be applied to all classes and interfaces of the infrastructure for symbol table persistence. Each generated DeSer class further has hook point methods for the customization of additional (de)serialization. The regular expression for file extensions of symbol table files in global scopes can be employed for configuring a language to load symbols tables of any language. Beyond this, the symbol DeSer map in global scopes can be used for configuring a language to load symbols of any kind. For this, the language infrastructure does not require to be aware of the kind, *i.e.,* it enables loading unknown kinds. This section demonstrates the customization of symbol table persistence based on a collection of typical customization scenarios.

5.5.1 Providing a Serialization Strategy for a Symbol Attribute

The (de)serialization of symbols or scopes can rely on types for which no serialization strategy is prepared. In this case, the DeSer for the symbol kind or scope is generated as an abstract class, and by applying the TOP mechanism, the abstract methods for the serialization and deserialization of the unknown serialization strategy have to be overridden. The serialization strategy must be realized for each attribute individually.

An example of an attribute for which no serialization strategy is available is the `color` attribute of `StateSymbols`. The handwritten class for the `StateSymbolDeSer` that extends the generated `StateSymbolDeSerTOP` class is depicted in Figure 5.24. In the `serializeColor(..)` method, a serialization for the attribute `color` of the type `Color` has to be provided. Colors can be serialized in various JSON representations:

- as a JSON String containing the hexadecimal representation of the RGB value, such as `"#73AC57"`
- as a JSON array with individual numeric entries for red, green, and blue values, like `[115,172,87]`
- as a JSON object with individual members for the red, green, and blue values, as in `{"red":115, "blue":172, "green":87}`

The example depicted in the top of Figure 5.24 serializes a color as a JSON array with three values. The JSON-encoded String is produced with the JSON printer contained in the `AutomataSymbols2Json` object that is passed to the method as an argument. The printer prints the beginning of an object member of the type JSON array with the member name `"color"`. The RGB values of the color are obtained from the color object passed as method argument and printed as values of the JSON array. As the last step in the serialization of a color, the end of the array is printed. The deserialization of the color attribute begins with obtaining the object member with the name `"color"` from the JSON object containing the serialized state symbol. The deserialization assumes that the member is a JSON array. Otherwise, the deserialization yields an error. The first three values of the array are read as integer values and stored in individual variables. JSON arrays are ordered, and hence the first value is set as the red value, the second as the green value, and the third as the blue value. If the serialized array contained more than three members, the deserialization would ignore these. If the array contained less than three values, the deserialization would yield an error. The deserialization returns a new `Color` object instantiated via a constructor that uses the color values from the variables. The bottom of Figure 5.24 depicts an example of a serialized state symbol with the color attribute as realized by the serialization strategy implemented in the `StateSymbolDeSer` depicted in the top of the figure.

5.5.2 Omitting Serialization of Symbols of a Certain Kind

The traversal of symbol tables with the purpose of their serialization is performed by the Symbols2Json class. The traversal performed by this class can be customized to ignore symbols of certain kinds. This is depicted by the example of the class `Automata-Symbols2Json` that extends the `AutomataSymbols2JsonTOP` class in Figure 5.25. The class implements the `AutomataHandler` interface that is part of the generated visitor infrastructure of the automata language. In the constructor, the current object

Java

```java
public class AutomataSymbols2Json extends AutomataSymbols2JsonTOP
                                  implements AutomataHandler {

  public AutomataSymbols2Json() {
    getTraverser().setAutomataHandler(this);
  }

  public AutomataSymbols2Json(AutomataTraverser t, JsonPrinter p) {
    super(t, p);
  }

  public void traverse(IAutomataScope s) {
    for (AutomatonSymbol sym : s.getLocalAutomatonSymbols()) {
      sym.accept(getTraverser());
    }
  }
}
```

JSON

```json
{ "kind": "automata._symboltable.AutomatonSymbol",
  "name": "H2O"
}
```

spanned scope is empty, because state symbols are not traversed

Figure 5.25: Example for omitting serialization of a certain symbol kind

of the class has to be set as the handler for the automata language in the traverser of the Symbols2Json class. As an effect, the `AutomataSymbols2Json` class can override the `traverse(..)` methods for any symbol kinds, the artifact scope, and the scope of the automata language. The other constructor of the `AutomataSymbols2Json` class delegates to the constructor of the TOP class. This constructor does not set the current object as a handler of the traverser to enable reconfiguration of the Symbols2Json class through subclasses.

To ignore the traversal of state symbols within scopes of the automata language, the `traverse(..)` method is overridden and iterates only over the automaton symbols and not over state symbols. The symbols have to be traversed with double dispatching [HKR21], and therefore the traverser is accepted by each automaton symbol instead of directly calling the `handle(..)` method with the automaton symbols as an argument.

The bottom of Figure 5.25 depicts a JSON-encoded String of a serialized automaton symbol `H2O`. Without the customization, the serialized automaton symbol would contain a member for the spanned scope that itself would contain state symbols. However, as state symbols are not serialized, the spanned scope is empty and is not serialized.

5.5.3 Realizing Serialization of an Additional Scope Attribute

There are applications of symbol table persistence in which information should be persisted that is not directly available from an attribute described by a symbol rule or scope rule. To this effect, the `serializeAddons(..)` methods are hook point

Java

```java
public class AutomatonSymbolDeSer extends AutomatonSymbolDeSerTOP {
  protected void serializeAddons(AutomatonSymbol aut, AutomataSymbols2Json s2j) {
    JsonPrinter printer = s2j.getJsonPrinter();
    printer.beginArray("states");
    for (StateSymbol state : aut.getSpannedScope().getLocalStateSymbols()) {
      printer.value(state.getName());
    }
    printer.endArray();
  }

  protected void deserializeAddons(AutomatonSymbol sym, JsonObject j) {
    IAutomataScope scope = sym.getSpannedScope();
    for (JsonElement e : j.getArrayMember("states")) {
      String n = e.getAsJsonString().getValue();
      StateSymbol s = AutomataMill.stateSymbolBuilder().setName(n).build();
      scope.add(s);
    }
  }
}
```

JSON

```json
{ "kind": "automata._symboltable.AutomatonSymbol",
  "name": "Game",
  "states": ["Main", "InGame", "Pause"]
}
```

Figure 5.26: Example for using the hook points for (de)serialization

methods for adding further object members to a serialized symbol or scope, while the `deserializeAddons(..)` methods are hook point methods enabling to reflect the serialized additional information in the symbol or scope object.

An example of an attribute for which the hook points for serialization strategies are used is the attribute for states of an `AutomatonSymbol`. This information is not captured in a symbol rule of the `AutomatonSymbol` because it is entirely derived from the scope that an automaton symbol spans. In fact, serializing the states as an enumeration of state names that itself is a member of an automaton is more compact than serializing the spanned scope and the state symbols as individual JSON objects. In combination with omitting the serialization of state symbols individually (*cf.* Section 5.5.2), this can significantly reduce the complexity of symbol table files of the automaton language.

The handwritten class for the `AutomatonSymbolDeSer` that extends the generated `AutomatonSymbolDeSerTOP` class and provides implementations for the hook point methods is depicted in the top of Figure 5.26. The serialization uses the `JsonPrinter` from the Symbols2Json object passed to the method as an argument. The symbols are printed as an object member of the type `JsonArray`. The first step is to indicate the beginning of the array with the name of the member, which is `"symbols"`. Then, a `for` loop iterates over the state symbols directly contained in the scope spanned by the automaton symbol. For each of these, the printer adds a value to the array. The value is the name of the state symbol as a JSON String. As the last step in the serialization of the states, the end of the array is printed. The deserialization begins with storing

```java
public class AutomataGlobalScope extends AutomataGlobalScopeTOP {

  public void loadFileForModelName(String modelName)  {
    ModelCoordinate model = ModelCoordinates
                .createQualifiedCoordinate(modelName, "aut");
    String filePath = model.getQualifiedPath().toString();
    if(!isFileLoaded(filePath)) {
      model = getModelPath().resolveModel(model);
      if(model.hasLocation()) {
        ASTAutomaton ast = parse(model);
         IAutomataArtifactScope artScope =
                 AutomataMill.scopesGenitor().createFromAST(ast);
        addSubScope(artScope);
        addLoadedFile(filePath);
      }
    }
  }

  public ASTAutomaton parse(ModelCooordinate model) { /* omitted here */ }
}
```

Figure 5.27: Load AST and symbol table from models

the scope spanned by the symbol to a variable. At the time the method is invoked from the deserialization algorithm, the scope has already been established because all regular steps of the deserialization that the serialization strategy for automaton symbols foresees have been executed before. For each value of the JSON array of the `"symbols"` member of a serialized automaton symbol, the deserialization performs three steps: (1) read the array value as a JSON String and store it to a variable containing the state name, (2) create a new `StateSymbol` object with the mill of the automata language and set the name of the symbol, and (3) add the symbol to the scope spanned by the automaton symbol. An example of a serialized automaton symbol that uses the compact form of state serialization realized in the hook point methods is depicted at the bottom of Figure 5.26.

5.5.4 Load ASTs together with Symbol Tables

In the symbol management infrastructure [MSN17], symbol tables are not persisted, but instead, artifact scopes can be loaded by searching for model files, parsing these models, and instantiating a novel artifact scope from their AST. While it is generally less efficient to load a symbol table through a model file than through a symbol table file, the presence of AST nodes in loaded symbol tables can be beneficial in certain use cases. Through customization of the typed symbol table infrastructure presented in this thesis, this behavior can be mimicked.

To do so, the method `loadFileForModelName(..)` in the global scope class has to be overridden by applying the TOP mechanism to the global scope class. Figure 5.27 depicts an example of loading models of the automata language. The first step in the

implementation of this method is to create a model coordinate for the location of the model. Other than for symbol table files, where the file extension is indicated via a regular expression, the file extension for the location of the model can, *e.g.,* be set to a fixed file extension of models such as `"aut"`.

The global scope mechanism for avoiding to load an artifact scope multiple times has to be applied to model files instead of symbol table files. Therefore, the model filename for each loaded artifact scope has to be added to the list of loaded files via `addLoadedFile(..)`. Furthermore, loading a model must only be attempted if loading the same model has not been attempted before, *i.e.,* if the filename is not contained in the list of loaded files. In the first attempt of loading a model, the model path of the global scope is used to find an absolute path of the model file that is relative to a model path entry. This is achieved by the `resolveModel(..)` method of the `ModelPath`. In case an existing file has been found, the model coordinate has a location in the form of a URL. This location can be passed as an argument to a `parse(..)` method of a language's parser. If the AST has been successfully created, it can be used as an argument for the `createFromAST(..)` method of the scope genitor of a language. The resulting artifact scope instance is added as subscope of the global scope.

5.5.5 Load Symbol Tables of a Single Language Only

By default, the regular expression for file extensions of symbol table files that are considered during symbol resolution is `".*sym"`. This includes all symbol table files whose naming schema follows the guidelines described in Section 5.2.2. However, the regular expression can be adjusted with the `setFileExt()` method of a global scope. For instance, via setting the file extension to `"autsym"`, only symbols of the automata language are considered.

5.5.6 Load Symbols as Instances of their Subkinds

A scenario for symbol reusability foresees that it shall be possible that a symbol stored as a kind `K` shall be loaded as a symbol of kind `M`, where `M` is a subkind of `K`. To achieve this in the realization of symbol table persistence, the global scope has to be reconfigured. In this scenario, the symbol map in the global scope, by default, has an entry for the symbol kind `K` that employs a `KSymbolDeSer` for the deserialization of stored `K` symbols. The map has to be reconfigured such that it uses an `MSymbolDeSer` for the deserialization of stored `K` symbols. Additional members of a symbol of kind `M` that do not exist in the stored symbols of kind `K` have to be initialized with default values. One approach for achieving this is to realize symbol DeSers in general in a way that these initialize all symbol-kind-specific attributes with default values if no information about the attribute is contained in a persisted symbol. This prevents errors if the symbol DeSers are reused for loading a symbol as a subkind.

However, when a language infrastructure including the symbol DeSers is reused, it might occur that a symbol DeSer does not follow this approach and cannot be modified. In this case, the default values for symbol attributes can also be established by realizing a subclass of the original DeSer that sets the attributes or by setting the attribute default values of such symbols as part of loading the artifact scope in a scope DeSer.

5.7 Load Symbols as Instances of their Super Kinds

Another scenario for symbol reusability foresees that it shall be possible that a symbol stored as a kind `K` shall be loaded as a symbol of kind `J`, where `J` is a super kind of `K`. To achieve this in the realization of symbol table persistence, the global scope has to be reconfigured. In this scenario, the language infrastructure is only aware of the symbol kind `J` but not aware of the symbol kind `K`. The map of symbol DeSers in the global scope requires an entry for the symbol kind `K` that indicates that the `JSymbolDeSer` for symbols of kind `J` shall be used to deserialize such symbols. This can either be achieved by the language engineer who designs the language or it can be passed for reconfiguration through the modelers, *e.g.,* by leaving the global scope reconfiguration open via an optional command-line interface argument. Additional members of a stored symbol of kind `K` that do not exist in symbols of kind `J` are ignored during deserialization.

5.6 Discussion

Both the DeSer classes for symbols and for scopes, as well as the Symbols2Json classes, are not compositional in the context of language composition. It is a deliberate decision not to integrate their instantiation into the language mill because it is not required to configure a language with subtypes of these classes.

The approach for (de-)serializing symbol tables stores artifact scopes top-down beginning with the artifact scope and continuing with the contained subscopes. The deserialization of artifact scopes begins with instantiating the artifact scope and then instantiates the local symbols of the artifact scope. If any of these symbols spans a scope, these scopes are instantiated afterward. Instantiation of objects, hence, is realized top-down from the artifact scope. After a symbol or scope is instantiated, it is connected with the remaining parts of the symbol table infrastructure. Therefore, if a symbol or scope requires information about other parts of the symbol table during its deserialization, only the enclosing parts of the infrastructure are available. Through the statically available global scope instance, information about all other known models is also available.

Java uses classpath entries for looking up both source files (.java) and symbol table files (.class) by default [www21b]. However, the Java compiler enables to specify a source path that deviates from the classpath optionally. If this is the case, Java searches classpath entries for class files only and source path entries for source files only. Our approach for storing symbol tables does not distinguish symbol path entries from model path entries.

Instead, both model files and symbol table files are searched in model path entries, such as in the default case of the Java compiler. Besides the additional complexity, a reason for the decision is to reduce misconfigurations by language engineers. A disadvantage of this is that directories containing model files have to be searched for stored symbol tables and vice versa, which may be inefficient. However, DSMLs often load either symbol tables or models, but rarely both. Furthermore, an optional separation between the model path and symbol path can be added to MontiCore with little effort.

Sometimes, marshaling is used as a synonym for serialization. In the context of Java and the lightweight directory access protocol (LDAP), there is a slight difference: "Marshalling is like serialization, except marshalling also records codebases." [www99] According to this, marshaling an object, therefore, comprises serialization of an object and record of the codebase. This codebase includes, inter alia, the exact location of the Java class to which the serialized object conforms. The purpose of this is that the type information has to be only available at the time of unmarshaling of a marshaled object, *i.e.,* at the time the software is executed and not at the time it is compiled. In the context of symbol table persistence, this is not the case, and therefore, we refer to the processes as serialization and deserialization.

The presented JSON infrastructure does not use the Java reflection API [McC98] and avoids explicit type casts. The avoidance of the Java reflection API requires serializing type information with each object or performing type reconstruction based on the available serialized information of the object. Sometimes it is also possible to reconstruct the type through information about the data structure to which the object structure conforms. To obtain the type information from object structures during serialization, the visitor pattern is employed, and the instantiation of objects based on serialized types during deserialization relies on builders. This, however, requires that visitors and builders for all serialized types (except primitive types) exist. The serialization infrastructure is intended to be used in the context of MontiCore symbol tables for the serialization of symbols and scopes. As the types realizing these are generated (*cf.* Section 4.3), builders and visitors are generated as well. Serialization strategies for scope or symbol attributes of non-primitive types, however, have to be added manually. For this, the serialization infrastructure of MontiCore symbol tables is extensible with handwritten customization through MontiCore's TOP mechanism [HKR21].

5.7 Related Work

Xtext [Bet16] supports the serialization of the abstract syntax in terms of a transformation from the EMF model to a textual representation. In this, serialization is a complement to the parsing and lexing of the textual representation. This is in contrast to the serialization of symbol tables presented in this thesis, which has the primary purpose of providing an interface for language composition on the level of models.

One requirement for a suitable serialization format of MontiCore symbol tables is to keep the symbol table files readable for humans to enable language engineers and modelers to inspect these. Thus, binary serialization formats, such as the Java serialization [www20b] or Protobuf [www20h], are not suitable for our approach. However, binary formats are typically both more efficient regarding the deserialization speed and the size of the serialized data [SM12]. Besides JSON, there are different formats for representing object structures in a textual form that is human-readable.

XML [PSMB+06] enables representing object structures in a textual syntax that, similar to JSON, is human-readable as it is not binary-encoded. In contrast to JSON, XML is a markup language and XML documents typically conform to an XML schema definition. Thus, sophisticated well-formedness checks with regard to their schema can be applied to XML documents. Beyond this, a wealth of languages that operate in the context of XML exist. For example, XPath [DC99] is a language for addressing data entries of an XML document, XQuery [FRF+10] is a language that enables querying data within XML documents, and XSLT [Cla99] is a language for describing transformations of XML documents. Compared to JSON, XML is more verbose, inter alia, since it relies on end tags that match the different kinds of start tags. Furthermore, JSON can be parsed more efficiently than XML [NPRI09] due to its simpler structure.

UML object diagrams [Rum16] are a concise form of representing object structures. The formal model of ODs is well understood [EFLR99]. However, to the best of our knowledge, no standardized textual syntax for ODs exists. Therefore, in contrast to XML and JSON, limited tool support such as sophisticated text editor infrastructure is provided for ODs. Furthermore, ODs allow for more complexity in models, which may complicate engineering efficient parsers and well-formedness checking for ODs. However, the exact efficiency mainly depends on the textual syntax. While XML documents conform to schemata and with JSON schema [www20f], there is similar a effort to realize this for JSON, UML ODs have the advantage that their underlying data structure can be modeled adequately in terms of UML class diagrams [MRR11].

Besides the JSON infrastructure described in this thesis, different libraries exist for serializing and deserializing Java objects with JSON. However, most of these libraries support direct instantiation of Java objects and thus, make use of the reflection API. Gson [www20a] is an open-source JSON library that focuses on intuitive usability. The two methods `fromJson` and `toJson` of the class `Gson` suffice to integrate serialization and deserialization into any Java application. Gson uses reflection to produce serialization strategies for any type automatically. However, Gson can be parametrized with custom serialization strategies as well. Due to the extensive use of final classes, the potential for customization is limited. The JSON infrastructure presented in this thesis is entirely customizable by introducing subclasses.

`org.json` [www20d], sometimes also referred to as `Json-java`, is a reference implementation of the JSON standard, which features encoding and decoding of JSON as well as conversion between JSON and XML, Cookies, and other formats. While the library

has no dependencies to external libraries and does not provide serialization strategies based on reflection, it makes use of unnecessary downcasts and instance-of checks that our infrastructure avoids.

JSON-P [www20e] is an API for the JSON standard that is standardized in Java through a Java specification request [www20g] and is also contained in Java EE [Hef14]. It strictly separates interfaces (the actual API) from the implementation. An implementation is contained, *e.g.,* in Java EE's Glassfish [Hef14] open-source reference implementation. The binding from API to interfaces is carried out by loading the classes via their names using a `ServiceLoader`. For the purpose of persisting symbol tables, this introduces avoidable complexity.

Chapter 6

Using Typed Symbol Tables for Language Composition

MontiCore supports four different kinds of language composition [KRV10, HLMSN+15a], which are language extension, language inheritance, language embedding, and language aggregation. The general notions of these forms of language compositions are introduced in Section 2.1. The symbol table infrastructure of a language is affected by all four kinds of language composition but is also a core enabler for achieving language composition. In MontiCore, language extension and language embedding are technically realized via language inheritance and can be considered special cases of the latter. This chapter explains language inheritance in the STI in Section 6.1. Section 6.2 describes the purpose and realization of symbol adapters translating between symbol kinds and their integration into the symbol resolution algorithm. Section 6.3 presents a means to reuse symbols from foreign technological spaces by the example of Java. Section 6.4 introduces the realization of language aggregation with the STI before Section 6.5 discusses the STI for realizing language composition, and Section 6.6 describes related work.

6.1 Language Inheritance in the Typed Symbol Table Infrastructure

Language engineers can indicate language inheritance in MontiCore through inheritance of the corresponding MontiCore grammars. This section describes how this affects and is assisted by the generation of symbol tables infrastructures in the STI. The effect of language inheritance on parts of the language infrastructure that are not directly associated with the STI – such as parsers and AST nodes – are not in the focus of this thesis and, hence, omitted. The composition of such parts of the infrastructure is explained in the MontiCore handbook [HKR21].

6.1.1 Language Inheritance of Scopes

Language inheritance reflects in the scopes, artifact scopes, and global scopes of a language because these have to be able to resolve for symbol kinds that are defined in

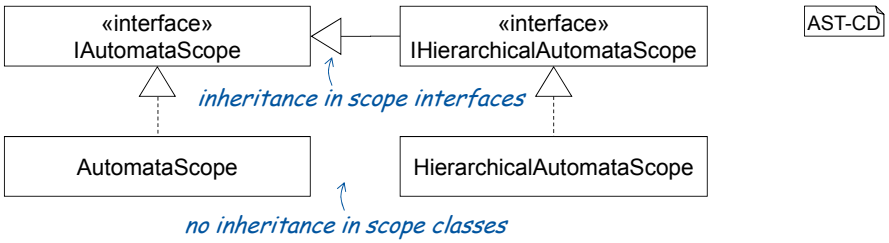


Figure 6.1: Effect of language inheritance on scopes

inherited languages. To realize this, the inheritance relationships in scopes, conceptually, follow the inheritance relationships of languages.

Multiple inheritance is allowed on MontiCore languages and on Java interfaces but not on Java classes. To overcome this, the scopes, artifact scopes, and global scopes are divided into scope classes and scope interfaces, as described in Chapter 4. The composition of languages is carried out through the (artifact, global) scope interfaces. This is depicted in Figure 6.1 by the example of the scope interfaces and scope classes of the automata language and the hierarchical automata language that extends the automata language. The scope interface `IHierarchicalAutomataScope` extends the interface `IAutomataScope`. The scope classes implement the interfaces of the corresponding languages but are not in a mutual inheritance relationship. Therefore, the class `HierarchicalAutomataScope` cannot use any methods defined in the class `AutomataScope`. However, the majority of the functionality of scopes, such as the resolve methods, is located as default implementation in the interfaces. Scope classes, for the most part, manage scope attributes and direct access to these. As the class `HierarchicalAutomataScope` transitively implements the `IAutomataScope` interface, all methods defined there can be reused for the inheriting language. If a language inherits from multiple languages, the scope interface of the language inherits from all scope interfaces of inherited languages. The same inheritance relationship for scope interfaces also holds for artifact scope interfaces and global scope interfaces. For instance, the interface `IHierarchicalAutomataArtifactScope` extends the interface `IAutomataArtifactScope`, but the artifact scope classes of the languages are not in a mutual inheritance relationship.

The reuse of default method implementations in scope interfaces implies that any handwritten adjustments of such method implementations through the application of the TOP mechanism to the scope interfaces can be reused for language inheritance. Contrary to this, handwritten adjustments to scope classes are not reused. The reason for this is that multiple inheritance on languages would only allow extending at most a single scope class of an inherited language. Artifact scope classes and global scope classes extend the scope class and, hence, inheritance between these and the artifact and global scope classes of inherited languages would always require multiple inheritance.

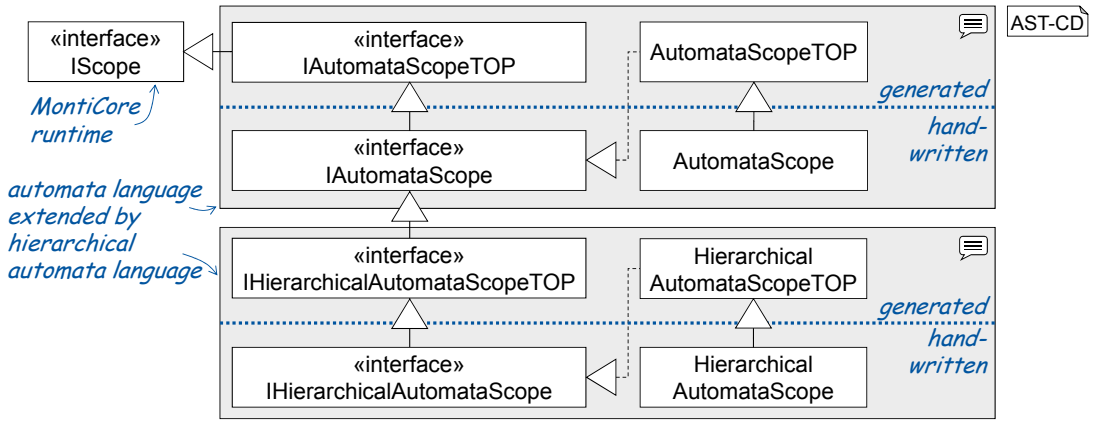


Figure 6.2: Integration of handwritten code into scopes

A scenario for handwritten adjustments of scope classes and scope interfaces for the automata language and for the hierarchical automata language is depicted in Figure 6.2. In this example, both scope classes and both scope interfaces are extended via the TOP mechanism. Hence, the handwritten class `AutomataScope` extends the generated class `AutomataScopeTOP` and the handwritten interface `IAutomataScope` extends the generated interface `IAutomataScopeTOP`. The same applies to the scope class and interface of the hierarchical automata language. The inheritance relationship that realizes the language inheritance is carried out through the generated interface `IHierarchicalAutomataScopeTOP` that extends the handwritten interface `IAutomataScope`. Through this, the handwritten adjustments to the `IAutomataScope` interface are considered in the hierarchical automata language. The adjustments, hence, are also transitively available in scope classes of the hierarchical automata language. Any handwritten adjustments to the scope class of the automata language, however, are not available in the scopes of the hierarchical automata language.

The language's mill creates all scope objects of a language and manages singleton instance of the global scope. In the presence of language inheritance, there is only a single global scope instance for all involved languages. By reconfiguring the mills of inherited languages (*cf.* Section 6.1.4), the infrastructures of inherited languages instantiate the intended scope types of the most concrete language.

6.1.2 Language Inheritance of Symbol Table Creation

As described in Section 4.3, the scope genitor of a language is responsible for instantiating symbol and scope objects for a model. To do so, the scope genitor traverses the AST with the visitor infrastructure of the language. The scope genitor delegator (*cf.* Section 4.3.9) employs a traverser (*cf.* Chapter 2) to enact the traversal of an AST and

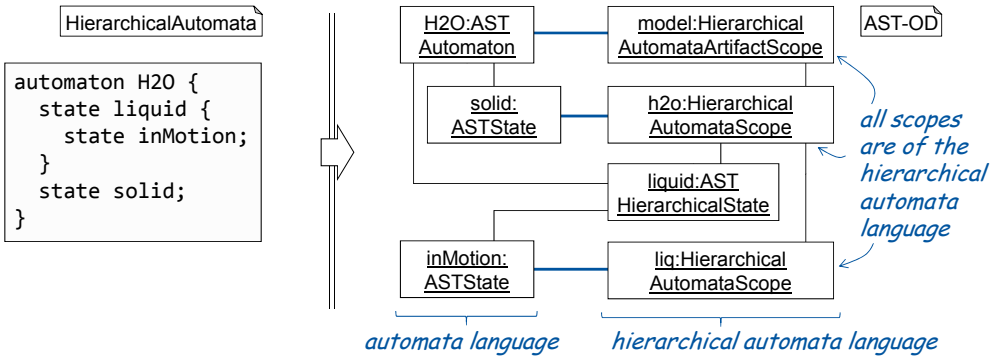


Figure 6.3: Model of a composed language (left) and excerpt of scopes and AST nodes instantiated by processing the model (right)

delegate the `traverse`, `handle`, `visit`, and `endVisit` methods to the corresponding scope genitor methods of the language. In the presence of language inheritance, traversers are able to traverse ASTs and delegate the visitor methods to visitors and handlers of the individual languages. An advantage of this delegation is that it does not require the visitor and handler classes to be in a relationship with visitors and handlers of other languages. To achieve this, the scope genitor delegator configures the traverser not only with the scope genitor of the own language but also with the scope genitors of all transitively inherited languages. This enables reusing handwritten adjustments to `visit` method implementations in all scope genitors of the individual languages.

If a model conforming to a language that inherits from one or more other languages is processed, the symbol table creation involves creating scope instances and setting these as enclosing scope, subscope, and spanned scope attributes of AST nodes, symbols, and other scopes. The ASTs of such models typically contain nodes from different languages, which are processed by different scope genitors controlled by the scope genitor delegator. Nonetheless, the symbol tables for all models of the language use only the scope type from the language and no scope types from inherited languages. The reason behind this is that only the scopes of the current language are able to resolve for symbols of all known kinds and consider potential handwritten scope adjustments. As the scope interfaces of a language extend the scope interfaces of inherited languages, using the more specific scope does not violate type restrictions, *e.g.,* of the methods for access and manipulation of the enclosing scope of an inherited language's AST node.

For example, an object diagram with AST nodes and scopes instantiated while processing the `H2O` automata model with the scope genitor delegator of the hierarchical automata language is depicted in Figure 6.3. The hierarchical automata model depicted on the left side has a non-hierarchical state `solid` and a hierarchical state `liquid` with a contained state `inMotion`. An excerpt of the AST nodes and scopes instan-

tiated while the model is processed is depicted on the right side of the figure. The relations between AST nodes and the enclosing scopes equal the relations that would have been created when instantiating the symbol tables for individual languages, as explained in Section 4.1.10. An exception to this is that the AST nodes of the automata language, such as the `ASTState`, have an enclosing scope that is an instance of the `HierarchicalAutomataScope`. All three relations of AST nodes from the automata language to scopes of the hierarchical automata language are visualized in the figure as bold lines. The same holds for symbols of the automata language that are omitted in the figure for better readability.

The associations across the "borders" of the languages are realized via reconfiguration of the scope instantiation process of inherited languages, which is performed through reconfiguration of the mills of the inherited languages as described in Section 6.1.4. The process of mill reconfiguration is contained in the generated language infrastructure and does not require manual interaction by language engineers. To this end, all scope instances created in the generated language infrastructure are instantiated via the mill. To support language inheritance, all scope instances that are created in handwritten code must be conceived via the mill as well.

6.1.3 Language Inheritance of Symbol Table Persistence

Similar to the instantiation of symbol tables, the symbol table persistence relies on the visitor infrastructure of a language (*cf.* Chapter 5). Other than the symbol table instantiation that traverses the AST, the symbol table persistence traverses the symbols and scopes. However, in both cases, the traversal is carried out by a traverser. Like the scope genitor delegator for symbol table instantiation, the Symbols2Json class holds an attribute of the language's traverser for storing symbol tables. Symbol DeSers for symbol kinds defined in inherited languages can be reused.

All symbol DeSers are obtained via the map of symbol DeSers in global scopes. For reusing symbol DeSers of inherited languages, the global scope has map entries for all symbol kinds defined in inherited languages as well. If a symbol kind spans a scope, the scope is serialized and deserialized with the scope DeSer obtained from the global scope in a procedure described in Chapter 5.

For language inheritance, the scope DeSer of a language is not reconfigured by default. The reason for this is that the inheritance of scope DeSer classes does not follow the inheritance relationship of scopes, *i.e.,* if a language inherits from another language, the scope DeSers are not in an inheritance relationship. An inheritance relationship between scope DeSers would require separating scope DeSer interfaces and classes to support multiple inheritance of languages in the same way as the separation for scopes. This would, among other things, introduce additional complexity into potential manual adjustments and reduce the compile time of the language infrastructure. However, the scope instances that are created during deserialization are created via the language's

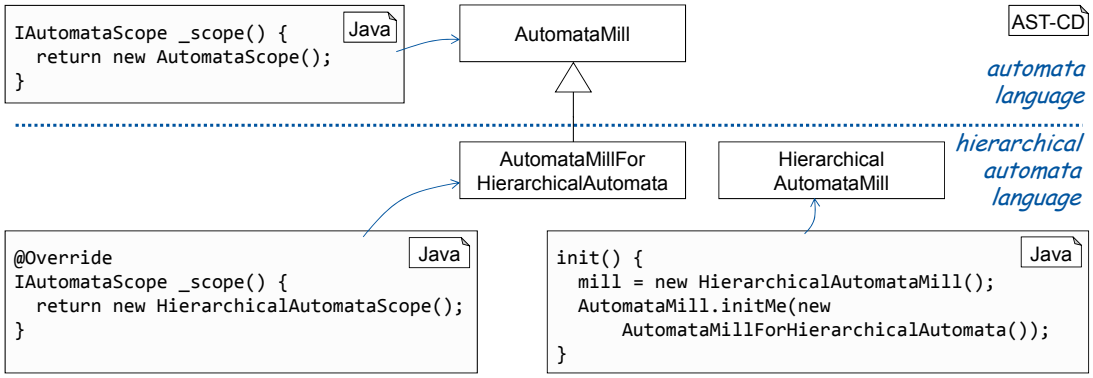


Figure 6.4: Reconfiguration of mills for language inheritance

mill. In language inheritance, the mill is reconfigured to create instances of the correct scope type, as described in Section 6.1.4.

6.1.4 Reconfiguration via Mills

As described in Section 4.3.1, language mills are singleton classes, inter alia, for the purpose of obtaining instances of symbol table classes. If a language inherits from one or more languages, the mills of the inherited languages are (re)configured to create instances of the desired types. For instance, if the automata language is reused for the hierarchical automata language through language inheritance, the mill of the automata language has to instantiate scopes of the hierarchical automata language instead of the scopes for the automata language it instantiates otherwise. As the scopes are in an inheritance relationship, the mill can be reconfigured by overriding the methods that instantiate scopes and let these instantiate the desired subtypes of the scopes to produce integrated symbol tables, as depicted by example in Figure 6.3.

This enables reusing the entire language infrastructure of inherited languages without any modification of the language infrastructure that requires manual interaction or a new generation of the language infrastructure for the reused languages. Instead, languages are configured via *bridge mills*. A bridge mill acts as a facade for a language's mill that reconfigures the original mill with the purpose of reusing the language as inherited language. For each language that a language inherits from – either directly or transitively through other languages – MontiCore generates a bridge mill as part of the language's infrastructure. For example, MontiCore generates the bridge mill `AutomataMill4HierarchicalAutomata` as part of the infrastructure for the hierarchical automata language. This bridge mill configures the `AutomataMill` for using the automata language as part of the hierarchical automata language. Technically, bridge mills are Java classes that extend the Java classes realizing language mills. Bridge mills

override the non-static methods of language mills for obtaining the global scope, artifact scope, and scope instances. Furthermore, bridge mills override the methods for obtaining the language's traverser. As the inheritance in all the above parts of the language infrastructure follows language inheritance, the overridden methods can return instances of the inheriting language instead of instances of the inherited language.

An example of the reconfiguration of the automata mill for reuse in the hierarchical automata language is depicted in Figure 6.4. The bridge mill class extends the `AutomataMill` and overrides all methods that create instances of language infrastructure constituents that the bridge mill reconfigures. For example, the class overrides the method `_scope` to return scope instances of the desired type. In the `init` method of the `HierarchicalAutomataMill`, the mills of all (directly and transitively) inherited languages are initialized with the bridge mills via the mills' `initMe` methods. In this example, the hierarchical automata language extends only the automata language. If a language inherits from multiple languages, all mills of inherited languages are reconfigured with bridge mills. With these reconfigurations, the infrastructure of the automata language that uses the automata mill to create scope instances creates scopes of the hierarchical automata language instead.

6.2 Adapting between Symbol Kinds

In all forms of language composition that MontiCore supports, it can occur that a symbol usage should refer to a symbol definition of a model conforming to another language. In language inheritance and language extension, a language is typically aware of other languages that it reuses. Hence, the language engineers can ensure that the (expected) symbol kind of the name usage and the (provided) symbol kind of the name definition are compatible. Language embedding and language aggregation, on the other hand, typically compose two or more independent languages that are not aware of one another.

For instance, a language `L1` can be embedded into a language `L2`, which creates a new language `L3`, and `L1` and `L2` remain unaware of each other. Furthermore, the input languages of language embedding and language aggregation typically should be immutable with respect to language composition. To this end, it may occur that a language engineer composing languages intends that the symbol usage points to a symbol definition of a model conforming to another language, although the expected and the provided kind of the symbol definitions differ. For this purpose, a symbol can be adapted from a source symbol kind to a target symbol kind with a *symbol adapter*. The following sections first introduce the concept of symbol adapters in the STI before explaining how resolving for adapted symbol kinds is integrated into the symbol resolution algorithm and how symbol adapters in combination with symbol table persistence foster efficient language aggregation.

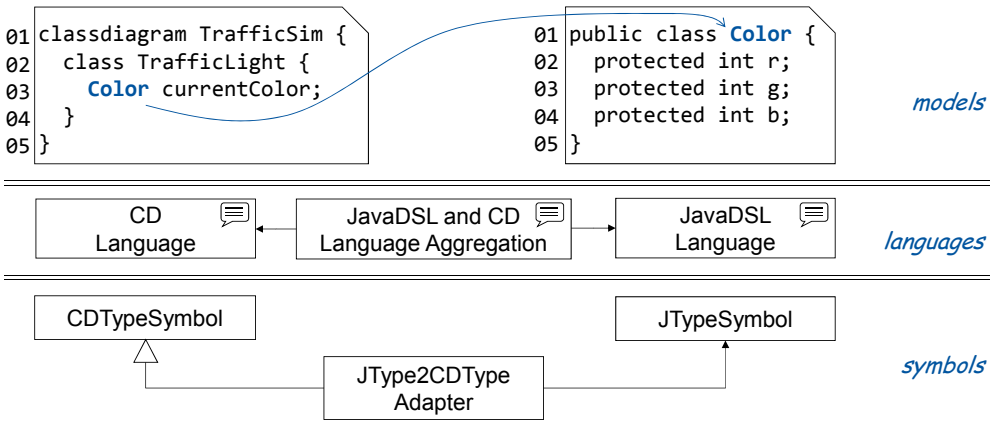


Figure 6.5: Example for adaption between symbol kinds on the level of models, languages, and symbols

6.2.1 Concept for Symbol Adapters

At the core of symbol adaptation are symbol adapters that translate a symbol from a source kind to a target kind. A symbol adapter realizes the adapter pattern [GHJV95] for symbol classes. In the technical realization, a symbol adapter is a (handwritten) Java class that extends the symbol class of the target symbol kind and has an attribute `adaptee` with the type of the source symbol kind. The adapter class overrides all necessary methods of the target symbol class and, in the method implementations, delegates to methods of the adaptee. Which methods are necessary to override depends both on the symbol classes and on the purpose of the symbol adapter. Symbol adapters do not alter the source symbol class, which enables the language engineers to resolve for the original symbol with the source symbol kind.

An example of a symbol adapter is depicted in Figure 6.5. The top row depicts two models. The model on the left is the CD model `TrafficSim` that uses the name `Color` as the type of the attribute `currentColor` of the class `TafficLight`. The model depicted on the right side is a model of the class `Color` that conforms to a `JavaDSL` language, which is a DSML with Java-based syntax. It is intended that the name usage of `Color` in the CD model refers to the name definition of `Color` in the Java class model. On the level of languages depicted in the center row, this is realized via language aggregation between the CD language to which the `TrafficSim` model conforms and the JavaDSL language to which the `Color` class conforms. The result is a language aggregation of the JavaDSL and the CD language. The bottom row of the figure depicts the symbol adapter class as well as the Java classes realizing the source and the target symbol kind. The source symbol kind `JTypeSymbol` depicted right is the symbol kind in which the symbol definition is provided. The target symbol kind `CDTypeSymbol`

depicted left is the symbol kind for which the symbol resolution searches. The middle depicts the adapter class that extends the class `CDTypeSymbol` and has an attribute of the `JTypeSymbol`. By convention, an adapter class for a source symbol kind `SSymbol` and a target symbol kind `TSymbol` has the name `S2TAdapter`[1]. In this example, the name of the adapter is `JType2CDTypeAdapter`.

6.2.2 Finding Symbol Adapters during Symbol Resolution

Symbol adapters are handwritten Java classes that can be integrated into a language. To integrate symbol adapters into a language, they must be found during resolution for symbols of the target symbol kind. The symbol resolution, as described in Section 4.1.8, takes into account that language engineers intend to resolve for adapted symbols during resolution for local symbols in a scope. Hence, the hook point method `resolveAdaptedTLocallyMany` is contained in a scope interface for every symbol kind `T` of the language. The default implementation of the hook point method is empty, but by applying the TOP mechanism to the scope interface, the hook point method(s) for adapted symbols can be overridden. A language engineer who intends to search for symbols of kind `S` when the language tool searches for symbols of kind `T` can override the method `resolveAdaptedTLocallyMany`. The new implementation of this method can also search for symbols of multiple source kinds. In the new implementation of the method, the resolution should delegate to the resolution for symbols of kind `S` with the method `resolveSLocallyMany`. Each element in the resulting list of symbols of kind `S` has to be adapted by instantiating a symbol adapter translating the symbol to a symbol of kind `T`. The list of adapters is then returned by the method.

With this approach, language engineers can integrate any adapter into the symbol resolution as far as the language is aware of the source symbol kind. If this is not the case, the resolution has to be continued in a global scope of a different language. The following describes how this is achieved in the STI with symbol resolvers.

Symbol Resolvers

Resolvers enable resolving symbols in global scopes of foreign languages. A resolver is always specific to a symbol kind `T`, for which it resolves. It resolves either for symbols of kind `T` in a foreign global scope or it resolves for symbols of another symbol kind `S` for which it instantiates an adapter translating the symbols to the symbol kind `T`. The language employing the resolver is not required to be aware of the symbol kind `S`.

For each symbol kind `T` of a language, MontiCore generates a resolver interface `ITResolver`. The global scope of a language manages a list of resolvers for each symbol kind of the language. During the inter-model resolution of symbols of a kind `T`, global scopes resolve for adapted symbols. For this, the global scope iterates over all resolvers

[1] `S` stands for the `Source` and `T` represents the `Target`

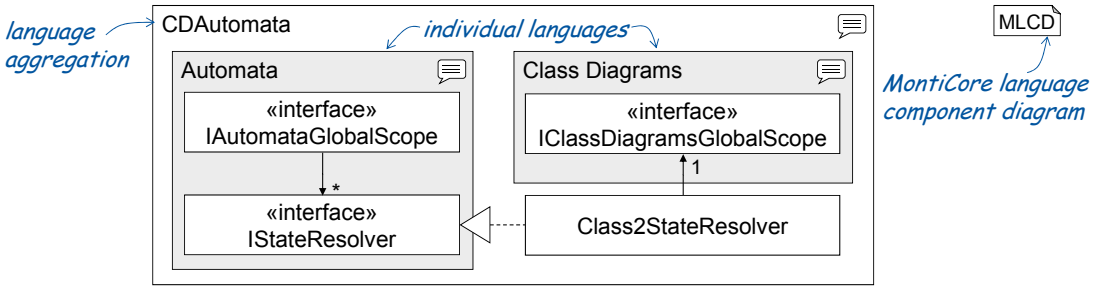


Figure 6.6: Example for a symbol resolver realizing language aggregation

for the kind `T` of which it is aware. By default, no resolvers are configured in a global scope. Language engineers can develop resolver classes that implement the generated resolver interfaces and perform custom symbol resolution. To consider a resolver during symbol resolution, language engineers have to add an instance of the resolver class to the global scope.

An example of using a resolver is depicted in Figure 6.6. The left side of the figure displays the type of the global scope of the automata language that manages a list of resolvers for state symbols. The top right part of the figure displays the type of the global scope of a class diagram language. The bottom right displays a resolver class that carries out a language aggregation between the automata language and the class diagram language if it is added to the global scope of the automata language. If added, the `Class2StateResolver` class is utilized during resolution for state symbols in the automata global scope and internally resolves for class symbols in the class diagram language. If a suitable class symbol is found, an instance of a `Class2StateSymbolAdapter` is created and added to the result of the resolution for state symbols. To decouple the languages, the resolver class is neither considered a part of the automata language nor a part of the class diagram language.

Avoiding Circular Resolution

If a language uses multiple symbol adapters, it might be possible to form cyclic adaptations. For example, we consider a language that uses two symbol adapters. One adapter adapts symbols of kind `K` to symbols of kind `J`, and another adapter adapts symbols of kind `J` to symbols of kind `K`. Without a mechanism to avoid cyclic adaptations, the resolution process described in Section 6.2.2 would not terminate. The resolution for symbols of kind `J` would trigger the resolution for kind `K`, which again would trigger resolution for the kind `J`.

To avoid such cycles, each scope instance must resolve for a symbol of a specific kind at most two times: the first resolution occurs during bottom-up resolution and the second resolution occurs during top-down resolution in the scope. The top-down resolution in

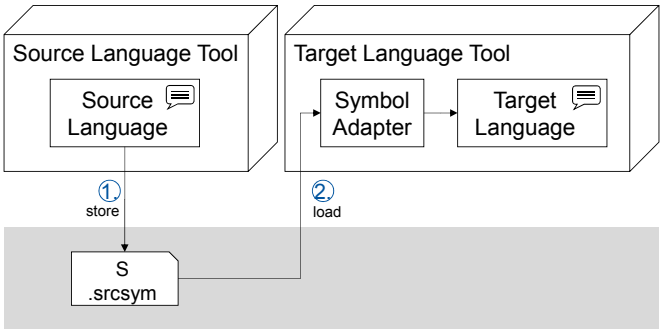


Figure 6.7: On-demand symbol adaptation: a symbol is loaded as its source kind and then adapted to the target kind

the same scope may find a symbol that has not been found during bottom-up resolution if the symbol name is qualified or unqualified as part of the inter-model resolution that is executed in between.

To this end, the resolution contains a mechanism to avoid such cyclic resolutions with a Boolean flag for each symbol kind contained in each scope instance. For a symbol kind `K`, the flag is called `kAlreadyResolved`. Each scope instance skips the resolution of a symbol at the beginning of the `resolveMany` and `resolveDownMany` methods if the corresponding flag has the value *true*. Global scopes further do not search for adapted symbols with the `resolveAdapted` methods if the flag is set to the value *true*. The flag is set to *true* before each call of the `resolveLocallyMany` method and is set to $false$ afterward. This prevents that the resolution of any adapted symbols continues with searching for symbols of a kind that has already been resolved for during both bottom-up and top-down resolution.

Local resolution of a symbol triggers resolution of adapted symbols (*cf.* Section 4.1.8) in the scope, which could lead to cyclic calls of the method without the mechanism for avoidance of circular resolution.

6.2.3 Combination of Symbol Adapters and Symbol Persistence

Together with symbol table persistence as described in Chapter 5, symbol adapters can support the realization of language aggregation with different forms of efficient translations between symbols of a source kind and symbols of a target kind. A requirement for this is that source language and target language must not be modified permanently, as both languages should be reusable in different contexts and independent of each other.

Temporal reconfiguration of a language tool, *e.g.,* with symbol resolvers as described in Section 6.2.2, is not problematic as it typically has no side effects if it is applied correctly. Hence, a symbol adapter that is part of a language infrastructure would per-

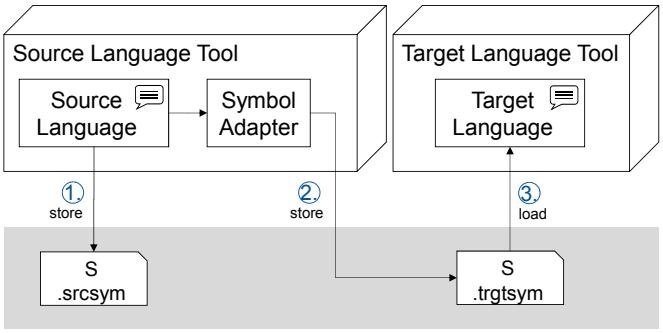


Figure 6.8: Polyglot symbol persistence: a symbol is persisted in multiple representations for different kinds

manently tie a language to another language. We distinguish three different approaches for realizing combinations of symbol adapters and symbol persistence that are explained in the following sections.

On-Demand Symbol Adaptation

For realizing on-demand symbol adaptation, the source language stores symbol tables as usual. In the example depicted in Figure 6.7, the source language tool produces and stores a symbol table for the model `S` in the file `S.srcsym`. By default, the target language is not able to load symbol tables of the source language. However, with (re)configuration of the target language, inter alia, with the regular expression enabling it to recognize `srcsym` files as symbol table files, the language tool is able to find and load the symbol table. Furthermore, the target language has to be configured to search for symbols of the source kind, instantiate a symbol adapter, and return the adapter as a result of resolution for the target kind as described in Section 6.2.2.

An advantage of this solution is that the symbol is only persisted in a single representation, which avoids inconsistencies that could occur if the same symbol was stored in different representations and individual files. Furthermore, transporting the symbol from the source language to the target language requires only two load/store operations.

However, loading a symbol with on-demand symbol adaptation requires instantiating the adapter first. Thus, loading an adapted symbol is less efficient than loading a symbol in the target symbol kind representation directly. It is feasible to assume that a symbol is loaded and adapted more often than it is stored and thus, the balancing may be disadvantageous. However, the instantiation of a symbol adapter typically does not cause much effort.

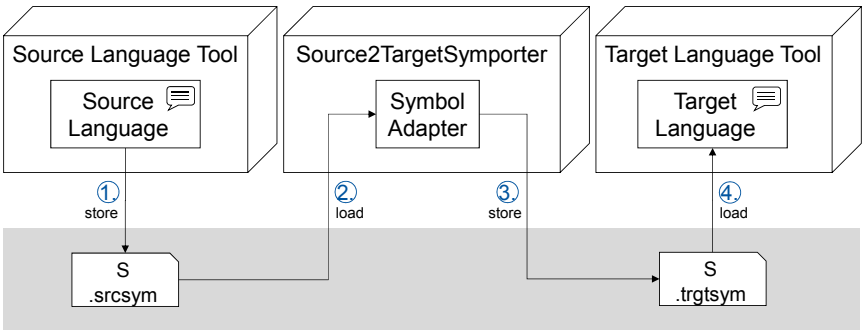


Figure 6.9: Standalone symbol translation: a standalone tool carries out the translation from source to target symbol kind

Polyglot Symbol Persistence

The concept of polyglot symbol persistence is depicted by example in Figure 6.8. The source language tool processes and stores the symbol table of a model `S`. For polyglot symbol persistence, the source language is configured to store symbols in different representations. Whenever the source language tool stores a symbol table file, it triggers adapters that translate the symbols into different kinds. Afterward, the resulting adapted symbols are stored. In the example of the figure, the symbol adapter translates symbols in the artifact scope of `S` and stores the artifact scope in the file `S.trgtsym`. The symbol adaptation should be triggered in the source language tool whenever a symbol table is stored. Language engineers have to conceive individual (re)configuration infrastructure in the language tools. A target language can load the symbol in the target symbol kind representation. This reduces the cost of loading a symbol compared to the on-demand symbol adaptation. Since all adapters that translate a symbol from a specific source kind are triggered together, all representations of a persisted symbol are produced at once, and the risk of inconsistencies between different stored symbol representations is relatively low. This solution requires three load/store operations.

Despite the advantages of this approach, the assumption that all potential representations of a symbol are available at the time a symbol is stored is unrealistic. Hence, polyglot symbol persistence does not support to reuse libraries of arbitrary (processed) source language models without configuring and executing the source language tool. This raises the problem that the source language tool co-exists in different configurations, and each configuration has to be handled individually. It might occur that target language tools invoke suitable configurations of the source language tool on-demand, and the source language tool has to re-produce symbol tables in different representations again. Thus, it cannot be assured that all symbol representations of a symbol are stored at once, which again raises the potential for inconsistencies.

Standalone Symbol Translation

In the standalone symbol translation, depicted by example in Figure 6.9, the source language tool that produces a symbol table for the model `S` stores it only in its source kind. A standalone tool loads symbols of a source kind, translates these symbols into a target representation, and stores these afterward. This tool is specific to the pair of source and target languages, and the translation requires four load/store operations. An advantage of this solution is that the process of symbol adaptation is neither bound to loading a symbol during resolution (*cf.* Figure 6.7) nor to the process of storing a symbol during model processing (*cf.* Figure 6.8). Instead, it can be performed independently and, hence, does not decrease the efficiency of processing models with source and target language tools.

However, standalone symbol translation requires a well-defined processing toolchain, as otherwise, it can lead to inconsistencies between the different stored representations of a symbol. If the source representation of a symbol table is modified and the tool of the source language persists this symbol table, the symbol translator tool is not triggered to translate the source representation into the target representation. Hence, the persisted representations of the symbol table remain inconsistent until the symbol translator tool is executed again. While inconsistencies between symbol table objects and their persisted representations can occur in all three approaches, the standalone symbol translation has a greater risk of producing inconsistencies due to the fact that a third tool is involved. The tool executions can be coordinated with a proper build management tool, such as Gradle [Mus14]. To this end, the approach is not suitable if the source symbol may potentially be modified. Despite this limitation, the approach is useful for importing a library of symbols from a foreign representation. Therefore, we also refer to the standalone symbol translator tool as a symbol importer (or short, `Symporter`). The tool that translates symbols from the `Source` language to the `Target` language, hence, is called `Source2TargetSymporter`.

6.3 Importing Symbols from Java with Class2MC

The techniques described in the previous sections are able to bridge the gap between different MontiCore languages and models. In practice, it may be the case that a MontiCore language should enable its models to reuse models produced by tools from different technological spaces. In other words, a MontiCore language should be able to refer to elements of languages that are not available as MontiCore languages. For example, the engineers of a MontiCore language for class diagrams intend to use Java types that are loaded from Java's `class` files in the class diagram language. More precisely, a class in the class diagram language should be able to indicate that it extends or implements a Java type that is available from a `class` file or to use a Java type as the type of a class member, return type of a method, or similar.

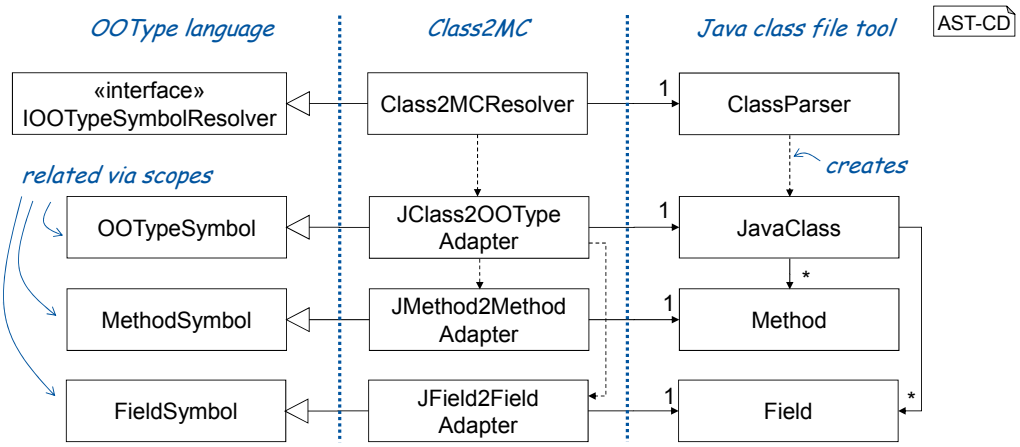


Figure 6.10: Central types of the Class2MC tool that enables importing Java types into MontiCore languages

The STI contains generated resolver interfaces that are integrated into the resolution for symbols of known kinds. For instance, the class diagram language contains the `ICDTypeResolver` that can be implemented by a handwritten class that is added to the global scope of the CD language. This handwritten class can resolve for adapted `CDTypes`, *e.g.,* in global scopes of foreign MontiCore languages. Moreover, the resolver can be utilized to search for foreign representations of `CDTypes` in technological spaces beyond MontiCore.

This section presents the Class2MC tool, which enables to import Java types from `class` files into the universe of MontiCore languages. MontiCore's built-in type system framework contains the `OOSymbols` grammar [HKR21] that defines the syntax for type definitions in object-oriented languages. MontiCore languages can extend this grammar to reuse the syntax for type definitions and reuse a type check for such symbols.

The Class2MC tool is realized as a resolver for `OOTypeSymbols`. The `Class2MC-Resolver` implements the interface `IOOTypeSymbolResolver` of the `OOSymbols` language. Whenever an `OOTypeSymbol` is resolved, the `Class2MCResolver` searches for suitable Java types loaded from class files, instantiates symbol adapters, and contributes these to the result of the symbol resolution.

The tool internally employs a tool that parses class files and instantiates the abstract syntax of a Java type. This is performed by the class `ClassParser`. If the resolved name identifies a class file that is located relative to a location indicated in the model path of the global scope of the language, the `ClassParser` parses the class file. The result of parsing the class file is an instance of the class `JavaClass` that realizes the abstract syntax of a class file and, hence, contains information about the content of the class file, such as the top-level type definition(s). The Class2MC tool instantiates an

artifact scope of the OOSymbols language for each class file that is loaded. Afterward, the symbol definitions of the class file have to be adapted to the OOSymbols language.

A symbol adapter adapts the JavaClass to an OOTypeSymbol. Furthermore, all methods and fields of the Java type are translated into the corresponding elements of the OOSymbols symbol table with suitable adapters, as depicted in Figure 6.10. In the abstract syntax instantiated by the ClassParser, methods and fields are realized as attributes of the JavaClass. In the symbol table infrastructure of the OOSymbols language, this is realized differently. MethodSymbols and FieldSymbols are located in scopes that the OOTypeSymbols span. Through this, the symbol resolution strategy of MontiCore is able to resolve for methods and fields as well. As the last step, the Class2MCResolver resolves for the symbol in the newly created artifact scope.

To avoid new instantiation of adapters if a particular Java type is resolved more than once, the Class2MC tool adds all adapters to the corresponding artifact scopes and all artifact scopes to the global scope.

The engineer of the class diagram language can use the Class2MC tool if the class diagram language inherits from the OOSymbols language. Beyond being able to use Class2MC, inheriting from the OOSymbols language enables the CD language engineer to use MontiCore's built-in type checker and simplifies integration with other MontiCore languages that define or use types via the built-in type system framework. The Class2MC tool is integrated into the CD language by adding the Class2MCResolver class to the CD global scope with the following statement:

```
1 CDMill.globalScope()
2     .addAdaptedOOTypeSymbolResolver(new Class2MCResolver())
```

The Class2MC tool realizes on-demand symbol adaptation. Instead, it could have been realized as a standalone symbol translation tool that translates a library of class files into a library of OOSymbols symbol table files. The latter is better suitable for translating the entire Java runtime environment into the MontiCore representation since loading a stored symbol table is more efficient than instantiating the adapters.

However, the standalone tool would be less flexible if an application intends to use local class files. In such situations, the standalone tool would have to be called individually before the remaining models of an application are processed. Therefore, the Class2MC tool is realized as an on-demand symbol adaptation that can be better integrated into the workflows of processing models in different applications. Furthermore, most languages use only a tiny part of the entire Java runtime environment. Translating all types of the Java runtime environment, hence, would create numerous symbol table files that are not used at all.

6.4 Aggregation of Languages

Language aggregation [HKR21] is a loose coupling between languages whose models remain in separate artifacts per language. A model of one language may, however, refer to elements of models conforming to other languages. In MontiCore, this is reflected through the symbol table infrastructure in which the resolution of name usages yields the symbol definitions that may be located in other models.

One purpose of symbols is being surrogates for foreign models, typed through the symbol kind. Languages, thus, use symbols for acquiring the information they require of other models. These models may conform to the same or to a different DSML. In the latter case, it is beneficial for language reusability to realize a loose coupling between the DSML that requires information of a model conforming to another DSML.

The following describes four alternative methods for realizing language aggregation with the STI, where each method has individual advantages and disadvantages. Language engineers have to decide on a suitable form of language aggregation for each application individually.

6.4.1 Aggregation through Shared Grammar

In the STI, the resolve methods are located in scope interfaces, and the inheritance of scope interfaces follows the inheritance of languages. Language engineers can utilize this if the languages that are intended to be aggregated are engineered anew. In such a scenario, a language engineer can extract all symbol-defining nonterminals to a common, shareable language that all languages-to-be-aggregated extend. As an effect of this, the resolve methods for all symbols are available in all languages and enable symbol resolution across different languages. This form of language aggregation requires little effort for the actual integration of the languages but requires a-priori knowledge about the aggregation at the time the languages are engineered. This form of language aggregation, hence, is suitable for preparing a language to be used in combination with other languages.

MontiCore's built-in type system framework contains grammars that define symbols for different kinds of type definitions. For instance, the grammar `OOSymbols` introduces nonterminals that define types in object-oriented languages. If a language, such as a class diagram language, reuses the nonterminals of the `OOSymbols` language to define the symbols of class diagram types, other languages can be integrated with the class diagram language through a shared grammar. For example, we consider an object diagram language that is intended to be aggregated with the class diagram language such that the type of an object in the object diagram refers to a type definition in the class diagram language. If the object diagram language extends the `OOSymbols` language, the scope of the object diagram language is able to resolve for type symbols that are inter alia, used for type definitions in class diagrams. Through the proper configuration of

the symbol table persistence infrastructure in the object diagram language, the symbol table files of the class diagram languages can be found, and the symbol tables can be instantiated.

An advantage of this form of language aggregation is its simplicity as ideally, no implementation dedicated to the language aggregation has to be conceived. Furthermore, the language aggregation can be extended with additional languages via language inheritance with little effort. However, a disadvantage is the prerequisite of languages to inherit from the common language. This causes a weak form of conceptual coupling between the aggregated languages since it is not possible to aggregate arbitrary languages.

6.4.2 Aggregation through Unifying Grammar

Another approach for achieving language aggregation with the STI relies on producing a novel grammar that inherits from all grammars of the languages that should be aggregated. Upon execution of the MontiCore generator, language infrastructure is generated from the unifying grammar. This grammar, however, is not intended for obtaining an integrated parser rather than an integrated symbol table infrastructure in which symbols of all aggregated languages can be resolved. The integrated grammar is necessary since large parts of the STI are generated from a language. Especially, this produces a global scope that is aware of all languages. The grammar that integrates the languages can optionally introduce a novel start rule that uses an alternative over all start rules of the aggregated languages on its right-hand side. Through this, the language infrastructure generated from the integrating grammar is capable of parsing models of all aggregated languages. This form of language composition, however, is merely language extension rather than language aggregation.

For example, an object diagram and a class diagram language can be aggregated by conceiving a novel grammar `CDandOD` that extends the grammars of the object diagram and the class diagram language. The tool for the novel grammar can realize a context condition that checks the validity of name usages across the border of the aggregated languages. In the example, a context condition could check whether all names that are used as types of objects in the object diagram refer to names of type definitions in a class diagram by resolving for type symbols in the class diagram.

The aggregation through a unifying grammar is especially suitable for language aggregations that are rarely extended with new languages. A disadvantage of the mechanism is that extending the aggregation with additional languages requires adjusting the grammar and re-generating the language infrastructure from the grammar.

6.4.3 Aggregation through Resolvers

A further method for achieving language aggregation relies on separate global scope instances for each aggregated language that are connected to each other via symbol

resolvers. Resolvers are explained in Section 4.3.5 and enable resolving for symbols in foreign global scopes. This form of language aggregation can be extended with new languages with little effort, as it requires adding other resolvers to the global scopes only instead of re-generating novel language infrastructure such as in the approaches described in Section 6.4.1 and Section 6.4.2. Furthermore, the aggregated languages do not have to be coupled, are not aware of each other, and do not share any common language infrastructure.

6.4.4 Aggregation through Symbol Files

With the STI and the mechanism for symbol table persistence, languages can be aggregated without any integrating infrastructure. As the type of scopes is not persisted, the scope type instantiated while a symbol table is loaded can be different from the type of the scope that was stored. However, this does not hold for symbols, whose kind is always persisted. To this end, two languages $L1$ and $L2$ can be aggregated by interpreting stored symbol tables of the language $L1$ as stored symbol tables of $L2$. This is feasible since all global scopes use a regular expression to identify the file extensions of files that are considered for loading symbol tables. Using a regular expression that includes symbol files of all aggregated languages as a filename extension considers the symbol tables of all these languages. By default, symbol tables are stored by translating the association between a symbol and its spanned scope into containment in JSON (*cf.* Chapter 5). Hence, the deserialization can only consider symbols and scopes contained in symbols of known kinds. If an unknown symbol kind occurs during deserialization, the deserialization of the symbol and all transitively contained scopes is omitted. For example, a statechart language that is aware of state symbols could load the state symbol `Pause` from a symbol table stored by an automaton language (*cf.* Figure 5.16) only if the language is aware of automaton symbols either directly or through a symbol adapter that translates these to a known kind. Therefore, language aggregation through symbol files can only be applied in scenarios in which the symbols that should be resolved across the borders of the language are available in the stored symbol tables through known symbol kinds or through symbol adapters that translate symbols into a known kind. Language engineers must consider this when performing language aggregation through symbol files.

An advantage of this approach is that it does not require any form of explicit integration, and the infrastructures remain completely separated. However, besides the restriction of its applications for scenarios in which symbols are exported to the artifact scopes only via known symbol kinds, the approach has certain disadvantages. A disadvantage of this approach is that through re-interpretation of the type of the stored scope instances, any scope-specific information that is stored in a symbol table is lost. For instance, if an automata language uses a scope attribute defined via a scope rule, the attribute values are lost when the symbol table is loaded with a different language. A further disadvantage is the risk of loading symbol tables unintendedly if these can be

identified through the regular expression for symbol table file extensions in global scopes. This can cause ambiguities if multiple matching symbols result from symbol resolution.

6.5 Discussion

Language composition is a complex endeavor for which MontiCore and the STI offer powerful tools and capabilities. This chapter has described different forms of language composition and alternative approaches for realizing certain aspects of language composition. However, there is no blueprint or single "best" solution for realizing language composition with the STI. Language engineers who compose languages should be aware of all concepts and techniques and decide on the approaches that are most suitable for a concrete application.

A language L may define name usages that refer to name definitions of models that conform to a language, for which the symbol kind of the name definition is not known or should be left underspecified intentionally. Without symbol tables that are integrated through any form of language composition, the connection between name usage and name definition could not be checked in such scenarios. With symbol adapters, language engineers can compose languages that are not intended to be used together. The STI does not allow to indicate a name usage that leaves the symbol kind unspecified. Hence, such an adapter could not be realized within L, and the correctness check of the name usage could not be located within the language L. However, language engineers of L can introduce a symbolic extension point by defining a symbol kind k to which L expects to refer. Other languages can then provide symbols of any kind j that can be used as name definition if the person who composes the languages provides a symbol adapter that translates j symbols into k symbols. Through this, any correctness checks can be implemented against symbols of kind k, although L never instantiates any symbols of this kind. Only if another language provides a suitable adapter, symbols of the kind k are resolved and satisfy the correctness checks. The feature diagram language presented in Chapter 8 uses this mechanism in a slightly modified form.

6.6 Related Work

Language composition exists in different shapes and guises [EGR12]. The four kinds of language composition that MontiCore supports are supported in similar form by related language workbenches and language engineering tools [EvdSV+13].

ableC [KKCVW17] is an extensible C language framework based on the extensible attribute grammar system Silver [WBGK08]. The core C language is extensible with independent language modules. ableC provides several mechanisms for composing modules. A focus of the automated process of language composition in ableC is that it produces correct composed attribute grammars and accompanying language infrastruc-

ture reliably. As a C language framework, ableC aims at providing a customizable programming language, while the focus of MontiCore is to provide tailored modeling languages without setting a fixed base language.

mbeddr [VRSK12] is a family of modular languages that extend the C language for use in various purposes. It is realized with the language workbench MPS [VP12], which relies on projectional editors for building languages. mbeddr supports three kinds of language composition for extending the base C language. At the center of language extension are inheritance relations in abstract syntax elements. With mbeddr, languages can either be composed via language references similar to language aggregation in MontiCore, via language extension, or via language embedding.

SugarJ [EKR+11] is an extensible Java language. Language extensions are centered around novel syntactical constructs that can be translated into the base syntax in a "desugaring" process. SugarJ organizes language extensions in syntactic sugar libraries.

The SMI [MSN17] does not rely on a strictly typed infrastructure for symbols and, hence, requires less reconfiguration for language composition compared to the STI. However, the general forms of language composition (*i.e.,* language inheritance, embedding, extension, and aggregation) are the same. A major difference for language composition between the SMI and the STI is that the `resolve` methods in the SMI use the symbol kind as a method argument, while the STI has the symbol kind encoded into the method names. This ensures type compatibility in the STI but restricts composing languages as flexible as in the SMI. Language aggregation in the SMI is realized via *language families*, which are realized as Java classes that aggregate the language infrastructures of all aggregated languages. The global scope in the SMI is a Java class in the runtime environment of MontiCore that can be initialized with a set of languages that should be aggregated. This is not feasible for the STI, as the strictly typed infrastructure would prohibit initializing a global scope with arbitrary languages. Furthermore, as the `resolve` methods of the STI have the symbol kind encoded into their names, composing arbitrary languages on the level of global scopes is not possible.

However, the typed symbol table infrastructure has the advantage that the type system of the host language of the language workbench (for MontiCore, *i.e.,* Java) can be employed for checking the type correctness of language composition rather than implementing custom type checks, *e.g.,* for the compatibility of symbol kinds, that are carried out during the execution of MontiCore. Furthermore, the STI relies on generating large parts of the infrastructure and, hence, consequently considers adjustments via the TOP mechanism also for language composition.

Chapter 7

Language Components

Component-based software engineering is a sub-discipline of software engineering that develops means to modularize software into components to foster their off-the-shelf reuse. This has proven helpful in software engineering [NR68]. As "software languages are software too" [FGLP10], the concepts and techniques of component-based software engineering can also be applied to software languages. In this sense, a software language is a collection of artifacts that describe the language and that realize tooling for processing models of the language.

As depicted in Figure 7.1, the technical realization of a language component constitutes a number of different artifacts. In the reverse direction, an artifact may be part of one or more language components. Software artifacts are realized in the context of a file system. Therefore, an artifact is always located in a file system directory. A software language typically comprises heterogeneous artifacts that are scattered across different directories. Modern build tools such as Maven [MVM10] or Gradle [Mus14] propose separating handwritten files from generated files, productive files from files for testing purposes, and source code files from resource files. This separation enables managing the files efficiently from each perspective. However, it harms the comprehensibility of the language infrastructure as the entirety of files cannot be easily overviewed at a glance.

The software language engineering literature has proposed a wealth of different definitions for the term *language component* [CBCR15] in various levels of abstraction and for different purposes. Some of these approaches and their notions of language components are described in Section 7.6. For the purpose of analyzing software languages and realizing language composition in MontiCore, we consider language components as sets of artifacts.

Definition 6 (Language Component)**.** *A language component is a reusable unit encapsulating a potentially incomplete language definition. A language definition comprises the realization of syntax and semantics of a software language.*

A language component, therefore, contains either a complete language definition or parts of a language definition. For simplicity, we sometimes refer to language components containing complete definitions as *languages* in the remainder of this thesis if the distinction to language components is clear from the context.

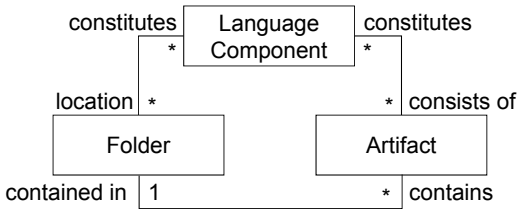


Figure 7.1: Relation between language components and artifacts

A language component can be left incomplete for different purposes:

- A language component can be an *interface provider*. Such language components create the basis for which other language components and languages can provide different realizations. For example, the language component `MCBasicTypes` of MontiCore provides an interface `MCType`, for which other language components provide different realizations [BEH+20].

- A language component can *realize a feature* of a language. Such language components realize a part of a language that is not intended to be a complete language, but a language feature that can be used within different languages or that is an optional part of a language. In the latter, the language component realizing a feature can be optionally added to the main language component. This realizes two variants of the language and is the basis for realizing language product lines in Chapter 9. For instance, the language component `CommonExpressions` realizes syntax and evaluation for commonly used expressions [BEH+20]. This is not a complete language but can be used by any language that requires such expressions.

- A language component can *group* other language components. Such language components are handy to reduce the effort for other languages to use a common combination of language components. For example, the MontiCore core language component `FullJavaStatements` does not introduce any new nonterminals but inherits from six other language components and thus, simplifies their reusability. Other languages can indicate to rely on this language component instead of indicating to rely on all inherited language components individually.

- A language component can *realize a language with holes*. Such language components realize entire languages (in contrast to features) but leave one or more concrete realizations for language concepts open to other language components. Languages with holes are explicitly parametrized and need the parameters to be defined to become a complete language. For example, the `BaseADL` introduced in Section 2.1 defines the interface nonterminal `IBehavior` without providing

a nonterminal implementing it. Instead, inheriting languages have to provide a nonterminal implementing the interface to complete the language component.

Definition 7 (Language Constituent)**.** *A language constituent is a kind of artifact in the infrastructure that realizes a software language.*

Examples for language constituents are grammar, parser, context condition, or a scope class. Language constituents can either be handwritten or generated. Considering the constituents of a language supports describing systematic handling of language components, inter alia, to describe their composition. In the following, we use the term *language component constituent* for a constituent of the (potentially incomplete) language defined within a language component.

This chapter describes several approaches for supporting the modular development of MontiCore languages. The chapter first details the notion of language components in MontiCore, then introduces a diagrammatic notation for describing language components with the purpose of documentation, before introducing a textual DSML for MontiCore language component models. Such models are the basis for operations and analyses against language components, for instance, for identifying the artifacts of a language or for unveiling forbidden artifact relations.

7.1 Language Component Models

A language component model is a model that describes which artifacts are part of a language and which artifacts a language is allowed to use. This serves several purposes. Using a language component requires *identifying all artifacts* of the language component. A language engineer using the language component should be able to identify all accessible constituents of the language by their kind. For example, identifying the context conditions of a language component enables executing a tool checking these.

Once all artifacts of a language are identified through a language component model, they can be packaged into an archive for *distribution*. In the case of Java as host language for the language infrastructure, jar files can be packaged to bundle a language infrastructure and provide access to the language for language users.

A straightforward solution to identifying all artifacts that constitute a language component is to use an explicit build module for each language component, where each module is located at a separate location in the file system. This form of modularization is often fostered by build tools such as Maven or Gradle. The modularization through such build tools alone, however, may not be sufficient as modularization for language components. Language components are typically fine-grained and may constitute a small number of artifacts only, while build tool modules usually capture larger software components. A one-to-one relationship between a build module and a language component would cause management overhead.

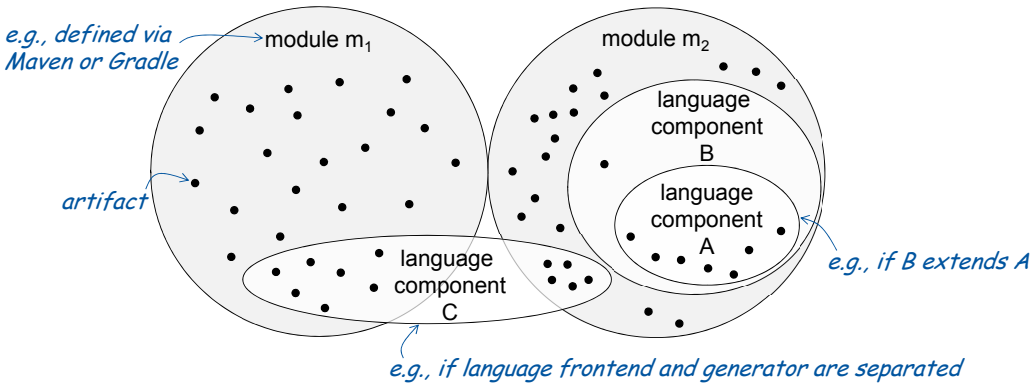


Figure 7.2: Relationship between artifacts in build modules and in language components

Furthermore, language components can have an artifact overlap if the intersection of their sets of artifacts is not empty. For instance, there might be two language components A and B that differ only in the fact that B contains a single additional artifact, as depicted in Figure 7.2. This could be an artifact realizing a context condition that B does not use. If each of these two language components was bundled into a separate build module, the module of B would either have to (1) clone all constituents from the module of A or (2) define a dependency relation to the module of A. The first solution leads to inconsistencies through evolution or co-evolution of the cloned artifacts. The second solution causes unnecessary management overhead because the module of B would constitute only a single artifact. The management overhead further increases if A constitutes a single artifact that B does not contain. In this scenario, the commonalities of A and B would have to be extracted to a common build module, and the modules of A and B each constitute a single artifact only.

If no one-to-one relationship between a build module and a language component is prescribed, the artifacts that belong to a language component may also be distributed over different build modules as depicted in Figure 7.2 by the language component C. This occurs, for instance, if language engineers build or use a library of code generator templates contained in a dedicated build module for different language components.

Because of these reasons, we propose to identify language component artifacts with a mechanism that supports fine-grained selection of artifacts cross-cutting to their location within the file system or within build modules. The MLC language presented in Section 7.4.1 supports selecting artifacts via regular expressions over their path within the model path of a language (*cf.* Section 2.1).

Language components rarely operate in isolation. Instead, they refer to the MontiCore runtime environment and may refer to further language components or any other software (*cf.* Figure 7.3). The task of managing such relations is usually performed by the dependency management of a build tool. Multiple language components may be

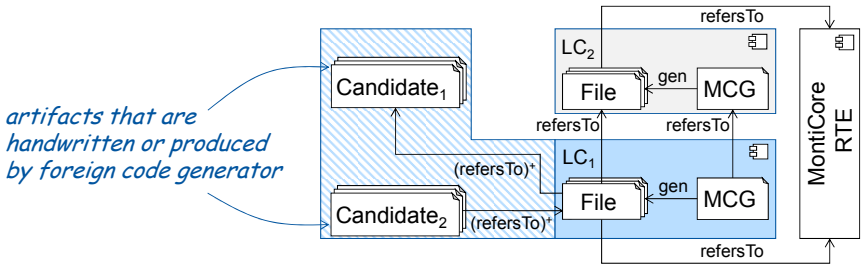


Figure 7.3: Artifacts of a language component LC_1 and their environment

located within a build tool module, so more fine-grained dependency management is required. Precise dependency management for language components fosters identifying relationships of a language component to external artifacts that may potentially harm their reusability. For example, a language tool for a language `A` uses a pretty printer of a language `B` that extends `A`. In this example, the language tool for `A` can only be used if the language `B` is available in this context, although other than for reusing the pretty printer, there is no reason for `B` to be required.

As we define the language components in the context of MontiCore, many artifacts that constitute a language component can be automatically identified. This includes all artifacts that are generated from a language's grammar, as depicted in Figure 7.3. However, often there are artifacts that are not part of the generated artifacts of a language to which the generated artifacts directly refer. This involves even more artifacts if transitively referred artifacts and, *e.g.,* subclasses of referred artifacts are included. For such artifacts, it cannot automatically be inferred whether these should be considered as part of the language component or not. We denote such artifacts as $Candidate_1$ artifacts. Examples of these artifacts are the handwritten classes in the context of the TOP mechanism [HKR21] or custom types used for symbol rule attributes. Furthermore, there are artifacts that directly or transitively refer to the generated artifacts of a language component. For such artifacts, which we refer to as $Candidate_2$ artifacts, it is unclear whether these are considered part of the language or not. Examples for such artifacts are handwritten context condition classes that implement generated context condition interfaces or classes realizing a code generator that rely on generated abstract syntax types of a language. Due to $Candidate_1$ and $Candidate_2$ artifacts, it is not helpful to generate language component models from MontiCore grammars. Instead, these have to be modeled manually by the language component designer.

Definition 8 (Language Component Model)**.** *A language component model describes the sets of own and allowed artifacts. Own artifacts are the artifacts that constitute a language component, and allowed artifacts are the artifacts that a language component is permitted to use.*

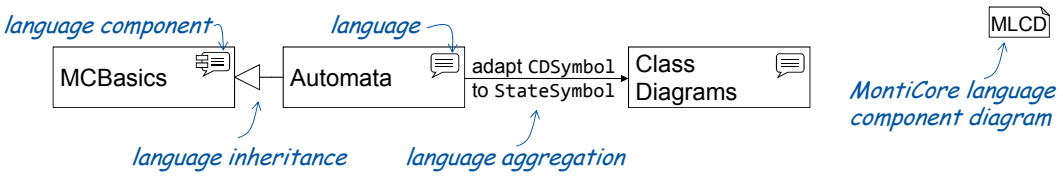


Figure 7.4: Syntax elements of MontiCore language component diagrams

While the own artifacts of a language component model serve the purpose of identifying artifacts that are part of the language, the allowed artifacts realize fine-grained dependency management on the level of individual artifacts.

7.2 MontiCore Language Component Diagrams

For visualizing the interrelations between different MontiCore language components, we use a graphical notation for *MontiCore language component diagrams* (MLCDs). An example presenting all syntactical elements of MLCDs is depicted in Figure 7.4. Languages and language components are represented by rectangles containing the name of the language (component) and a dedicated marker in the upper right corner. To distinguish a complete language from a language component, both use a different marker. The marker for a language is a speech bubble with three lines as a symbolic representation of text and language. The marker for language components includes two additional small rectangles in analogy to ports in markers of UML component diagrams. The name of a language or language component must be unique in an MLCD. Figure 7.4 depicts the language component `MCBasics` and two languages `Automata` and `ClassDiagrams`.

MLCDs distinguish two kinds of relations between language components, namely language inheritance and language aggregation. Language inheritance models the MontiCore language inheritance that is also employed for realizing language extension and language embedding. These three forms of language composition are not distinguished in MLCDs as there exist various forms of achieving each of these in the implementation. Furthermore, the distinction between these is not required for analyzing the interrelations between language components, which is a major motivation for creating MLCD models. Language inheritance between a language and a language that it extends is visually represented by an inheritance arrow lent from the notation for the inheritance relation in UML class diagrams.

As MontiCore language composition supports multiple inheritance, a language in an MLCD may inherit from multiple languages. Both complete languages and language components may extend other language components or complete languages. In the example depicted in Figure 7.4, the language `Automata` inherits from the language component `MCBasics`. As usual for language inheritance in MontiCore, this is modeled

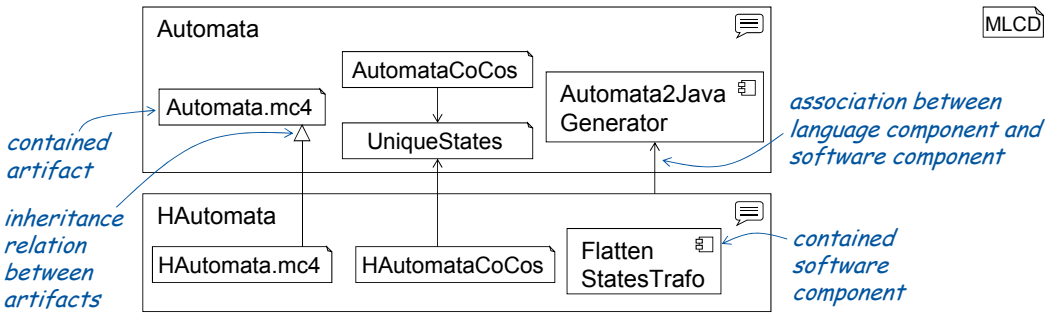


Figure 7.5: Internal elements of MontiCore language component diagrams

in the grammar of the automata language that extends the grammar of the MCBasics language component.

Language aggregation in MLCDs is represented by lines as lent from the notation of associations in UML class diagrams. The line usually has an arrow from the language (component) providing the global scope instance to the language (component) that is reused, *e.g.,* through employed resolvers. The direction of the arrow may be left underspecified if the language providing the global scope has not (yet) been decided. Optionally, the line can be labeled with an indicator that describes how the symbols of the languages are adapted if any adapters are employed. If any adapters are used, these typically adapt between a symbol kind defined in the language that does not provide the global scope to a symbol kind defined by the language that provides the global scope of the aggregation. If this is the case, it can be realized in MontiCore by configuring the global scope to use a resolver adapting a foreign symbol kind to a known symbol kind. Both languages and language components can be aggregated with other language components and other languages.

The MLCDs, as presented so far, enable modeling an overview of the relationships between language components. Sometimes it is helpful to visualize the internals of a language component as well. For this purpose, MLCDs lend the notation of UML component diagrams for describing the internals of a language component. The box visualizing a language component can have nested software components or nested individual artifacts. These artifacts or components are usually excerpts of the entire set of contained artifacts, as a visualization of all artifacts of a language is highly complex. The artifacts are visualized as boxes with a folded edge symbolizing sheets of papers, and software components are denoted as boxes with a software component icon known from UML component diagrams. Arrows visualizing inheritance and other forms of associations between contained individual artifacts, software components, or the language components themselves can be used to depict the relationships between parts of one or more language components.

An example of an MLCD with nested elements is depicted in Figure 7.5. The example comprises two language components `Automata` and `HAutomata`. Each language component contains a grammar, and the grammar `HAutomata.mc4` extends the grammar `Automata.mc4`. The artifact `AutomataCoCos` is associated with the artifact `UniqueStates`, which are both located in the `Automata` language component. However, the artifact `HAutomataCoCos` of the language component `HAutomata` is also associated with `UniqueStates`. The software component `Automata2Java` is associated from the language component `HAutomata`, but it is underspecified from which part of it.

While with language component diagrams, the contents of language components can be overviewed, a complete visualization of all artifacts that are part of the language components is rarely feasible. The MLC language presented in Section 7.4.1 enables precise specification of all artifacts via textual models. The following describes a concept for identifying artifacts that are part of a language component in a semi-automated process. This concept is the basis for the MontiCore language component language.

7.3 Concept for Identifying Artifacts of Language Components

The own and allowed artifacts contained in language component models describe the entirety of artifacts that a language contains and that it is allowed to use. However, specifying each of such artifacts individually would be cumbersome, and the concrete syntax of such models would be inherently complex. Instead, language component models can address such artifacts via regular expressions that we refer to as *artifact selectors*. If an artifact analysis is carried out on a file system, the artifacts that a language component actually uses can be determined. With this information, forbidden artifact usages can be identified, and self-contained language component archives can be bundled. The following sections describe the concepts for this.

7.3.1 Address Artifacts of a Language Component

Artifact selectors are statements that rely on a glob [Fri06] pattern that can be evaluated against a file system. With a single glob expression, an artifact selector can describe the selection of a set of artifacts. The combination of several artifact selector statements, therefore, enables describing large sets of artifacts with little effort.

Glob expressions identify file paths in which parts of a path can be specified with different wildcard characters. A star `*` is a wildcard that identifies a sequence of characters, and a question mark `?` identifies a single character. For example, the glob expression `foo/*.aut` identifies all files in the directory `foo` with the file extension `aut`. Similarly, `foo/*/PingPong.*` identifies all files in direct subfolders of the folder `foo` with the name `PingPong` and an arbitrary file extension. To address transitive subfolders of a folder, glob expressions can use a double star `**`, like in `foo/**/PingPong.aut`.

The correct identification of artifacts requires their fully qualified path in the file system, beginning with the root of the file system. In the context of language component models, however, fully qualified paths prevent the language component models from being applicable in different file systems. This is a severe restriction because language component models should be transferable. To overcome this restriction, artifact selectors can be relativized with respect to a *base indicator*. A base indicator, thus, is a variable that can be used as the prefix of a glob expression to absolutize a relative artifact selector in a way that the relativized part can be evaluated against different file systems. Currently, we distinguish three kinds of base indicators:

callDir is a base indicator that evaluates to the directory, from which the tool that processes the language component model is invoked. This behavior matches the typical behavior of relative file system indications in command-line interfaces. However, this base indicator has to be handled with care as invocations of the tool from a different directory may yield different evaluations, which ultimately may identify different language components.

projectDir is a base indicator that evaluates to a common root directory that the language component is located in. Typically, this is the root directory of the build tool module or project. For the proper evaluation of this base indicator, the directory must be passed as an argument to the evaluating tool.

mp is a base indicator that evaluates to a set of directories that are entries of the model path of the language. Currently, this is the only base identifier that can evaluate to multiple directories. If this is the case, each artifact identifier is evaluated against multiple base locations. The `mp` base indicator enables specifying artifacts that a language is allowed to use that are located in a different language component or software component whose absolute location relative to the location of the current language component should not be explicated. Furthermore, in the context of build tools, this enables identifying artifacts across the borders of build tool modules.

The MLC language described in Section 7.4.1 uses artifact selectors and base indicators to enable evaluating language component models on different file systems.

7.3.2 Artifact Analysis

The own and allowed artifacts of a language have to be modeled in a manual process to avoid omitting or including unintended artifacts. The artifacts that a language actually uses, however, can be extracted in an automated process. An artifact analysis can unveil `refersTo` associations between artifacts [BGRW17, GHR17]. The kinds of `refersTo` relations between artifacts are modeled in an artifact model and depend on the kinds of artifacts that are involved. For instance, MontiCore languages typically involve Java

artifacts, grammars, and FreeMarker templates. A MontiCore grammar can refer to other MontiCore grammars only via grammar inheritance. Java artifacts can refer to other Java artifacts in multiple reifications of `refersTo` associations including, among others, via inheritance between the Java types or via usage of a Java type as the type of an attribute, as the return type of a method, or as a super type of a generic type argument. Given an input artifact, the artifact analysis calculates the set of all artifacts to which the input artifact directly refers. Through iterative application, the transitive closure of referred artifacts can be calculated as well.

Given the set of own artifacts as input, an artifact analysis can be performed against a file system to calculate several sets of artifacts that are of importance for handling language components.

Used artifacts are artifacts that a language component uses. More precisely, this set of artifacts includes all artifacts to which at least one own artifact of a language component refers. Own artifacts of a language are not included in the used artifacts. Therefore, if an own artifact refers to another own artifact of the language, the referred artifact is not contained in the set of used artifacts.

Forbidden artifacts are artifacts that a language uses illegally. Forbidden artifacts are all artifacts that a language uses that are not allowed artifacts of the language. Hence, the set is calculated with a set difference operation. If the set of forbidden artifacts of a language is not empty, the language violates the specification given in the language component model.

Unused artifacts are artifacts that are allowed through a language component model but are not actually used by the language. This set of artifacts is an indicator for potential refinements of artifact selectors specifying the allowed artifacts. Refining the artifact selectors fosters immutability of the language component model against evolution in the file system.

To simplify the calculation of forbidden artifacts of a language component, we assume that each allowed artifact of a language component is an own artifact of another language or software component that is modeled explicitly. Under this assumption, it suffices to check the allowed artifacts against artifacts that are directly used by a language rather than checking these against transitively used artifacts.

7.3.3 Building Self-Contained Language Component Archives

Self-contained language component archives can be built after evaluating a language component model to distribute language components. To be self-contained, the language component archive must contain not only the own files of a language component but also the files that the language component uses transitively.

There are different use cases for distributing a language component via an archive. A language component can be distributed to access the language component via a language tool that is realized as a command-line interface tool. For this use case, it is not relevant that the archive includes the source files of a language component. Instead, it suffices to include compiled class files and, if the language includes a code generator, the templates of the code generator. Tool archives should usually be self-contained and thus should include transitively used artifacts.

Another use case is the distribution of language components for reusing these in the context of engineering another language component. For this purpose, language component archives must include grammar files in addition to the compiled Java class files and templates required for tool archives. Usually, it is not required to provide Java source files as the language composition of MontiCore languages does not require their re-compilation. An exception to this is a language that applies the TOP mechanism to a class generated for an inherited language. However, we strongly recommend avoiding this due to the required re-compilation. Alternative solutions for employing the TOP mechanism to a Java type of an imported language include reconfiguration of the language mill with a handwritten subtype or overriding a nonterminal and applying customization of the generated code for classes synthesized from this nonterminal.

A third use case for language component archives is the distribution of the language components for evolving or customizing the language component. This use case requires including all source files of a language component in an archive. Generated artifacts are not required to be included, as these can be reproduced from the source files.

All language component archives that do not have the purpose of using the language component through a (command-line interface) tool should not include the artifacts that a language component uses. The reason is that including used files increases the likelihood of clashes between files with the same name if different language components are reused at once. However, a single artifact may also be part of multiple language components, and hence, such clashes cannot be avoided completely. To this effect, in case that a build tool is used in combination with language component models, we recommend using the archives produced by build tools for engineering languages and self-contained archives only for distributing language tools. Build tools usually prohibit that a file is part of multiple (atomic) modules at once. A further advantage of using build tools for archiving language components is that their sophisticated dependency management can be reused to identify archives by unique coordinates rather than by their fully qualified location. Such coordinates usually contain the name of the module, its version, and an optional classifier.

```
01 mlc Automata {                                                    MLC
02   files "$projectDir/src/main/grammars" {
03     include "Automata.mc4" ;
04   }
05
06   files "$projectDir/src/main/java" {
07     include "automata/**/*.java" ;
08   }
09
10   files "$projectDir/target/generated-sources/monticore/sourcecode" {
11     include "**/*.java" ;
12   }
13
14   uses {
15     language "MCCommonLiterals" ;
16     include "$mp/java/**.class" ;
17     include "$mp/java/lang/reflect/**/*.class" ;
18   }
19 }
```

Figure 7.6: Example model of the MLC language

7.4 Realization of Language Components

This section presents the realization of language component models in MontiCore through the MontiCore language component (MLC) language and the tool that processes MLC models and can perform artifact analysis.

7.4.1 The MLC Language

The syntax of the MLC language is explained based on an exemplary model for the language component `Automata` depicted in Figure 7.6. The language component definition in an MLC model begins with the keyword `mlc` followed by the name of the language component (l. 1). The body of the language component is enclosed by curly brackets and may contain *file blocks*, *include statements*, and *use blocks* in arbitrary order. The first element of the MLC body in the example is the file block in ll. 2-4. While file blocks and include statements in the body of MLC models define own artifacts of a language component, use blocks indicate artifacts and languages that a language component is allowed to use.

File blocks begin with the keyword `file`, followed by an artifact selector that begins with a base indicator (*cf.* Section 7.3). The consecutive block enclosed by curly brackets contains a non-empty sequence of include statements followed by optional exclude statements. Include statements and exclude statements have an artifact selector String that uses a glob expression to address artifact sets. Include statements indicate artifacts that should be included in a file block, while exclude statements exclude artifacts that

```
01 grammar MLC extends MCBasicTypes, MCCommonLiterals {                    MCG
02    MLCCompilationUnit = ("package" package:MCQualifiedName ";")? MLCDef;
03    symbol MLCDef = "mlc" Name "{" MLCElement* "}";
04    interface MLCElement;
05    interface UseElement;
06    FileBlock implements MLCElement, UseElement = "files" StringLiteral
07                "{" IncludeStatement+ ExcludeStatement* "}";
08    UseBlock implements MLCElement = "uses" "{" UseElement+ "}";
09    IncludeStatement implements MLCElement, UseElement = "include" StringLiteral ";";
10    ExcludeStatement implements UseElement = "exclude" StringLiteral ";";
11    LanguageStatement implements UseElement = "language" StringLiteral ";";
12    symbolrule MLCDef = includedFiles:Multimap<String, String>;
13 }
```

Figure 7.7: MontiCore grammar of the MLC language

have been included before. The include and exclude statements of a single file block are evaluated in the order of their appearance within a model. The order of different file blocks in an MLC model does not influence the selection of artifacts.

The file block in ll. 2-4 includes artifacts from the folder `src/main/grammars` relative to the project root directory. In the body of the block, only the artifact `Automata.mc4` is included. The second file block in ll. 6-8 uses a glob expression to include all files with the file extension `java` that are located in transitive subdirectories of the directory `automata` relative to the path specified in the artifact selector in the head of the file block.

Use blocks begin with the keyword `uses`, followed by a block of include and exclude statements enclosed by curly brackets. Additionally, a use block may refer to other MLC models whose own artifacts are allowed to be used. The use block in ll. 14-18 defines that the language component may use all artifacts of the language component `MCCommonLiterals`. Furthermore, all Java class artifacts that are located in any subfolders of the folder `java` relative to any model path entry may be used, except for those that are located in any subfolders of `java/lang/reflect`.

In summary, the language component includes the grammar of the `Automata` language as well as all handwritten and generated Java artifacts for the language. Furthermore, the language may use any artifacts of the inherited language `MCCommonLiterals` and any Java API artifact except for the artifacts of the Java reflection API.

Grammar

The following describes the syntax in more detail through the grammar of the language, which is the `MLC` grammar depicted in Figure 7.7. The grammar inherits from the grammar `MCBasicTypes` to reuse the nonterminal `MCQualifiedName` and from the grammar `MCCommonLiterals` to reuse the nonterminal `StringLiteral` (l. 1). To create a hierarchical namespace of MLC models, each model may begin with a package statement (l. 2) in the same syntax that Java uses for package statements. Import statements are not allowed in an MLC model. The definition of an MLC is realized through the nonterminal `MLCDef` (l. 3) that defines an `MLCDefSymbol`. The elements in the body of an MLC definition are realized through the interface nonterminal `MLCElement` (l. 4). Currently, the nonterminals `FileBlock`, `UseBlock`, and `IncludeStatement` implement this interface nonterminal. The MLC language can be extended with additional elements if further nonterminals implement this interface.

The nonterminal `FileBlock` (ll. 6-7) defines the syntax for file blocks that begin with the keyword `file` followed by a `StringLiteral` that realizes the artifact selector. The body of a file block contains a non-empty list of include statements followed by a list of exclude statements that may be empty. Hence, within a single file block, it is not allowed that an include statement follows an exclude statement.

A use block may contain a list of `UseElements`, which are realized as another interface nonterminal (l. 5). Currently, the nonterminals `IncludeStatements` (l. 9), `ExcludeStatements` (l. 10), and `LanguageStatements` (l. 11) implement this interface. Each of these statements begins with a specific keyword followed by a `StringLiteral` that realizes an artifact selector and by a semicolon. Furthermore, the nonterminal `FileBlock` implements the interface. Because of this, a use block may have nested file blocks. Modelers can use such nested file blocks as an alternative syntax for individual include and exclude statements relative to a common base directory.

With the symbol rule for the nonterminal `MLCDef` in l. 12, the grammar adds an additional attribute to the `MLCDefSymbols` Java class, for which MontiCore also generates access and modification methods. This attribute has the name `includedFiles` and is a String multimap, *i.e.,* a data type that maps Strings to a list of Strings. The entries in the map are the paths of the own files of a language. The map is explained in more detail in Section 7.4.1.

Context Conditions

The implementation of the MLC language currently relies on two context conditions that ensure well-formedness of the syntax of models beyond the restrictions of the grammar.

The restriction that include statements must not follow exclude statements is achieved through the grammar for file blocks. For use blocks, however, the combination of include and exclude statements with language statements would require more complex handling

of this restriction through the grammar. Furthermore, the use block elements are realized via interface nonterminals because checking the restriction through the grammar would impede the extensibility of the language with further nonterminals that implement the interface. Therefore, this restriction is realized through the context condition `UseBlockOrder`. This context condition uses a visitor for traversing the AST of use blocks and, during the traversal, ignores any contained file blocks. It sets a Boolean flag once it encounters an exclude statement in an AST and yields an error on any consecutive occurrence of an include statement.

The second context condition of the MLC language ensures proper usage of base indicators. The artifact selector in the head of file blocks must always start with a base indicator. If no explicit base indicator is given, the default base indicator is the `$callDir`. All include and exclude statements within file blocks are relative to the path in the file block's head and, therefore, must not begin with an explicit base indicator. If this condition is violated, the context condition yields an error. For include statements outside of file blocks and include and exclude statements contained in use blocks, no well-formedness checks are required. Instead, the artifact selectors in these statements can have an optional explicit base indicator that evaluates to the default if it is omitted.

Symbol Table

The symbol table of the MLC language captures the essence of a language component that other language components have to be aware of for their evaluation. To this end, the symbol table of the MLC language performs the evaluation of the artifact selector statements to obtain sets of absolute file locations for each artifact selector. Only the absolute locations identify a consistent set of artifacts that other language components can rely on for their evaluation. The exported symbol table of an MLC model includes only the set of own artifacts of the language component, as the set of allowed artifacts is not relevant for other language components.

As described in Section 7.4.1, the `MLCDefSymbol` derived from the grammar of the MLC language contains an attribute `includedFiles` that manages the set of own artifacts of the language component. Each entry in this set is the file path of an own artifact of the language and is realized by an entry of the map of included files. The paths are, however, separated into two parts, where one part is the key of the map and the second part is a value for the key in the map. The keys of the map are model path entries in which the artifact is located. The values are the file locations relative to the model path entry. This separation is required for bundling the files of a language component into archives. While the absolute qualifier of own files, *i.e.,* their model path entry, is omitted in archives, the folder structure of relative qualifiers is rebuilt in archives.

The `includedFiles` attribute is realized as multimap and, thus, MontiCore cannot generate a built-in serialization strategy for this attribute. The custom serialization translates the multimap into a JSON array with a value for each entry of the map. The

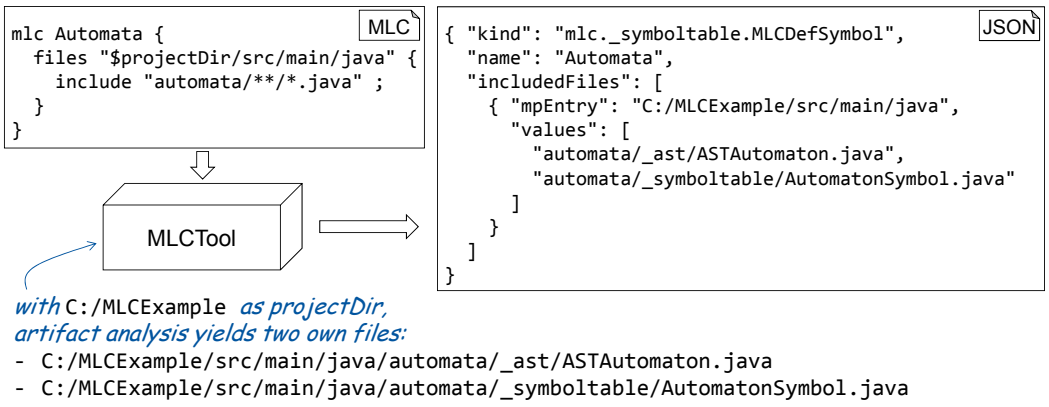


Figure 7.8: Example for an MLC model and the exported `MLCDefSymbol`

individual entries are realized as JSON objects where the key of the map is realized as object member with the key `mpEntry` and are of the JSON type String. The values of the `includedFiles` are realized as a JSON object member with the key `values` of the type `JsonArray`. The values of the array are the values of the `includedFiles` map for the given key and are realized as JSON Strings.

An example of a minimalistic MLC model for the language component `Automata` is depicted in the left side of Figure 7.8. The model contains a single file block with the model path entry `src/main/java` relative to the `projectDir` base indicator. The model is processed with the MLC tool (*cf.* Section 7.4.2) that performs an artifact analysis and exports the symbol table for the `Automata` model. As an argument of the tool, the project directory is set to the absolute location `C:/MLCExample`. The results of the artifact analysis are the absolute locations of two Java source files `ASTAutomaton` and `AutomatonSymbol`. The resulting exported symbol is serialized as the JSON object depicted on the right side of Figure 7.8. The `includedFiles` map contains a single map entry only. The key of this map entry is a model path entry with an evaluated base indicator. For both files that have been found during the artifact analysis, a value is contained in the entry. These values have locations relative to the `mpEntry` key. The artifact paths in the stored symbol tables are specific to the file system on which the artifacts are located. Hence, stored symbol tables of MLC models must not be exchanged or distributed, *e.g.,* as part of symbol libraries of language components. This is a deliberate decision because it prevents re-evaluation of the regular expressions in the models if the symbol table is loaded.

The symbol tables of the MLC language are loaded while evaluating the language statements (*cf.* Figure 7.7). If a symbol file is found for the resolved name, it is loaded, and re-evaluation of the regular expressions can be omitted. Otherwise, the language attempts to find a suitable MLC model, parses it, and instantiates its symbol table.

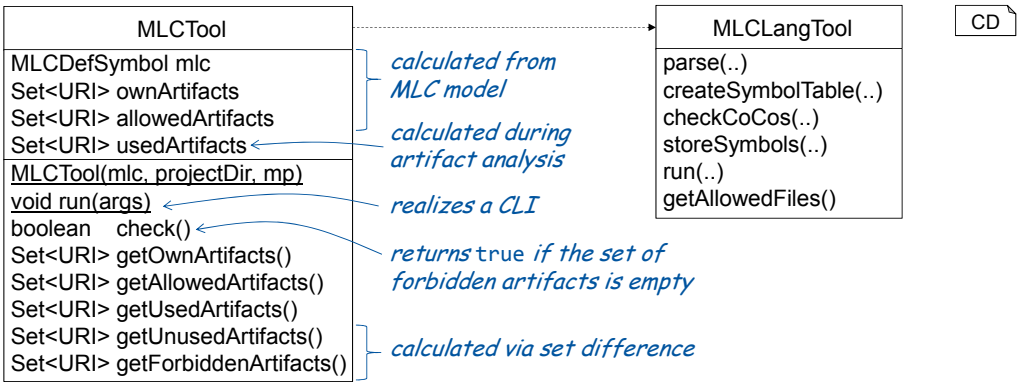


Figure 7.9: Tool for using the MLC language

7.4.2 Tool for Processing MLC Models

The class `MLCTool` implements the functionality to process language components in the form of MLC models. The usual operations for processing MLC models with the MLC language, such as parsing a model, creating the symbol table for a given AST, checking the context conditions, and storing a symbol table, are realized in the class `MLCLangTool`. The signatures for the static methods realizing the operations are as follows:

parse(..) takes a path to an MLC model as an argument and returns the parsed model as an instance of the class `ASTMLCCompilationUnit`. This method yields an error if the model cannot be parsed.

createSymbolTable(..) uses an instance of the AST, such as the result of executing the `parse` method, a String to the projectDir location, and a `ModelPath` as arguments. The input is used to calculate the own files of the language component and instantiate the symbol table for the passed AST. The result is an instance of the `IMLCArtifactScope`.

checkCoCos(..) uses a passed AST and checks the context conditions of the MLC language. The method yields errors on a violation of context conditions.

storeSymbols(..) stores a passed artifact scope instance to a file at a file path passed as String. As a side effect, the symbol table of the model is persisted.

run(..) executes all methods described above. As input, it requires the path to an MLC model as a String argument, the `projectDir` path as a String argument, and the `ModelPath` as another argument. The result is an instance of an `MLCDefSymbol`. As a side effect, the symbol table of the model is persisted if the input model is parseable and well-formed.

getAllowedFiles(..) calculates the set of allowed files as an instance of `Set<URI>`, *i.e.,* a set of Uniform Resource Identifiers [www05]. These identify artifacts uniquely regardless of whether these are located in the local file system, in an archive, or at a remote web server. As input, the method requires an AST, the path to the `projectDir` location, and the `ModelPath`.

The `MLCTool` uses the `MLCLangTool` for processing an MLC model and, based on the symbol table of this model, executes an artifact analysis to calculate the derived artifact sets described in Section 7.3.2. An overview of the methods of the `MLCTool` is depicted in Figure 7.9. An input MLC model is processed with the `MLCLangTool` during instantiation of the `MLCTool` and, as a result, the `MLCDefSymbol` of the model as well as the sets of own and allowed artifacts are available. The set of used artifacts is calculated with an artifact analysis whenever it is required. Once calculated, the set is cached for further requests. With the set of used artifacts, the sets of forbidden artifacts and unused artifacts can be calculated by applying set differences. The method `check` returns a Boolean value that is *true* if the set of forbidden artifacts is empty and *false* otherwise. As a side effect in the latter case, the tool prints a list of forbidden artifacts. The `MLCTool` realizes a command-line interface in the static method `run`. The arguments are as follows:

-input reads the input MLC file at the passed location. This argument is mandatory.

-out sets the output directory for exported symbol tables. The default value is `target/`.

-projectDir sets the project directory for the evaluation of the MLC model.

-path sets the model path for the evaluation of the MLC model either as a single path or as multiple path entries separated by space.

-prettyprint prints the MLC file to the console or to a specified output file.

-store serializes and prints the symbol table to the specified output file.

-check checks whether the language uses forbidden artifacts. If so, the tool reports these and terminates with an error.

-own prints a list of all artifacts that are part of the language.

-allowed prints a list of all artifacts that the language is allowed to use based on the MLC model.

-used prints a list of all artifacts that are used from artifacts of the language but are not part of the language.

-forbidden prints a list of all artifacts that the language uses illegally.

```
> java -jar MLCTool.jar -input Automata.mlc -projectDir . -own -forbidden
own: file:///C:/aut-lng/src/Automata.mc4
own: file:///C:/aut-lng/src/automata/_ast/ASTState.java
own: file:///C:/aut-lng/src/automata/_symboltable/StateSymbol.java
 ⋮
forbidden: file:///C:/gradle-repository/Statecharts.jar!statecharts/_ast/ASTState.java
```

Figure 7.10: Example output of an execution of the `MLCTool`.

The tool has several arguments that print lists of artifacts. Within these lists, the URL of each artifact is printed in a new line. Each line starts with a short name identifying the artifact list and a double colon, like in `allowed:`, followed by the URL of the artifact. The prefix of the lines supports, among other things, reading in the tool output with another tool. If the tool is called with multiple such arguments, the corresponding lists are concatenated. An example of the output of a tool execution is depicted in Figure 7.10. The first line depicted the tool call. The tool is executed for an MLC model `Automata.mlc` with the directory of the tool call set as project directory. Furthermore, the tool call uses the arguments `own` and `forbidden` to print the respective artifacts. As a result of the tool call, the absolute locations of all own artifacts and all forbidden artifacts are printed as described above. The figure depicts three own artifacts (the grammar, an AST class, and a symbol class) and one artifact that is illegally used. Three artifacts are files in the file system and one artifact is a file contained in a jar file. A realistic tool execution would print larger numbers of artifacts, which are omitted in the figure for presentational reasons.

7.5 Discussion

A typical characteristic of software components is that they have an interface that describes the interaction of the component with its environment [BHP+98]. The presented approach does not proclaim explicit interfaces for language components. Instead, our intention is that language component interfaces should be realized as part of the documentation of the language rather than through explicit development artifacts. Some approaches for language componentization use explicit *provided* and *required* interfaces of a language component [BPRW20, CBCR15]. Explicit interfaces enable language engineers to oversee the important part of a language component, such as extension points, without required in-depth knowledge about the internals of a language component. Such internals are hidden from language engineers who want to reuse the language components and, thus, realize a black-box character of language components. In practice, explicit interfaces require additional effort for developing, evolving, and maintaining language components that can lead to inconsistencies between the explicit interface and the imple-

mentation. Furthermore, a provided interface of a language component prevents reusing parts of the language component that are not part of the provided interface and, thus, requires language engineers to foresee all potential extension points of a language during the design time of a language. Explicit interfaces cannot completely replace documentation about how to reuse a language component. Overall, the benefit of explicit interfaces is more significant the higher the complexity of an individual language component. Our approach, on the contrary, pursues language components with low complexity to foster their independent reusability.

The artifact analysis performed in the context of language components can be extended to identify proper access to the language component's constituents. With the name and location of the constituents alone, as it would be available in previous work [BEK+18b, BEK+19], a fixed assumption about the access to objects, such as via a zero-argument constructor, has to be made.

The symbol table of the MLC language exports the absolute location of all own artifacts of a language. This prevents distributing exported symbol tables between different physical locations, *e.g.,* between different machines. However, the set of absolute locations of the own artifacts of a language component is the information that other language components have to know of a particular language component. For example, an MLC model for the language A indicates that artifacts of A are allowed to refer to artifacts of a language B. If the own files of B were not captured by the exported symbol table of B, the MLC model of B would have to be processed anew completely. For constellations of language components with many interrelations, this would be a severe reduction of the performance.

With the artifact selectors presented in Section 7.3, it is possible to address artifacts either precisely through their individual names or imprecisely by using wildcard symbols. This affects the evolution of language components. For example, the include statement in l. 7 of Figure 7.6 includes all Java artifacts that are located relative to the folder `automata`. The include statement in l. 11 of the same figure includes all Java artifacts of the base location. If the language component is renamed, the statement in l. 7 has to be adjusted accordingly while the statement in l. 11 is still correct. However, if another language component `HAutomata` is conceived next to the location of the language component `Automata`, the statement in l. 11 would include the artifacts of `HAutomata`, which may be unintended.

During the evolution of a language component, the MLC models have to be evolved accordingly. If this is not the case, the conceptual boundary of a language component's artifacts and the inclusion and exclusion of artifacts expressed in the MLC model diverge. Future work should investigate tool support for the recommendation of adjustments to the MLC model in accordance with the evolution of the infrastructure of a language component.

In the MLC language, the modularization of build tools co-exists with the modularization induced by language components. More precisely, MLC models are intended to be

used upon the modules of build tools to foster more concise and finer-grained modularization. If MLC modules were intended to replace build tool modules, the dependency management would have to be more sophisticated, for instance, in terms of different versions of the artifacts used. Furthermore, MLCs currently lack support for executing the actual build through a build script.

The implementation of the MLC language realizes the identification of own artifacts through include and exclude statements within file blocks. Within each file block, it is prohibited that any include statements follow an exclude statement. The reason behind this is that an inclusion of files after the exclusion might lead to the unintended inclusion of artifacts that have previously been excluded. To circumvent include statements that follow exclude statements, these can be extracted to a separate file block with the same artifact selector.

7.6 Related Work

Componentization of languages is investigated in various related approaches. To foster reusability of languages, many language workbenches [EvdSV+13] such as GEMOC Studio [www21a], MPS [VP12], Neverlang [VCPC13], Spoofax [KV10], and XText [Bet16] support engineering language modules. The modularization of software languages requires means to perform language composition [EGR12] of such modules. Other related approaches for such mechanisms are described in Chapter 6.

The definition for language components presented in this chapter is similar to the definition of GEMOC [CBCR15]. The major difference between the two definitions is that the definition in this thesis does not prescribe that language components have required and provided interfaces.

Melange [DCB+15], as part of GEMOC Studio [DMW17], separates language interfaces from language implementations to foster their reusability. Language interfaces expose information about language constituents for a specific purpose. A primary motivation behind the language interfaces in Melange is that these abstract from the implementation and thus, enable modifying the implementation while the interface remains the same. Melange languages can be composed with different composition operators such as *merge* and *slice* through their interfaces. Language implementations in GEMOC Studio rely on Ecore [SBPM09] metamodels for the definition of the abstract syntax, Sirius or Xtext for realizing concrete syntax, as well as different meta-languages for defining the execution semantics of a language.

In the revisitor [LDC+17, LDC18, Led19] approach, modular executable languages are based on Ecore metamodels and have attached semantics that can be composed with the revisitor pattern. In the metamodels, required elements can be used to realize underspecification in the syntax. Bindings [LDC18] can relate elements of different metamodels to realize language composition with the revisitor pattern [LDC+17]. With

this approach, language engineers can reuse language modules without foreseeing explicit extension points. The focus of this approach is to support independent language extensions [LDWC19].

Concern-oriented language development (COLD) [CKM+18] is a paradigm for engineering modular languages with the purpose of their reuse. A language concern in COLD is similar to language components in our approach. Languages in COLD are the analog of complete language components, and a facet in COLD is similar to a constituent of a language component. Language concerns have three interfaces. The variation interface makes closed variability within a language concern available for reuse, and the customization interface realizes this for open variability within a language concern. The usage interface defines the access to relevant operations and information of the concern. In MontiCore, the usage interface is typically realized via the tool of a language, and the customization interface is induced by the nonterminals of a language's grammar. With MontiCore language components alone, no variability interface is available. In combination with a feature model as described in Chapter 9, however, closed variability can also be described.

In previous work in the context of product lines of languages [BEK+18b, BEK+19], we described language components through tuples of grammar, context conditions, and code generators. This approach is well suited for identifying the constituents of language components by their kind. Through the grammar, all artifacts generated by the grammar can be identified, and by strict usage of the TOP mechanism, handwritten adjustments to these can be identified as well. With context conditions and code generators, many commonly used handwritten artifacts that constitute a language are covered as well. However, further handwritten constituents of a language cannot be addressed.

LISA [LDA13, Mer13] employs attribute grammars that describe integrated concrete and abstract syntax and semantics of language modules. To compose these language modules, LISA relies on inheritance as known from object-oriented programming. The inheritance can be applied to the attribute grammars as well as to individual rules of the grammars.

Component-based semantics [Mos19] is an approach for realizing semantics modules based on context-free grammars for describing language syntax. Through using nonterminals that are not defined in a grammar, extension points can be realized. When the language modules are composed, another grammar has to provide definitions for such nonterminals. However, the focus of this approach is the compositional semantics based on "funcons" as individual fundamental programming constructs. To give meaning to a model, the syntax is translated into funcons that an interpreter executes.

Chapter 8

The MontiCore Feature Diagram Language Family

The MontiCore feature diagram language family realizes feature diagrams (*cf.* Section 2.3.3) and two different notations for feature configurations as textual DSMLs. The language family comprises three languages:

The feature diagram language (FDL) is used to describe the variability of a product line in the form of a feature model. Feature models may import other feature models to foster their modularization. The language contains extension points to enable the customization of feature diagrams for different application scenarios.

The feature configuration language (FCL) is used to model feature configurations that describe a product of the product line. Feature configurations refer to a feature model and contain a set of selected features. They do not make assumptions about features that are not selected. Therefore, all features of a feature diagram that are not explicitly selected in a feature configuration model are either excluded or no choice has been made for these. The FCL does not distinguish this.

The partial feature configuration language (PFCL) is used to model partial feature configurations. In addition to selecting features, partial feature configuration models can explicitly exclude features that should not be part of the described product. Compared to models of the FCL, partial feature configurations, thus, distinguish excluded features from those for which no decision has been made yet. The set of features for which no decision has been made contains exactly those features of the feature model, which are neither explicitly selected nor excluded. This fosters, among other things, the staged configuration [CHE05] of products from the product line.

The advantage of engineering a feature-oriented software product line with the feature diagram language family is that the realizations of features in software can be strongly coupled with the features of the feature diagram. The powerful language composition mechanisms of MontiCore can integrate feature diagrams with software artifacts that implement the features. Such integrations can be realized in different forms, as explained in Section 8.4. In contrast, feature diagram tools without language composition use

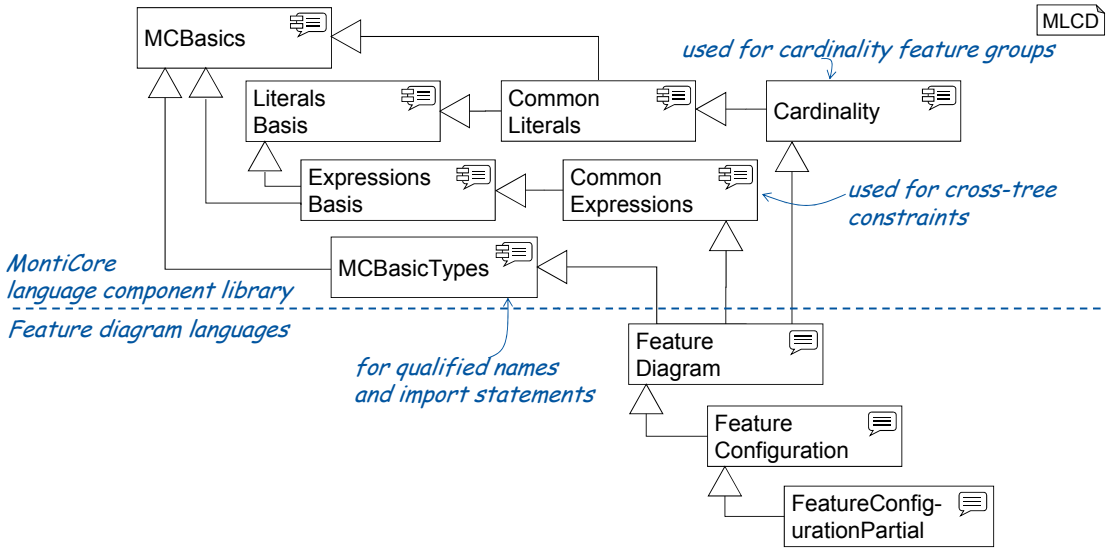


Figure 8.1: Overview of relations between language components and languages realizing the feature diagram language family

concepts such as bindings between features and software artifacts or annotations in source code artifacts that mark the feature to which these belong. The consistency between the feature model and the features in the source code often has to be checked manually or with external tools. In the feature diagram language family, such integrations can be realized in different forms, as explained in Section 8.4.

The languages of the feature diagram language family and the language components that these reuse from the MontiCore language component library [BEH+20] are overviewed in Figure 8.1. The `FeatureDiagram` language extends three language components. It extends the language component `MCBasicTypes` for reusing qualified names and import statements, the `CommonExpressions` language component for realizing cross-tree constraints, and `Cardinality` for realizing cardinality feature groups. These language components again reuse other language components, as depicted in Figure 8.1. MontiCore's library of language components and their interrelations are explained in more detail in the MontiCore reference manual [HKR21].

The `FeatureConfiguration` language extends the `FeatureDiagram` language to reuse the symbols of features and feature diagrams. The `FeatureConfiguration-Partial` language extends the `FeatureConfiguration` language and adds selection and exclude statements. The following sections explain the syntax for the languages of the feature diagram language family in more detail.

```
01 package car.pl;                                                    FD
02 featurediagram Navigation {        optional feature   mandatory feature
03   Navigation -> Display & GPS & PreinstalledMaps? & Memory;
04   Navigation -> VoiceControl ^ TouchControl;   alternative feature group
05   Memory -> Small ^ Medium ^ Large;            selection feature group
06   Display -> SmallScreen | LargeScreen;        cardinality feature group
07   PreinstalledMaps -> [1..3] of {Europe, NorthAmerica, SouthAmerica, Asia, Africa};
08   TouchControl requires LargeScreen;   equal to: !TouchControl || LargeScreen
09   SmallScreen excludes TouchControl;   equal to: !(SmallScreen && TouchControl)
10   (Europe && NorthAmerica && Asia) requires (Large || Medium);
11 }                          cross-tree constraints can be nested expressions
```

Figure 8.2: Exemplary model conforming to the feature diagram language

8.1 The Feature Diagram Language

The feature diagram language describes textual feature models with cross-tree constraint expressions and cardinality groups. The language provides extension points for extending it with novel feature diagram concepts as language extension or for embedding other languages. This can be used to realize attributed feature diagrams.

The textual syntax of feature models is depicted by the example of a feature model `Navigation` in Figure 8.2. Each feature model has a name (l. 2). Similar to artifacts in programming languages such as Java, feature model artifacts can be arranged in packages with package statements (l. 1), which yields a hierarchical namespace for feature diagram names. The example feature model has the simple name `Navigation` and the full name `car.pl.Navigation`. The body of a feature diagram is enclosed by curly brackets and contains feature tree rules as well as cross-tree constraints. The root feature `Navigation` of the feature model has the same name as the feature model (l. 3) and four subfeatures. In this example, the root feature is the first feature tree rule. In general, however, this is not necessarily the case. Instead, the root feature is identified as the only feature that is not contained in a right-hand side of a feature tree rule. The subfeatures of `Navigation` are not arranged in a feature group, indicated by the separating character `'&'`. The features `Display`, `GPS`, and `Memory` are mandatory subfeatures of `Navigation`. `PreinstalledMaps` is an optional subfeature, indicated by a subsequent question mark. Subfeatures of a feature can be part of a feature group. The supported kinds of feature groups are

- alternative feature groups (l. 4), indicated by the separating character `'^'`,
- selection feature groups (l. 6), indicated by the separating character `'|'`, and

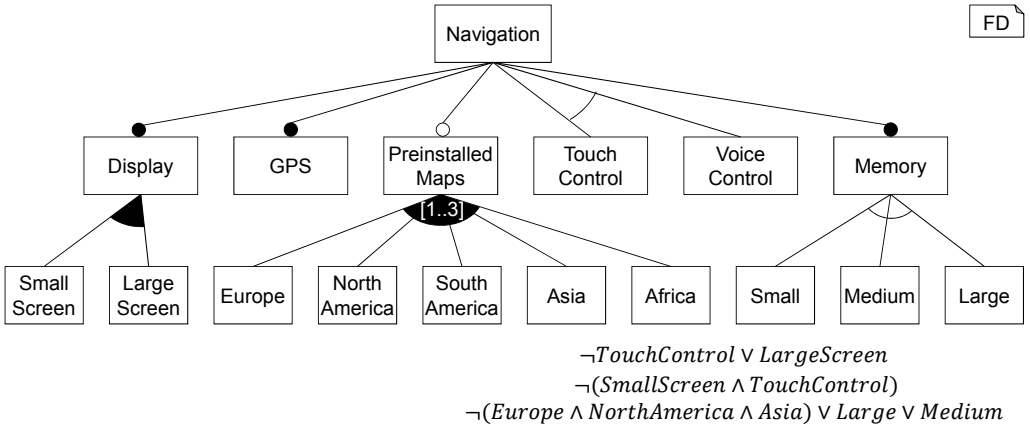


Figure 8.3: Graphical representation of the textual feature model presented in Figure 8.2

- cardinality feature groups (l. 7), indicated by a cardinality range from the minimum to the maximum number of selected features and a set with the subfeatures that are part of this group.

A feature must not be part of more than one feature group. However, a feature may have more than one group of subfeatures. In the textual notation, this is realized through multiple feature tree rules that have the same feature on the left-hand side. For example, the feature `Navigation` is on the left-hand side of two feature tree rules (ll. 3-4).

For realizing cross-tree constraints, the feature diagram language has two dedicated built-in operators. The `requires` operator (l. 8) indicates that the selection of a feature on the left-hand side of the operator requires the selection of the feature on the right-hand side in every valid feature configuration. The `excludes` operator (l. 9) indicates that the feature on the left-hand side of the operator and the feature on the right-hand side of the operator may not both be part of a valid configuration. Cross-tree constraints can be comprised of arbitrary nested Boolean expressions, including the dedicated built-in operators (l. 10). The meaning of the cross-tree constraint in l. 10 is that if `Europe`, `NorthAmerica`, and `Asia` are selected in a configuration, this requires either `Large` or `Medium` to be selected as well. A graphical representation of the textual feature model presented in Figure 8.2 is depicted in Figure 8.3.

For modularization of feature diagrams, the feature diagram language supports that a feature diagram imports other feature diagrams. If a feature diagram B imports a feature diagram A, all feature tree rules and cross-tree constraints of A hold for the feature diagram B as well. There is a flat namespace of feature names, which means that A and B may not use the same name for different features. If a feature name occurs in both diagrams, it identifies the same feature. Thus, any feature name f of A can be used in the feature tree of B. This integrates the feature f and the feature subtree

```
01 grammar FeatureDiagram extends Cardinality, MCBasicTypes, CommonExpressions {   MCG
02   FDCompilationUnit = ("package" package:MCQualifiedName ";")?
03                       MCImportStatement* FeatureDiagram ;
04   symbol scope FeatureDiagram = "featurediagram" Name "{" FDElement* "}" ;
05   interface FDElement ;
06   symbol Feature = Name ;
07   FeatureTreeRule implements FDElement = Name@Feature "->" FeatureGroup ";" ;
08   FeatureConstraint implements FDElement = constraint:Expression ";" ;
09   interface FeatureGroup = GroupPart+ ;
10   GroupPart = Name@Feature optional:["?"]? ;
11   OrGroup implements FeatureGroup = GroupPart ( "|" GroupPart )+ ;
12   Requires implements Expression <115>, InfixExpression =
13     left:Expression operator:"requires" right:Expression;
14 }
```

Figure 8.4: Excerpt of the grammar realizing the syntax of the feature diagram language

that the feature induces into the feature tree of B. However, all constraints for feature configurations of A that are described through feature groups, cross-tree constraints, and parent-child relationships between features, then also hold for the feature diagram B. With this import mechanism semantics, the root feature can be defined either in the imported feature diagram or in the local feature diagram. Any feature can be imported, not only the root feature of an imported feature diagram. If the local and imported feature tree rules do not describe a tree of features but a forest[1] of features or a feature that has more than a single parent, the model is not considered well-formed anymore. By importing other feature diagrams, a feature diagram may have feature tree rules that describe a tree but yield no valid feature configuration. We call such feature diagrams *void* feature diagrams and consider these well-formed. Suitable feature analysis [BSRC10] can detect void feature diagrams.

An excerpt of the grammar realizing the syntax of the feature diagram language is depicted in Figure 8.4. An `FDCompilationUnit` (ll. 2-3) contains the optional package declaration, a list of import statements, and a `FeatureDiagram`. A `FeatureDiagram` begins with the keyword `featurediagram`, has a `Name`, and contains `FDElements`. The nonterminal defines a symbol with the name of the feature diagram and spans a scope to create a namespace for contained feature names. The interface nonterminal `FDElement` (l. 5) is implemented by the nonterminals `FeatureTreeRules` (l. 7) and `FeatureConstraints` (l. 8). It further acts as an extension point. If a language ex-

[1] With *forest*, we denote a set of trees.

Common Expressions	A.b()	A.b	~	!	*	/	%	+	-	<=	>=	<	>	==	!=	&&	||	A?B:C	(A)
Feature Diagram	✘	✘	✘	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✔	✔	✔	✔	✔	✔

Figure 8.5: Some expressions of `CommonExpressions` are reused for the feature diagram language (✔), and some are not (✘)

tends the feature diagram language and adds another nonterminal implementation for the interface, novel kinds of feature diagram elements, such as for instance, blocks for declaring feature attributes, can be added. Each feature tree rule has the name of a feature on its left-hand side, followed by an arrow separating the left- and right-hand side and a feature group on its right-hand side. `FeatureGroup` (l. 9) is an interface nonterminal for which different implementations are included in the feature diagram language. The excerpt of the grammar only introduces the `OrGroup` (l. 11) as implementation, but similar nonterminals exist for `XorGroups`, `AndGroups`, and `CardinalizedGroups`. The interface nonterminal `FeatureGroup` is a further extension point of the feature diagram grammar. Novel kinds of feature groups can be added through language extension by adding further nonterminals implementing the interface.

The nonterminal `FeatureConstraint` introduces cross-tee constraints in feature models. For these, it reuses the `Expression` interface nonterminal from grammar `ExpressionsBasis` of the MontiCore language component library. As the feature diagram grammar further extends the `CommonExpressions` grammar, all implementations of the Expression nonterminal can be used for cross-tree constraints as well. The grammar for `CommonExpressions` introduces more kinds of expressions than the feature diagram intends to reuse. Hence, some expressions are explicitly forbidden to be used in feature models. This is realized as a context condition that yields an error if a forbidden expression is part of an AST. Figure 8.5 depicts which kinds of expressions are allowed to be used and which expressions are forbidden. For simplicity, the expression kinds are identified by the syntax of their operators.

The built-in operators for feature diagrams are added as implementations of the `Expression` nonterminal. The excerpt of the grammar depicts the `Requires` nonterminal (ll. 12-13) that implements the nonterminal for expressions and integrates a novel kind of expression with a parser priority of 115. It uses the `Expression` nonterminal for the left- and right-hand sides of the cross-tree constraint to enable nesting with other kinds of expressions.

All feature names within the feature diagram grammar are marked as name usages. This holds for feature names on the left-hand side of feature tree rules (l. 7) as well as for feature names on their right-hand side (l. 10). At the same time, there is a nonterminal introducing feature names in l. 6, but the nonterminal is not reachable from the start rule of the grammar and, therefore, is never parsed. Instead, the nonterminal has the

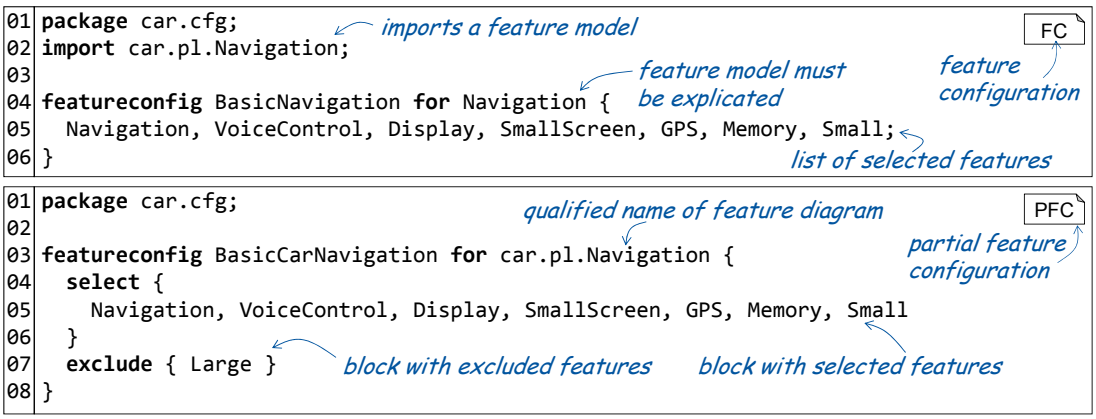


Figure 8.6: Exemplary models of the feature configuration language (top) and the partial feature configuration language (bottom)

sole purpose of defining feature symbols with the effect that MontiCore generates symbol table infrastructure for handling feature symbols. During symbol table creation of feature diagram models, the first occurrence of a feature name in a model introduces a feature symbol for this name. All subsequent occurrences of the name in the model are handled as usages of this name. This deviates from the default handling of MontiCore symbol definitions in which symbols are created if an instance of a symbol-defining nonterminal is contained in the AST of a model. Feature diagrams have this special behavior because the name of a feature is neither uniquely defined on the left-hand side of a feature tree rule nor on its right-hand side.

If feature names would be defined on the left-hand side of feature tree rules, multiple feature groups for the same parent feature could not be defined through individual feature tree rules (*cf.* ll. 3-4 in Figure 8.2) as their names would not be unique. Furthermore, leaf features would require special treatment, as their names only occur on the right-hand sides of feature-tree rules. If feature names were defined on the right-hand side of feature tree rules, the root feature would require special treatment because its name occurs only on the left-hand side of feature tree rules.

8.2 The Feature Configuration Languages

The feature diagram language family includes two languages for modeling feature configurations. An example of a model conforming to the `FeatureConfiguration` language is depicted in the top of Figure 8.6. The bottom of the figure depicts an example model conforming to the `FeatureConfigurationPartial` language. Models of both languages can contain package definitions and import statements that refer to feature

models. Each (partial) feature configuration model has a name and refers to a feature model, which is indicated by using the name of a feature diagram. Feature configuration models have a body that may only contain a comma-separated list of feature names (l. 5) that are contained in the configuration. Partial feature configuration models, instead, contain blocks with selected features (ll. 4-6) and blocks with excluded features (l. 7). Features for which no selection has been made are not explicated.

The grammars for these languages are not included here for reasons of brevity. However, the entire source code of the feature diagram language family, including these grammars, is available on GitHub[2].

8.3 The Feature Diagram Analysis Tool

Models of all three languages that constitute the feature diagram language family are processed by three individual language tools, where each is available as CLI and as Java API. The `FeatureDiagramCLI` tool processes feature models, while the `FeatureConfigurationCLI` tool processes feature configuration models, and the `FeatureConfigurationPartialCLI` tool processes partial feature configurations. These tools support the usual steps in language processing in a language's frontend, such as parsing a model to an instance of the AST, creating the symbol table for an instance of the AST, checking context conditions, storing a symbol table to a file, and pretty printing an AST. Each of these steps is realized as an individual method of the respective Java class of the tool. The main methods of the Java classes invoke `run(String[] args)` methods that realize the CLI.

The feature diagram analysis CLI tool (or short: FACT) utilizes the individual language tools for feature diagrams and (partial) feature configurations and realizes several analyses against feature models and feature configuration models. FACT can be used both as Java API or via the command line. It processes a feature model passed as an argument with the feature diagram tool and then pretty prints the feature model to FlatZinc [NSB+07], which is a DSL for constraint satisfaction problems. In general, feature models can be represented as constraint satisfaction problems [Bat05], which enables solving feature analyses through integration with SAT solvers.

Some kinds of analyses require a feature configuration model as an additional argument. If such an analysis is performed, the input feature configuration model is processed with the feature configuration tool and is also translated to FlatZinc. Afterward, FACT passes the FlatZinc representation of the feature model (and optionally, the feature configuration) to a constraint solver. The FlatZinc notation is a supported input format

[2]The feature diagram language family on GitHub: `https://github.com/MontiCore/feature-diagram`

of several constraint solvers such as Choco[3], Gecode[4], or OR Tools[5]. The indirection through FlatZinc decouples FACT from a concrete (third party) solver. Furthermore, it enables inspection of the translated feature diagram by language engineers and modelers. To integrate a solver with the FACT, the feature diagram language family provides the interface `ISolver`. This interface prescribes three methods that have a FlatZinc model as an argument. A concrete solver has to implement the methods.

hasSolution returns *true* if the passed FlatZinc model has a solution and $false$, otherwise. This can be used for analyses that check whether a configuration is valid.

getAnySolution returns a single solution (*i.e.,* values for open variables) for the passed FlatZinc model. This can be used to find a valid configuration of a feature model.

getAllSolutions returns all solutions for the passed FlatZinc model. This enumerates all valid configurations of a feature model. Unsurprisingly, calculating all configurations is inefficient, which may result in long execution times for large feature models.

8.4 Composing Feature Models with Domain Models

In MDD applications, feature diagram tools enable modeling a connection between the feature model and the domain models that can provide far more analyses for the correctness of the product line and can detect inconsistencies between the participating models. If the feature diagram language and the domain languages are not engineered as compositional languages in the same technological space (*i.e.,* in the same language workbench), realizing such connections is cumbersome. In this case, the languages are typically only loosely integrated by annotating/stereotyping the domain models with the names of the feature model or by using comments in the domain language to indicate the connection to a feature. With compositional languages, sophisticated forms of language composition can be achieved, among other things, to support bidirectional traceability between a feature and a domain model or proper type checking between features models and domain models.

As the feature diagram language family is engineered with the language workbench MontiCore, it can build upon MontiCore's powerful language composition mechanisms. This enables different forms of connections between the feature diagram language and domain languages in which features are realized. The following sections describe three of these forms.

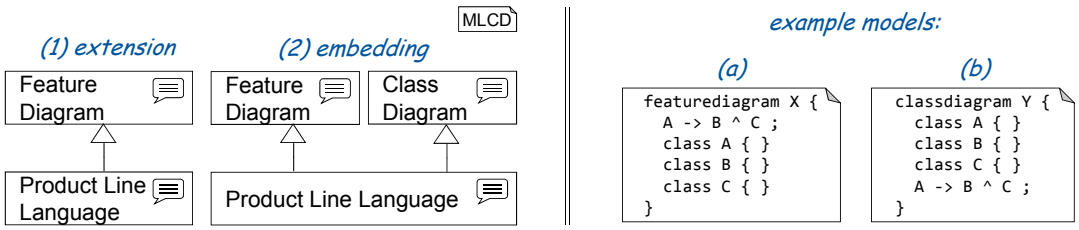


Figure 8.7: Examples for internal feature realizations on the language level (left) and models conforming to these languages (right)

8.4.1 Internal Feature Realizations

A simplistic approach for integrating feature models with the realization of features in domain models is to include both in a single model artifact. The produced integrated artifact is a 150% model with an integrated feature model and, thus, has similar advantages and disadvantages as regular 150% models. Technically, the product line language can extend the feature diagram language to reuse the feature diagram syntax and add novel concepts for the domain model parts of the language. However, internal feature realizations can alternatively be achieved by language embedding, where the feature diagram language is either the host language or the embedded language. Two examples for internal feature realizations with class diagrams as domain language are depicted in Figure 8.7. The left side shows the different options to conceive a language for internal feature realizations. The product line language can be realized either (1) as extension to the feature diagram language or (2) by embedding the class diagram language into the feature diagram language or vice versa. The latter is realized in MontiCore through multiple inheritance. The right side of Figure 8.7 depicts exemplary models for both cases that (a) the feature diagram language is the host language or (b) the class diagram language is the host language. Either alternative can be realized by both language extension or embedding.

An advantage of the integrated notation is that no inconsistencies between a feature model and the domain model may arise. Furthermore, a product line engineered with this approach comprises only a single artifact, preventing the scattering of domain artifacts. However, integrated feature realization models tend to become complex for larger product lines, and therefore this approach does not scale well. In addition, all parts of the domain model are bound to the feature model, which reduces the ability to reuse the domain model, or parts of it, with different language product lines. As domain model and feature model do not remain in separate models, each combination of feature language

[3]The Choco Website: `https://choco-solver.org/`
[4]The Gecode Website: `https://www.gecode.org/`
[5]The OR Tools Website: `https://developers.google.com/optimization`

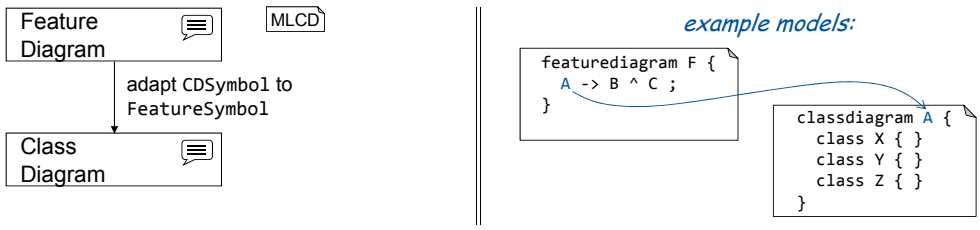


Figure 8.8: Example for feature diagrams referring to external feature realizations on the level of languages (left) and models (right)

and domain languages requires individual language tooling such as a parser. Therefore, internal feature realizations have limited reusability for domain language tools.

8.4.2 Referring to Feature Realizations

Through language aggregation between the feature diagram language and domain languages, features of the feature model can be related with feature realizations in the domain language via symbolic links. This requires that the feature realization in the domain language can be identified through a named model element that has the same name as the feature in the feature model. Technically, this approach demands implementing a symbol adapter from the symbol kind of the named language element in the domain language to the feature symbol kind defined in the feature diagram language. If the symbol adapter is integrated into the feature diagram language via configuration of its global scope, a context condition can check the existence of a feature realization for each feature of the feature model. This can also be the basis for further analyses and transformations operating on the integrated abstract syntax between the feature diagram language and the domain language. An example of feature diagrams that refer to feature realizations in the form of a class diagram model is depicted in Figure 8.8. The feature `A` of the feature diagram `F` refers to a class diagram with the name `A`. Other features of the feature diagram refer to further class diagram models.

Compared to internal feature realizations, this approach enables independent reusability of feature model and domain models, as these are only loosely coupled through language aggregation. Furthermore, the language infrastructure for these languages can be reused without modifications as well. The separation into different models further fosters the readability of the models, as these have fewer lines of code and are less complex in their structure. For language aggregation, it is viable that the name of the model element of the feature realization is identical with the name of the feature in the feature model. The flat namespace for feature names further prohibits using feature realizations with hierarchical names. Hence, this approach cannot enable reusing any language elements as feature realizations. Furthermore, the separation into individual models can yield

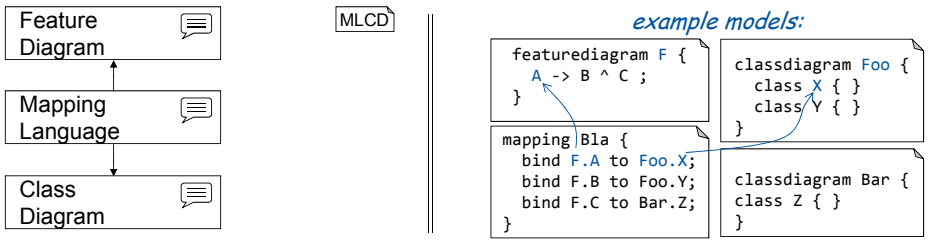


Figure 8.9: Examples for mapped feature realizations on the level of languages (left) and models (right)

inconsistencies between feature models and domain models. For example, renaming a feature realization makes it unreachable from the feature model unless the feature is renamed there as well.

8.4.3 Mapping to Feature Realizations

Introducing a layer of indirection between feature model and domain models helps to mitigate some disadvantages of the approach in Section 8.4.1 but also introduces more complexity. The indirection can be realized by introducing a mapping model or a tagging model [GLRR15] to draw the connection between a feature model and domain models. This mapping (or tag) model conforms to a mapping (or tagging) language aggregating the feature diagram language with the domain language(s). An example of feature diagrams that map features to their realizations in the form of classes of class diagram models via a mapping model is depicted in Figure 8.9. The feature `A` of the feature diagram `F` refers to a class `X` in the class diagram with the name `Foo`. Other features of the feature diagram refer to further classes of the same or different class diagram models.

An advantage of mapping features to their realizations is the loose coupling between a feature model and domain models. Any named model element can be reused as a domain model with a suitable mapping language. The feature realization can be identified with a name that does not match the feature name and thus can also be identified in a hierarchical namespace. Furthermore, this approach enables using heterogeneous models for realizing features and supports mapping a single feature to multiple solution space artifacts. The approach, thus, is beneficial for larger product lines in which solution space artifacts are to be engineered individually and separated from the product line. However, mapping models increase the complexity of the product line engineering as at least three models conforming to three different languages are required to realize a product line. The indirection can lead to inconsistencies between all three kinds of models, which have to be taken care of, *e.g.,* by proper tool support.

All presented approaches have individual strengths and weaknesses that make these suitable for different applications. The MontiCore feature diagram language family sup-

ports all three approaches and moreover, supports combining the approaches to achieve optimal results for an application. For instance, the language product line language (*cf.* Section 9.2) is a combination of integrated and external feature realizations. While feature models have integrated feature realizations in terms of referenced language component models and binding rules, the majority of feature realizations (including the grammars, parsers, and context conditions) are contained in external artifacts.

8.5 Discussion

The presented approach for realizing feature diagrams in the context of a language workbench that supports language composition is currently limited to the presented notations and language elements. While the language includes syntax for cardinalized feature groups [RBSP02], it does not support cardinalized features [MCHB11]. In addition, it would be possible to extend the language with additional feature diagram language concepts such as abstract features [TKES11]. Extending the language entails adjusting the feature analyses that are affected by the novel language elements accordingly. However, the visitor infrastructure generated by MontiCore enables extending analyses for new language elements with little effort [BEH+20]. A further advantage of building the feature diagram language on top of a language workbench is the fact that it can be integrated with other modeling languages to realize product lines in pervasive model-driven applications.

There is a strong correspondence between feature diagrams and context-free grammars [dJV02]. Both feature diagrams and grammars describe tree structures, where a feature diagram describes the configuration space of feature configurations and a grammar describes the valid parse trees. In this, a feature diagram can be modeled through a grammar in which each valid model of the grammar is a feature configuration. The concrete syntax of the presented feature diagram language is similar to the concrete syntax of MontiCore's grammar language by intention. Reusing notations when engineering a new language increases the usability of the language for language users [Hoa73].

Many approaches for modeling feature diagrams support a graphical syntax. The presented approach, however, is realized as a textual language, which may be less intuitive to understand for users familiar with the graphical feature diagram syntax. However, a textual syntax bears advantages such as better support for version control or no limitations regarding the model editor tools [GKR+07]. A pretty printer that translates the textual syntax of a model into a representation that can be imported by a graphical feature diagram editor of choice can be realized with little effort.

8.6 Related Work

Originally developed in the context of the feature-oriented domain analysis [KCH+90], feature diagrams have been adapted and extended in various contexts [CE00]. There is a variety of feature diagram tools, such as FAMILIAR [ACLF13], FeatureIDE [MTS+17], FeaturePlugin [AC04], pure::variants [Beu12], and SPLOT [MBC09].

FAMILIAR [ACLF13] is a textual DSML for feature diagrams and operations on feature diagrams. It aims at large-scale management of feature diagrams and, hence, supports modularization and composition of feature models. In contrast to the approach presented in this thesis, a model artifact in FAMILIAR may contain more than a single feature model or feature configuration. It may contain multiple feature models captured as individual variables as well as operations to modify feature models and to check properties of the feature models. These operations enable realizing both anomaly detection and correction. In the feature diagram language family presented in this thesis, the models do not contain operations on features or feature diagrams. Instead, such analyses are located in external tools such as the FACT tool, which makes the models less complex.

FeatureIDE [MTS+17] is a tool suite that aims at supporting all phases of feature-oriented software development and is realized as a plug-in for the Eclipse IDE. In FeatureIDE, feature diagrams and feature configurations are mainly viewed and edited in a graphical concrete syntax. The tool suite contains editors for feature diagrams, cross-tree constraints, and feature configurations. Furthermore, it supports several feature analyses and different integrations for assets that realize features in the source code, *e.g.,* for Java artifacts. In contrast to the feature diagram language family presented in this thesis, FeatureIDE is centered around editors and the Eclipse IDE instead of providing command-line interface tools. The feature diagram language family is built with the language workbench MontiCore and can be extended and integrated with other languages through the language composition mechanisms of MontiCore.

Beyond feature diagrams, which have found wide application in modeling variability of software, there are different variability modeling languages [BSL+13, CGR+12] such as orthogonal variability models or decision models. Orthogonal variability models represent variation points orthogonal to a base that contains the commonalities between all variants of a product line. Decision models focus on the decisions that have to be made during the derivation of a product. This is in contrast to feature models, which focus on the features that users can experience to distinguish between variants of a product line. There are different approaches for realizing decision models [SRG11].

Chapter 9

Engineering Feature-Oriented Language Product Lines with MontiCore

Throughout the past decades, software has become ubiquitous and, thus, the application domains for which software is engineered are nearly endless. The ability to encode problems of a domain into well-designed software depends on the means of the software language in which the problem is described [MHS05]. MDD with DSMLs reduces the gap between problem description and the solution domain [FR07]. Using a programming language, such as C++ or Java, that is agnostic of the domain for encoding such problems can lead to complex programs [BGM10]. Furthermore, implementing such programs in a programming language can be challenging since there are typically numerous ways of describing problems, which have individual advantages and disadvantages that only experienced programmers can assess.

Due to the variety of different application domains for software, the domain experts are, *e.g.,* mechanical engineers, legal experts, robotics experts, or dispatchers but rarely software engineering experts. Encoding, for instance, a law into a dedicated DSML enables realizing a compact yet understandable syntax compared to encoding the same law into a programming language. Among other things, this is because DSMLs can use domain vocabulary with which domain experts are familiar. Using an overly general software language, on the other hand, usually leads to accidental complexity [Bro87] for the modelers as these have to learn the complex language constructs of programming languages, although they require understanding only a small portion of the overall language's features for describing their problems.

The language composition mechanisms, as explained in Chapter 6, and modular languages in terms of the reusable language components, as described in Chapter 7, reduce the effort for engineering new languages and, hence, foster agile language engineering. Reusing language modules improves the quality of the language implementation – because modules can be tested independently – and reduces the time for engineering a new language because less additional implementation effort is required. Beyond this, a well-designed language uses concepts that its users are familiar with, *i.e.,* a language designer's task is "consolidation, not innovation" [Hoa73]. This motivates developing numerous DSMLs while reusing language parts whenever it is considered beneficial.

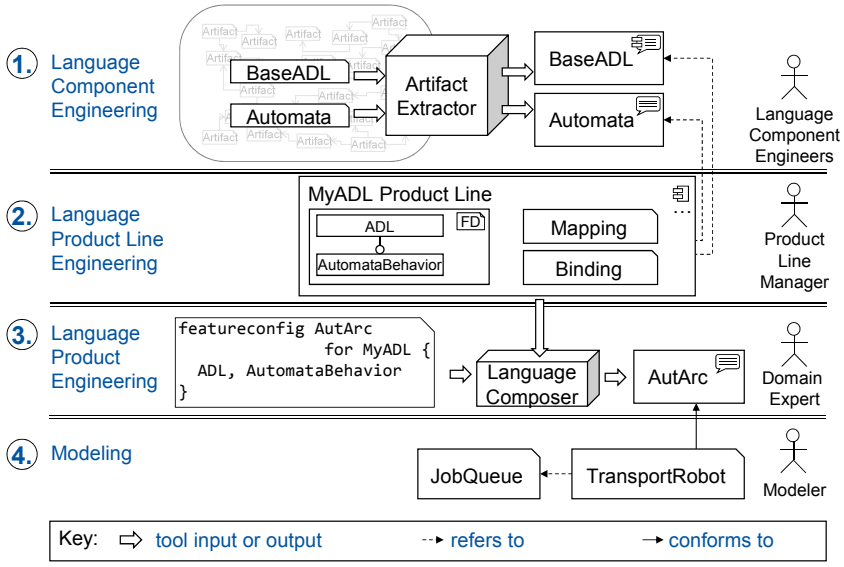


Figure 9.1: Overview of activities and roles involved in engineering and using language product lines by the example of the product line `MyADL`

A *language product line* [WHT+09] is a product line of (domain-specific) languages that supports managing families of similar languages coherently by fostering systematic reuse of languages. Product lines typically represent closed variability [SVGB05] in terms of a finite number of products that are part of the product line (*cf.* Section 2.3). Thus, language product lines differ from underspecification in other forms of language variability that typically realize open variability in terms of languages with extension points, hook points, or customization points. With language product lines, language engineers are able to express intended constellations of language components that can be used together. Moreover, they can restrict constellations of languages that should not be used together through constraints in the problem-space variability described, for example, by means of a feature model.

Language engineering is a complex endeavor in which providing a closed set of tried-and-tested combinations of language components in the form of a product line supports assuring the quality of the languages. Furthermore, language product lines enable separating the concerns of engineering languages and language components from the concern of composing languages for a concrete application or a family of similar applications.

This chapter describes the concept and implementation of the language component product line (LCPL) approach for feature-oriented language product lines in the context of reusable language components. The general ideas behind the concept are explained in Chapter 3. In the following, Section 9.1 describes the concept in more detail, and Section 9.2 describes its implementation. The implementation for language product lines

in MontiCore is based on the feature diagram language family introduced in Chapter 8. Section 9.3 discusses the LCPL approach and Section 9.4 describes related work.

9.1 Concept of a Feature-Oriented Language Product Line

The concept for language product lines in MontiCore relies on MLC models that identify the involved language components and on models of a novel LCPL language that describes the product line's feature model as well as mapping rules and binding rules. Mapping rules map a feature of the feature model to a language component, and binding rules bind an extension point of a language component to an extension of another language component. The following sections explain the processes and the roles involved in engineering language product lines, how the composition of language components in the product line is achieved, and how variants are derived from the product line.

9.1.1 Engineering a Language Product Line

The process of developing language product lines comprises four different engineering phases. An overview of these phases is depicted in Figure 9.1 by an excerpt of a language product line `MyADL`. The product line describes different variants of architecture description languages [MT00] (ADLs). For demonstration purposes, the product line uses a feature model with two features only and comprises two language components. The language component `BaseADL`, modeled through a language component model, defines language elements for definitions of components, ports, and connectors. The language component `Automata` defines the syntax and well-formedness checks for input/output automata [Rum96]. Through the root feature `ADL` that is mapped to the language component `BaseADL`, all languages that can be derived from the product line include language elements for components, ports, and connectors [RRW14]. The root feature has an optional child feature, `AutomataBehavior`, that maps to the `Automata` language component. If the optional feature is contained in a product of the product line, the resulting language can express automata models for the behavior description of atomic components [BHH+17, BRW16].

Language Component Engineering is the process of engineering individual language components and complete languages. For increasing off-the-shelf reuse of language components in language product lines, language components should minimize their dependencies to other language components. Different language engineers can develop individual language components independent of each other. To identify the constituents of a language component for being used as part of a language product line, it suffices to provide a language component model of the language. Other than that, no specific requirements have to hold for language components that are being used within language product lines. Thus, language components can either be engineered for the dedicated

purpose of being used within product lines but also any other language component or complete language can be reused for this purpose. From a language component model, an artifact extractor gathers the artifacts that are part of the language component and bundles these as described in Section 7.3.

In the example of Figure 9.1, language component engineers have produced two language components `BaseADL` and `Automata`. The language components are described by language component models (*cf.* Section 7.4.1). An artifact extractor tool collects the artifacts that realize the language components based on the language component models. The tool bundles these artifacts to archives, which are units of reuse for the product line.

Language Product Line Engineering is the realization of the language product line in terms of its feature model, mapping rules, and binding rules by product line managers. This process determines which variants or language products should be part of the product line and how these can be realized by the systematic composition of language components and languages. The process, thus, includes the selection of suitable languages and language components. If no suitable language can be reused for a product line feature, product line managers can also specify requirements for novel languages that language engineers implement. For instance, a product line manager who engineers a language product line for ADLs intends to offer an optional language feature for dynamic reconfigurations of connectors [BHK+17]. If no suitable language component exists, she may define specific requirements that a novel language should meet to best integrate with the other language components of the product line.

In the example of Figure 9.1, the product line contains a feature model with the root feature `ADL` and its optional subfeature `AutomataBehavior`. The root feature is mapped to the language component `BaseADL`, and the other feature uses the language component `Automata`. The product line uses a single binding rule that indicates that language embedding should be performed at the extension point realized through the grammar rule `Behavior` of the `BaseADL` and the grammar rule `Automaton` of the grammar `Automata`. This binding rule suffices for realizing language embedding as described in Chapter 6.

Language Product Engineering is the process of deriving a language (product) from a product line. Sometimes this is also referred to as a *language variant*. A domain engineer determines which features of the product line have to be selected to build a language suitable for a specific application or application domain. This means that a domain expert creates a feature configuration model that describes a selection of language components – through the features of the feature model – that should be composed to form the product. The derivation of products from the product line is performed by a language composer tool, which automatically composes the selected language components and, as a result, creates a novel language component or standalone language. This re-

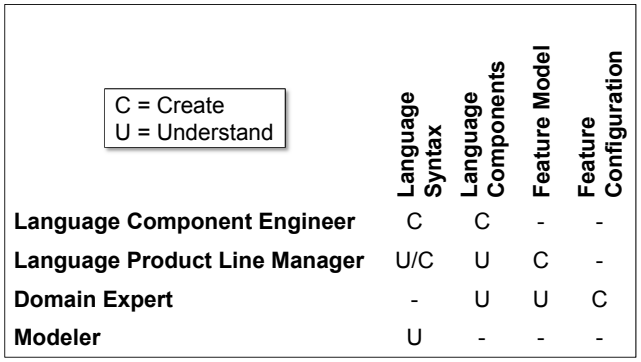


Figure 9.2: Roles involved in engineering languages through language product lines

sulting language component can be customized with MontiCore's means for integration of handwritten code (e.g., the TOP mechanism, hook points, an extension of the grammar). If languages are composed via language aggregation (*cf.* Section 2.2), the result of language product engineering can also be a set of language components.

In the example of Figure 9.1, a domain engineer selects both features of the feature model `MyADL` in the feature configuration `AutArc`. The language composer tool composes the language components of the selected features and produces a new language component with the name of the feature configuration.

Modeling in this context means that a modeler crafts models conforming to a language component that is derived from a language component as a result of language product engineering. However, for modelers, a language that has been derived from a product line should not be distinguishable from a language that has not been derived from a product line.

In the example of Figure 9.1, modelers create models such as `TransportRobot` and `JobQueue`, which conform to the `AutArc` language. They use the `AutArc` language component through a language tool that is contained in the language component.

9.1.2 Roles Involved in Language Product Lines

Software language engineering in general always involves a language engineer who is able to design, conceptualize, and implement the software language using a language workbench or similar language engineering tools. In the context of modeling languages, *modelers* are users of the software language and are rarely experts in language engineering – sometimes, they are not even software engineers. The distinction between these two roles already has certain challenges for language engineering techniques. For instance, the tools language engineers prefer to use can differ largely from the tools modelers prefer.

However, the distinction into different roles helps to separate the concerns of the different roles in terms of tasks, responsibilities, and required capabilities. The engineering of language product lines involves the four roles *language component engineers*, *product line managers*, *domain experts*, and *modelers*. All four roles have specific tasks and require specific expertise in engineering a language product line. The roles in the process of engineering languages product lines are introduced in Figure 9.1. Figure 9.2 explains the tasks for each role in more detail. For some roles, it suffices to understand certain parts of the product line only, while other roles have to create these parts. Creating parts of the infrastructure includes understanding these parts. As usual with roles in software engineering, a single physical person can embody multiple roles at once.

Language Component Engineers are language engineers whose task is to build language components. To this end, they must have the expertise to engineer languages. Beyond this, language component engineers must be able to design reusable languages and, due to this, focus on documentation of the language's extension points and extensions. Language component engineers do not need to be familiar with engineering product lines through feature models. Furthermore, they are not necessarily required to be familiar with language composition in general. Instead, the language composition can also be described by language product line managers.

Language Product Line Managers gather language components for realizing a language product line. To this end, they create feature models and mapping rules from features to language components. Furthermore, they create binding rules to integrate the syntax of the language components. Ideally, language product line managers do not have to be language engineers. This is the case if extension points and extensions are well documented, and all language components can be used together as intended. In practice, reusing language components with the purpose of composing these requires an understanding of the language component's internals and thus, requires language engineering expertise. In some cases, it might be required for language product line managers to implement "glue" code for missing pieces of integrating syntax.

Domain Experts create feature configuration models, *i.e.,* select features of the feature model to build a product of the product line. For realizing a useful product, they should be experts in the application domain of the DSML. Domain experts do not have to be language engineers for the purpose of deriving a product from the product line. However, domain experts can optionally customize the language component that results from the language derivation tooling.

Modelers are domain experts who use the language component that is the result of deriving a product from the product line to produce models for concrete applications. Modelers do not need to be aware of any product-line-related languages and tools such as feature models or the language derivation tooling. In fact, modelers do not have to

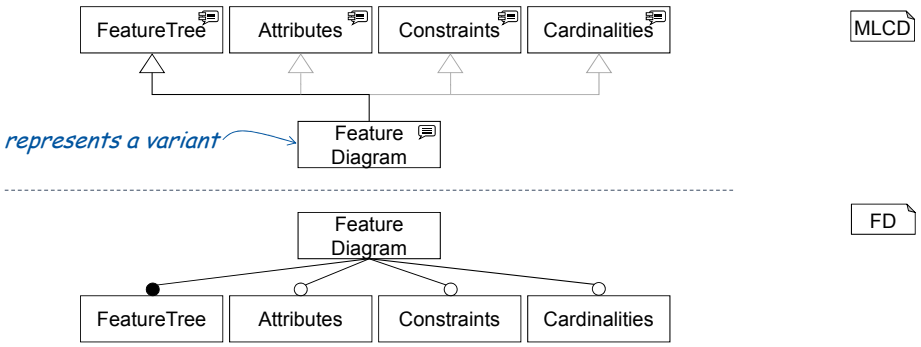


Figure 9.3: Example for composition of language components in language product lines

be language engineers. Nevertheless, to understand the syntax and semantics of the language they use, it is helpful if they have language engineering expertise.

All four roles require specific expertise, and the separation relieves roles from requiring expertise in all fields. Although almost all roles require language engineering expertise, the level of detail in which specific knowledge is required differs. While language component engineers have to be experts in engineering every aspect of a software language, language product line managers mainly have to be familiar with language infrastructure for which manual composition is required. It is impossible to alleviate language product line engineers from software language engineering expertise [VCPC13, VCCA14].

9.1.3 Describing the Composition of Language Components

In our language product line approach, the features from the feature diagram refer to language features that are realized by language components. The explicit composition of language components is described by binding rules. The approach also allows implicit relationships between the language components, *i.e.,* language composition not described by binding rules. LCPL, thus, does not prescribe how, *e.g.,* inheritance relationships between language components and the arrangement of features in the feature tree influence each other.

A simplistic approach for realizing language product lines from a set of given language components can be achieved by translating all feature-subfeature relationships in the feature model to equivalent but inverted inheritance relationships between the language components. This is depicted by the example of a product line of feature diagram languages in Figure 9.3, where the feature model of the language product line is depicted at the bottom, and the language components are depicted at the top. Each variant of the product line should contain the language component `FeatureTree` and may optionally contain the language components `Attributes`, `Constraints`, and `Cardinalities`.

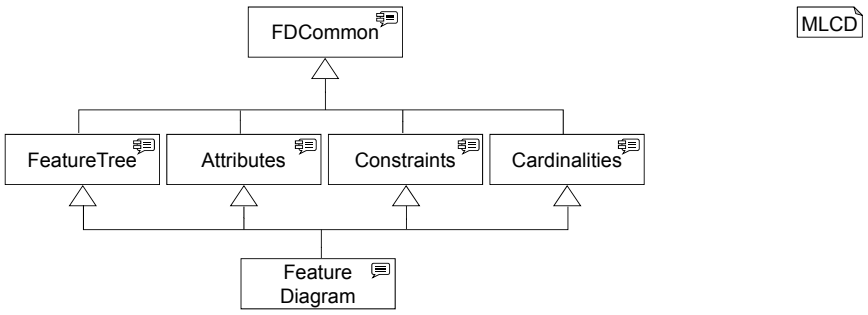


Figure 9.4: Example for the implicit composition of language components in language product lines with a common base

In the feature model of the product line, all language components are denoted by features with the same name as the language components to which these refer. The root feature of the product line has the name `FeatureDiagram`. The language derivation tooling synthesizes a new language component for each product of the product line. The new language component extends all language components that correspond to features that are selected in the variant and integrates their syntax.

The language product line can also be established from a set of existing language components. By constructing a feature tree from the (inverted) inheritance relationships, the feature model describes constellations of allowed or forbidden subsets of the set of inheritance relationships between the language components. For this, the feature model can use mandatory and optional subfeatures, subfeature groups, and cross-tree constraints. The set of language components and all inheritance relationships realize the product line, while each subset of inheritance relationships describes a legal or illegal variant of the product line that is the realization of the respective feature configuration.

In this example, the parts of the feature diagram language are developed independently and, hence, they cannot rely on a common language component from which they inherit. Therefore, it is impossible that the composition of the syntax of the individual parts can be realized, for instance, by implementing a common interface nonterminal. Instead, the composition has to be realized via explicit grammar rules in the `FeatureDiagram` grammar. For the product line, this means that the composition has to be described through explicit binding rules instead of implicit composition. The reason for this is that any description for integrating the languages in the `FeatureDiagram` grammar would cause an invalid grammar if the inheritance relationship is not established because the respective feature has not been selected for a language variant. With binding rules, the description of the language composition between two language components `MLC1` and `MLC2`, is only included in the resulting language component if both `MLC1` and `MLC2` are part of the variant through the selection of features that map to these.

To circumvent explicit statements of the composition in binding rules, a base language component can be introduced that all language components of the product line implement. This base grammar can introduce common (interface) nonterminals acting as extension points that some grammars use and for which other grammars provide extensions by implementing the interface. In this scenario, the integrating grammar is not required to provide any grammar rules to describe the language composition except for deciding which start nonterminal to use. Thus, the language product line does not require any binding rules and can instead rely on the implicit composition of the languages. This is depicted by the example of the `FeatureDiagram` product line in Figure 9.4.

However, both forms of language composition rely on language inheritance between language components used by the features. This produces an inherent coupling between the language components that prevents reusing an individual language component in a different context or in another language product line. Furthermore, this coupling complicates replacing a language component with a new one if other language components that are used in the same product line rely on the language component that is to be replaced. The evolution of the product line in terms of adding novel features and language components also requires more effort if the language components of the product line are already strongly coupled. Other forms of language composition, such as language embedding and language aggregation, can compose languages that do not depend on one another.

As introduced in Section 2.2, language embedding describes the embedding of an *embedded* language into a *host* language. In MontiCore, language embedding constitutes the *composed* grammar, which extends both the grammars of host and embedded languages. Through the inheritance relationships in the grammar alone, it is not clear which grammar is the host grammar and which is the embedded grammar. Instead, this is determined by integrating syntax in the composed grammar. For language product lines, an intuitive idea is to perform language embedding along the feature-subfeature relationships. In this, the language component of the root feature becomes the host language, and the language components of subfeatures of the root feature become embedded languages. However, in practice, this approach is too restrictive. Sometimes it is helpful to perform embedding, *e.g.,* with languages of sibling features. For realizing embedding in language product lines, our concept defines binding rules that describe embedding of a language with a nonterminal acting as an *extension* into a nonterminal of another language acting as an *extension point*.

For instance, the language component `BaseADL` contains an interface nonterminal `IBehavior` that acts as an extension point for integrating component behavior descriptions. A binding rule in the product line describes that the nonterminal `Automaton` of the language `Automata` should be used as an extension bound to the extension point through language embedding. When deriving a language variant from the product line, the language composer tool generates syntax that integrates the languages. The realization of this is described in Section 9.2.

For language aggregation, the symbol tables of language components have to be integrated. To describe this in a language product line, it suffices to indicate which symbol adapters should be established between symbol kinds of the involved languages. A dedicated kind of binding rules describes the adaptation from symbols of a *source* kind to symbols of a *target* kind. From such rules, the language composer tool generates an abstract class for the symbol adapter. The implementation of the adapter must be provided through the TOP mechanism as it cannot be generated. This is due to the fact that the adapter implementation depends not only on the source and target symbol kind but also on the purpose of the adapter. For instance, depending on the purpose, there might be different adapters that translate between state symbols of an automata language and class symbols of a class diagram language. Thus, the number of potential adapters that can be realized for each pair of source and target kind is infinite. However, in the context of product lines, the adapter is specific to a pair of language features whose language components' symbol tables are to be integrated. Therefore, the adapter must be contained in every product of the product line that includes both features and, otherwise, should not be part of the product. To keep the language components of aggregated languages independent of another, the symbol adapter should not be part of either of the language components. Instead, it can be either part of the product line or of a specific language variant. To foster the reusability of adapters, it is preferable to locate these at the level of the product line.

The composition of context conditions does not need to be described through binding rules. Instead, the sets of context conditions of the individual languages are unified during language variant derivation and can be checked against the composed languages after applying the language composer tool. Sometimes, additional context conditions should be applied that are not part of any of the used language components but are specific to the combination of two or more language components. Therefore, our concept for language product lines also enables specifying context conditions at the level of the product line.

Following the concept of product-line-specific symbol adapter implementations and context conditions, other parts of the language infrastructure can be added to the product line as well.

9.1.4 Language Variant Derivation

The composition operators for language components presented in Chapter 6 enable composing languages according to the binding rules as presented in Section 9.1.3. After a language product line has been modeled, variants can be derived from the product line. As LCPL is based on feature diagrams, variants are described by feature configurations. The process of deriving a language from a given language product line and a feature configuration is called the *language variant derivation*. This process is performed by a language composer tool, which processes an input product line comprising feature model,

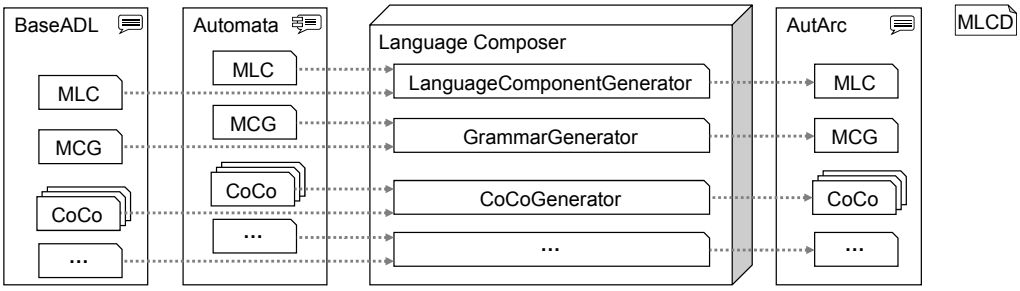


Figure 9.5: Structure of a language composer

binding rules, mapping rules, and language components as well as a feature configuration and produces a composed language component according to the selection of features. If language aggregation is used through binding rules, the result can be a set of multiple language components.

To realize the language variant derivation, the language composer tool comprises individual code generators for all handwritten constituents of the language components. For instance, one generator composes the grammars, another generator composes the language component models, and a further generator composes the context conditions. Which generators are required depends on the technological space in which language components are realized and on the forms of language composition that are supported. Dedicated generators for generated language constituents are not required to be part of the language composer, as these can either be re-generated from other composed artifacts or can remain separated. An overview of the internal architecture of a language composer is depicted in Figure 9.5. The figure visualizes the composition of the language components `BaseADL` and `Automata` producing the composed language component `AutArc`. The language composer contains an individual generator for each handwritten constituent of a language component, including a generator responsible for composing the MLC models.

Conceptually, LCPL does not prescribe an order in which language components have to be composed during language variant derivation. In fact, all languages can be composed at once. The result of language variant derivation is a (set of) novel language components that can be reused by another language product line as well. Therefore, any order of compositions can be simulated by composing pairs of language components and replacing these and the features in the product line with the result of the composition. Whether a particular order of composition has to be considered in the implementation, however, depends on the concrete language composition operators. Some composition operators may prescribe to preserve a specific order, while others do not.

The start nonterminals of the composed languages are already specified through the binding rules and the implicit forms of language composition. With language aggregation

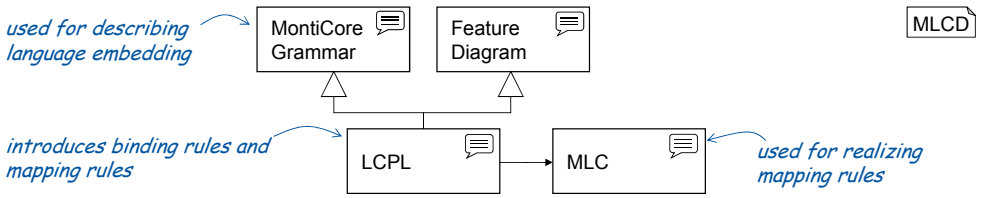


Figure 9.6: The LCPL language reuses other languages

used in at least one binding rule, the outcome of language variant derivation is a set of language components. Each language component has a start nonterminal, which is determined by the host language of embedded languages. For other forms of language composition, the start nonterminal has to be derivable as well.

The context conditions of the composed language is the set union of the context conditions of all language components of selected features. Checking the context condition of the composed language is realized with the generated visitor-based context condition checker of the composed language. Our concept for language product lines allows product line engineers to reuse a language component not only through its start nonterminal but through any nonterminal. For example, the nonterminal `State` of the language component `Automata` can be used for embedding, even if it is not the start nonterminal of the `Automata` grammar. As an effect, it might occur that some AST nodes from nonterminals of original language components are not part of any parsed model of the composed language. Therefore, context conditions implemented against such nonterminals do not influence the well-formedness of models of the composed languages. With a suitable analysis, such nonterminals can be identified and removed during composition.

9.2 Realizing Language Product Lines in MontiCore

The MontiCore feature diagram language (*cf.* Chapter 8) is the basis of LCPL according to the concept described in Section 9.1. For this purpose, the feature diagram language has been extended with novel language elements that describe (1) the connection between features of the feature model and language components that realize these (*i.e.,* mapping rules) and (2) the connection between two language components to describe their language composition (*i.e.,* binding rules). We refer to this extension of the feature diagram language as the language product line or LCPL language.

The LCPL language describes product lines as integrated models. In previous realizations of the approach [BEK+18a, BEK+19], the feature diagram and binding rules are located in separate model artifacts. This enables reusing different notations for feature diagrams, such as from Feature IDE [TKB+14] because the feature diagram notation does not have to be extended with language-product-line-specific syntax. However, this

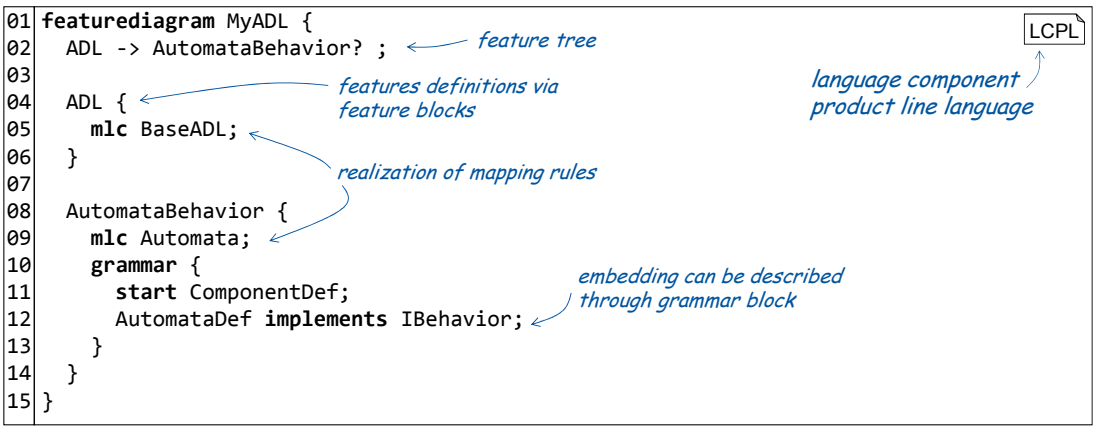


Figure 9.7: A model of the LCPL language demonstrating language embedding

separation introduces a gap between the feature model and the model containing the mapping and binding rules. The gap can lead to inconsistencies between the models and complicates the configuration of the product line, as more artifacts are involved. The separation of artifacts enables reusing a single feature model with different binding models or using a binding model with different feature models. This is not possible with integrated product line models. Nevertheless, our experiences have shown that using the same feature model for different language product lines is rarely feasible.

9.2.1 The Language Product Line Language

The LCPL language is a MontiCore language used to create models that describe language product lines. It extends the feature diagram language (*cf.* Chapter 8) with novel feature definition blocks. Inside these definition blocks, a feature can reference an MLC model via its name. These references realize the mapping rules that map a feature to a language component. The LCPL language further reuses grammar rules from the MontiCore grammar language through language inheritance and uses `MLCDefSymbols` defined in the MLC language (*cf.* Section 7.4.1). Figure 9.6 depicts the LCPL language and its relations to reused language components.

Binding rules are realized inside of feature definition blocks. The LCPL language currently supports two forms of binding rules. With the first form, product line engineers can specify binding "glue" inside of a block with grammar statements. This is especially useful for realizing language embedding, where different patterns can be used to connect an extension point to an extension nonterminal. Furthermore, such grammar blocks can indicate the start rule for the composed language's grammar. The grammar block may use nonterminals of all grammars that are part of the product line. The second form of binding rules describes the types of adapters between two symbol kinds.

```
01 featurediagram MyADL {                                          LCPL
02   ADL -> AutomataBehavior? ;
03   ADL -> CDTypes | JavaTypes ;
04
05   CDTypes {
06     mlc CD4A;
07     adapt CDTypeSymbol to PortTypeSymbol;
08   }
09                                          describing language aggregation
10    JavaTypes {
11     mlc JavaDSL;
12     adapt JavaTypeSymbol to PortTypeSymbol;
13   }
14
15   // further feature definitions omitted
16 }
```

Figure 9.8: A model of the LCPL language demonstrating language aggregation

The syntax of the LCPL language is depicted in Figure 9.7 by the `MyADL` example as introduced in Figure 9.1. By inheriting from the feature diagram language, the LCPL language reuses the definition of the feature diagram type (l. 1) and the syntax for specifying the feature tree (l. 2). The definition for cross-tree constraints is also reused. Feature definition blocks begin with the name of the feature, followed by a block embraced in curly brackets. The block contains statements for defining mapping rules and binding rules. The mapping rule statements begin with the keyword `mlc`, followed by the name of the language component model (*cf.* l. 5 and l .9) to which the feature maps. Feature definition blocks can further contain binding rules in the form of embedded grammar blocks, as depicted in ll. 10-13. In this example, the grammar block contains two grammar rules. The first rule in l. 11 indicates that the nonterminal `ComponentDef` should be used as start nonterminal of the grammar generated for language variants that include the feature `AutomataBehavior`. Alternatively, the product line manager could have indicated the start rule in a grammar block contained in the body of the definition of the root feature. The second rule (l. 12) overrides the nonterminal `AutomataDef` from the grammar of the language component `Automata` and implements the interface nonterminal `IBehavior` defined in the grammar of the language component `BaseADL`. In the context of language embedding, `IBehavior` is an extension point for which, as modeled through the grammar rule, `AutomataDef` is an extension.

To demonstrate language aggregation, the `MyADL` product line in Figure 9.7 is evolved by adding a selection of two new subfeatures to the feature `ADL`. An excerpt of the new LCPL model is depicted in Figure 9.8. Both new features realize types that are used by component port definitions [BRW16]. In the feature `CDTypes`, the port types are defined in classes or interfaces of CD models. With the selection of the feature `JavaTypes`, types defined in Java can be used as port types. Both features are defined in feature definition

blocks and reuse individual languages, namely the `JavaDSL` language and the `CD4A` language. The `CD4A` language is composed with the language component `ADL` through language aggregation, as described by the binding rule in l. 7. The rule indicates that `CDTypeSymbols` defined in class diagrams are adapted to `PortTypeSymbols` of ADL components. Through this, name usages in the ADL may refer to name definitions of types in class diagrams. In analogy to CD4A, the `JavaTypeSymbols` of the `JavaDSL` are adapted to `PortTypeSymbols`, as described in l. 12. The composition of the language components based on a selection of features is described in Section 9.2.2.

9.2.2 The Composition Infrastructure

This section overviews the implementation of the language composer tool for language product lines as introduced in Figure 9.1 in the context of MontiCore language components and the LCPL language. The tool comprises individual code generators for the different handwritten constituents of language components, as depicted in Figure 9.5.

Composing Language Component Models

For each language component that is the result of an execution of the language composer, the `LanguageComponentGenerator` produces an MLC model. The name of the generated language component equals the name of the feature configuration that is used to describe the product derived from the product line. Furthermore, the language component declares the set of all artifacts that are part of the language and the set of artifacts and languages that the language component is allowed to use. Both sets are calculated based on the respective information contained in all MLC models (*cf.* Section 7.4.1) that are input to the composition. For the artifacts that are part of the language component, the generator evaluates the regular expressions of the input MLC models to obtain a set of artifacts. The own artifacts of a generated language component are obtained by forming the set union of the input artifact sets. Furthermore, all artifacts that are defined as part of selected features of the product line (such as adapter implementations or additional context conditions on the level of product lines) are added to the set of own artifacts. The generated language component model does not include any regular expressions to denote own artifacts to avoid undesired interactions between the include and exclude statements of individual input language components. The set of artifacts that the generated language component is allowed to use is the union of the sets of artifacts that input language components are allowed to use.

Composing MontiCore Grammars

The generator that performs the composition of grammars in the context of language product lines realizes language inheritance, language extension, and language embedding. For language aggregation, no composition of MontiCore grammars is required. The

`GrammarGenerator` produces a grammar with the name of the feature configuration that is used to describe the product derived from the product line. The grammar extends all grammars of language components to which the selected features map. Through the LCPL language, each binding rule is attached to a feature. All binding rules of all selected features that are realized via blocks of grammar statements are relevant for the grammar generator as well. All grammar statements contained in such blocks are added to the generated grammar. These binding rules not only realize the part of language composition that integrates the concrete and abstract syntax but also determine which grammar rule has to be used as the start rule.

Composing MontiCore Context Conditions

As described in Section 9.1.4, the context conditions are composed during language inheritance and language embedding via set union. For language aggregation, no context conditions need to be composed.

Unifying the sets of context conditions first requires identifying the sets of context conditions for each language. In previous realizations of LCPL [BEK+18b, BEK+19], language engineers had to explicate the names of all context conditions of a language as part of the language component. With the current approach, language engineers are relieved from that. In MontiCore, context conditions implement context conditions interfaces generated from a grammar. With an MLC model of a language component and an underlying artifact model of MontiCore languages, the artifacts realizing the context conditions can automatically be identified by the language composer tool.

Context conditions are checked against the AST of a language with visitor-based context condition checkers. As MontiCore generates these checkers for each language component, it is not necessary that the language product line tooling performs any additional composition. However, checking the context condition has to be triggered, which is typically realized within a language tool. To support language engineers in this, the language composer generator for context conditions synthesizes a `ContextConditions` class, which prepares a checker by adding all context conditions of all languages that are part of the composed languages and the context conditions that are part of the language product line as described in Section 9.1.4. Through adjusting the context conditions added to the checker, the context conditions produced for composed language components can be customized. Language engineers can customize context condition checkers via the TOP mechanism.

Composing MontiCore Symbol Tables

Symbol tables are a central part of the language infrastructure for realizing language composition. The language composer tool, however, does not have to consider the composition of most parts of the symbol tables as the composition is performed by Monti-

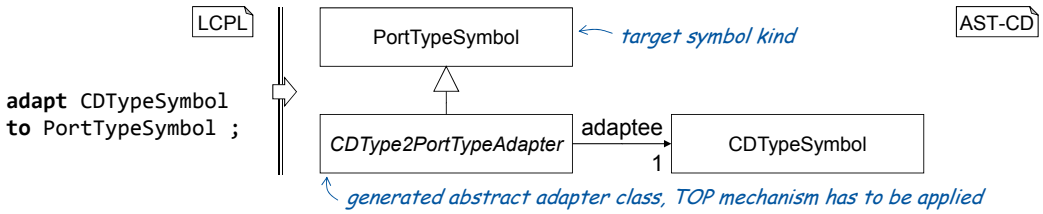


Figure 9.9: Example for a binding rule (left) and the adapter generated from this rule while deriving a product from the product line (right)

Core during language inheritance (*cf.* Section 6.1). Through the composed grammar produced by the language composer tool, MontiCore generates an integrated symbol table infrastructure for the language product. For language aggregation, the symbol table infrastructures are not integrated through integrated types for scopes and symbols. Instead, the global scope of a language is configured with suitable symbol resolvers that realize resolving in symbol tables of foreign languages (*cf.* Section 6.4).

With binding rules describing symbol adapters, LCPL enables the reuse of language components via language aggregation in product lines. As described in Section 9.1.3, the language composer produces abstract symbol adapter classes for which product line engineers provide realizations. In the implementation, the TOP mechanism [HKR21] is employed for integrating the generated abstract class and the handwritten extension that realizes the adapter. If product line engineers do not provide a handwritten adapter class, only the generated abstract adapter exists, which yields compilation errors. Figure 9.9 depicts an example of a binding rule for symbol adapters and the symbol adapter class generated while deriving a language from the product line. The left side of the figure displays a binding rule that indicates that `CDTypeSymbols` should be adapted to `PortTypeSymbols`. In the generated abstract syntax, this results in the abstract class `CDType2PortTypeAdapter` depicted on the right side of the figure. The abstract class extends the class `PortTypeSymbol` and has an attribute of the type `CDTypeSymbol`. A handwritten extension of the adapter class must carry out the actual adaption.

9.3 Discussion

The approach for realizing language product lines presented in Section 9.1 uses grammars for describing concrete and abstract syntax in an integrated fashion, as the approach is realized with the MontiCore language workbench (*cf.* Section 9.2). Other language workbenches for textual languages may use grammars only for the description of the concrete syntax and realize the specification of abstract syntax via metamodels. This requires only a slight adaption to LCPL, as composition operators are defined per constituent of the language.

Languages defined via grammars are typically textual. Building product lines for graphical (*e.g.,* diagrammatical) languages with the presented approach can be achieved by building a graphical visualization on top of the textual model representation [RRW13].

The presented language product line approach supports language embedding, extension, inheritance, and aggregation. However, the approach can be extended with novel forms of language composition with little effort. Adding a novel language composition technique requires introducing a new syntax element for connecting extension points and extensions in the LCPL language, as explained in Section 9.2. Furthermore, the generator of the language composer has to be adjusted.

Some forms of language composition enable integrating languages or language components that are developed independently of another. For example, language aggregation is a loose coupling of languages whose infrastructures remain separated. The integration of the languages is realized either via symbolic interfaces or symbol adapters. Language embedding in MontiCore means that the composed language inherits from the host language and the embedded language, whose infrastructure in large parts remains independent of another as well. These forms of language composition enable better "off-the-shelf" reusability of language components and, thus, are suitable for being reused in language product lines. However, our experiences have shown that realizing product lines is also beneficial for realizing a family of similar languages or language components that a single language engineer has designed to be reused for various purposes. In such use cases, language product lines also can employ forms of language composition that require dependencies between the composed languages, such as language inheritance and language extension.

The MontiCore language component library is a basis used by most languages engineered with MontiCore. Language engineers can choose between different grammars for types, expressions, literals, and statements [BEH+20]. These grammars are in complex inheritance relationships that language engineers have to untangle and understand for reusing the optimal constellation of grammars for engineering their language. A language product line for the language components induced by these grammars helps to manage this complexity. It can provide a deliberate set of language variants that are commonly used together as a basis for engineering languages. This reduces the effort for creating new languages because numerous irrelevant constellations of language components can be forbidden in the feature model of the product line.

Previous versions of LCPL presented in this thesis rely on explicit [BEK+18a] or implicit [BEK+19] marking of extension points in terms of interface nonterminals of the grammars. In these approaches, language components contribute extension points to *interfaces* of the language components. We deliberately removed these interfaces as the disadvantages described in the following outweigh the advantages. An advantage of such an interface is that it separates parts of the language components that are relevant for language composition from internal grammar nonterminals and rules that are not planned to be used either as extensions or as extension points of the language

components. This hides language component internals and fosters the black-box fashion of software components. Another advantage of explicit interfaces of language components is that there can be different realizations for the same interface. Reifying this in a concrete language workbench, however, complicates language component engineering, as interfaces and implementations have to be realized as separate artifacts.

A disadvantage of explicit marking of language component interfaces is the separation of the interface of the language component and the interface nonterminal in the language component's grammar that realizes a particular extension point into different artifacts. Both artifacts have to be documented independently, and the evolution of the language can cause inconsistencies between both. Another disadvantage of explicit language component interfaces is that language engineers have to foresee all extension points and establish these on the language component interfaces. An unforeseen nonterminal cannot be used as an extension point, limiting the reusability of the language component. Implicit interfaces based on strict rules for language components, for example, by using all interface nonterminals as extension points, however, have the disadvantage that interface nonterminals can unintendedly be marked as extension points. This can complicate reusing foreign grammars via their extension points.

Our concept for language variability, hence, relies on underspecification in the abstract syntax through extension points in the form of grammar nonterminals. In MontiCore, these can be, for instance, interface nonterminals that prescribe certain abstract syntax elements or external nonterminals that must be provided by inheriting grammars or by class nonterminals that are extended in inheriting grammars. In related workbenches that rely on different technological spaces, extension points can be realized, for instance, through interfaces or abstract classes in metamodel-based languages [SBPM09], through merging of abstract syntax elements [DCB+15], or through underspecification in grammars, such as binding elements of different languages by name [VC15].

A major challenge in engineering language product lines is finding a suitable granularity of language components. To achieve optimal reusability of language syntax, language components are finest-grained, meaning that each nonterminal would have to be handled individually as an independent language component. In MontiCore, this would require encapsulating each nonterminal into a separate grammar. However, the consequence is a scattering of languages and a significant overhead in managing the individual language components. Moreover, the additional complexity has to be handled in the feature model, which complicates product line engineering. Coarse-grained language components, on the other hand, yield less complex feature models and reduce the effort for managing language components. However, coarse-grained language components have the disadvantage that when only parts of the language components shall be reused, the remaining parts have to be handled during language composition as well. Coarse-grained language components are inefficient because more artifacts are involved in the composition compared to fine-grained components. This fosters ambiguities since more name definitions (such as those of nonterminals) are involved.

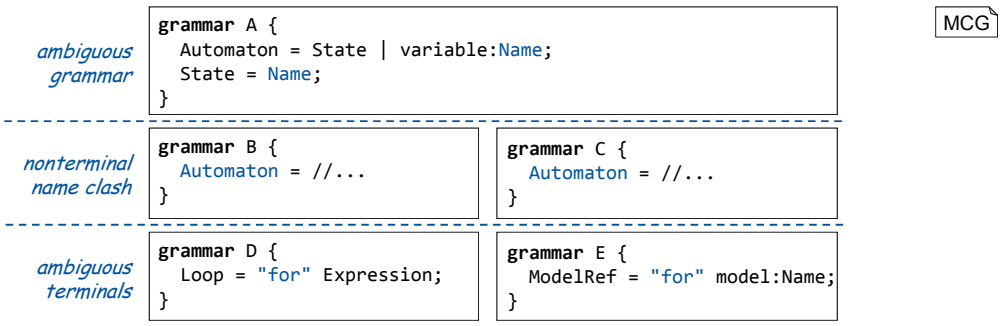


Figure 9.10: Potential ambiguities in grammar composition

A nonterminal that is not the start nonterminal of the grammar can be used as an extension to an extension point. This cuts off all parts of the language's concrete and abstract syntax that are not reachable as child elements of the abstract syntax induced by the new start nonterminal. Being able to use any nonterminal of a grammar as an extension increases the reusability of a grammar compared to only using the root nonterminal. However, it can cause parts of the language infrastructure to be included in a language variant, although these parts are not relevant there. This can be avoided by applying sophisticated checks in the language composer tool to identify such artifacts and exclude these in the generated language component model.

The generators composing the language components produce grammars, which extend all grammars of language components that are selected in the feature configuration. This mechanism produces correct results, enables reusing most parts of the language infrastructure, and requires generating only a few additional infrastructure constituents. However, grammars are usually leveraged as documentation of the syntax for language users. In the case of the generated grammar extending all reused grammars, this documentation is scattered across different artifacts. This can be solved by generating a single grammar including all nonterminals of the current grammar and transitively extended grammars. However, such a grammar should only be used for documentation and not for the generation of language-processing infrastructure due to performance reasons.

Language embedding in MontiCore is carried out via multiple language inheritance. Language embedding during derivation of language products from the product line produces a novel language that extends the languages and language components of selected features. Therefore, it is impossible that cyclic language inheritance is introduced through the selection of any combination of features.

The combination of grammars, in general, can yield ambiguous abstract and concrete syntax. We distinguish three forms of such ambiguities: (1) The grammar is ambiguous, as depicted in the top of Figure 9.10 if the same nonterminal is expected in more than one alternative. This ambiguity can lead to the generation of an ambiguous parser.

MontiCore generates parsers by employing ANTLR [Par13], which is able to detect several kinds of such ambiguities. Language engineers can add semantic predicates to the grammar to circumvent grammar ambiguities. (2) A nonterminal name clash (*cf.* Figure 9.10 center) can occur if a grammar reuses multiple other grammars, which define a nonterminal with the same name. This can only occur in language workbenches that have a flat namespace for nonterminals, which is the case for MontiCore. In case of a nonterminal name clash, the conflicting nonterminals have to be renamed. (3) There may be ambiguous terminals if the same concrete syntax is used in different grammar rules. An example of this is depicted in the bottom of Figure 9.10. In the engineering of language product lines, these ambiguities can be tested in the product line, which is more efficient than testing the property for each language individually during the derivation process. Product line managers can use exclude constraints in the feature model to indicate that two features use incompatible language components. Future work should investigate how these ambiguities can be detected efficiently and how constraints for their avoidance can be proposed automatically.

The employed feature diagram language enables the modularization of feature models into different artifacts. This fosters both the scalability of language product lines in general and the reuse of language product lines as part of other language product lines. Furthermore, the compositional approach based on language components enables better modularization and thus improved testability, maintainability, and reusability of the individual language components compared to employing a single 150% language. Nonetheless, large language product lines can rely on a quantity of language components that is hardly manageable. How this can be supported well is subject to further research.

Research and practice have produced numerous realizations for feature diagram languages. These often share commonalities but also have notational particularities. This makes feature diagrams themselves a good candidate for realizing a language product line that captures different optional language extensions of classical feature diagrams. Moreover, the LCPL language can be derived as a product of this language product line, *i.e.,* the LCPL language could be bootstrapped.

The property of conservative extension [HKR21] of language syntax during language composition increases the reusability of visitor-based language tooling. Without conservative extension, the language tooling produced for a language could only be applied to precisely this language. In language inheritance, for instance, conservative extension can be circumvented by overriding a nonterminal. If parts of a nonterminal body are removed, this may cause syntactical and semantical problems when analyses implemented against the original nonterminal are applied to the overridden nonterminal. With conservative extension, syntactical problems can be avoided for visitor-based tooling. For engineering language product lines, conservative extension of the language composition mechanism is an essential property as it enables reusing analysis implemented against a language feature for all language products that include this feature. Moreover, it en-

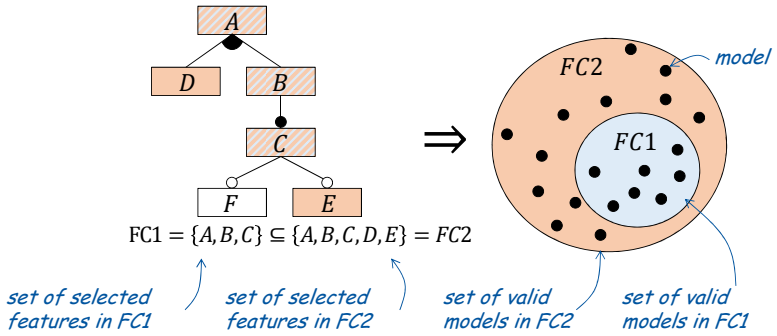


Figure 9.11: Conservative extension in language product lines and its effect for valid models

ables reusing tooling for language variants that comprise a superset of the set of currently employed language components.

The benefit of conservative extension in language product lines is depicted by example in Figure 9.11. If a language variant $FC2$ comprises a set of features that are a superset of another language variant $FC1$, then the employed language components also form a superset. With language composition mechanisms that realize conservative extension, the set of valid models in $FC2$ is a superset of valid models in $FC1$. The composition mechanism for language embedding in MontiCore realizes the conservative extension only with respect to the composed grammars and the parsers and visitors generated for the grammars. Context conditions invalidate models that the generated parser regards as correct. Therefore, adding more context conditions to a language reduces the number of valid models.

Language composition in MontiCore enables reusing context conditions implemented against the individual language components for the composed language. However, it may occur that a context condition checked against a certain nonterminal is not applied if the nonterminal is not reachable from the starting nonterminal of a composed language.

Furthermore, the composition of context conditions and any other analyses against the abstract syntax cannot guarantee to produce the results intended by the language engineers. For instance, an analysis for an automata language counts the states of an automaton model. This analysis can be applied to a language that extends the automaton language and introduces hierarchical states. Depending on the realization of language composition and the implementation of the analysis, nested states are taken into account for the result or not. This has to be taken into consideration by language engineers and language product line managers during language composition.

A type system for language components could improve reusing language components off-the-shelf when engineering a language product line. However, it is currently unclear into which dimension languages should be typed for this purpose. Languages could

be typed differently such as according to their purpose (like expression language, behavior language, statement language), according to their language paradigm (such as, being imperative or declarative), or according to their constituents (*e.g.,* code generators translate to the same target language, concrete syntax relies on the same tokens).

Language product lines can be developed top-down or bottom-up [KC16]. Applying the paradigm of top-down language product line engineering to our approach means that the feature model is modeled before the individual languages or language components are (re)used in the language product line. In bottom-up language product line engineering, the feature model is conceived after the individual language components have been implemented. Approaches for bottom-up engineering of product lines can be extended to extract product lines [Sch19] from a set of applications that are considered the products of the product line. This extraction can be applied to language product lines as well [VCPC13]. LCPL is not limited to either bottom-up or top-down engineering of product lines. The loose coupling between the feature model and language components supports both paradigms.

Presentational variability [CGR09] is variability between languages that only affects the concrete syntax. MontiCore grammars are an integrated definition for the abstract and concrete syntax of a language. Therefore, pure presentational variability cannot be realized for languages in our approach for realizing language product lines.

The language composition mechanisms introduced in this chapter include elements for concrete and abstract syntax, well-formedness checking, as well as additional infrastructure such as visitors. However, a holistic approach to software language product line should consider the realization of the semantics of the languages. Software languages assign behavior to models typically via interpretation of models or model-to-text transformations. Thus, composition operators for modular code generators [Bet16, CE00] or interpreters [BDV+16, VDKV00] have to be integrated into the language component definitions. The implementation presented in this thesis does not prescribe a composition mechanism for one of these, but an approach for integrating modular code generators into language product lines is described in [BEK+18a].

9.4 Related Work

Introducing variability in technical realizations of software languages is supported by language workbenches and similar language development tools to different extents and in various shapes [MAGD+16]. Reusing languages often relies on language composition [EvdSV+13]. The available forms of language composition differ in each language workbench and also depend on the language infrastructure constituents. Language workbenches with grammar-based syntax definitions (such as Rascal [vdS11], MontiCore [HLMSN+15b], Neverlang [VC15], Spoofax [KV10], or Xtext [Bet16]) realize different language composition techniques than, for instance, language workbenches with

metamodel-based syntax definitions (*e.g.,* EMF [SBPM09], GEMOC Studio [DCB+15], or MetaEdit+ [TK09]).

Several approaches support modular development of languages with language composition techniques for building specific languages but – to the best of our knowledge – do not support realizing dedicated language product lines for arranging language modules and their interrelations. These include mbeddr [VRSK12], ableC [KKCVW17], LISA [Mer13], CBS modules [Mos19], and the revisitor approach [LDC18]. These approaches are described in detail in Section 7.6. Some related approaches [HOKU15, KC16, KCO15, LDA13, VCPC13, WHT+09] use feature diagrams to restrict combinations of modular languages, *i.e.,* realize feature-oriented language product lines.

Neverlang [VC15] is a language workbench for textual DSLs that enables modular language development. A language module in Neverlang uses a grammar to define the language's syntax and a pipeline of evaluation phases to process models of the language. Evaluation phases can include well-formedness checking, type checking, and code generation [VCPC13]. Languages can define extension points by using placeholder nonterminals that are nonterminals not defined in a grammar. Through language composition, another grammar can provide implementations for placeholder nonterminals. Compared to this, MontiCore uses, interface or external nonterminals to denote underspecification of syntax. This reduces the risk of defining extension points unintendedly (*e.g.,* by misspelling a nonterminal) and enables making assumptions on the required abstract syntax through the right-hand side of interface nonterminals.

Neverlang has been integrated with FeatureIDE [MTS+17] to model feature diagrams and feature configurations and AiDE [KCO15] to derive a feature model from a family of interrelated language modules. With these extensions, Neverlang provides means to realize feature-oriented language product lines [FKC20]. The process of engineering product lines of languages in this approach begins with the decomposition of languages, which results in the generation of a feature model. The processes of deriving products from the product line and customizing language products follow concepts similar to those in LCPL. Another extension of Neverlang [VCPC13] uses the common variability language to realize language product lines.

Another approach [LDA13] for language product lines uses SDF for realizing modular languages with Spoofax [KV10] and FeatureHouse [AKL09] to model variability and perform derivation of languages. Language modules can be composed through superimposition, weaving, or inheritance between modules. In contrast to LCPL, this approach distinguishes two different dimensions of variability. One dimension is the variability of languages features in terms of concepts of the language, and the other dimension describes variability in the language infrastructure.

A related approach for language product lines based on MontiCore [BPRW20, Wor19] separates between language interfaces and implementations more strictly. A *family* model contains both a feature model describing the problem space variability of the product line and feature definitions, which connect features of the feature model with

their realization in terms of a `DSL component` model and bindings for this component. A DSL component model describes a language component in terms of grammar, well-formedness rules, and code generator. Contrary to the approach described in this thesis, other parts of language infrastructure, such as AST classes or symbol classes, are not explicated. Instead, a DSL component describes provided and required elements of the language interface. Furthermore, the concept of reusing languages through nonterminals in the language interface prohibits reuse through other nonterminals of the language. Thus, all provided and required extensions of the language must be foreseen by language engineers at the design time of the DSL component. While this approach fosters reusing language components as black boxes, it requires additional effort for reflecting the language interfaces in the product line explicitly. This additional effort targets language components that contain complex languages, as the interface hides language component internals. For language components defining a small number of nonterminals only, the benefit of explicating the language interface is questionable. However, the abstraction from MontiCore-specific composition techniques supports the application of this approach with other language workbenches. Contrary to the approach described in this thesis, this approach distinguishes optional and mandatory extension points of a language component. MontiCore distinguishes complete and incomplete languages based on the grammar, which is marked as component grammar in case the language is incomplete. If a language is technically complete but should be marked as incomplete, this approach uses mandatory extension points. The approach in this thesis relies on optional and mandatory features of the feature model instead, as the completeness of a language component can differ based on the context in which it is used.

Chapter 10

Application-Based Evaluation

This chapter summarizes several evaluations of different forms that have been performed using the individual results of this thesis. As stated in Chapter 3, the provided development steps of this thesis build upon each other, but each step can be used for different purposes than those detailed in this thesis. Hence, the steps are evaluated independently. Most steps are evaluated by applying them to different DSMLs. Figure 10.1 depicts an overview of the central development steps described in this thesis, which are evaluated in the following sections. Section 10.1 describes the evaluation of the STI, which is carried out by using the STI in various MontiCore languages. Section 10.2 explains the measurement of the parser performance of the developed JSON infrastructure. Section 10.3 describes the evaluation of the approach for loading and storing symbol tables, and Section 10.4 describes the evaluation of the language composition. The application of the feature diagram language family is explained in Section 10.6, and an outlook on the evaluation for the LCPL is given in Section 10.7.

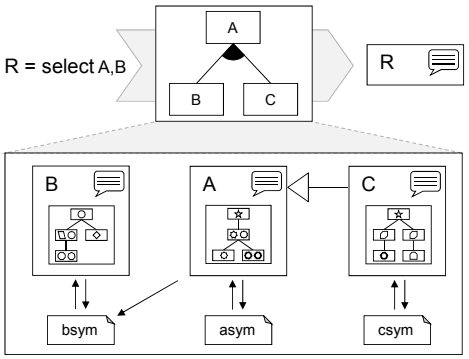


Figure 10.1: Development steps described in this thesis

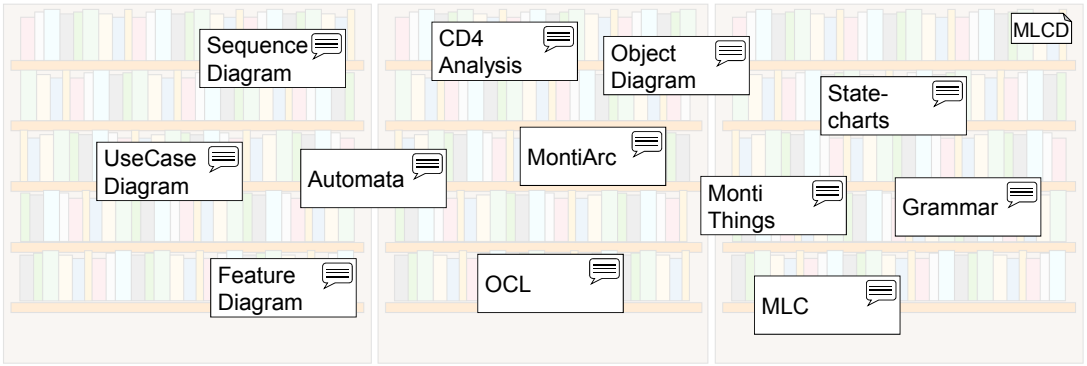


Figure 10.2: MontiCore languages that rely on the STI

10.1 Application of the STI

The STI has been applied to and constantly evaluated in the engineering of numerous DSMLs (*cf.* Figure 10.2), some of which are available open-source[1]. During the conception of the STI, the findings of the constant evaluation have been used as a basis for iterative improvements. For instance, the application to several languages unveiled that the symbol table instantiation is often customized at similar spots. This led to the introduction of the hook point methods in scope genitors as described in Section 4.3.8. Moreover, the STI has been used for building the MLC language presented in Chapter 7 of this thesis and for building the feature diagram languages presented in Chapter 8.

The symbol table infrastructure of the MLC language, as described in Section 7.4.1, defines a single symbol kind `MLCDefSymbol` that is introduced in the language's grammar. Attributes of the symbol kind are defined via a symbol rule. The generated symbol table infrastructure is customized for the symbol table instantiation in the scope genitor class by applying the TOP mechanism. The handwritten adjustments collect the regular expressions over artifacts and evaluate these against the actual file system. The results of the evaluations are set to the corresponding attributes in the `MLCDefSymbol`.

The feature diagram language enables a feature diagram model to import other feature diagram models. The symbol table infrastructure of the feature diagram language customizes the generated default infrastructure because it loads feature diagram models instead of the corresponding symbol tables for each imported feature diagram. This is realized by extending the language's scope genitor via the TOP mechanism (*cf.* Section 8.1). Furthermore, the scope genitor realizes the behavior that feature symbols are defined by the first occurrence of their name in a model.

The feature configuration and the partial feature configuration languages extend the generated scope genitors to add the specific behavior for import statements in their mod-

[1]Open-source MontiCore languages on GitHub: `https://github.com/MontiCore`

els. Each of these models must have at most a single import statement, and this statement must refer to the corresponding feature diagram (*cf.* Section 8.2). This is achieved by overriding the `createFromAST` method in the scope genitors of both languages. Furthermore, the `FeatureConfigurationSymbol` that is used in both languages contains attributes for the `FeatureDiagramSymbol` and the `FeatureSymbols` of selected features. These are realized through a symbol rule and, hence, can be generated from the grammar. The instantiation, however, has to be conceived by hand, which is achieved by overriding the `visit` methods for feature configurations and feature selections in the scope genitors of the languages.

Lessons learned: The application of the STI to numerous languages led to the iterative improvements of the STI. Moreover, the application demonstrates the ability of the STI to support the engineering of symbol tables for a variety of modeling languages. As symbol tables are used for multiple purposes, the requirements and quality criteria for symbol tables in different modeling languages can vary. Despite that, the generated symbol table infrastructures form a solid base to engineer symbol kinds, scopes, and most parts of the symbol resolution. Nevertheless, our experiences are that most languages require handwritten adjustments to the generated infrastructure. The STI consequently generates most parts of the symbol table infrastructure of a language. Hence, with the TOP mechanism that can be applied to all Java classes and interfaces of the generated symbol table infrastructures, such customizations can be achieved with little effort.

10.2 Performance of Json Infrastructure

A key motivation for engineering a JSON parser infrastructure by hand is the speed of the parser because loading symbol tables should be efficient. Generating language infrastructure based on a JSON grammar and processing this with MontiCore or a different language workbench would result in a less efficient parser. Language workbenches for textual languages are typically capable of generating parsers that parse syntax described by any context-free grammar. However, these generated parsers are rarely optimized with regard to parsing speed. As the syntax of the JSON language has a rather simplistic structure, a handwritten parser and lexer can be optimized for improving the parsing speed. We concluded this by measuring the parsing speed of different JSON parsers for comparison. This comparison of parsing speeds includes

- the parser of Gson [www20a] as a related, commonly used optimized parser,
- the presented handwritten JSON parser (in Table 10.1 referred to as “MontiCore handwritten”),
- the parser generated from a MontiCore grammar that optimizes parsing speed by avoiding splitting token definitions into token fragments (“Inline literal grammar”),

Parser	Parsing Time
Gson	1592 ms
MontiCore handwritten	1601 ms
Inline literal grammar	9718 ms
RFC grammar	11248 ms
Modular JSON grammar	11659 ms
Flat grammar	22164 ms

Table 10.1: Comparison of parsing speed of different JSON parsers ordered by decreasing parsing speed.

- the parser generated from a MontiCore grammar that directly realizes the grammar in the RFC standard of JSON ("RFC grammar"),
- the parser generated from a MontiCore grammar that is engineered for being well extensible ("Modular JSON grammar"), and
- the parser generated from a MontiCore grammar that accepts many malformed JSON documents and postpones more of the well-formedness to context conditions ("Flat grammar").

We measured the time as the mean parsing speed of 100 iterations of parsing two generated JSON files, where one of these is a tree with a depth of 7 and a branching factor of 7, and the other is a tree with depth 3 and a branching factor of 10. The results of the comparison are depicted in Table 10.1. The measurements include only the pure parsing and lexing that produce an instance of an abstract syntax model of JSON. The instantiation of the serialized object structure from the serialized String is not taken into account. The measurements were carried out on a machine with a 4 x 2.6GHz CPU and 16GB RAM.

Lessons learned: Although the measurements have a significant variance, among other things, due to garbage collection, it is visible that optimized JSON parsers (Google Gson and MontiCore handwritten) are significantly faster than the remaining investigated parsers that were generated.

10.3 Application of Loading and Storing Symbol Tables

Serialization for symbol tables has been established for a larger number of MontiCore languages, including those that are available open-source. Similar to the symbol table infrastructure in general, the mechanisms for loading and storing symbol tables have been constantly evaluated by employing the approaches to these languages. Therefore,

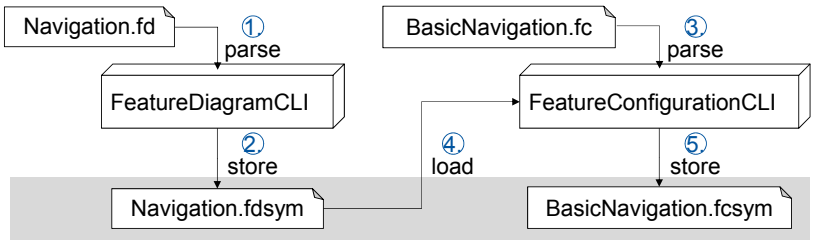


Figure 10.3: Example of loading and storing symbols in the feature diagram language family

the generated serialization and deserialization infrastructure is utilized, sometimes in a customized form, in most of the open-source MontiCore languages. Since the speed of loading and storing symbol tables largely depends on the complexity of the language and its serialization strategies as well as on the complexity of the models, the results of a speed measurement for loading and storing symbol tables could hardly be generalized. Hence, it is a deliberate decision not to measure this speed. Loading and storing of symbol tables is also used in the MLC language presented in Chapter 7 of this thesis and for building the feature diagram languages presented in Chapter 8 of this thesis.

Although the tooling of the feature diagram language loads models instead of symbol tables of models, the loading and storing of feature diagram models is implemented as part of the language infrastructure. It is used for the case that a feature configuration model is processed without executing feature analyses. In this case, the feature configuration can check the conformance to the linked feature diagram model by loading the symbol table. The tools of the two feature configuration languages are able to store symbol tables of feature configuration models. However, these files currently have the sole purpose of enabling developers to investigate the symbol tables of models. An example of this is depicted in Figure 10.3, where first, the feature diagram tool processes a feature model `Navigation` and stores the corresponding symbol file. Afterward, the feature configuration tool processes the feature configuration `BasicNavigation` that refers to the feature model `Navigation`. To check that the feature configuration refers to the correct feature diagram, the feature configuration tool loads the symbol file of `Navigation` and resolves for the name of the feature diagram symbol and the names of the feature symbols. If all names exist at the expected locations, the tool stores the symbol file for the feature configuration.

The MLC language relies on stored symbol tables to increase the performance of processing MLC models. If an MLC model refers to another MLC model, loading the corresponding symbol table file avoids re-evaluation of the regular expressions in MLC models. The serialization strategy for `MLCDefSymbols` is customized to serialize the map of exported files. This is explained in more detail in Section 7.4.1.

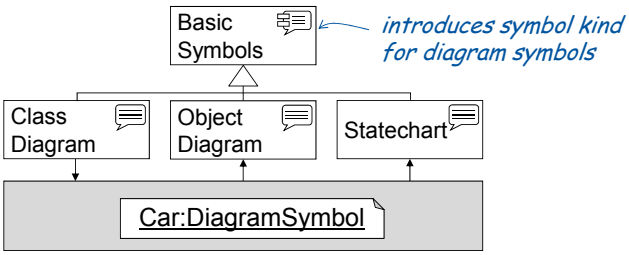


Figure 10.4: Example of language composition through shared grammar

Lessons learned: The approach for loading and storing symbol tables generates default serialization strategies for custom symbol and scope attributes that are of basic data types only. Other serialization strategies have to be conceived manually and integrated into the corresponding DeSers. Our experiences have shown that most symbol and scope attributes that the languages use are either Boolean flags, String attributes, or related to type expressions. For the former two, MontiCore generates the serialization strategies. For type expressions, the runtime environment of MontiCore provides DeSer classes for the serialization strategies of different kinds of `SymTypeExpressions` (*cf.* Section 4.1.7). Hence, conceiving serialization strategies by hand is rarely necessary.

The loading and storing of symbol tables integrates into the process of processing models seamlessly. In most cases, loading symbol tables of foreign models instead of parsing and instantiating the symbol table of models anew suffices to realize type checks and, hence, to check the correctness of a model. However, for sophisticated analyses and code generation, it is sometimes necessary to have the AST of foreign models available as well. For instance, the feature diagram language requires the AST of imported feature diagrams to carry out feature analyses.

10.4 Application of Language Composition via Symbol Tables

MontiCore languages rarely exist in isolation. Most languages reuse language components from the MontiCore language component library. Hence, most languages rely on forms of language composition, as described in Chapter 6.

The language aggregation through a shared grammar (*cf.* Section 6.4.1) is used to realize MontiCore's type checking framework [HKR21]. The nonterminals for type definitions are located in a grammar that can be extended by languages that use the type system. Similarly, language engineers can conceive further grammars that introduce symbol-defining nonterminals that languages can use via language extension. For example, UML [Rum16, Rum17] and SysML [www21f] comprise heterogeneous languages for specific diagrammatical models such as class diagrams, activity diagrams, or block-definition diagrams. Applications are typically modeled through heterogeneous models,

where different aspects of the application are modeled with suitable modeling languages. In this, the models usually refer to elements of other diagrams. In such scenarios, a common grammar can be used to introduce a symbol-defining nonterminal `Diagram`. Each individual language that extends the common grammar can refer to other diagrams with `Name@Diagram`. This further enables to underspecify the concrete kind of diagram and fosters extensibility with new languages. An example of the diagram symbol exchanged between different languages is depicted in Figure 10.4. In this, the diagram symbol can be either exchanged as a Java object or via symbol table persistence. Optionally, the individual languages can define language-specific diagram symbol kinds that extend the general symbol kind `DiagramSymbol`. This supports distinguishing different forms of diagrams by resolving for language-specific diagram kinds and, at the same time, resolving for any suitable diagram regardless of the concrete kind of diagram by resolving for symbols of the more general kind `DiagramSymbol`.

The feature diagram language presented in Chapter 8 as part of the feature diagram language family reuses three languages from the MontiCore language component library that again reuse further language components (*cf.* Figure 8.1). This demonstrates the reusability of language components through language inheritance. The feature configuration languages inherit from the feature diagram language, inter alia, to enable their scopes to resolve for symbol kinds defined in the feature diagram language. Moreover, the different forms of language composition presented in Chapter 6 of this thesis are utilized to integrate the feature diagram language with other languages through the mechanism described in Section 8.4.

Lessons learned: The combination of kind-typed symbol tables and symbol table persistence enables novel techniques of conceiving language composition. For instance, in the STI, there are different ways to achieve language aggregation. Instead, in the SMI [MSN17], language aggregation is always realized through a `ModelingLanguage-Family` and `AdaptedResolvingFilters`.

The novel techniques for language composition enable adjusting the extensibility of language aggregations flexibly. By reconfiguring the global scope's DeSers and considered symbol file extensions, novel languages that provide previously unknown symbol kinds can be integrated. On the other hand, symbol adapters can translate between individual symbol kinds to precisely control which symbol kinds should be considered in the aggregation.

In the STI, the symbol kinds are encoded into the names of `resolve` methods. The SMI [MSN17], on the other hand, uses the symbol kinds as an argument of the resolution. Handling foreign symbol kinds, hence, requires more effort in the STI because the global scope cannot handle unknown kinds in its `resolve` methods. However, this encourages language engineers to explicitly consider the ability of a language to participate in language aggregation.

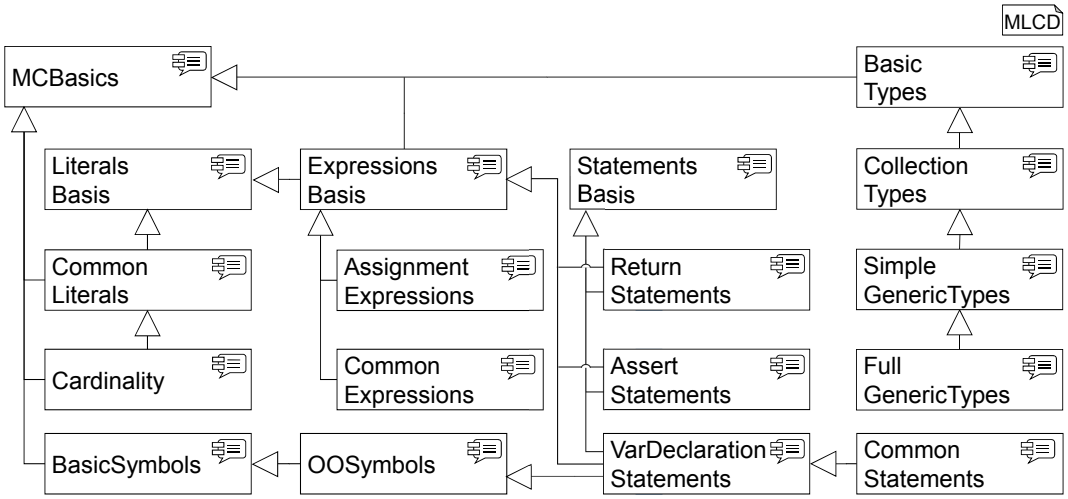


Figure 10.5: Excerpt of MontiCore's language component library

10.5 Application of MontiCore Language Components

Most MontiCore languages rely on languages of the MontiCore language component library by extending one or more of its grammars. With the MLC language as presented in Chapter 7, a language component can indicate that it is allowed to use one or more other languages by referring to their MLC models. To this end, creating MLC models for the language components of the MontiCore language component library is a valuable base for creating MLC models of other, more complex languages.

Since all language components of the library are located within a single build tool module, introducing MLC models for each of the language components helps to untangle the relations between the artifacts of the build tool module. Hence, the language component library is suited well for evaluating the MLC language and tools.

We created MLC models for all 39 language components of the MontiCore language component library. Most language components follow the same schema: they comprise a grammar, all Java files that are generated from the grammar, and some Java classes that are handwritten extensions to the generated classes and result from applying the TOP mechanism. Some language components, such as the language components for type expressions, share several Java classes that realize the type check. Furthermore, some language components are documented in a markdown file located next to the grammar, while for others, the documentation is contained in common markdown files. By applying the MLC files to the language component library, several unintended relationships between language components have been identified. For instance, in a previous version of the language component `BasicSymbols`, some handwritten AST classes used classes of the `OOSymbols` language. Because the `OOSymbols` language component extends the

`BasicSymbols`, this relationship is undesired and has been removed. An excerpt of the language components in the language component library and some of their interrelations are depicted in Figure 10.5.

Apart from the language components of the language component library in MontiCore, the MLC tool has been integrated into different DSMLs, such as into the DSMLs of the feature diagram language family. We modeled MLC models for each of the three language components of the language family. The MLC models assure that only the language components depicted in Figure 8.1 are allowed to be used. As the feature analysis tooling cannot be associated directly with the feature diagram language or a feature configuration language, it is not included in any of the language components. Instead, we regard it as a separate software component.

Lessons learned: Engineering MLC models is a manual activity. For some language components, creating MLC models is straightforward. For other languages, however, creating MLC models requires effort to understand the interrelations of a language component's internal artifacts and their relations with the artifacts in their environment to include or exclude these in the MLC model.

Most MontiCore languages use the same build tool configuration. Hence, MLC models contain a part of boilerplate include statements for the grammar, the generated code, and the handwritten code that extends the generated code through the TOP mechanism. Similarly, each MLC model indicates that the language component is allowed to use the same artifacts, such as the MontiCore runtime environment.

10.6 Application of the Feature Diagram Language Family

For assuring the intended functionality of the language tools, the languages of the feature diagram language family are tested with unit tests that utilize different valid and invalid models of the individual languages. The languages form the basis for realizing the LCPL language and tooling described in Chapter 9 of this thesis. At the same time, the LCPL language evaluates the extensibility of the feature diagram language to customize it for a particular application.

To demonstrate a different form of application, we integrated the feature diagram language with a class diagram language to create 150% class diagram models that we refer to as *product line class diagram* (PLCD) models. The syntax of this language is realized through multiple inheritance of a novel grammar from the feature diagram grammar and the class diagram grammar. The novel grammar introduces a new start rule `PLCD` for the language[2]. A PLCD model has a name and a body that contains `FDElements` reused from the feature diagram language as well as `FeatureBlocks`.

`FeatureBlocks` are defined anew by the language and associate a feature name with

[2]Hence, the form of composition is language inheritance and not language embedding

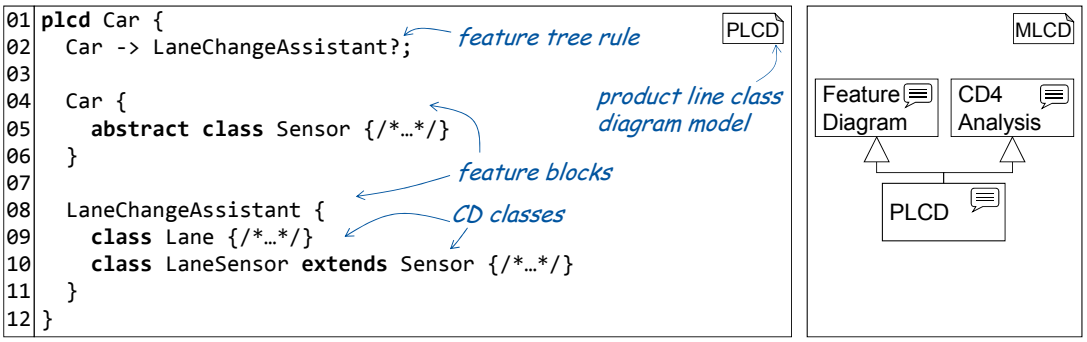


Figure 10.6: The PLCD language reuses the feature diagram language

a set of `CDClasses` and `CDAssociations` reused from the class diagram language. The left side of Figure 10.6 depicts an example model of the PLCD language, and the right side of the figure displays the relations between the involved language components. We implemented tooling based on the generated visitor infrastructure of the language that transforms any PLCD model into a feature diagram model. The resulting feature diagrams enable reusing the feature analysis tooling from the feature diagram language family (*cf.* Section 8.3). The visitor infrastructure can further be used to implement a transformation that creates a class diagram model based on a given PLCD model and feature configuration. This class diagram contains only classes and associations that belong to features selected in the given feature configuration. To this end, this tool derives a product from the product line modeled within the PLCD model. Such forms of language composition can be achieved analogously with similar modeling languages.

Lessons learned: As feature modeling is a central activity in many software product line applications, there are numerous feature modeling tools. Despite that, we created a new feature diagram language in MontiCore. The languages of the feature diagram language family can be integrated with other languages through MontiCore's means of language composition with little effort. We demonstrated this by applying the feature diagram language in the context of PLCDs and the LCPL language. Furthermore, the languages of the feature diagram language family can be extended with additional feature modeling capabilities that are taken into account by feature analyses. For these additional capabilities, analyses and code generators can be integrated with the existing tooling. However, industrial-scale feature modeling tools are more optimized regarding the efficiency and scalability of feature analyses than the current implementation of the feature analyses tools presented in this thesis.

10.7 Evaluation of the LCPL

The concepts behind the LCPL that is described in Chapter 9 have been conceived, evaluated, and updated continuously over the course of several years. This is reflected in numerous publications [BEK+18a, BEK+18b, BEK+19, BPRW20, BW21]. In addition, the concepts have been applied to several example product lines, including an architecture description language that is customizable for different application domains. However, a proper evaluation with large-scale language product lines has yet to be conducted.

Chapter 11

Conclusion

This thesis presents different means that support language engineering in the large. The results of this thesis serve the purpose of being able to engineer high-quality DSMLs tailored to specific applications with little effort. Such languages typically use domain vocabulary and, hence, reduce the gap between problem and implementation [FR07]. This fosters domain engineers to implement software without the accidental complexity of learning complex programming languages. Symbol tables enable efficient composition of modular languages, and language components promote language reuse through precise identification of language artifacts and analyses against languages. Language product lines support the structured reuse of languages in product lines. This chapter summarizes the main results of the thesis in Section 11.1 and describes how these results contribute answers to the research questions introduced in Chapter 1. Section 11.2 concludes the thesis with an outlook on directions for further research.

11.1 Summary

The thesis has developed approaches that enable engineering modular language components with MontiCore and composing these language components, inter alia, via their symbol tables. This fosters language reuse and is the basis for realizing language product lines of languages. The following list summarizes the main results and explains how these answer the partial research questions:

Typed Symbol Table Infrastructures Chapter 4 presents the typed symbol table infrastructure and describes how large parts of it can be generated. The STI has been successfully applied to a larger number of MontiCore language, including the MontiCore grammar language as well as the MLC language and the feature diagram language family presented in this thesis. The research question **RQ1** ("Can a typed symbol table infrastructure support language composition?") can be answered positively because Chapter 8 shows several forms of language composition for the feature diagram language family that rely on the STI and can be employed to integrate the FD language with other languages. In general, Chapter 6 describes how to achieve language composition with the

STI through typed symbol tables and, hence, answers the research question **RQ3** ("How to compose languages via typed symbol tables?").

Persistence of Symbol Tables An approach for realizing load and store operations for symbol tables is described in Chapter 5. The approach enables reusing processed models of a language and fosters reuse of models across the borders of languages through different forms of language composition that rely on stored symbol tables, as presented in Chapter 6. Furthermore, a tool for importing class files, *i.e.,* processed models of Java, exemplary demonstrates how processed models from non-MontiCore languages can be imported into MontiCore languages. Thus, the persistence of symbol tables answers the research question **RQ2** ("How can we reuse processed models of a language and of foreign languages?").

Language Components Chapter 7 introduces the notion of language components in MontiCore by means of the artifacts that realize a language component. The MLC language enables modeling language components, and MLC tools can be employed to identify the artifacts that constitute a language component. Furthermore, language engineers can identify undesired relations between language components that complicate language reusability with the MLC tool. The definition of language components realized through the MLC language and tools answers the research question **RQ4** ("What constitutes a reusable language component?").

Language Product Lines As described in Chapter 9, language product lines in MontiCore can be realized with the MLC language introduced in Section 7.4.1 and the feature diagram language family explained in Chapter 8. The approach for language product lines in MontiCore restricts undesired combinations of language components via feature models and fosters reusability of language compositions among families of similar languages. This answers the research question **RQ5** ("How can a family/product line describing similar languages be modeled?").

All these results contribute to the main research question of the thesis, *i.e.,* present approaches for the composition of DSMLs via their symbol tables using reusable language components. These language components can be arranged in language product lines. With the language composition techniques, languages can be derived from the product line in a semi-automated process.

11.2 Potential for Future Work

This thesis has answered the research questions introduced in Chapter 1 in the technological space of MontiCore. In general, engineering languages in the large is a complex endeavor with a number of further open research questions to solve. Despite this, the presented approaches can be evolved and extended into various dimensions.

Other forms of Symbol Table Persistence The presented approach for symbol table persistence stores symbols and scopes and re-establishes the object structure from stored artifact scopes. As presented in Chapter 5, an alternative for this is to calculate qualified names for each persisted symbol and store only symbols instead of both symbols and scopes. Realizing an approach for key-value storage of symbols enables exploring the advantages and disadvantages beyond the points discussed in Chapter 5 and enables a comparison of both approaches in terms of the performances for symbol resolution and in terms of suitability for language composition.

Synthesizing Language Component Models The MLC language presented in Chapter 7 enables modeling regular expressions for artifact sets that realize/constitute a language. However, the configuration of the MontiCore generator already determines the location of all artifacts that are generated from a language's grammar. Furthermore, all MontiCore languages require access to the MontiCore runtime environment. The identification of such artifacts in a manual process, hence, is not necessary. Instead, parts of an MLC model can be synthesized with a code generator. Future work should investigate to generate parts of MLC models, *e.g.,* via generating adjustable proposals for MLC models from a MontiCore grammar. The generated proposals can be customizable by making the TOP mechanism available for the MLC language via inheritance between MLC models or by adding detailed import mechanisms per artifact set of an MLC model.

Extending the MLC Language The MLC language presented in Chapter 7 enables specifying the artifacts of a language component and the artifacts that a language component is allowed to use. Currently, the reuse of MLC models is limited to indicating that a language component may use all artifacts of another language component by indicating its MLC model. This is due to the fact that the MLC language distinguishes only between *own* and *allowed* artifacts. If a language is allowed to use another language, all own artifacts of the other language become allowed artifacts of the language. By distinguishing own artifacts that are exported for other languages and own artifacts that are only visible locally, the MLC language could allow finer-grained specifications of artifact sets. This could be achieved with a syntax similar to visibility modifiers from programming languages such as Java. Furthermore, allowed artifacts of a language could be split up into artifacts that are promoted for transitive use and artifacts that are not promoted. However, such language extensions make the MLC language more complex to understand and use. Future work should evaluate the benefits and the additional complexity of such language extensions for language users and for evaluating artifact sets with MLC tools.

Enhanced Language Component Tools Currently, language-specific artifact extractors of the MLC tooling presented in Chapter 7 identify sets of artifacts that an artifact uses. For instance, an artifact extractor for the Java language identifies all artifacts that a Java artifact uses. Another artifact extractor for the MontiCore grammar language identifies

artifacts that a grammar artifact uses. The extractors can be improved to consider relationships across artifacts of different languages. For instance, code generator template artifacts are typically used from Java artifacts. Suitable cross-language extractors can consider such relations as well if such inter-language relations are modeled in an artifact model [BGRW18].

Delta-Oriented Language Product Lines The approach for language product lines presented in Chapter 9 uses modular language components that are composed to realize language variants. In this, the features of the feature diagram refer to language components. Future work could investigate a different approach to realize language product lines by applying deltas [SBB+10] on the level of languages. This approach would use a MontiCore language as the basis of the product line and root of the feature diagram. Each further feature would constitute delta operations that are applied to the language if the feature is selected. This could be suitable for language product lines in which the variants differ only in a few nonterminals or context conditions. However, the application of negative delta operations breaks with the property of conservative extension and, hence, endangers the reusability of language tools. From a technical viewpoint, the language describing deltas for MontiCore languages could be derived from the grammar language [HHK+15].

Outreach to Generation Composition The presented approach for conceiving language product lines does not explicitly handle code generators as part of language components. However, the process of modular code generation requires some forms of composition [BW21]. There are several approaches for realizing modular code generators with MontiCore. Some approaches are suitable for realizing language product lines but are applicable with restrictions on the code generators only [BEK+18b, BW21, BPRW20]. Other approaches require information about foreign code generators to compose these [GMR+16] and, hence, are not suitable well to be applied in language composition as part of language product lines. A more general approach for generator composition is yet to be conceived.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines:Concepts and Implementation*. Springer, 2013.

[AC04] Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange*, pages 67–72, 2004.

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.

[AK03] Colin Atkinson and Thomas Kuhne. Model-Driven Development: A Metamodeling Foundation. *IEEE software*, 20(5):36–41, 2003.

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In *2009 IEEE 31st International Conference on Software Engineering*, pages 221–231. IEEE, 2009.

[ALSU07] Alfred V. Aho, Monika S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2007.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[BDV+16] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 84–89. ACM, 2016.

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design.

Journal of Object Technology, 19(3):3:1–16, October 2020. Special Issue dedicated to Martin Gogolla on his 65th Birthday.

[BEK+18a] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*, pages 75–82. ACM, January 2018.

[BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

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

[Bet16] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[Beu12] Danilo Beuche. Modeling and Building Software Product Lines with pure::variants. In *Proceedings of the 16th International Software Product Line Conference*, volume 2, pages 255–255, 2012.

[BGM10] Barrett R Bryant, Jeff Gray, and Marjan Mernik. Domain-Specific Software Engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 65–68, 2010.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. Part of the Grand Challenges in Modeling (GRAND'17) Workshop, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language.

In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BMSN21] Arvid Butting and Pedram Mir Seyed Nazari. Symbol Management Infrastructure. In Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe, editors, *MontiCore Language Workbench and Library Handbook: Edition 2021*, Aachener Informatik-Berichte, Software Engineering, chapter 9, pages 155–210. Shaker Verlag, 2021.

[BPRW20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35—-46. ACM, October 2020.

[Bro87] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

[BRW16] Arvid Butting, Bernhard Rumpe, and Andreas Wortmann. Embedding Component Behavior DSLs into the MontiArcAutomaton ADL. In *Globalization of Modeling Languages Workshop (GEMOC'16)*, volume 1731 of *CEUR Workshop Proceedings*, October 2016.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer Science & Business Media, 2001.

[BSL+13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, 2013.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 years later: A Literature Review. *Information systems*, 35(6):615–636, 2010.

[BvdBH+15] 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. Modular Language Implementation in Rascal– Experience Report. *Science of Computer Programming*, 114:7–19, 2015.

[BW21] Arvid Butting and Andreas Wortmann. *Language Engineering for Heterogeneous Collaborative Embedded Systems*, pages 239–253. Springer, January 2021.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CE99] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and Generative Programming. *ACM SIGSOFT Software Engineering Notes*, 24(6):2–19, 1999.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison

of Variability Modeling Approaches. In *Proceedings of the 6th international Workshop on Variability Modeling of Software-Intensive Systems*, pages 173–182, 2012.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 45(3):1–17, 2003.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CKM+18] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018.

[Cla99] James Clark. XSL Transformations (XSLT) Version 1.0. W3C recommendation, W3C, November 1999. https://www.w3.org/TR/1999/REC-xslt-19991116.

[CN02] Paul Clement and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[DC99] Steven DeRose and James Clark. XML Path Language (XPath) Version 1.0. W3C recommendation, W3C, November 1999. https://www.w3.org/TR/1999/REC-xpath-19991116/.

[DCB+15] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*, Pittsburgh, United States, 2015.

[dJV02] Merijn de Jonge and Joost Visser. Grammars as Feature Diagrams. In *ICSR7 Workshop on Generative Programming*, pages 23–24, 2002.

[DMW17] Thomas Degueule, Tanja Mayerhofer, and Andreas Wortmann. Engineering a ROVER Language in GEMOC STUDIO & MONTICORE: A Comparison of Language Reuse Support. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34. IEEE, 2013.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. <<UML>> '98: Beyond the Notation*, volume 1618 of *LNCS*, pages 336–348. Springer, Germany, 1999.

[EGR12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language Composition Untangled. In *Twelfth Workshop on Language Descriptions, Tools, and Applications*, pages 1–8, 2012.

[EKR+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*, pages 17–18. ACM, 2011.

[EvdSV+13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In *Software Language Engineering*. Springer International Publishing, 2013.

[FGLP10] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical Language Analysis in Software Linguistics. In *International Conference on Software Language Engineering*, pages 316–326. Springer, 2010.

[FK05] William Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[FKC20] Luca Favalli, Thomas Kühn, and Walter Cazzola. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line*, pages 1–11, 2020.

[Fla05] David Flanagan. *Java in a Nutshell*. O'Reilly Media, Inc., 2005.

[For22] Henry Ford. *My Life and Work*. Cosimo, Inc., 1922.

[For13] Charles Forsythe. *Instant FreeMarker Starter*. Packt Publishing, 2013.

[Fow05] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages, 2005.

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, pages 37–54, May 2007.

[FRF+10] Daniela Florescu, Jonathan Robie, Mary Fernandez, Don Chamberlin, Jerome Simeon, and Scott Boag. XQuery 1.0: An XML Query Language (Second Edition). W3C recommendation, W3C, December 2010. https://www.w3.org/TR/2010/REC-xquery-20101214/.

[Fri06] Jeffrey EF Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[FT96] William Frakes and Carol Terry. Software Reuse: Metrics and Models. *ACM Computing Surveys (CSUR)*, 28(2):415–435, 1996.

[GGdL+19] Antonio Garmendia, Esther Guerra, Juan de Lara, Antonio García-Domínguez, and Dimitris Kolovos. Scaling-Up Domain-Specific Modelling Languages through Modularity Services. *Information and Software Technology*, 115:97–118, 2019.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GJR79] Susan L Graham, William N Joy, and Olivier Roubine. Hashed Symbol Tables for Languages with Explicit Scope Control. In *Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, pages 50–57, 1979.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, volume 254 of *LNI*, pages 141–156. Bonner Köllen Verlag, March 2016.

[GNT+07] Jeff Gray, Sandeep Neema, Juha-Pekka Tolvanen, Aniruddha S Gokhale, Steven Kelly, and Jonathan Sprinkle. Domain-Specific Modeling. In *Handbook of Dynamic System Modeling*, pages 7–1 – 7–20, 2007.

[GP15] Terje Gjøsæter and Andreas Prinz. LanguageLab-A Meta-modelling Environment. In *International SDL Forum*, pages 91–105. Springer, 2015.

[Hef14] David R. Heffelfinger. *Java EE 7 with GlassFish 4 Application Server*. Packt Publishing Ltd, 2014.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener

Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLMSN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science*, pages 45–66. Springer, 2015.

[HLMSN+15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. *Acm Sigact News*, 32(1):60–65, 2001.

[Hoa73] Charles A. R. Hoare. Hints on Programming Language Design. Technical report, Stanford University, 1973.

[Hoa09] Charles A. R. Hoare. Null References: The Billion Dollar Mistake. *Presentation at QCon London*, 298, 2009.

[HOKU15] C. Huang, A. Osaka, Y. Kamei, and N. Ubayashi. Automated DSL Construction Based On Software Product Lines. In *3rd International Conference on Model-Driven Engineering and Software Development*, pages 1–8, 2015.

[Höl18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[KC16] Thomas Kühn and Walter Cazzola. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 50–59. ACM, 2016.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University, 1990.

[KCO15] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Software Product Line Conference*, pages 71–80. ACM, 2015.

[KKCVW17] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proc. ACM Program. Lang.*, 1(OOPSLA):98:1–98:29, October 2017.

[KKWV12] Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative Name Binding and Scope Rules. In *International Conference on Software Language Engineering*, pages 311–331. Springer, 2012.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels.* Pearson Education, 2008.

[KMC12] Tomaž Kosar, Marjan Mernik, and Jeffrey C Carver. Program Comprehension of Domain-Specific and General-Purpose Languages: Comparison Using a Family of Experiments. *Empirical software engineering*, 17(3):276–304, 2012.

[KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* John Wiley & Sons, 2008.

[KV10] Lennart CL 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 (OOPSLA 2010)*, pages 444–463. ACM, 2010.

[KVW13] Ted Kaminski and Eric Van Wyk. Creating and Using Domain-Specific Language Features. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 18–21, 2013.

[LDA13] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented Language Families: A Case Study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

[LDC+17] Manuel Leduc, Thomas Degueule, Benoit Combemale, Tijs van der Storm, and Olivier Barais. Revisiting Visitors for Modular Extension of Executable DSMLs. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, sep 2017.

[LDC18] Manuel Leduc, Thomas Degueule, and Benoit Combemale. Modular Language Composition for the Masses. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, pages 47–59. ACM, 2018.

[LDWC19] Manuel Leduc, Thomas Degueule, Eric Van Wyk, and Benoit Combemale. The Software Language Extension Problem. *Software and Systems Modeling*, dec 2019.

[Led19] Manuel Leduc. *On Modularity and Performance of External Domain-Specific Language Implementations*. PhD thesis, University of Rennes 1, France, 2019.

[LW94] Barbara H Liskov and Jeannette M Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[Mac99] Bruce J. MacLennan. *Principles of Programming Languages Design, Evaluation, and Implementation*. Oxford University Press, 1999.

[MAGD+16] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures*, 46:206–235, 2016.

[Mar17] Tom Marrs. *JSON at Work: Practical Data Integration for the Web*. O'Reilly, 2017.

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. SPLOT: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 761–762, 2009.

[McC98] Glen McCluskey. Documentation of the Java Reflection API. `https://www.oracle.com/technical-resources/articles/java/javareflection.html`, 1998. [Online; accessed 24-August-2020].

[MCHB11] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 82–89, 2011.

[Mer13] Marjan Mernik. An Object-Oriented Approach to Language Compositions for Software Language Engineering. *Journal of Systems and Software*, 86(9), 2013.

[MGVB16] Sadaf Mustafiz, Cláudio Gomes, Hans Vangheluwe, and Bruno Barroca. Modular Design of Hybrid Languages by Explicit Modeling of Semantic Adaptation. In *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*, pages 1–8. IEEE, 2016.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. When And How to Develop Domain-Specific Languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[Mos19] Peter D Mosses. Software Meta-Language Engineering and CBS. *Journal of Computer Languages*, 50:39–48, 2019.

[MRR11] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[MT00] Nenad Medvidovic and Richard N Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[Mus14] Benjamin Muschko. *Gradle in Action*. Manning, 2014.

[MVG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[MVM10] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Apache Maven*. Alpha Press, 2010.

[NES17] Michael Nieke, Gil Engel, and Christoph Seidl. DarwinSPL: An Iintegrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 92–99, 2017.

[NPRI09] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of JSON and XML Data Interchange Formats: A Case Study. *Caine*, 9:157–162, 2009.

[NR68] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1968.

[NSB+07] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[PH04] Frits K. Pil and Matthias Holweg. Linking Product Variety to Order-Fulfillment Strategies. *Interfaces*, 34(5):394–403, 2004.

[PSMB+06] Jean Paoli, Michael Sperberg-McQueen, Tim Bray, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). W3C recommendation, W3C, August 2006. https://www.w3.org/TR/2006/REC-xml11-20060816/.

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending Feature Diagrams with UML Multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, volume 23, pages 1–7, 2002.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, volume 215 of *LNI*, pages 155–170, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *International Conference on Software Product Lines*, pages 77–91. Springer, 2010.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Sch19] Christoph Schulze. *Agile Software-Produktlinienentwicklung im Kontext heterogener Projektlandschaften*. Aachener Informatik-Berichte, Software Engineering, Band 40. Shaker Verlag, May 2019.

[SM12] Audie Sumaray and S. Kami Makki. A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, pages 1–6, 2012.

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In Patrick Heymans, Krzysztof Czarnecki, and Ulrich W. Eisenecker, editors, *Proceedings of the 5th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'11)*, pages 119–126. ACM, 2011.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[SVGB05] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and experience*, 35(8):705–754, 2005.

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014.

[TK09] Juha-Pekka Tolvanen and Steven Kelly. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70–85, 2014.

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *15th International Software Product Line Conference*, pages 191–200. IEEE, 2011.

[VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A Framework for Feature-oriented Language Development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[VCCA14] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the 18th International Software Product Line Conference*, pages 167–176. ACM, 2014.

[VCPC13] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. Variability Support in Domain-Specific Language Development. In M. Erwig, R.F. Paige, and E. Van Wyk, editors, *SLE 2013*, volume 8225 of *LNCS*, page 76–95. Springer, 2013.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[vdML04] Thomas von der Maßen and Horst Lichter. Deficiencies in Feature Models. In *Workshop on Software Variability Management for Product Derivation*, volume 44, page 21, 2004.

[vdS11] Tijs van der Storm. *The Rascal Language Workbench*. CWI. Software Engineering [SEN], 2011.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[VP12] Markus Voelter and Vaclav Pech. Language Modularity With the MPS Language Workbench. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1449–1450. IEEE, 2012.

[VRSK12] 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*, pages 121–140. ACM, 2012.

[VSB+13] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2013.

[VSBK14] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.

[WBGK08] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 2008.

[WHT+09] Jules White, James H. Hill, Sumant Tambe, Aniruddha S. Gokhale, Douglas C. Schmidt, and Jeff Gray. Improving Domain- Specific Language Reuse with Software Product Line Techniques. *IEEE software*, 26(4):47–53, 2009.

[Wor19] Andreas Wortmann. Towards Component-Based Development of Textual Domain-Specific Languages. In Luigi Lavazza, Herwig Mannaert,

and Krishna Kavi, editors, *International Conference on Software Engineering Advances (ICSEA 2019)*, pages 68–73. IARIA XPS Press, November 2019.

[www99] RFC2713 - Schema for Representing Java(tm) Objects in an LDAP Directory. https://tools.ietf.org/html/rfc2713, 1999. [Online; accessed 16-July-2020].

[www05] RFC3986 - Uniform Resource Identifier (URI): Generic Syntax. https://tools.ietf.org/html/rfc3986, 2005. [Online; accessed 16-July-2020].

[www17] ECMA-404 - The JSON Data Interchange Standard. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, 2017. [Online; accessed 16-July-2020].

[www20a] Google GSON on GitHub. https://github.com/google/gson/, 2020. [Online; accessed 28-August-2020].

[www20b] Documentation of the Java Object Serialization. https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/, 2020. [Online; accessed 16-August-2020].

[www20c] Documentation of the JSON Syntax. https://www.json.org/json-en.html, 2020. [Online; accessed 16-July-2020].

[www20d] JSON-java on GitHub. https://github.com/stleary/JSON-java/tree/master/src/main/java/org/json, 2020. [Online; accessed 28-August-2020].

[www20e] JSON-P API Description. https://javaee.github.io/jsonp, 2020. [Online; accessed 28-August-2020].

[www20f] The JSON Schema Wbsite. https://json-schema.org/, 2020. [Online; accessed 16-July-2020].

[www20g] Java Specification Request 374 - Java API for JSON Processing 1.1. https://www.jcp.org/en/jsr/detail?id=374, 2020. [Online; accessed 28-August-2020].

[www20h] Google Protocol Buffers API Reference. https://developers.google.com/protocol-buffers/docs/reference/overview, 2020. [Online; accessed 28-February-2021].

[www21a] The GEMOC Studio - A Language and Modeling Workbench for Executable Modeling. http://gemoc.org/studio.html, 2021. [Online; accessed 26-January-2021].

[www21b] javac - Java programming language compiler. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html, 2021. [Online; accessed 03-02-2021].

[www21c] Chapter 4. The class File Format. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html, 2021. [Online; accessed 03-02-2021].

[www21d] Static Semantics Definition with NaBL2. http://www.metaborg.org/en/latest/source/langdev/meta/lang/nabl2/index.html, 2021. [Online; accessed 16-August-2021].

[www21e] Object Management Group: About the Unified Modeling Language, v2.5.1. https://www.omg.org/spec/UML/About-UML/, 2021. [Online; accessed 15-May-2021].

[www21f] Object Management Group: SysML V2. http://www.omgsysml.org/SysML-2.htm, 2021. [Online; accessed 07-March-2021].

List of Figures

Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview of related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR+19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04c]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question of how digital and organizational techniques help to cope with the physical distance of developers and [RRSW17] addresses how to teach agile modeling.

Modeling will increasingly be used in development projects if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKR+06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum11, Rum12] and [Rum16, Rum17], the UML/P, a variant of the UML especially designed for programming, refactoring, and evolution is defined.

The language workbench MontiCore [GKR+06, GKR+08, HKR21] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09], and refactoring in various modeling and programming languages [PR03]. To better understand the effect of an agile evolving design, we discuss the need for semantic differencing in [MRR10].

In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation. In [KMA+16] we have also introduced a classification of ways to reuse modeled software components.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, the size, and the number of artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17].

To keep track of relevant structures, artifacts, and their relations in order to be able e.g. to evolve or adapt models and their implementing code, the *artifact model* [GHR17, Gre19] was introduced. [BGRW18] and [HJK+21] explain its applicability in systems engineering based on MDSE projects and [BHR+18] applies a variant of the artifact model to evolutionary development, especially for CPS.

An artifact model is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model, therefore, covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development

and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks.

For analysis of MLOps in an agile development, a software 2.0 artifact model distinguishing different kinds of artifacts is given in [AKK+21].

According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Such operators have been defined for Activity Diagrams [MRR11d], Class Diagrams [MRR11b], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18a].

We apply logic, knowledge representation, and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests, or find counterexamples using a theorem prover. We have defined a core theory in [BKR+20], which is based on the core concepts of Broy's Focus theory [RR11, BR07], and applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, KMP+21, HRR12].

Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams (ADs) and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11, KLPR12] and city quarters [GLPR15] to optimize operational efficiency and prevent unneeded CO_2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P, [Hab16] for MontiArc is used in domains such as cars or robotics [HRR12], and [AMN+20a] for enterprise information systems based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HKR21].

In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKR+06, GHK+15, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12, HRW15, Hoe18], we demonstrate how to systematically derive a transformation language in concrete syntax and e.g. in [HHR+15, AHRW17] we have applied this technique successfully for several UML sub-languages and DSLs.

[HNRW16] presents how to generate extensible and statically type-safe visitors. In [NRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLN+15, HLN+15a].

To understand the implications of executability for UML, we discuss the needs and the advantages of executable modeling with UML in agile projects in [Rum04c], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML) & the UML-P Tool

Starting with the early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] and is implemented in [Sch12].

Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09], [BCGR09a], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied when checking variants of class diagrams [MRR11e] and object diagrams [MRR11c] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11a] which allows us to check for semantic differences in activity diagrams [MRR11d]. The basic semantics for ADs and their semantic variation points are given in [GRR10].

We also discuss how to ensure and identify model quality [FHR08], how models, views, and the system under development correlate to each other [BGH+98b], and how to use modeling in agile development projects [Rum04c], [Rum03] and [Rum02].

The question of how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99a], [FEL+98] and [SRVK10].

The UML-P tool was conceptually defined in [Rum16, Rum17, Rum12, Rum11], got the first realization in [Sch12], and is extended in various ways, such as logically or physically distributed computation [BKRW17a]. Based on a detailed examination [JPR+22], insights are also transferred to the SysML 2.

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use than general-purpose programming languages but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HKR21] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HKR21] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Voe11, HLN+15, HLN+15a, HRW18, BEK+18b, BEK+19, Sch12] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages already in [KRV06] and elaborate on the engineering aspect of DSL development in [CFJ+16].

[Wei12, HRW15, Hoe18] present an approach that allows the creation of transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses, and model evolution have been discussed in [LRSS10] and [SRVK10].

[BJRW18] describes a mapping bridge between both. DSL quality in [FHR08], instructions for defining views [GHK+07] and [PFR02], guidelines to define DSLs [KKP+09], and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

A broader discussion on the *global* integration of DSMLs is given in [CBCR15] as part of [CCF+15a], and [TAB+21] discusses the compositionality of analysis techniques for models.

The MontiCore language workbench has been successfully applied to a larger number of domains, resulting in a variety of languages documented e.g. in [AHRW17, BEH+20, BHR+21, BPR+20, HHR+15, HJRW20, HMR+19, HRR12, PBI+16, RRW15] and Ph.D. theses like [Ber10, Gre19, Hab16, Her19, Kus21, Loo17, Pin14, Plo18, Rei16, Rot17, Sch12, Wor16].

Software Language Engineering

For a systematic definition of languages using a composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15a]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HKR21, HRW18, BPR+20, BEK+19].

In [SRVK10] we discuss the possibilities and the challenges of using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Voe11, Naz17, KRV08, HLN+15, HLN+15a, HNRW16, HKR21, BEK+18b, BEK+19] and the backend [RRRW15b, NRR16, GMR+16, HKR21, BEK+18b, BBC+18]. In [GHK+15, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV10] describes the roles in software development using domain specific languages.

Language derivation is to our belief a promising technique to develop new languages for a specific purpose, e.g., model transformation, that relies on existing basic languages [HRW18].

How to automatically derive such a transformation language using a concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs.

We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15, HHK+13] that are derived from base languages to be able to constructively describe differences between model variants usable to build feature sets.

The derivation of internal DSLs from grammars is discussed in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services.

We use streams, statemachines, and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR10, HRR12] for architecture design and extensions for states [RRW13c, BKRW17a, RRW14a, Wor16]. In [RRW13], we introduce a code generation framework for MontiArc. [RRRW15b] describes how the language is composed of individual sublanguages.

MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [PR13], security in [HHR+15], and the robotics domain [AHRW17, AHRW17b]. Extension mechanisms for MontiArc are generally discussed in [BHH+17].

[GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants.

[MRR14b] provides a precise technique for verifying the consistency of architectural views [Rin14, MRR13] against a complete architecture to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. An experience report [MRRW16] and a methodological embedding [DGH+19] complete the core approach.

Extensions for co-evolution of architecture are discussed in [MMR10], for powerful analyses of software architecture behavior evolution provided in [BKRW19], techniques for understanding semantic differences presented in [BKRW17], and modeling techniques to describe dynamic architectures shown in [HRR98, HKR+16, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09, TAB+21] motivate the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HKR21] that can even be used to develop modeling tools in a compositional form [HKR21, HLN+15, HLN+15a, HNRW16, NRR16, HRW18, BEK+18b, BEK+19, BPR+20, KRV07b]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09].

[Voe11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15b] applies compositionality to robotics control.

[CBCR15] (published in [CCF+15a]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information, we have developed the concept of tagging languages in [GLRR15, MRRW16]. It allows the description of additional information for model elements in separated documents, facilitates reuse, and allows typing tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision, and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96, RK96]. An extended version especially suited for the UML is given in [GRR09], [BCGR09a] and in [BCGR09] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08] or sequence diagrams in [BGH+98a].

To better understand the effect of an evolved design, detection of semantic differencing, as opposed to pure syntactical differences, is needed [MRR10]. [MRR11d, MRR11a] encode a part of the semantics to handle semantic differences of activity diagrams. [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. And [BKRW17] compares component and connector architectures similar to SysML' block definition diagrams.

In [BR07, RR11], a precise mathematical model for distributed systems based on black-box behaviors of components is defined and accompanied by automata in [Rum96]. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of exemplary object interaction, today called sequence diagram. [BGH+98b] discusses the relationships between a system, a view, and a complete model in the context of the UML.

[GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these to class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10].

[Rum12] defines the semantics in a variety of code and test case generation, refactoring, and evolution techniques. [LRSS10] discusses the evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied to the development of CPS.

A first encoding of these theories in the Isabelle verification tool is defined in [BKR+20].

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code, they are not initially correct and need to be changed, evolved, and maintained over time. Model transformation is therefore essential to effectively deal with models [CFJ+16].

Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04c, MRR10], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11e, Rum12], systematic model transformation language development [Wei12, HRW15, Hoe18, HHR+15], repair of failed model evolution [KR18a].

[Rum04c] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97] and refining pipe-and-filter architectures is explained in [PR99]. This has e.g. been applied for robotics in [AHRW17, AHRW17b].

Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. [HRRS11, HRR+11, HRRS12] encode these in constructive Delta transformations, which are defined in derivable Delta languages [HHK+13].

Translation between languages, e.g., from class diagrams into Alloy [MRR11e] allows for comparing class diagrams on a semantic level. Similarly, semantic differences of evolved activity diagrams are identified via techniques from [MRR11d] and for Simulink models in [RSW+15].

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example, cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08, GKPR08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12].

Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRRS11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe special variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13, HHK+15] and [HRW15] describe an approach to systematically derive delta languages.

We also apply variability modeling languages to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic

way to define variants of modeling languages [CGR09], leverage features for their compositional reuse [BEK+18b, BEK+19], and applied it as a semantic language refinement on Statecharts in [GR11].

Digital Twins and Digital Shadows in Engineering and Production

The digital transformation of production changes the life cycle of the design, the production, and the use of products [BDJ+22]. To support this transformation, we can use Digital Twins (DTs) and Digital Shadows (DSs). In [DMR+20] we define: "A digital twin of a system consists of a set of models of the system, a set of digital shadows, and provides a set of services to use the data and models purposefully with respect to the original system."

We have investigated how to synthesize self-adaptive DT architectures with model-driven methods [BBD+21a] and have applied it e.g. on a digital twin for injection molding [BDH+20]. In [BDR+21] we investigate the economic implications of digital twin services.

Digital twins also need user interaction and visualization, why we have extended the infrastructure by generating DT cockpits [DMR+20]. To support the DevOps approach in DT engineering, we have created a generator for low-code development platforms for digital twins [DHM+22] and sophisticated tool chains to generate process-aware digital twin cockpits that also include condensed forms of event logs [BMR+22].

[BBD+21b] describes a conceptual model for digital shadows covering the purpose, relevant assets, data, and metadata as well as connections to engineering models. These can be used during the runtime of a DT, e.g. when using process prediction services within DTs [BHK+21].

Integration challenges for digital twin systems-of-systems [MPRW22] include, e.g., the horizontal integration of digital twin parts, the composition of DTs for different perspectives, or how to handle different lifecycle representations of the original system.

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12, BBR20] are complex, distributed systems that control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied to the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12, KRRW17], autonomous driving [BR12b, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07].

In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest to European avionics [ZPK+11]. Optimized [KRS+18a] and domain specific code generation [AHRW17b], and the extension to product lines of CPS [RSW+15, KRR+16, RRS+16] are key for CPS.

A component and connector architecture description language (ADL) suitable for the specific challenges in robotics is discussed in [RRW13c, RRW14a, Wor16, RRSW17, Wor21]. In [RRW12], we use this language for describing requirements and in [RRW13], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as an Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition of contributing to systems engineering in automotive [FND+98] and [GHK+08a], which culminated in a new compre-

hensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration.

To facilitate the modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for production engineering based on these concepts [BKL+18] and addressed to bridge the gap between functions and the physical product architecture by modeling mechanical functional architectures in SysML [DRW+20]. For that purpose, we also did a detailed examination of the upcoming SysML 2.0 standard [JPR+22] and examined how to extend the SPES/CrEST methodology for a systems engineering approach [BBR20].

Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20], and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09a, BCGR09], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems.

In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07].

We apply these techniques, e.g., in MontiArcAutomaton [RRW13, RRW14a, RRW13, RW18], in a robot task modeling language [THR+13], and in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behavior (2) based on information from previously stored and real-time monitored structural context and behavior data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20, MRZ21] or UML/P based languages [MNRV19]. [MM15] describes a process of how languages for assistive systems can be created. MontiGem [AMN+20a] is used as the underlying generator technology.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18a] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view of the system design

allows to track who does what, when, why, where, and how with personal data, makes information about it available via information services and provides support using assistive services.

Modeling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires the composition and the interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers the broad propagation of robotics applications.

The MontiArcAutomaton language [RRW12, RRW14a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13c, RRRW15b, HKR21] that perfectly fit robotic architectural modeling.

The iserveU modeling framework describes domains, actors, goals, and tasks of service robotics applications [ABH+16, ABH+17] with declarative models. Goals and tasks are translated into models of the planning domain definition language (PDDL) and then solved [ABK+17]. Thus, domain experts focus on describing the domain and its properties only.

The LightRocks [THR+13, BRS+15] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible with different robotics middleware platforms.

Many of the concepts in robotics were derived from automotive software [BBR07, BR12b].

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment, and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed, and tested. A consistent requirement management connecting requirements with features in all development phases for the automotive domain is described in [GRJA12].

The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. A methodical embedding of the resulting function nets and their quality assurance using automated testing is given in the SMaRDT method [DGH+19, KMS+18].

[HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses the means to extract a well-defined Software Product Line from a set of copy and paste variants.

Potential variants of components in product lines can be identified using similarity analysis of interfaces [KRR+16], or execute tests to identify similar behavior [RRS+16]. [RSW+15] describes an approach to using model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we model dynamic reconfiguration of architectures applied to cooperating vehicles.

Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12b, BR12], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in the development and the evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12b].

[MMR10] gives an overview of the state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions.

MontiSim simulates autonomous and cooperative driving behavior [GKR+17] for testing various forms of errors as well as spatial distance [FIK+18, KKRZ19]. As tooling infrastructure, the SSELab storage, versioning, and management services [HKR12] are essential for many projects.

Internet of Things, Industry 4.0 & the MontiThings Tool

The Internet of Things (IoT) requires the development of increasingly complex distributed systems. The MontiThings ecosystem [KRS+22] provides an end-to-end solution to modeling, deploying [KKR+22], and analyzing [KMR21] failure-tolerant [KRS+22] IoT systems and connecting them to synthesized digital twins [KMR+20]. We have investigated how model-driven methods can support the development of privacy-aware [ELR+17, HHK+14] cloud systems [PR13], distributed systems security [HHR+15], privacy-aware process mining [MKM+19], and distributed robotics applications [RRRW15b].

In the course of Industry 4.0, we have also turned our attention to mechanical or electrical applications [DRW+20]. We identified the digital representation, integration, and (re-)configuration of automation systems as primary Industry 4.0 concerns [WCB17]. Using a multi-level modeling framework, we support machine as a service approaches [BKL+18].

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO_2 emissions are important challenges. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales.

During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for the technical specification of building services already.

We adapted the well-known concept of statemachines to be able to describe different states of a facility and validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing and Services

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, facilitate web-based innovations, and increase the efficiency and cost-effectiveness of web development [KRR14].

Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15a], Big Data, Apps, and Service Ecosystems bring attention to aspects like responsiveness, privacy, and open platforms. Regardless of the application domain, developers of such systems need robust methods and efficient, easy-to-use languages and tools [KRS12].

We tackle these challenges by perusing a model-based, generative approach [PR13]. At the core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale.

We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our development platforms. New

services, e.g., for collecting data from temperature sensors, cars, etc. are now easily developed and deployed, e.g. in production or Internet-of-Things environments.

Security aspects and architectures of cloud services for the digital me in a privacy-aware environment are addressed in [ELR+17].

Model-Driven Engineering of Information Systems & the MontiGem Tool

Information Systems provide information to different user groups as the main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HKR21], we developed several generators for such data-centric information systems.

MontiGem [AMN+20a] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular, and incremental, handwritten and generated code pieces are well integrated [GHK+15a, NRR15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

We are using MontiGem for financial management [GHK+20, ANV+18], for creating digital twin cockpits [DMR+20], and various industrial projects. MontiGem makes it easier to create low-code development platforms for digital twins [DHM+22]. When using additional DSLs, we can develop assistive systems providing user support based on goal models [MRV20], privacy-preserving information systems using privacy models and purpose trees [MNRV19], and process-aware digital twin cockpits using BPMN models [BMR+22].

We have also developed an architecture of cloud services for the digital me in a privacy-aware environment [ELR+17] and a method for retrofitting generative aspects into existing applications [DGM+21].

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016.

[ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse.* Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.

[ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AKK+21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 55–68. ACM SIGPLAN, October 2021.

[AMN+20a] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sand-

kuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, CEUR Workshop Proceedings 2097, pages 75–79. CEUR-WS.org, May 2018.

[BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics*, 12, 2018.

[BBD+21b] Fabian Becker, Pascal Bibow, Manuela Dalibor, Aymen Gannouni, Viviane Hahn, Christian Hopmann, Matthias Jarke, Istvan Koren, Moritz Kröger, Johannes Lipp, Judith Maibaum, Judith Michael, Bernhard Rumpe, Patrick Sapel, Niklas Schäfer, Georg J. Schmitz, Günther Schuh, and Andreas Wortmann. A Conceptual Model for Digital Shadows in Industry and its Application. In Aditya Ghose, Jennifer Horkoff, Vitor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling, ER 2021*, pages 271–281. Springer, October 2021.

[BBD+21a] Tim Bolender, Gereon Bürvenich, Manuela Dalibor, Bernhard Rumpe, and Andreas Wortmann. Self-Adaptive Manufacturing with Digital Twins. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 156–166. IEEE Computer Society, May 2021.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BBR20] Manfred Broy, Wolfgang Böhm, and Bernhard Rumpe. Advanced Systems Engineering - Die Systeme der Zukunft. White paper, fortiss. Forschungsinstitut für softwareintensive Systeme, Munich, July 2020.

[BCGR09] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *ACM Trans. Internet Things*, 3:1–32, February 2022.

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BDR+21] Christian Brecher, Manuela Dalibor, Bernhard Rumpe, Katrin Schilling, and Andreas Wortmann. An Ecosystem for Digital Shadows in Manufacturing. In *54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0*. Elsevier, September 2021.

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology*, 19(3):3:1–16, October 2020. Special Issue dedicated to Martin Gogolla on his 65th Birthday.

[BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

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

[Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98a] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. *Computer Standards & Interfaces*, 19(7):335–345, November 1998.

[BGH+98b] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND'17) Workshop*, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language.

In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHK+21] Tobias Brockhoff, Malte Heithoff, István Koren, Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, Merih Seran Uysal, Wil M. P. van der Aalst, and Andreas Wortmann. Process Prediction with Digital Twins. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 182–187. ACM/IEEE, October 2021.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BHR+18] Arvid Butting, Steffen Hillemacher, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Shepherding Model Evolution in Model-Driven Development. In *Joint Proceedings of the Workshops at Modellierung 2018 (MOD-WS 2018)*, CEUR Workshop Proceedings 2060, pages 67–77. CEUR-WS.org, February 2018.

[BHR+21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings*

of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18). ACM, September 2018.

[BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle Theory for distributed, interactive systems - the untimed case*. Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.

[BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software*, 149:437–461, March 2019.

[BMR+22] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)*, 70, June 2022.

[BPR+20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12b] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*, 2015.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15a] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, LNCS 8378, pages 101–136. Springer International Publishing, Switzerland, 2014.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In

Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[DGH+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. *Software: Practice and Experience*, 49(2):301–328, February 2019.

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology*, 20:1–24, November 2021.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, 70, June 2022.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[EFLR99a] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, LNCS 1618, pages 336–348. Springer, Germany, 1999.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

[ELR+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 Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[FEL+98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FND+98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. BMW-ROOM An Object-Oriented Method for ASCET. In *SAE'98, Cobo Center (Detroit, Michigan, USA)*, Society of Automotive Engineers, 1998.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.

[GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects.* Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[Gre19] Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte*. Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-based Definition of Modeling Language Semantics. In *Proc. of FMOODS/FORTE 2009, LNCS 5522*, Lisbon, Portugal, 2009.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Her19] Lars Hermerschmidt. *Agile Modellgetriebene Entwicklung von Software Security & Privacy*. Aachener Informatik-Berichte, Software Engineering, Band 41. Shaker Verlag, June 2019.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15a] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HHR+15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, CEUR Workshop Proceedings 1463, pages 18–23, 2015.

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021.

[HJRW20] Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 189–190. Gesellschaft für Informatik e.V., February 2020.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore.

In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, LNCS 9839, pages 175–182. Springer, December 2016.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HLN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, 18(1):1–60, July 2019.

[HNRW16] 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 (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[Hoe18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformation-*

ssprachen. Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter-Systeme VI*, Informatik-Bericht 2010-01, pages 13 – 22. fortiss GmbH, Germany, 2010.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

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

[JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology*, 21, July 2022.

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35(6):40–47, 2018.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications.

[KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *ACM Transactions on Internet of Things*, November 2022.

[KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti

Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR'16)*, LNCS 9679, pages 122–137. Springer, June 2016.

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

[KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 197–209. ACM SIGPLAN, October 2021.

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.

[KR18a] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of

Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016.

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[KRRW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012.

[KRS+18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[KRS+22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software*, 183:1–21, January 2022.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020.

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems.* Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021.

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE.* Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017.

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[MPRW22] Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Integration Challenges for Digital Twin Systems-of-Systems. In *10th IEEE/ACM International Workshop on Software Engineering for*

Systems-of-Systems and Software Ecosystems, pages 9–12. IEEE, May 2022.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting*

of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13), pages 444–454. ACM New York, 2013.

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, CEUR Workshop Proceedings 1723, pages 19–24, October 2016.

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.

[MRZ21] Judith Michael, Bernhard Rumpe, and Lukas Tim Zimmermann. Goal Modeling and MDSE for Behavior Assistance. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 370–379. ACM/IEEE, October 2021.

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials.* Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[NRR15a] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-

centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.

[NRR16] 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*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

[PR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, LNI 254, pages 93–108. Bonner Köllen Verlag, March 2016.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen.* Aachener Informatik-Berichte, Software Engineering, Band 17. Shaker Verlag, 2014.

[Plo18] Dimitri Plotnikov. *NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project.* Aachener Informatik-Berichte, Software Engineering, Band 33. Shaker Verlag, February 2018.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15.* Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen.* Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems.* Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex.* Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRRW15b] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering:*

Software Engineering and Education Track (ICSE'17), pages 127–136. IEEE, May 2017.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04c] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.

[SM18a] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018.

[SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communi-

cations and Networks, pages 227–246. Springer International Publishing, 2020.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70. Springer, July 2021.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Voe11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen.* Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling.* Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems.* Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, Oktober 2021.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On De-

mand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.