

RWTH Aachen University
Software Engineering Group

MontiCore
Language Workbench and Library
Handbook

Edition 2021



Bernhard Rumpe
Katrin Hölldobler
Oliver Kautz

<http://www.se-rwth.de/>
<http://www.monticore.de/>



[HKR21a] M. Heithoff, E. Kusmenko, B. Rumpe:

Synchronous Execution Semantics for Component & Connector Models.

In: Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 264-273, ACM/IEEE, Oct. 2021.

<https://www.se-rwth.de/publications/>

Foreword

MontiCore is a language workbench, which is developed since 2004. We have started its development because at that time the available tools for model management were often very poor in their functionalities and also not extensible, but closed shops. In 2004 the first version of the UML/P was published (and is now available as [Rum16, Rum17]) and demonstrated that the family of languages that the UML is made of can be substantiated with useful transformation, refinement, refactoring and semantic diffing techniques [KRW20, BEK⁺18b]. Code and test code generation as well as flexible combination of language fragments, such as OCL within Statecharts or Class Diagrams for typing in Component and Connector Diagrams, were the techniques of primary interest. However, at that time available modeling tools were mainly editors and thus not helpful in realizing these advanced and smart techniques. This was the original motivation for MontiCore that can also be found in the first foundational theses in [Kra10, Völ11, Sch12, Höl18].

Later, it became apparent that the UML will be complemented by SysML as well as domain specific languages (DSLs) that will be connected to software development or execution in various ways. The definition of DSLs encounters the same difficulties as the definition of the UML faced, i.e., they are often built from scratch, reuse is pretty bad, and the same concepts get different syntactic shapes. Thus, combining DSLs is rather impossible. We therefore extended the focus of MontiCore to become a general language workbench that allows to define languages and language fragments and to derive as much as possible from an integrated and therefore compact definition.

In this version of the MontiCore Reference Manual, the core facilities of MontiCore are described. Extensions are available through various projects either using or enhancing MontiCore with more functionality. MontiCore provides sophisticated techniques to generate transformation languages and their transformation engines based on DSLs [Höl18, HRW15, AHRW17b, RRW15, HHRW15, Wei12, HMR⁺19], MontiCore was used to explore tagging languages [Loo17, MRRW16, GLRR15, HMR⁺19], various forms of the UML and its derivatives [Sch12, Wor16, Hab16, Rei16, Rot17, HMR⁺19], sophisticated forms of language composition and derivation techniques including the generated code [GHK⁺15b, HLMSN⁺15a, HMSNRW16, MSN17, HRW18, BEK⁺18a, BEK⁺18b, BEK⁺19]. MontiCore also explored novel comfortable code generation techniques [MSNRR16, EHRR19] as well as plenty of domain specific languages.

Despite MontiCore originated as academic tool to explore modeling and meta-modeling techniques, after 17 years of development, it has reached an extraordinary strength and is thus increasingly used in industrial projects. The small excerpt of topics below demonstrates this: energy management [Pin14], program planning in the television domain [DHH⁺20], modelling and execution of tax laws, assistive systems [MRV20], AutoSAR

communication and architecture modelling, autonomous driving [KKRZ19, KKR19], accounting and management [GMN⁺20, AMN⁺20, SHH⁺20], orchestrating digital twins [JvdAB⁺21, DMR⁺20, BDH⁺20], internet of things, as well as in scientific projects of entirely different nature, such as simulation of city scenarios for autonomous driving [Ber10] or human brain modeling [PBI⁺16]. MontiCore, however, does not primarily focus on comfort, e.g., graphical editing, but advanced functionality for model-based analysis or synthesis of software intensive systems and quick textual editing for experienced users.

We would like to thank all current and former members of our group as well as all students and apprentices who helped to develop MontiCore in its current shape. Namely, we would like to thank Kai Adam, Daoud Ali, Vassily Aliseyko, Professor Dr. Christian Berger, Vincent Bertram, Miriam Boß, Arvid Butting, Joel Charles, Manuela Dalibor, Anabel Derlam, Niklas Dienstknecht, Imke Drave, Robert Eikermann, Christoph Engels, Arkadii Gerasimov, Dr. Timo Greifenberg, Dr. Hans Grönniger, Dr. Tim Gülke, Dr. Arne Haber, Guido Hansen, Olga Haubrich, Malte Heithoff, Dr. Lars Hermerschmidt, Dr. Christoph Herrmann, Gabi Heuschen, Steffen Hillemacher, Nico Jansen, Hendrik Kausch, Christian Kirchhof, Carsten Kolassa, Dr. Anne-Therese Körtgen, Thomas Kurpick, Evgeny Kusmenko, Dr. Holger Krahn, Dr. Stefan Kriebel, Achim Lindt, Dr. Markus Look, Daniel Maibach, Professor Dr. Shahar Maoz, Matthias Markthaler, Dr. Dan Matheson, Dr. Judith Michael, Joshua Mingers, Dr. Klaus Müller, Dr. Pedram Mir Seyed Nazari, Antonio Navarro Pérez, Lukas Netz, Mathias Pfeiffer, Nina Pichler, Dr. Claas Pinkernell, Dr. Dimitri Plotnikov, Deni Raco, Professor Dr. Jan Ringert, Dr. Holger Rendel, Dr. Dirk Reiss, Dr. Daniel Retkowitz, Dr. Alexander Roth, Dr. Martin Schindler, David Schmalzing, Steffi Schrader, Dr. Frank Schroven, Dr. Christoph Schulze, Igor Shumeiko, Brian Sinkovec, Sebastian Stüber, Simon Varga, Dr. Steven Völkel, Louis Wachtmeister, Dr. Ingo Weisemöller, Dr. Michael von Wenckstern, and Professor Dr. Andreas Wortmann. The individual contributions to MontiCore and its derivatives resulted in numerous publications¹. Special thanks go to Marita Breuer and Galina Volkova, who maintain and extend MontiCore, and in particular to Sylvia Gunder and Sonja Müßigbrodt, who ensure that all financial and project activities supporting our language workbench project MontiCore are running perfectly.

We also would like to thank the authors or co-authors of several chapters, for describing relevant parts of MontiCore directly in this handbook.

To all readers of this handbook: We hope you enjoy reading this manual and trying out our language workbench MontiCore as well as the tools generated with MontiCore. In case you have any suggestions or questions do not hesitate to contact us.

Aachen, 27.03.2021

Bernhard Rumpe, Katrin Hölldobler, Oliver Kautz

¹www.se-rwth.de/publications/

Contents

1	Introduction to Tool Generation	1
1.1	MontiCore Language Workbench	1
1.2	Notational Conventions	3
1.3	Textual Modeling	4
1.4	Methodical Considerations: Agile Modeling	5
2	Getting Started with MontiCore	7
2.1	Prerequisites: Installing the Java Development Kit	7
2.2	Install and Use the MontiCore Command Line Interface	8
2.2.1	Installation	9
2.2.2	Inspect the Example Grammar	9
2.2.3	Run MontiCore	12
2.2.4	Compile the Target	22
2.2.5	Run the Tool	28
2.3	Using MontiCore via Gradle From the Command Line	29
2.4	Using MontiCore in Eclipse	29
2.4.1	Setting up Eclipse	30
2.4.2	Importing the Example	30
2.4.3	Running MontiCore	30
2.5	Using MontiCore in IntelliJ IDEA	31
2.5.1	Setting up IntelliJ IDEA	31
2.5.2	Importing the Example	31
2.5.3	Running MontiCore	32
3	Architecture of a Model Processor	33
3.1	Structure of a Model Processor - External View	33
3.2	Internal Architecture of a Generator - Component View	34
3.3	Tool Workflow	36
4	MontiCore Grammar for Language and AST Definitions	39
4.1	Lexical Tokens for the Scanner	40
4.1.1	Definition of Tokens using Regular Expressions	41
4.1.2	Actions to Process a Token	42
4.1.3	Predefined Tokens in Importable Grammars	43
4.2	Productions in the Grammar	46
4.2.1	Terminals	48
4.2.2	Enumeration	49
4.2.3	Nonterminals	50
4.2.4	Interface Nonterminals: implements	51

4.2.5	Extending Nonterminals: extends	52
4.2.6	Abstract Nonterminals	52
4.2.7	Starting Nonterminal	54
4.2.8	Infix Operations and Priorities	54
4.2.9	Restricting the Cardinality of a Nonterminal	55
4.2.10	Symbols and Scopes	56
4.2.11	Passing Code to the ANTLR Parser	57
4.2.12	Annotations for Nonterminals and Grammars	58
4.2.13	Predefined Nonterminals in Importable Grammars	59
4.3	Additional Control Directives in the MCG Language	59
4.3.1	Splitting Tokens	60
4.3.2	Local Keywords: Avoid handling Keywords as Tokens	61
4.4	Context Conditions for the MCG Language	62
4.5	Semantic Predicates and Actions	69
4.6	EBNF of the MCG Language	70
5	Abstract Syntax Tree	77
5.1	Mapping Nonterminals to the AST	78
5.2	Interface and Abstract Nonterminals	79
5.3	Extending Nonterminals: astimplements, astextends	79
5.4	Extending the Abstract Syntax Implementation	81
5.5	Terminals in the AST	83
5.6	Enumerations	84
5.7	ASTNode: A Base Interface for AST Classes	85
5.8	Generated ASTNode Subclasses	86
5.9	Node Construction Using the Node Builder Mill	91
5.10	Handwritten Extension of AST Classes and Node Builders	96
5.10.1	Handwritten Extension of AST Classes: TOP-Mechanism	96
5.10.2	Handwritten Extension of AST Builders and Mills	97
6	Parser Generation and Use	101
6.1	Generating a Parser and a Lexer, as done in MontiCore	101
6.2	Interface of the Generated Parser Classes	104
6.3	Executing a Generated Parser	106
7	Language Composition	109
7.1	Introduction to Language Composition	109
7.2	Language Composition at a Glance	113
7.3	Grammar Constructs for Language Composition	116
7.3.1	Component Grammar	117
7.3.2	External Nonterminals	117
7.3.3	Importing and Extending Grammars	119
7.4	Language Inheritance	120
7.4.1	Redefining / Overriding Productions of Grammars	121
7.4.2	Extending the Implementation Structure of a Nonterminal	123
7.4.3	Extending Multiple Inherited Grammars	124

7.5	Language Embedding	125
7.6	Composing the Builder Infrastructure	126
7.7	Composing Parsers	127
7.8	Composition of Visitors and Context Conditions	128
7.9	Conservative Extension	129
7.9.1	Conservative Extension of the Concrete Syntax	129
7.9.2	Access-Conservative Extension of the Abstract Syntax	131
7.9.3	Modification-Conservative Extension of the Abstract Syntax	132
7.9.4	AST Signatures Causing Java Type Errors	133
8	Visitors for AST Traversal	135
8.1	Visitor Infrastructure for a Language	136
8.1.1	Traverser Interface and Implementing Class	136
8.1.2	Visitor2 Interface	140
8.1.3	Handler Interface	141
8.1.4	Inheritance Handler for Explicit Visit of Supertypes	142
8.2	Visitors for Composed Languages	144
8.2.1	Visitor Infrastructure for Language Inheritance and Extension	145
8.2.2	Visitor for Language Inheritance with Overriding Nonterminal	147
8.2.3	Visitors for Compositional Language Embedding	149
9	Symbol Management Infrastructure	155
9.1	Introduction to Symbol Table Concepts	157
9.1.1	Symbols	157
9.1.2	Scopes	158
9.2	Defining Symbols	160
9.2.1	Runtime (RTE) Classes For Symbols	162
9.2.2	Generated Classes For Symbols	164
9.2.3	Defining Additional Symbol Attributes via <code>symbolrule</code>	166
9.3	Defining Scopes	167
9.3.1	Artifact Scope and Global Scope	167
9.3.2	Runtime Environment Classes for Scopes	168
9.3.3	Generated Classes For Scopes	172
9.3.4	Defining Scope Attributes and Methods via <code>scoperule</code>	175
9.4	Collaboration between AST, Symbol, and Scope	176
9.5	Using Symbols	177
9.6	Instantiating Symbol Tables	179
9.6.1	Phase 1: Symbols and Scope Skeletons	179
9.6.2	Phase 2+: Filling Symbols with Value	182
9.7	Loading and Storing Symbol Tables	182
9.7.1	Stored Symbol Tables	183
9.7.2	RTE Classes For Symbol Table Persistence	184
9.7.3	Generated Classes for Symbol Storage and Their Adaptation	187
9.7.4	Loading Symbol Tables	190
9.7.5	Storing Symbol Tables	191
9.7.6	Realizing Custom Serialization Strategies	192

9.8	Resolving Symbols in Scopes	195
9.8.1	How to Use Symbol Resolution	195
9.8.2	Concept Of Symbol Resolution	196
9.8.3	Generated Implementation for Symbol Resolution	199
9.8.4	Customizing Symbol Resolution	201
9.9	Visitors Also Handle Symbol Tables	203
9.10	Symbol Tables in Composed Languages	204
9.10.1	Symbol Management Infrastructure for Language Inheritance	205
9.10.2	Symbol Management Infrastructure for Language Aggregation	206
9.10.3	Symbol Adapters	206
9.10.4	Resolving for Adapted Symbols	208
10	Realizing Context Conditions	211
10.1	Context Condition Infrastructure	212
10.2	Implementation of Context Conditions	214
10.3	Testing Context Conditions	216
10.3.1	Testing a Context Condition on a Valid Model	217
10.3.2	Testing a Context Condition on an Invalid Model	218
11	Design Patterns Used and Invented for MontiCore	221
11.1	Static Delegator Design Pattern	221
11.2	RealThis Object Composition Pattern	223
11.3	Attribute and Association Access Pattern	227
11.3.1	Attribute Access Pattern	227
11.3.2	Association Access Pattern	230
11.3.3	The Extended Builder Pattern	231
11.4	Template Hook Pattern	232
11.5	Mill Pattern to Assist Composition	233
11.6	Multiple Interface Composition Pattern	236
12	FreeMarker for Code Generation	237
12.1	The FreeMarker Template Languages	237
12.2	Expressions in FreeMarker	238
12.3	Control Directives in FreeMarker	240
12.4	FreeMarker Add Ons	242
13	Generator Engine using Flexible Templates	245
13.1	Methodical Considerations	245
13.2	Generator API	247
13.3	Configuring the Generation Process	250
13.4	MontiCore APIs for Templates	252
13.4.1	Shortcuts: Aliases in Templates	252
13.4.2	The TemplateController	254
13.4.3	Logging within a Template	259
13.4.4	Variables in the Templates with GlobalExtensionManagement	260
13.5	Hook Points for Adaptation	262
13.5.1	The Concept of Hook Points	262

13.5.2	Forms of Hook Points	266
13.5.3	Defining Explicit Hook Points in Templates	268
13.5.4	Binding Hook Points	270
13.5.5	Replacing and Decorating Hook Points	272
13.5.6	HookPoint Replacement and Decoration Strategy	272
13.5.7	A HookPoint Replacement and Decoration Example	274
14	Integrating Handwritten Code	279
14.1	Integration of Handwritten Code	279
14.2	Adaptation of Generated Code by Subclassing	280
14.3	Adaptation of Generated Code using the TOP Mechanism	281
15	Error Handling, Logging and Reporting	285
15.1	Where to find Concrete Help for an Error, Warning, or other Message . . .	285
15.2	Errors, Warnings and Log Messages	286
15.2.1	Errors	286
15.2.2	Warnings and Information	287
15.2.3	Form of Errors, Warnings and Log Messages	288
15.3	The Error and Logging Component	289
15.4	Logging Configurations in MontiCore	292
15.4.1	Selecting one of the given Configurations	292
15.4.2	Using a Custom logback Configuration	293
15.4.3	Initializing the Log within Java	293
15.4.4	Providing a Custom Log Implementation	294
15.5	Reports	294
15.5.1	Where to Find Reports	294
15.5.2	How to Configure Reporting	294
15.5.3	Identifiers contained in the Reports	295
15.5.4	List of the Reports	297
15.6	For Developers: How to Deal with Errors and Warnings	300
16	MontiCore Use and Configuration from CLI or Gradle	303
16.1	MontiCore from Commandline	304
16.1.1	How to Call the CLI	304
16.1.2	Parameters of the CLI	305
16.2	Embedding the CLI in a Makefile Build Process	306
16.3	MontiCore Used via Gradle Plugin	310
16.3.1	Defining a MontiCore Task	311
16.3.2	Compilation and Packaging	314
16.3.3	Defining external Dependencies	314
16.3.4	Example Build Script	315
16.4	MontiCore in Maven	317
16.5	MontiCore Workflow Configuration with Groovy	318
16.5.1	The Standard Groovy Generation Script	319
16.5.2	MontiCore Base Class for Groovy Scripts	321
16.5.3	Methods Available within Groovy Scripts	323

16.5.4	Variables Available within Groovy Scripts	324
16.5.5	Available preimported Classes within Groovy Scripts	325
17	Example MontiCore Grammars	327
17.1	Component Grammar MCBasics.mc4	328
17.2	Component Grammar StringLiterals.mc4	329
17.3	Component Grammars for Numbers	331
17.3.1	Component Grammar MCNumbers.mc4	331
17.3.2	Component Grammar MCHexNumbers.mc4	333
17.4	Component Grammars for UML Languages	334
17.4.1	Component Grammar UMLStereotype.mc4	335
17.4.2	Component Grammar Cardinality.mc4	335
17.4.3	Component Grammar UMLModifier.mc4	336
17.4.4	Component Grammar Completeness.mc4	337
17.4.5	Component Grammar MCCCommon.mc4	338
18	Expression and Type Language Components	339
18.1	Literals as Basis for Expressions	340
18.1.1	MCLiteralsBasis	341
18.1.2	MCCCommonLiterals	341
18.1.3	MCJavaLiterals	345
18.2	Expressions in various Variants	346
18.2.1	ExpressionsBasis	347
18.2.2	CommonExpressions	348
18.2.3	BitExpressions	349
18.2.4	AssignmentExpressions	349
18.2.5	JavaClassExpressions	350
18.3	Symbols	352
18.3.1	BasicSymbols	352
18.3.2	OOSymbols	354
18.4	Types: From Simple To Generic	355
18.4.1	MCBasicTypes	356
18.4.2	MCCollectionTypes	357
18.4.3	MCSimpleGenericTypes	358
18.4.4	MCFullGenericTypes	358
18.4.5	MCArrayTypes	359
18.5	Using Base Grammars	359
18.6	Type Checking in MontiCore Languages	361
18.6.1	Types in a Symbol Table: SymTypes	362
18.6.2	Using Type Checks: the Type Check API	363
18.6.3	How the Type Check is Configured	365
19	Statement Language Components	371
19.1	MCStatementsBasis	372
19.2	MCVarDeclarationStatements	373
19.3	MCArrayStatements	374

19.4	MCCCommonStatements	374
19.5	MCReturnStatements	376
19.6	MCAssertStatements	376
19.7	MCSynchronizedStatements	377
19.8	MCExceptionStatements	377
19.9	MCLowLevelStatements	378
19.10	MCFullJavaStatements	379
20	The JavaLight Language	381
20.1	Sublanguage Hierarchy of JavaLight	381
20.2	Nonterminals of JavaLight	382
20.2.1	Methods, Constructors, and Attributes	383
20.2.2	Java Annotations	386
20.2.3	Java-Specific Array Initialization	387
21	Some Demonstrating Example Languages	389
21.1	A Simple Automata Language	389
21.2	Hierarchical Automata	396
21.3	A Language for Automata with Invariants	398
21.4	Scannerless Parsing to Handle Complex Tokens	400
21.4.1	Parsing with Whitespaces	400
21.4.2	Temporarily Parsing with Whitespaces	402
21.4.3	Preventing Whitespaces between Tokens	403
21.5	Tip: Testing Grammars and their Models	404
21.6	ColoredGraph Language	405
21.7	Questionnaire Language	408
22	Developer’s View on MontiCore	413
22.1	MontiCore’s GitHub Repository	414
22.2	For External Developers: How to Contribute	416
22.3	MontiCore’s Gradle Projects	416
22.4	Further Source Code Locations	417
23	Further Reading and Related Work	419
	List of Figures	429
	Listings	433
	References	441
	Index	463

Chapter 1

Introduction to Tool Generation

This handbook describes how to generate tools that deal with language processing.

These tools are partially generated by the *language workbench MontiCore* and partially need handcoded extensions. This handbook explains how to do this efficiently.

We assume that the reader is familiar with a variety of computer science concepts, such as *grammars*, UML and in particular their `class` diagrams and Java. If not, [HMU06] is suggested for grammars, [Rum16] for UML, and [GJS05] for Java.

As we will further explain, MontiCore is a meta-tool, actually a language workbench: It generates tools. It may well be that the generated tooling is itself a generator. That is fine, but in order to avoid confusion, we should be clear that there are two levels. Furthermore, the generated tooling can not only be a generator, but can be used for transformation, simple and complex analysis, simulation or the connection of runtime data with the originating models.

All the tooling is about *processing models* in standard or *domain specific languages* (DSLs). MontiCore generates infrastructure, such that many models as well as heterogeneous models, that means of different languages, can be processed. *Modeling in the large* is well assisted.

MontiCore is not only about generation of tools, but in particular about *reuse* of tool components that have been developed independently. In particular MontiCore provides a number of techniques to systematically reuse *language components* by *composing*, *extending* or *inheriting* them. MontiCore assists an easy development and extension of languages and thus should be a good solution for tool development.

MontiCore also includes a number of plug-ins e.g. for Eclipse or EMF-compatible generation and thus supposedly has a rather useful development environment. We strongly encourage the reader to download and install MontiCore.

1.1 MontiCore Language Workbench

The MontiCore language workbench can be used both as a closed product out-of-the-box for the generation of software as well as an open, customizable framework for tool development. MontiCore itself is a generator with the speciality that the products it produces are generators themselves. As already said, MontiCore is therefore a meta-tool.



Tip 1.1: Where to find MontiCore

The MontiCore language workbench as well as a number of language components are available as open source. More interesting information can be found at:

```
1 www.monticore.de // Newest info about
2 // MontiCore and the
3 // MontiCore generator
4 https://github.com/MontiCore // Sources of the core
5 // project on GitHub
```

At a glimpse, the features of MontiCore are:

- Modular definition of languages and language components
- Explicit interfaces between models, allowing heterogeneous composition of models.
- Techniques for composition of languages, thus allowing:
 - independent language development,
 - language extension,
 - language inheritance including concept replacement, and
 - composition of language tools.
- Assistance for model analysis.
- Assistance for model transformation by reusing the concrete syntax of the modeling language.
- Only a single source is necessary for the definition of concrete syntax, abstract syntax, parser and internal representation of models.
- Easy definition of e.g. Eclipse language specific editors.
- Explicit management of variability in both, languages and their generation tools.

Numerous tools for domain specific modeling languages as well as general purpose languages have been developed by using MontiCore. Among them are MontiCore itself, a larger part of the UML set of languages, the architectural description language MontiArc, Java, a subset of Ansi-C++ and a feature diagram DSL (see e.g. Figure 1.2). Various applications in engineering domains (AutoSar, autonomous driving simulation, flight control, building facilities, energy management, cloud service configuration) and natural science (human brain, control software for physical experiments) demonstrate the usability of MontiCore.

In addition to the above mentioned bullets this document also discusses:

- How a generator architecture looks like
- Out-of-the box use of MontiCore

- | | | | |
|--|--|---|---|
| <ul style="list-style-type: none"> ▪ MontiCore <ul style="list-style-type: none"> • Bootstrapping ▪ UML <ul style="list-style-type: none"> • Class diagrams • Object diagrams • Statecharts • Activity diagrams • Sequence diagrams • OCL ▪ MontiArc <ul style="list-style-type: none"> • Architectural models / ADL, function nets • + automata + Java + views ▪ Java <ul style="list-style-type: none"> • Java 5.0 ▪ C++ <ul style="list-style-type: none"> • Ansi-C++ | 


 | <ul style="list-style-type: none"> ▪ MontiCore transformations <ul style="list-style-type: none"> • Pattern matching • Extended by Java ▪ FeatureDSL <ul style="list-style-type: none"> • Feature diagram & config. ▪ AutosarDSL <ul style="list-style-type: none"> • Components, deployment, interfaces ▪ Flight control: constraint language ▪ Building facility specification ▪ Curriculum ▪ Cloud Service Configurator <ul style="list-style-type: none"> • Management of Services | 

 |
|--|--|---|---|

Figure 1.2: Some languages MontiCore provides

- Language definition
- How an abstract syntax (AST) looks like
- Managing symbols and visibility of definitions
- Model composition
- Language components and their composition
- Navigation an manipulation of the AST with compositional visitors
- Generation using FreeMarker's templates
- Integration of handwritten code

There is more to say about MontiCore. However, this document explicitly omits the internal architecture of MontiCore, how to define and apply transformations in concrete syntax, how to manage variability of languages, and various application languages, such as UML, MontiArc etc. The MontiCore website provides additional information.

1.2 Notational Conventions

Although MontiCore mainly relies on textual models, a diagrammatic representation is sometimes convenient. To be clear to what language the model, code snippet, etc. belongs to, it will be marked with a flag. An example is shown in the upper right corner of Listing 1.3, which is an excerpt of a generated Java class called `Person`.

We use various abbreviations, such as CD for class diagrams, etc. Especially class diagrams serve multiple purposes, therefore, it is necessary to understand precisely, what is modeled by a CD. In several chapters we use the modeling language *Class Diagrams for Analysis* (abbreviated: CD4A) as source for the generation process. But, we also use class diagrams

```

1 class Person {
2   private String firstname, surname;
3   Adress adr;
4 }

```

Java «gen» Person

Listing 1.3: Example in Java

to exhibit concrete situations in the tool or product, such as for example the extension of a generated class GroupTOP by the handcoded class Group, which are both present in the product, i.e., the final target of the development process. We notate this as depicted in Figure 1.4

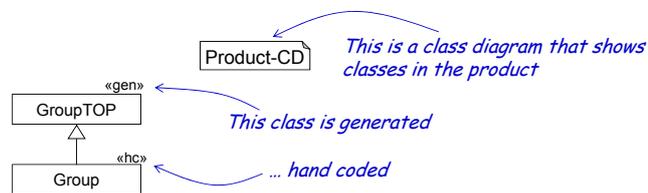


Figure 1.4: Notational conventions

1.3 Textual Modeling

Many experts think that the mental model in the conscious human brain is the most important form of models. Thus, it is not so important how to represent the modeling information on the screen or on paper, but that the model communicates the right information and concepts. However, for easy understanding, quick adaptations, logical manipulations, refactorings or similar purposes, it seems not so unimportant to use an appropriate representation.

It is an ongoing debate, whether and where textual or graphical models are better for software development. It also depends on the background of the reader, which model is easier to be used. Both forms of models do have advantages. Experts for example are quicker to produce the model in text form, because they are not distracted by "pushing boxes around" to produce nice diagrams. And for tool developers it is easier to write a text processor than a diagram processing tool, especially when using this handbook.

Our experience is that computer scientists tend towards textual models due to higher compactness, more efficient creation, refactoring and use and less dealing with graphical layout.

Diagrams and text will coexist in the future and may be even closely integrated. MontiCore currently focusses on text as the main form of input and output. Thus, the infrastructure is easier for tool development.

1.4 Methodical Considerations: Agile Modeling

There are a variety of development processes, ranging from traditional document oriented approaches, such as the V-Model, up to several incarnations of agile development, such as Extreme Programming (XP) [BA04] and Scrum [SB01]. It would go beyond the scope of this handbook to talk about methodological issues. However, we would like to hint towards discussions that a development job could well be assisted by models and high-level modeling languages, if the generation process for code and tests is efficient and robust. In [Rum11, Rum12, Rum16] this is discussed in detail.

[Rum12] for example suggests to combine the advantages of agile development with use of models by concentrating on a set of complementing models with as little redundancy as possible in order to represent each piece of information only once and as compact as possible. Figure 1.5 depicts this idea in an abstract form, mainly focussing on the UML.

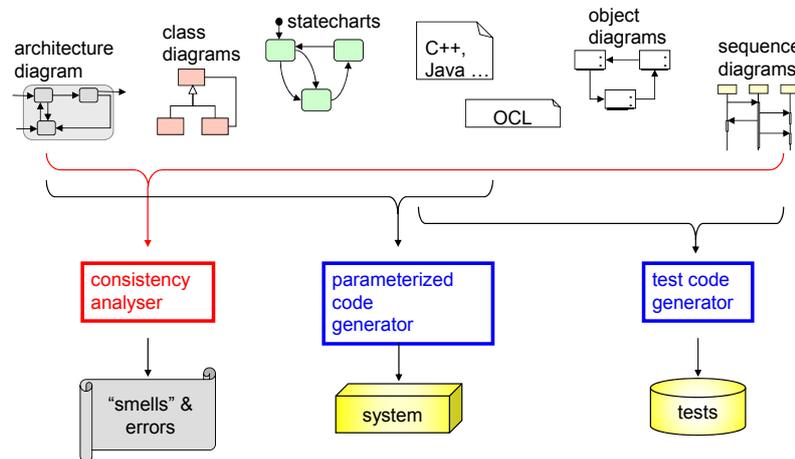


Figure 1.5: Agile use of models for coding and testing

Some generators concentrate on the system while other generators derive automatically running testing code similar to JUnit tests [Bec15]. If the software to be developed is part of a larger system, it would also be possible to derive automatically running simulations for the complete product or some of its components to check the correctness of the system, e.g. done in [BR12b].

As a consequence, we suggest to base larger parts of the development project on modeling artifacts. Models can be used for

- rapid prototyping,
- code generation,
- generation of automated tests,
- documentation,

1. Introduction to Tool Generation

- static analysis, and
- refactoring and evolution.

In a generative software development project, models serve as central artifacts. They are used for programming, testing and specifying.



Tip 1.6: Agile Model-Based Development

Agile software development and model-based generation fit together.

First and foremost, generation obviously increases the speed of development. However, this only becomes an advantage, when two important criteria are fulfilled:

(1) It is important to easily *rerun the generator* each time a slight change was made in the models. A one-shot generation is not helpful, because it does not assist any form of evolution, but only the waterfall model. So, it is best to not manually touch generated code.

(2) To keep the pace of development, generation must be *quick*. In particular when generating lots of code from lots of models, incremental generation based on detected changes is necessary. So, it is optimal to use an intelligent dependency management that allows automatic incremental and thus efficiently minimal re-generation.

Then agile generative software development becomes possible.



Tip 1.7: Current Version of this Document is Online

MontiCore is an evolving tool. Therefore, more material describing the capabilities and forms of usage will evolve over time.

Therefore, you might also have a look at Monticore's website.

However, your feedback will definitely be appreciated, e.g. by emails to monticore@se-rwth.de or through sending a printout with comments.

Chapter 2

Getting Started with MontiCore

This chapter describes the technical installation and usage of MontiCore for language developers. This chapter further inspects a simple example grammar and the Java classes and other artifacts generated from this grammar. After installing MontiCore as described in this chapter, it can be used to develop new modeling languages and generators as described in subsequent chapters.

MontiCore provides a command line interface (CLI) tool and can easily be used with Gradle. The Gradle integration enables developers to easily employ MontiCore in commonly used integrated development environments (IDEs), such as Eclipse and IntelliJ IDEA. We strongly recommend to work through the section about the CLI tool first. The CLI section contains information about an example MontiCore project and the files generated by MontiCore. It also shortly explains some key features of MontiCore.

A potentially newer explanation can be found on the MontiCore website. Detailed information about all configuration options that can be used in the MontiCore CLI tool and in MontiCore Gradle projects are explained in Chapter 16. More information about the example Automata language are available in Section 21.1.



Tip 2.1: MontiCore Website: Where to find MontiCore

The MontiCore language workbench as well as a number of language components are available as open source. More interesting information can be found at:

```
1 www.monticore.de/           // Newest info about
2                             // MontiCore and the
3                             // MontiCore generator
4
5 www.monticore.de/gettingstarted // Introductory tutorial
```

2.1 Prerequisites: Installing the Java Development Kit

We start with the JDK: Please perform the following steps to install the Java Development Kit (JDK) and validate that the installation was successful:



Tip 2.2: MontiCore Website: Best Practices

Many best practices for the development with MontiCore can be found on the website.

```
1 www.monticore.de/bestpractices
```

- Install a JDK with at least version 8 provided by Oracle or OpenJDK.
- Make sure the environment variable `JAVA_HOME` points to the installed JDK, and *not* to the JRE, e.g., the following would be good:

- `/usr/lib/jvm/java-8-openjdk` on UNIX or
- `C:\Program Files\Java\jdk1.8.*` on Windows.

You will need this in order to run the Java compiler for compiling the generated Java source files.

- Also make sure that the `PATH` system variable is set such that the Java compiler can be used from any directory. JDK installations on UNIX systems do this automatically. On Windows systems, the `bin` directory of the JDK installation needs to be appended to the `PATH` variable, e.g. `%PATH%;%JAVA_HOME%\bin .`
- Test whether the setup was successful. Open a command line shell in any directory. Execute the command `javac -version`. If this command is recognized and the shell displays the version of the installed JDK (e.g., `javac 1.8.0_192`), then the setup was successful.

Now we have the prerequisites to run MontiCore from the command line. The JDK installation is also required for using MontiCore with Gradle.

2.2 Install and Use the MontiCore Command Line Interface

This section describes instructions to perform the following first steps to use MontiCore as an CLI tool:

- Installation of the MontiCore distribution file.
- Grammar inspection
- Running the MontiCore generator
- Compiling the product
- Running the product, i.e. the Automata tool with an example model `example/PingPong.aut`.

2.2.1 Installation

For installing MontiCore, perform the following steps:

1. Download the example Automata MontiCore project:

```
1 // MontiCore zip distribution source
2 http://www.monticore.de/download/monticore.tar.gz
```

2. Unzip the archive. The unzipped files include a directory called `mc-workspace` containing the executable MontiCore CLI `monticore-cli.jar` along with a directory `src` containing handwritten Automata DSL infrastructure, a directory `hwc` containing handwritten code that is incorporated into the generated code, and a directory `example` containing an example model of the Automata DSL.

```
1 // MontiCore zip distribution content in directory mc-workspace
2 Automata.mc4
3 monticore-cli.jar
4 src/automata/AutomataTool.java
5 src/automata/visitors/CountStates.java
6 src/automata/prettyprint/PrettyPrinter.java
7 src/automata/cocos/AtLeastOneInitialAndFinalState.java
8 src/automata/cocos/StateNameStartsWithCapitalLetter.java
9 src/automata/cocos/TransitionSourceExists.java
10 hwc/automata/_ast/ASTState.java
11 hwc/automata/_symboltable/AutomatonSymbol.java
12 hwc/automata/_symboltable/AutomataSymbols2Json.java
13 hwc/automata/_symboltable/AutomatonSymbolDeser.java
14 hwc/automata/_symboltable/AutomataGlobalScope.java
15 example/PingPong.aut
```

2.2.2 Inspect the Example Grammar

MontiCore is a language workbench. It supports developers in developing modular modeling languages. The core of MontiCore is its grammar modeling language (cf. Chapter 4), which is used by developers for modeling context-free grammars. A MontiCore grammar defines (parts of) the abstract and concrete syntax of a language. Each grammar contains nonterminals, production rules, and may extend other grammars. At most one rule is marked as the start rule.

It is a *key feature of MontiCore* that it allows a grammar to *reuse and extend other grammars*. In an extension all of the nonterminals defined in the extended grammars can be reused or even overridden. This form of extension allows to achieve several effects:

- Language (i.e. grammar) components can be reused and integrated in larger languages, composed of several components.
- Individual nonterminals can be reused (like classes) from a library.

2. Getting Started with MontiCore

- A given language can be extended, which enables developers to add additional alternatives inside a language.

Component grammars and grammar extensions are detailedly discussed in Chapter 4.

```
1 grammar Automata extends de.monticore.MCBasics {
2
3   symbol scope Automaton =
4     "automaton" Name "{" (State | Transition)* "}" ;
5
6   symbol State =
7     "state" Name
8     ((("<<" ["initial"] ">>" ) | ("<<" ["final"] ">>" )) *
9     ( ("{" (State | Transition)* "}") | ";" ) ;
10
11  Transition =
12    from:Name "-" input:Name ">" to:Name ";" ;
13 }
```

Listing 2.3: The Automata grammar

In the following, we inspect the MontiCore grammar of the Automata language. Navigate your file explorer to the unzipped mc-workspace directory. The directory contains the file Automata.mc4. This file contains the MontiCore grammar depicted in Listing 2.3. MontiCore grammars end with .mc4.

The definition of a MontiCore grammar starts with the keyword `grammar`, followed by the grammar's name (l. 1). In this example, the grammar is called `Automata`. The grammar's name is optionally followed by the keyword `extends` and a list of grammars that are extended by the grammar. In this example, the `Automata` grammar extends the grammar `de.monticore.MCBasics`.



Tip 2.4: MontiCore Key Feature: Composition

The MontiCore language workbench allows to compose language components by composing grammars and also to reuse all infrastructure, such as context conditions, symbol table infrastructure, generator parts and handwritten extensions.

In the example the `Automata` grammar extends the grammar `de.monticore.MCBasics` and thus reuses its functionality.

MontiCore comes with an extensive library of predefined language components.

Grammars can also have a package and import other grammars. If a grammar has a package, then the package declaration must be the first statement in the grammar and is of the form `package QualifiedName` where `package` is a keyword and `QualifiedName` is an arbitrary qualified name (e.g., `de.monticore`). The optional grammar imports follow the package definition. Every import is of the form `import QualifiedName`. The `Automata` example grammar file does neither contain a package declaration nor imports. The grammar extended by the `Automata` grammar is specified by its fully qualified name.

```
1 automaton PingPong {
2     state NoGame <<initial>>;
3     state Ping;
4     state Pong <<final>>;
5
6     NoGame - startGame > Ping;
7
8     Ping - stopGame > NoGame;
9     Pong - stopGame > NoGame;
10
11    Ping - returnBall > Pong;
12    Pong - returnBall > Ping;
13 }
```

Listing 2.5: A model conforming to the Automata grammar

As usual in context-free grammars, production rules have a left-hand side and a right-hand side. The left-hand side contains the possibly annotated name of a nonterminal. The left-hand side is followed by the terminal = and the right-hand side. Nonterminal names start with an upper-case letter. For instance, the Automata grammar contains the nonterminals Automaton, State, and Transition. A single nonterminal can be annotated with the start keyword. Then, the nonterminal is the starting symbol of the grammar. If no nonterminal is annotated with start, then the first nonterminal of the grammar becomes the starting symbol by default. In the Automata grammar, the Automaton nonterminal is the starting symbol.

The other possible annotations for nonterminals influence the generated classes for the abstract syntax tree as well as the generated symbol table infrastructure. Details can be found in Chapter 4 and Chapter 9. For example, the Automaton nonterminal is annotated with symbol and scope. The annotation symbol makes the MontiCore generator generate a symbol class for the nonterminal. Intuitively stated, the annotation scope instructs MontiCore to construct a symbol table infrastructure that opens a scope when the production is processed. The following sections explain the effects of annotating the Automaton nonterminal with the keywords symbol and scope in more detail. Terminals are surrounded by quotation marks. The Automata grammar, for example, inter alia contains the terminals automaton, state, {, }, and ;.

The right-hand sides of grammar productions consist of nonterminals, terminals, and semantic predicates, may use cardinalities (*, +, ?), and introduce alternatives via the terminal | as known from regular expressions. Details can be found in Chapter 4. The right-hand side of the production defining the nonterminal Automaton, for example, uses the terminal automaton and the nonterminals Name, State, and Transition. The nonterminal Name is not defined in the grammar Automata. Thus, it must be defined in one of the extended grammars. In this case, Name is defined in the grammar MCBasics and is reused by the grammar Automata. For distinguishing different usages of nonterminals on right-hand sides, they can be named. For example, the right-hand side of the production defining the nonterminal Transition uses the Name nonterminal twice. The first usage is named input and the second usage is named to. MontiCore also supports

interface and external nonterminals for introducing extension points as detailedly described in Chapter 4. However, the example grammar does not use these concepts.

Listing 2.5 depicts an example model conforming to the Automata grammar in its concrete syntax. You can find the model in the file `PingPong.aut` contained in the `example` directory of the unzipped `mc-workspace` directory.

2.2.3 Run MontiCore

The MontiCore generator takes a MontiCore grammar as input and generates an infrastructure for processing models conforming to the grammar. When a grammar `E` extends another grammar `G`, then all of the infrastructure generated for the grammar `G` is reused and only the extending part from `E` is generated.



Tip 2.6: Infrastructure Generated by MontiCore

MontiCore itself as well as the infrastructure generated by the MontiCore generator are implemented in Java. This infrastructure includes:

- a *parser* for parsing models conforming to the grammar and transforming textual models into abstract syntax tree instances abstracting from the concrete syntax.
- a *symbol table infrastructure* to handle the symbols introduced or used by models conforming to the grammar. The symbol table infrastructure is used for resolving dependencies between model elements that are possibly defined in different files.
- a *context-condition checking framework* for checking well-formedness rules that cannot be captured by context-free languages.
- a *visitor infrastructure* for traversing models respectively their abstract syntax instances. The abstract syntax of a model consists of its internal representation as an abstract syntax tree abstracting from the concrete syntax of the model (the instance of the data structure obtained from parsing) and the symbol table of the model.
- a *mill infrastructure* for retrieving objects for language processing, such as parsers, builders for abstract syntax trees, visitors and objects for the symbol tables of the language. The possibility to configure the mills is crucial for reusing the functionality implemented for a sublanguage (cf. Section 5.9, Section 5.10.2, and Section 11.5 for details).
- a *code generating framework* that extends the FreeMarker template engine [Fre21] by various modularity enhancing features.

For executing MontiCore using the Automata grammar as input, perform the following two steps:

1. Open a command line shell and change the working directory to the unzipped directory (`mc-workspace`).
2. Execute the following command in order to generate the language infrastructure of the Automata DSL:

```
java -jar monticore-cli.jar Automata.mc4 -hcp hwc/
```



The only required argument `Automata.mc4` denotes the input grammar that shall be processed by MontiCore. The processing includes the generation of the language infrastructure. Using the option `-hcp` enables specifying the path to a directory containing the handwritten code that is to be incorporated into the generated infrastructure. In this case, passing the argument `hwc/` to the option `-hcp` makes MontiCore consider the handwritten code located in the directory `hwc/`. Providing handwritten code enables to easily incorporate additional functionality into the generated code. For example, this enables developers to extend generated abstract syntax classes as detailedly described in Section 5.10.

Executing the command launches MontiCore, which results in the executing of the following steps:

- a) The specified grammar is parsed and processed by MontiCore.
- b) Java source files for the corresponding DSL infrastructure are generated into the default output directory `out`. This infrastructure consists of the directories
 - `out/automata/` containing the mill (cf. Section 5.9, Section 5.10.2, Section 11.5).
 - `out/automata/_ast` containing the abstract syntax tree data structure (cf. Chapter 5).
 - `out/automata/_auxiliary` containing adapted mills of sublanguages, which are required for configuring the mills of sublanguages (cf. Chapter 11).
 - `out/automata/_cocos` containing the infrastructure for context conditions (cf. Chapter 10).
 - `out/automata/_od` containing the infrastructure for printing object diagrams for reports produced during processing the models.
 - `out/automata/_parser` containing the generated parsers, which are based on ANTLR (cf. Chapter 6).
 - `out/automata/_symboltable` containing the infrastructure for the symbol table (cf. Chapter 6).
 - `out/automata/_visitor` containing the infrastructure for visitors (cf. Chapter 9).
 - `out/reports/automata` containing reports created during the processing of the grammar.

- c) The output directory also contains a log file of the executed generation process `monticore.YYYY-MM-DD-HHmss.log` with the generation time in its name.

In the following, we review the classes and interfaces generated from the Automata grammar that are relevant for language engineers in more detail. We do not review the classes and interfaces that are only internally relevant for MontiCore and are usually not intended to be used by language engineers.

Abstract Syntax Tree Data Structure

The abstract syntax tree data structure is generated into the directory `out/automata/_ast`. Details about the generation of AST classes can be found in (cf. Chapter 5). For each nonterminal contained in the grammar, the MontiCore generator produces AST and corresponding builder classes. The AST classes implement the abstract syntax tree data structure.

The builder classes implement the builder pattern for constructing instances of the respective AST classes as usual. For example, the class `ASTAutomaton` is the AST class generated for the `Automaton` nonterminal (cf. Listing 2.3, l. 3) and the class `ASTAutomatonBuilder` is the corresponding generated builder class.

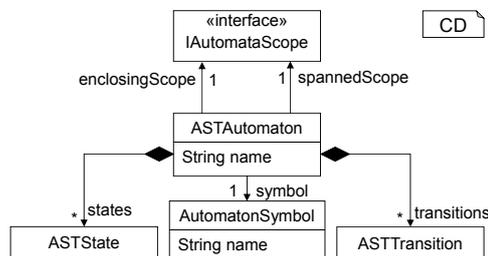


Figure 2.7: Parts of the AST data structure generated for the Automata grammar

The contents of the AST and builder classes are generated systematically from the grammar. The attributes of each AST class resemble the right-hand side of the corresponding production rule. In the following, we mainly speak of attributes, but please be aware that all attributes come fully equipped with access and modification methods, which should normally be used.

For instance, Figure 2.7 depicts parts of the generated AST infrastructure for the Automata grammar. The class `ASTAutomaton` contains the attributes `name`, `states`, and `transitions`. The AST class does not contain an attribute for the terminal automaton as it is part of every word conforming to the production of the `Automaton` nonterminal. The type of the attribute `name` is `String` whereas the attributes `states` and `transitions` are lists of the types of the AST classes corresponding to the used non-terminals. This is the case because exactly one `Name` is parsed with the right-hand side

of the production of the nonterminal `Automaton`, whereas multiple states and transitions can be parsed.

The `ASTAutomaton` class further contains the attributes `symbol`, `spannedScope`, and `enclosingScope`. These attributes are specific to the symbol table of `Automata` models and are used for linking the symbol table of a model with its abstract syntax tree. Details can be found in Chapter 9.



Tip 2.8: Generated Symbols and Scopes in the AST

Each AST class contains access to the `enclosingScope`.

When a production contains the keyword `symbol`, the generated AST class contains the attribute `symbol` (see Chapter 9).

Keyword `scope` indicates that a nonterminal also defines a new local scope, stored in attribute `spannedScope`.

The parser builds the abstract syntax tree of a model and the available scope genitor creates the symbol table of the model, consisting of symbols and scopes.

The `ASTAutomaton` class further contains several straight-forward methods for checking different instances for equality and accessing the attributes. Similar to the `ASTAutomaton` class, the `ASTAutomatonBuilder` class contains attributes resembling the right-hand side of the corresponding production. It further contains methods for changing the values of the attributes (e.g., `addState`), checking whether the AST instance that would be constructed from the current builder state is valid (cf. `isValid`), and for building the AST instance corresponding to the builder's state (cf. `build`). The contents of the other `AST` and `Builder` classes are constructed analogously.



Tip 2.9: Handwritten AST Class Extensions

If the generator detects that an AST class for a nonterminal is already implemented in the handwritten code, then it produces a corresponding `TOP` AST class instead.

This `TOP` mechanism allows developers to add handwritten extensions to any generated class, while reusing the generated `TOP` class via extension.

This gives a *very close integration* between handwritten and generated code that even adapts builders accordingly, while preventing the very bad habit of performing manual changes to the generated code.

Option `-hcp` tells the generator where to look for handwritten integrations.

The following section presents the methods of the classes for parsing textual models (possibly stored in files) into AST class instances at runtime. For now, it suffices for you to understand that (1) MontiCore generates an extensible AST data structure that resembles the nonterminals and productions of the grammar in a straight-forward way and (2) that all models of a grammar have an AST data structure representation for internal processing.

Parser

The parser infrastructure is generated into the directory `out/automata/_parser`. Details about the generated parsers and their uses are described in Chapter 6.

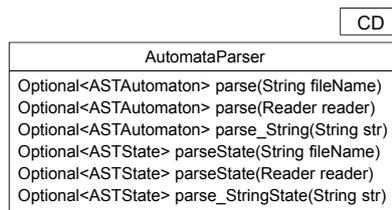


Figure 2.10: Parts of the class `AutomataParser`, which is generated from the Automata grammar

Parts of the generated class `AutomataParser` are depicted in Figure 2.10. The class implements the generated parser for the Automata grammar. Usually, developers are solely concerned with the methods `parse(String)` and `parse_String(String)`. For now, it suffices if you remember that parsing textual Automata models stored in files is possible by calling the method `parse(String)` of an `AutomataParser` object with the fully qualified name of the file as input.



Tip 2.11: Methods for Parsing

The class `AutomataParser` contains the methods

- `parse(Reader r)`,
- `parse(String filename)`, and
- `parse_String(String content)`.

All of the methods return an object of type `Optional<ASTAutomaton>`, where absence means failure of parsing and errors have been issued.

For each nonterminal in the grammar, the class further contains methods for parsing a sub-model described by this nonterminal.

Symbol Table

The symbol table infrastructure is generated into the directory `out/automata/_symboltable`. Details about the generated symbol table infrastructure and its use are described in Chapter 9. The symbol table infrastructure is used for resolving cross-references concerning information defined in different model elements that are potentially defined in different models stored in different files.

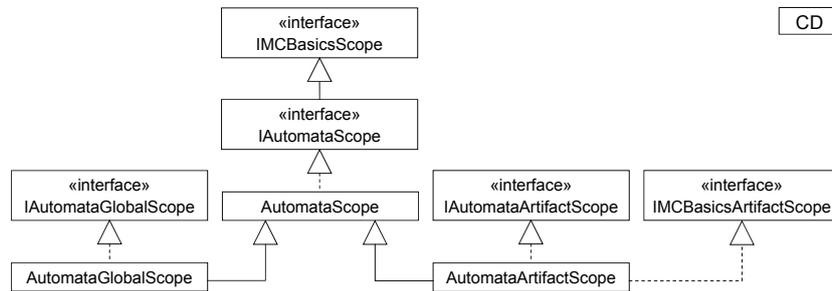


Figure 2.12: The scope classes generated from the Automata grammar



Tip 2.13: Scope Classes

For the Automata grammar, the generator produces the classes

- AutomataScope,
- AutomataArtifactScope, and
- AutomataGlobalScope

as well as respective interfaces. The relationships between these classes and interfaces are depicted in Figure 2.12.

The singleton AutomataGlobalScope contains all AutomataArtifactScopes of all loaded Automata artifacts. AutomataScopes represent scopes spanned inside of models.

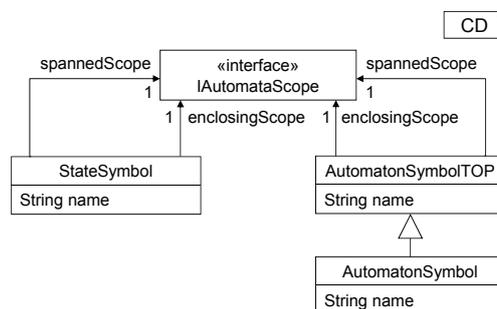


Figure 2.14: Parts of the symbol classes generated from the Automata grammar

Figure 2.14 depicts parts of the symbol classes generated for the Automata grammar. As the nonterminal State is annotated with symbol in the Automata grammar, the

2. Getting Started with MontiCore

generator produces the class `StateSymbol`. The `StateSymbol` class, inter alia, contains the attributes `name`, `enclosingScope`, and `spannedScope`. The attribute `name` stores the name of the symbol. The attributes `enclosingScope` and `spannedScope` store the enclosing and spanned scopes of the symbol. The class further contains methods for accessing and setting the attributes. For all symbol classes, the MontiCore generator also produces builder classes (e.g., `AutomataArtifactScopeBuilder` and `StateSymbolBuilder`).



Tip 2.15: Extending Symbol Classes

It is possible to add further methods and attributes in two ways:

- adding a *symbol rule* in the grammar (described in Chapter 9) or
- using the *TOP mechanism* applied to the generated symbols.

The generated class `AutomataScopesGenitor` is responsible for creating the scope structure of Automata artifacts and linking the scope structure with the corresponding AST nodes. For this task, it provides the method `createFromAST` that takes an `ASTAutomaton` instance as input and returns an `IAutomataArtifactScope` instance. The returned `IAutomataArtifactScope` instance can be added as a subscope to the (during runtime unique and administrated by the mill) `AutomataGlobalScope` instance.

Developers can create visitors for complementing the symbol table (creating symbols and filling the extensions introduced via symbol rules or the TOP mechanism) of an Automata artifact. After creating the scope structure, the visitor should be used to traverse the AST instance of the artifact for complementing the symbols and scopes. The following sections explain the generated visitor infrastructure in more detail.

```
1 Optional<AutomatonSymbol> resolveAutomaton(String name)
2 List<AutomatonSymbol> resolveAutomatonMany(String name)
3 Optional<StateSymbol> resolveState(String name)
4 List<StateSymbol> resolveStateMany(String name)
```



Listing 2.16: Different `resolve` methods

For each nonterminal annotated with `symbol` in the grammar Automata, the scope interfaces contain a symbol-specific `resolve` method taking a string as input. The method can be called to resolve symbol instances by their names. The name given as input to a `resolve` method should be as qualified as needed to find the symbol. For instance, Listing 2.16 lists the signatures of four of the `resolve` methods provided by the interface `IAutomataScope`.

For now, it suffices for you to understand that (1) MontiCore generates an extensible symbol table data structure that resembles the scope and symbol structure as specified in the grammar in a straight-forward way and (2) that all models of a grammar have a symbol table data structure representation for internal processing and (3) that symbols can be resolved from scopes via calling the `resolve` methods.

(De)Serialization of Symbol Tables

MontiCore also supports the serialization and deserialization of symbol tables. The (de)serialization is crucial for incremental code generation and efficient language composition via aggregation. Details about this are explained in Chapter 7 and Chapter 9.

For the (de)serialization, the generator produces the class `AutomataSymbolToJson`. It provides the public methods `store` and `load`. The former can be used to serialize `IAutomataScope` instances into their string representations encoded in JSON and persisting these to a file at a location that is passed as method argument. The latter can be used to load a stored `IAutomataScope` into its objects representation. For now, it suffices that you understand which methods to call for the (de)serialization.

Visitor

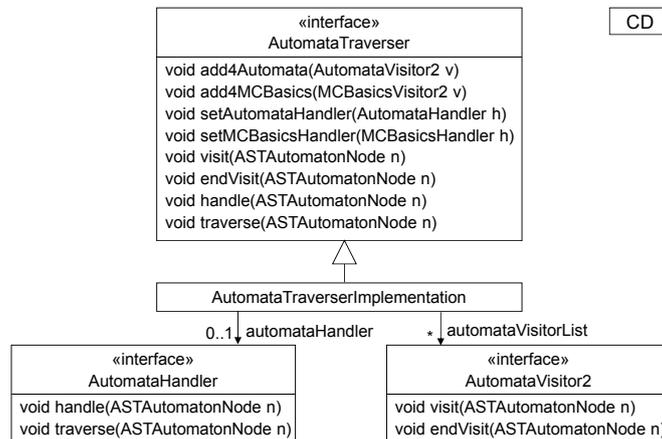


Figure 2.17: Parts of the visitor infrastructure generated from the Automata grammar

The visitor infrastructure is generated into the directory `out/automata/_visitor`. Details about the generated visitor infrastructure are described in Chapter 8. For each grammar, the generator systematically produces several classes and interfaces implementing the visitor infrastructure. For the Automata grammar, for example, the generator produces the interfaces `AutomataTraverser`, `AutomataVisitor2`, and `AutomataHandler` and the class `AutomataTraverserImplementation`. The relationships between these interfaces and classes are depicted in Figure 2.17.

The interfaces `Traverser`, `Visitor2` and `Handler` together realize the Visitor pattern. Conceptually, the traverser is the entry point for traversing. The traverser manages visitors for the different sublanguages and realizes the default traversing strategy. Whenever an AST node is traversed, the traverser delegates the visit to the corresponding visitor implementation. If a special traversal is to be implemented that differs from the default, it is possible to add handlers to the traverser that realize the alternative traversal. For a more detailed explanation consider reading Chapter 8.



Tip 2.18: Visitors

MontiCore provides the visitor pattern in a detangled and thus flexible variant. `AutomataTraverser` is traversing the AST. `AutomataVisitor2` contain the actual functionalities, added through subclassing. Many visitors can be added to the traverser for parallel execution via the method `add4Automata`.

The visitors are compositional, allowing to maximize reuse of visitors from sub-languages, and they can be adapted through the TOP mechanism.

For example, the handwritten class `PrettyPrinter`, which can be found in the directory `mc-workspace/src/automata/prettyprint`, implements functionality for pretty printing an Automata model, which is given by its abstract syntax tree. Listing 2.19 depicts the attributes and the constructor of the class. The `PrettyPrinter` class implements the `AutomataHandler` interface. Its constructor instantiates a printer (a helper for printing indented strings) and retrieves an `AutomataTraverser` object from the mill (which is explained later on). It sets the handler of the traverser to itself. This ensures that the pretty printer becomes the handler of the traverser. We will execute it in a following section.

```
1 public class PrettyPrinter implements AutomataHandler {
2     private final IndentPrinter printer;
3     private AutomataTraverser traverser;
4
5
6     public PrettyPrinter() {
7         this.printer = new IndentPrinter();
8         this.traverser = AutomataMill.traverser();
9         traverser.setAutomataHandler(this);
10    }
11 }
```

Listing 2.19: Attributes and constructor of the `PrettyPrinter` for the Automata language

For now, you should understand that (1) for implementing visitors it is often sufficient to implement the visitor interfaces and to add them to a traverser and (2) custom traversals can be realized by implementing handlers and adding those to the traverser.

Context Conditions

The context condition infrastructure is generated into the directory `out/automata/_cocos`. Details about the generated context condition infrastructure are described in Chapter 10.

For each nonterminal of a grammar, the generator produces a context condition interface for implementing context conditions for this nonterminal. For the Automata grammar, for

example, the generator produced the interface `AutomataASTStateCoCo`. The interface solely contains the method `check(ASTState)`. Each class implementing the interface should represent a predicate over subtrees of abstract syntax trees starting at a node with the type corresponding to the nonterminal.

The `check` method should be implemented such that it reports an error or a warning if the input node does not satisfy the predicate. Thus, context conditions implement well-formedness rules that cannot be captured by context-free grammars (or that are intentionally not captured by the grammar to achieve a specific AST data structure). For producing the error or warning, the static methods `error` and `warning` of the MontiCore runtime class `Log` should be used.

For the `Automata` grammar, the generator also produced the class `AutomataCocoChecker`. For each nonterminal of the grammar, the class contains a method for adding context condition instances to an `AutomataCocoChecker` instance. For checking whether an AST node satisfies all registered context conditions, the method `checkAll` can be called with the AST node as input. Calling the method makes the checker traverse the abstract syntax tree and check whether each node satisfies the context conditions registered for the node. Thus, `AutomataCocoChecker` instances represent sets of context conditions that are required to be satisfied by abstract syntax tree instances.

For now, you should understand that (1) implementing context conditions is possible via implementing the generated `CoCo` interfaces and (2) context conditions can be checked via instantiating the `Checker` class, adding the `CoCos`, and calling the `checkAll` method.

Mill as Factory for Builders

The mill for the `Automata` language is generated into the directory `out/automata/`. Details about the generated mill and the mill pattern in general are described in Section 11.5. The generated mill class `AutomataMill` is responsible for providing ready to use and correct parser, scope genitor, scope, and builder instances. The mill of each language is a singleton.



Tip 2.20: Mill Use and Automatic Initialization

A mill is a factory for builders and other commonly used functions, such as parsers or visitors. The mill was introduced to ensure compositionality of languages, while retaining reusability of functions developed for sublanguages.

Only one mill instance exists, even though in composed languages it is available under several static signatures. Let language `G2` extend another language `G1`. Then `G2Mill` initializes the `G1Mill` appropriately, such that all of the code of the sublanguage `G1` can be reused in the tools developed for the language `G2`, even when creating new AST nodes, symbols, etc.

Cool mechanism and the developers don't have to bother.

```
1  public static IAutomataGlobalScope globalScope ()
2  public static IAutomataArtifactScope artifactScope ()
3  public static IAutomataScope scope ()
4  public static AutomataScopesGenitor scopesGenitor ()
5  public static AutomataScopesGenitorDelegator
6                               scopesGenitorDelegator ()
7  public static ASTAutomatonBuilder automatonBuilder ()
8  public static AutomatonSymbolBuilder automatonSymbolBuilder ()
9  public static AutomataParser parser ()
10 public static AutomataTraverser traverser ()
```

Listing 2.21: Some methods of the AutomataMill API

Developers should retrieve all instances of the classes and interfaces provided by the mill by using the mill. Instances of the classes and interfaces that are provided by the mill should never be instantiated manually. Otherwise, it may be the case that not all of the code implemented for the language can be reused as expected in other languages extending the language. Listing 2.21 shows some signatures of the methods of the AutomataMill.



Tip 2.22: Mill Methods

A mill provides `public static` methods for retrieving the instances of the parsers, scope genitors, scopes, and builders. For that it acts like a factory. Because a mill is realized using the static delegator pattern (cf. 11.1), it still can be adapted at will.

This combines the advantage of general availability with the advantage of being able to override the functions.

For now, you should understand that (1) the methods of the mill should be used for creating ready to use and correct parser, scope genitor, scope, and builder instances and (2) how to call these methods.

2.2.4 Compile the Target

Section 2.2.3 describes how to generate the desired Java code from a MontiCore grammar. For compiling these Java classes, generated for the Automata DSL, execute the following command:

```
javac -cp monticore-cli.jar -sourcepath "src/;out/;hwc/" \
                                             src/automata/AutomataTool.java
```

Please note, on Unix systems paths are separated using ":" (colon) instead of semicolons.

Providing the option `-cp` with the argument `monticore-cli.jar` makes the Java compiler consider the compiled MontiCore runtime classes contained in the file `monticore-cli.jar`.

The option `-sourcepath` enables to specify paths to directories containing the source files that should be considered during the compilation.

In this case, executing the command makes the Java compiler consider all generated classes located in `out` and all handwritten classes located in `src` and `hwc`. The last argument instructs the Java compiler to compile the class `src/automata/AutomataTool.java`.

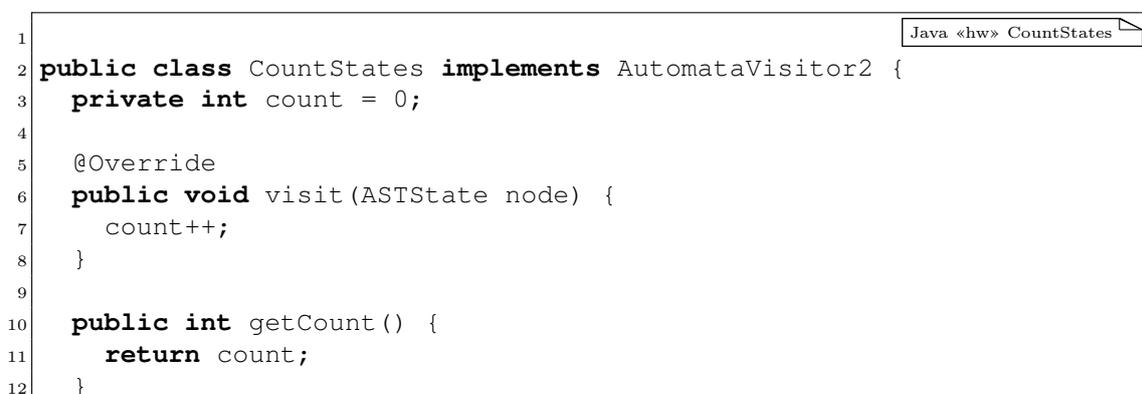
Please note that the structure of the handwritten classes follows the package layout of the generated code, i.e. there are the following subdirectories (Java packages):

- `src/automata` contains the top-level language realization for using the generated DSL infrastructure. In this case the class `src/automata/AutomataTool.java` constitutes a main class executable for processing automata models with the automata DSL.
- `src/automata/cocos` contains infrastructure for context condition of the automata DSL.
- `src/automata/prettyprint` contains an exemplary use of the generated visitor infrastructure for processing parsed models for pretty printing.
- `src/automata/visitors` contains an exemplary analysis using the visitor infrastructure. The exemplary analysis counts the states contained in the parsed automata model.
- `hwc/automata/_ast` contains an exemplary usage of the handwritten code integration mechanism for modifying the AST for the automata DSL. Details about the integration mechanism are described in Section 5.10.
- `hwc/automata/_symboltable` contains handwritten extensions of the generated symbol table infrastructure. Details about implementing handwritten symbol table infrastructure extensions are described in Chapter 9.

Please, also do not mix the code for the Automata tool vs. the code for the final product, generated from that tool, although both have a similar package structure.

We already described the contents of the directories `hwc/automata/_ast` and `hwc/automata/_symboltable` in the previous section. They contain handwritten extensions of the abstract syntax of the Automata language.

```
1 public class CountStates implements AutomataVisitor2 {
2     private int count = 0;
3
4     @Override
5     public void visit(ASTState node) {
6         count++;
7     }
8
9     public int getCount() {
10        return count;
11    }
12 }
```



2. Getting Started with MontiCore

```
13 }
```

Listing 2.23: The CountStates visitor implementation

The directory `src/automata/visitors` contains the file `CountStates.java`. The class is depicted in Listing 2.23. It implements a simple visitor for counting the number of states contained in an Automata model. To this effect, it implements the `AutomataVisitor2` interface. It has an attribute `count` of type `int` for storing the current number of counted nodes. It overrides the `visit` method for `ASTState` to increase the counter whenever a state is visited.

The directory `src/automata/cocos` contains the context-condition implementations for the Automata language.

```
1                                     Java «hw» AtLeastOneInitialAndFinalState
2 public class AtLeastOneInitialAndFinalState
3     implements AutomataASTAutomatonCoCo {
4     @Override
5     public void check(ASTAutomaton automaton) {
6         boolean initialState = false;
7         boolean finalState = false;
8
9         for (ASTState state : automaton.getStateList()) {
10            if (state.isInitial()) {
11                initialState = true;
12            }
13            if (state.isFinal()) {
14                finalState = true;
15            }
16        }
17
18        if (!initialState || !finalState) {
19            // Issue error...
20            Log.error("0xA0116 An automaton must have at least one "
21                    + "initial and one final state.",
22                    automaton.getSourcePositionStart());
23        }
24    }
25 }
```

Listing 2.24: Context condition implementation for checking that there exist at least one initial and at least one final state

Listing 2.24 depicts the class `AtLeastOneInitialAndFinalState`. The class implements a context condition for checking whether an Automata model contains at least one initial and at least one final state. To this effect, the class implements the interface `AutomataASTAutomatonCoCo`. The class `StateNameStartsWithCapitalLetter` is implemented similarly.

```

1 public class TransitionSourceExists Java «hw» TransitionSourceExists
2     implements AutomataASTTransitionCoCo {
3
4     @Override
5     public void check(ASTTransition node) {
6
7         IAutomataScope enclosingScope = node.getEnclosingScope();
8         Optional<StateSymbol> sourceState =
9             enclosingScope.resolveState(node.getFrom());
10
11         if (!sourceState.isPresent()) {
12             // Issue error...
13             Log.error(
14                 "0xADD03 Source state of transition missing.",
15                 node.getSourcePositionStart());
16         }
17     }
18 }

```

Listing 2.25: Context condition implementation for checking that states used in transitions exist

Listing 2.25 presents the implementation of the class `TransitionSourceExists`. The class implements a context condition for checking whether the source states used in transitions are defined. To this effect, the class uses the resolving mechanisms of the symbol table. For each transition, the context conditions tries to resolve the state symbol corresponding to the source state of the transition. If the resolving fails for the state, then the context condition logs an error.

The class `AutomataTool` is the main class of the Automata language. It is defined in the file `AutomataTool.java` contained in the directory `src/automata`.

```

1 public ASTAutomaton parse(String model) { Java «hw» AutomataTool
2     try {
3         AutomataParser parser = new AutomataParser();
4         Optional<ASTAutomaton> optAutomaton = parser.parse(model);
5
6         if (!parser.hasErrors() && optAutomaton.isPresent()) {
7             return optAutomaton.get();
8         }
9         Log.error("0xEE840 Model could not be parsed.");
10    }
11    catch (RecognitionException | IOException e) {
12        Log.error("0xEE640 Failed to parse " + model, e);
13    }
14    System.exit(1);
15    return null;
16 }
17
18 public IAutomataArtifactScope createSymbolTable(ASTAutomaton ast) {

```

2. Getting Started with MontiCore

```
19
20   IAutomataGlobalScope globalScope = AutomataMill.globalScope();
21   globalScope.setModelPath(new ModelPath());
22   globalScope.setFileExt("aut");
23
24   AutomataScopesGenitorDelegator symbolTable = AutomataMill
25       .scopesGenitorDelegator();
26
27   return symbolTable.createFromAST(ast);
28 }
```

Listing 2.26: Methods for parsing and creating symbol tables

Listing 2.26 presents the implementation of the methods `parse` and `createSymbolTable` of the `AutomataTool` class. The methods can be used for parsing and creating symbol tables for Automata. The methods also demonstrate the usage of the mill for retrieving global scopes and genitors.

```
1 public static void main(String[] args) { Java «hw» AutomataTool
2     // delegate main to instantiatable method for better integration,
3     // reuse, etc.
4     new AutomataTool().run(args);
5 }
6
7 public void run(String[] args) {
8     // use normal logging (no DEBUG, TRACE)
9     Log.ensureInitialization();
10
11     // Retrieve the model name
12     if (args.length != 2) {
13         Log.error("0xEE7400 Arguments are: (1) input "
14             + "model and (2) symbol store.");
15         return;
16     }
17     Log.info("Automata DSL Tool", "AutomataTool");
18     String model = args[0];
19
20     // parse the model and create the AST representation
21     ASTAutomaton ast = parse(model);
22     Log.info(model + " parsed successfully!", "AutomataTool");
23
24     // setup the symbol table
25     IAutomataArtifactScope modelTopScope =
26         createSymbolTable(ast);
27
28     // can be used for resolving names in the model
29     Optional<StateSymbol> aSymbol =
30         modelTopScope.resolveState("Ping");
31
32     if (aSymbol.isPresent()) {
33         Log.info("Resolved state symbol \"Ping\"; FQN = "
```

```

34         + aSymbol.get().toString(),
35         "AutomataTool");
36     } else {
37         Log.info("This automaton does not contain a state "
38             + "called \"Ping\";", "AutomataTool");
39     }
40
41     // setup context condition infrastructure
42     AutomataCoCoChecker checker = new AutomataCoCoChecker();
43
44     // add a custom set of context conditions
45     checker.addCoCo(new StateNameStartsWithCapitalLetter());
46     checker.addCoCo(new AtLeastOneInitialAndFinalState());
47     checker.addCoCo(new TransitionSourceExists());
48
49     // check the CoCos
50     checker.checkAll(ast);
51
52     // Now we know the model is well-formed and start backend
53
54     // store artifact scope and its symbols
55     AutomataSymbols2Json deser = new AutomataSymbols2Json();
56     deser.store(modelTopScope, args[1]);
57
58     // analyze the model with a visitor
59     CountStates cs = new CountStates();
60     AutomataTraverser traverser = AutomataMill.traverser();
61     traverser.add4Automata(cs);
62     ast.accept(traverser);
63     Log.info("Automaton has " + cs.getCount() + " states.",
64         "AutomataTool");
65
66     // execute a pretty printer
67     PrettyPrinter pp = new PrettyPrinter();
68     AutomataTraverser traverser2 = AutomataMill.traverser();
69     traverser2.setAutomataHandler(pp);
70     ast.accept(traverser2);
71     Log.info("Pretty printing automaton into console:",
72         "AutomataTool");
73     // print the result
74     Log.println(pp.getResult());
75 }

```

Listing 2.27: Main method of the AutomataTool class

The AutomataTool provides a main method, which can be called from the command line. The implementation of the method is depicted in Listing 2.27. It expects two inputs. The first is the name of a file containing an Automata model. The second input is the name of the file in which the tool should store the symbol table of the model that is given as first input.

The method

- parses the input model (l. 21),
- creates the symbol table (l. 25),
- resolves a state (l. 29),
- executes context conditions (ll. 41-50),
- stores the symbol table by using the serialization (ll. 54-56),
- executes the visitor for counting the states (ll. 59-64), and
- pretty prints the model to the standard output (ll. 66-74).

Inspect the main method and try to understand the implementation for the executed tasks. Read the above descriptions again if necessary.

2.2.5 Run the Tool

The previous command compiles the handwritten and generated code including the Automata tool class `AutomataTool`. For running the Automata DSL tool, execute the following command:

```
java -cp "src/;out/;hwc/;monticore-cli.jar" \
      automata.AutomataTool example/PingPong.aut \
      st/PingPong.autsym
```

Please note again, on Unix systems paths are separated using ":" (colon) instead of semi-colons. Executing the command runs the Automata DSL tool.

Using the option `-cp` makes the Java interpreter consider the compiled classes contained in the paths specified by the argument.

The argument `automata.AutomataTool` makes the Java interpreter execute the main method of the class `automata.AutomataTool` contained in the directory `src`.

The argument `example/PingPong.aut` is passed to the main method of the Automata DSL tool class as input. Inspect the output on the command line, which displays log messages concerning the processing of the example Automata model.

The last argument `st/PingPong.autsym` is also passed to the main method. It makes the tool store the serialized symbol table of the input model into the file `example/PingPong.aut`.

The shipped example Automata DSL (all sources contained in `mc-workspace/src` and `mc-workspace/hwc`) can be used as a starting point for creating your own language. It can easily be altered to specify your own DSL by adjusting the grammar and the handwritten Java sources and rerunning MontiCore as described above.

2.3 Using MontiCore via Gradle From the Command Line

It is possible to execute MontiCore via the MontiCore Gradle plugin. A detailed description about using the MontiCore Gradle plugin is given in Chapter 16. This section describes the execution of MontiCore via a Gradle plugin from the command line shell by example.

First, install Gradle via executing the instructions mentioned on the following website and make sure that the `PATH` system variable is set such that the `gradle` command can be used from any directory:

```
1 // Gradle installation
2 https://gradle.org/install/
```

The shipped example Automata DSL can be used as a starting point and can be downloaded here:

```
1 http://www.monticore.de/download/Automaton.zip
```

The build script (file `build.gradle`) can easily be adapted for creating build scripts for other languages. For executing MontiCore via the Gradle plugin from the command line shell by example of the Automata DSL, perform the following steps:

1. Download the Automata example (cf. Listing 2.3).
2. Unzip the downloaded zip file into an arbitrary directory.
3. Open a shell and change your working directory to the directory in which you unzipped the downloaded file (the directory containing the file `build.gradle`).
4. Execute Gradle in the shell:
 - If you are using a Windows shell, execute the command `gradle build`.
 - If you are using a Unix shell, execute the command `./gradle build`.

When executing the above commands, MontiCore launches, which results in the execution of the following steps:

1. The grammars specified in the `build.gradle` are incrementally parsed and processed by MontiCore.
2. Java source files for the corresponding DSL infrastructure are generated into the default output directory `../target/generated-sources/monticore/sourcecode`. The contents of this generated directory are equal to the contents of the generated directory out as described in Section 2.2.3.

2.4 Using MontiCore in Eclipse

The MontiCore Gradle plugin can be used in Eclipse. Section 2.4.1 describes the process of setting up Eclipse. Section 2.4.2 presents how to import the example project in Eclipse. Finally, Section 2.4.3 explains how the MontiCore Gradle plugin can be executed in Eclipse.

2.4.1 Setting up Eclipse

Before you import the example project and run MontiCore as a Gradle plugin, please make sure that a current version of the Gradle plugin is installed in Eclipse. When installing a new version of Eclipse, the Gradle plugin is installed by default. If the Gradle plugin is not yet integrated into your Eclipse installation, download the latest Eclipse version or perform the following steps to install the Eclipse plugin:

1. Download and install Eclipse (or use an existing one).
2. Open Eclipse.
3. Install the needed Plugins.
 - Help > Eclipse Marketplace...
 - Type 'gradle' in the search box and click Enter.
 - Install the 'Buildship Gradle Integration' plugin.
4. Make sure to configure Eclipse to use an JDK instead of an JRE.
 - Window > Preferences > Java > Installed JREs.

2.4.2 Importing the Example

The shipped example Automata DSL can be used as a starting point. Once imported into Eclipse, it can easily be altered to specify your own DSL by adjusting the grammar and the handwritten Java sources and rerunning MontiCore as described in Section 2.4.3. To import the example, perform the following steps:

1. Download and unzip the Automata example (cf. Listing 2.3)
2. Open Eclipse and select
 - File > Import > Gradle (if you are required to choose a Gradle version, then choose version 6.7.1) > Existing Gradle Projects > Next.
 - Click on the *Browse..* button and import the directory that contains the file `build.gradle` from the Automata example.

2.4.3 Running MontiCore

To execute the MontiCore Gradle plugin, perform the following steps:

- Select the *Gradle Task* menu (at the top or bottom, depending on your installed Eclipse version).
- There select `automaton > build > build` (double click).

This makes Eclipse execute the MontiCore Gradle plugin as described in Section 2.3. After installing and executing MontiCore in Eclipse, your workspace should look similar to Figure 2.28.

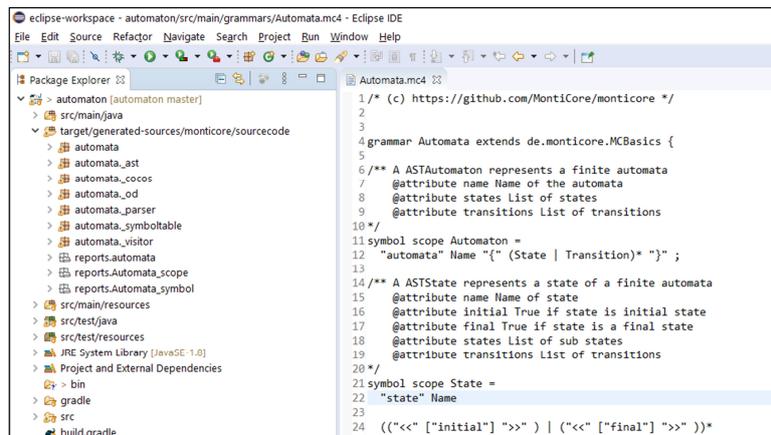


Figure 2.28: Eclipse after importing the example project and executing MontiCore

2.5 Using MontiCore in IntelliJ IDEA

The MontiCore Gradle plugin can be used in IntelliJ IDEA. Section 2.5.1 describes the process of setting up IntelliJ IDEA. Afterwards, Section 2.5.2 presents how to import the example project in Eclipse. Finally, Section 2.5.3 explains how the MontiCore Gradle plugin can be executed in IntelliJ IDEA.

2.5.1 Setting up IntelliJ IDEA

For setting up IntelliJ IDEA, perform the following steps:

1. Download and install IntelliJ IDEA (or use your existing installation).
 - Hint for Students: You get the Ultimate version of IntelliJ IDEA for free.
2. Open IntelliJ IDEA.

2.5.2 Importing the Example

The shipped example Automata DSL can be used as a starting point. Once imported into IntelliJ IDEA, it can easily be altered to specify your own DSL by adjusting the grammar and the handwritten Java sources and rerunning MontiCore as described in Section 2.5.3. For importing the example, perform the following steps:

1. Download and unzip the Automata Example (cf. Listing 2.3).
2. In the IDE select: File > Open.
3. Select the directory containing the build.gradle (if you are required to choose a Gradle version, then choose version 6.7.1).

2. Getting Started with MontiCore

2.5.3 Running MontiCore

To execute the MontiCore Gradle plugin, perform the following steps:

- Select the Gradle Projects menu on the right.
- From there select automaton > Tasks> build > build (double click).

This makes IntelliJ IDEA execute the Gradle plugin as described in Section 2.3. If you do not see the Gradle Projects menu yet, right-click on the build.gradle file and select 'Import Gradle Project'. Now the Gradle Projects menu should occur on the right side and you can follow the above mentioned steps for the execution. After installing and executing MontiCore in IntelliJ IDEA, your workspace should look similar to Figure 2.29.

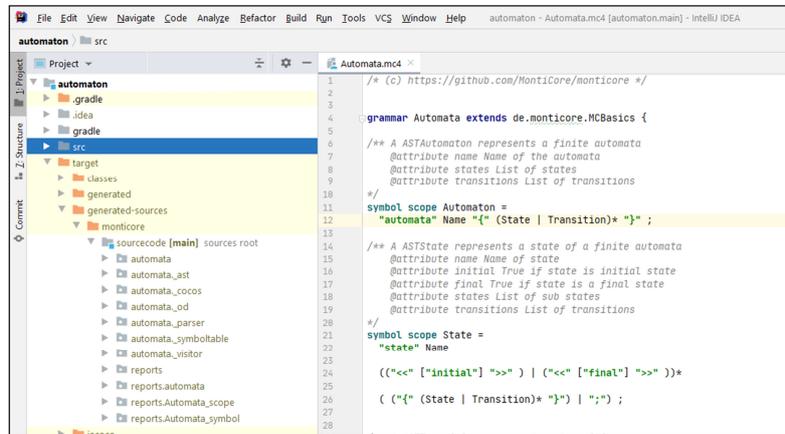


Figure 2.29: IntelliJ IDEA after importing the example project and executing MontiCore

Chapter 3

Architecture of a Model Processor

This chapter provides an overview of the standard architecture of the MontiCore tool, including which features and capabilities it provides. This is a prerequisite for understanding the definition, usage and adaptation of model-based tool processors.

Note that MontiCore as well as its components and derivatives are not exclusively used for generation, but also for deep analysis techniques, general transformations or language interpretation at runtime. This chapter focusses on the widely used generative aspect.

Familiarity with the concepts of *model*, *modeling language*, *model transformation* and *generator*, as discussed in [CFJ⁺16, Rum16, Rum17], is a prerequisite to understand the contents of this chapter.

3.1 Structure of a Model Processor - External View

There are many forms of model processing, but a typical processor would function like this: firstly process one or more models, then apply some internal transformations to them in order to produce related artifacts. Often these are partially or even completely executable programs written in a General Purpose Language. It is also possible to produce models of another language, websites in HTML, relevant documentation, overview drawings, or proof obligations to be handled by a model checker or verifier.

The generated code typically makes use of the generators operating system, some existing frameworks and other platform specific code. However, generators need high flexibility and intelligence in order to enable the incorporation of handwritten code, platform specific adaptations, predefined components, and potentially even project or user specific preferences.

Classical compilers embed concepts to manage these features within a programming language through the import of external frameworks, compiler directives or a macro preprocessor. Model-based generation contrast this by normally only carrying domain knowledge in the model, whilst the *parameterized generator* adds technical details and allows for filling of adaptation points in form of *generator scripts* and *templates*.

Figure 3.1 shows the external view of a generator along with the artifacts used and produced. Within a model-based generative project, people may adopt different roles in order to provide and use artifacts within it.

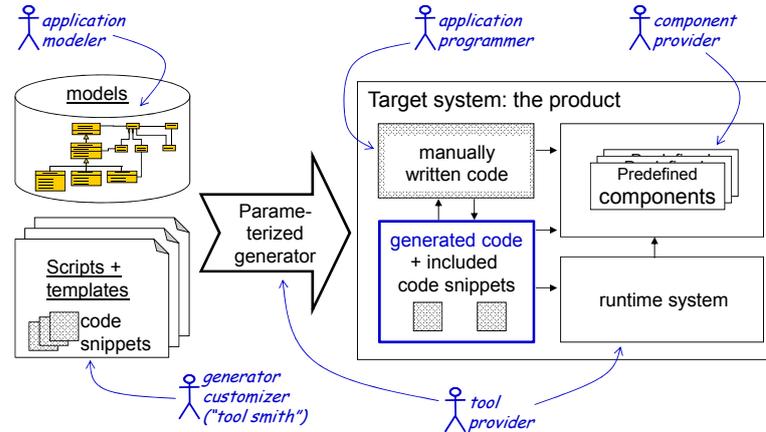


Figure 3.1: Structure of a generator - external view

The *tool provider* develops the generator as well as the runtime system that interacts with the generated code.

The *tool smith* customizes the generator. The customization can be used to add many project-specific scripts in order to orchestrate code generation or to introduce templates which provide code snippets to be copied into the generated code. Both, scripts and templates, are usually dependent on the target platform, operating system, hardware, frameworks, and external components included into the system, etc. Unlike scripts and templates, models only contain application or domain specific information, but are independent of the target technology.

3.2 Internal Architecture of a Generator - Component View

A generator is typically decomposed into several components as shown in Figure 3.2.

A *model loader* handles the loading of all needed models and their subsequent transformation into an internally accessible structure. The MontiCore language workbench mainly uses parsers, assuming that the models are stored textually in individual file artifacts. This assumption may be violated if the models are stored in a database or only one large file is used to store all models.

Parsers produce the internal representation, called *abstract syntax (AST)*¹ of the loaded models. Secondly the *frontend* contains a library of data structures and functions needed to check the context conditions on the input models, to load further needed models, or to ensure the resulting input AST is well formed.

The *central* part of a generator *transforms* the input AST into an output AST. This may be a rather complex transformation, mapping one kind of model to another, or a

¹"AST" traditionally also stands for *abstract syntax tree*, but our ASTs are often full graphs, because they contain a spanning sub-tree plus useful extra information and links.

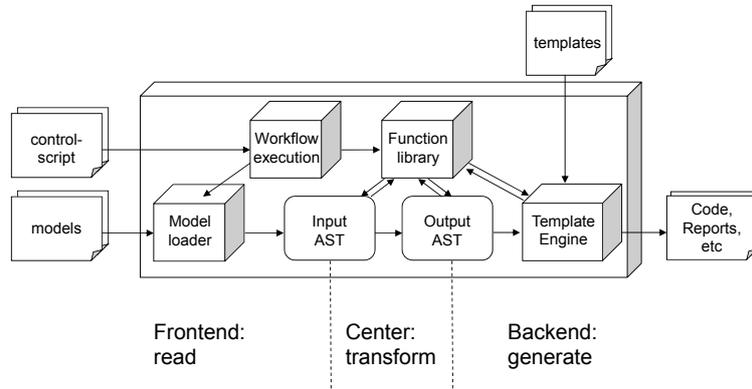


Figure 3.2: Internal architecture of a generator

relatively simple augmentation like adding additional information. MontiCore provides the capability to translate between different kinds of ASTs including the translation from any source language to Java. MontiCore also supports the attachment of specific templates to AST nodes, such that parts of the generators intelligence can be deferred to the templates themselves, while selection of the appropriate templates is done as part of the augmentation of the output AST and is consequently decided in the transformation part.

The *backend* of a generator focuses on the generation of artifacts such as code, analysis results, documentation or other forms of models. It consists of a *template engine* that processes the output AST together with a number of standardized and project-specific templates, which describe the concrete shape of the resultant artifacts. This process is highly configurable and adaptive, e.g. using the *hook point* mechanism provided by MontiCore.

The three processing steps are connected via a main control as shown in Section 3.3 or a *workflow* written in a Groovy script. These workflows control code generation and transformation, as well as storing of a larger function library that facilitates the building of symbol tables, arrangement of transformations, and the setting of specific template configurations, etc. Section 16.5 describes the standard Groovy script that MontiCore uses and that is explicitly dedicated for customization.

Figure 3.2 is of course an abstraction of the real MontiCore infrastructure that also contains components for reporting, logging, the symbol table infrastructure including symbols and scopes, parametrization and customization techniques, and mechanisms for adapting the template-based generation, such as hook points described in the subsequent chapters.

Figure 3.3 provides an overview of all the chapters containing relevant information on how to develop components for your own model parser.

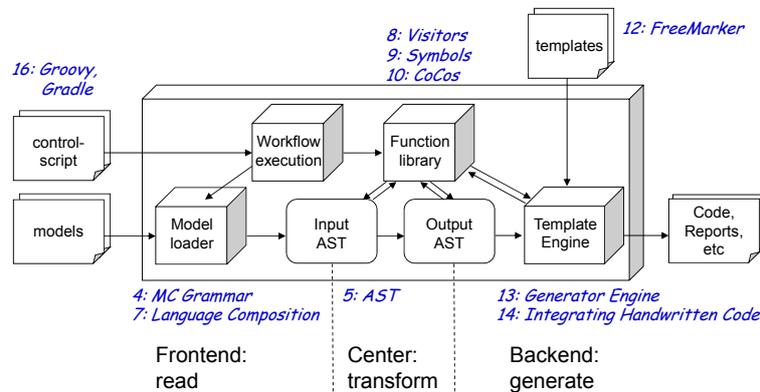


Figure 3.3: Chapter structure of the handbook

3.3 Tool Workflow

Figure 3.2 has shown the general architecture of a tool. However, many tools might be simplified, for example by not including a Groovy script engine, but being directly coded within a Java main class.

Listing 3.4 shows such a Java main method that connects different modules of a DSL tool into a linear workflow. It defines the tool's main control structure and might serve as a blueprint for other DSL processing tools, although it does not contain any handling of parameters and only processes a simple language without imports of foreign symbol tables. This example uses the Automata DSL which is also used and explained later in this handbook. Details, such as the methods used, can be found in their respective chapters or the complete example in the MontiCore project.

Line 18 identifies the file of an automata model from the arguments passed to the main method. In line 21 the parser `AutomataParser` (cf. Chapter 6) parses the automata model to create the AST. The symbol table (cf. Chapter 9) is then created (l. 26) based on the resulting AST. Lines 29-39 show an example resolution of a state symbol.

```

1 public static void main(String[] args) {
2     // delegate main to instantiatable method for better integration,
3     // reuse, etc.
4     new AutomataTool().run(args);
5 }
6
7 public void run(String[] args) {
8     // use normal logging (no DEBUG, TRACE)
9     Log.ensureInitialization();
10
11     // Retrieve the model name
12     if (args.length != 2) {
13         Log.error("0xEE7400 Arguments are: (1) input "

```

```

14         +"model and (2) symbol store.");
15     return;
16 }
17 Log.info("Automata DSL Tool", "AutomataTool");
18 String model = args[0];
19
20 // parse the model and create the AST representation
21 ASTAutomaton ast = parse(model);
22 Log.info(model + " parsed successfully!", "AutomataTool");
23
24 // setup the symbol table
25 IAutomataArtifactScope modelTopScope =
26     createSymbolTable(ast);
27
28 // can be used for resolving names in the model
29 Optional<StateSymbol> aSymbol =
30     modelTopScope.resolveState("Ping");
31
32 if (aSymbol.isPresent()) {
33     Log.info("Resolved state symbol \"Ping\"; FQN = "
34         + aSymbol.get().toString(),
35         "AutomataTool");
36 } else {
37     Log.info("This automaton does not contain a state "
38         + "called \"Ping\";", "AutomataTool");
39 }
40
41 // setup context condition infrastructure
42 AutomataCoCoChecker checker = new AutomataCoCoChecker();
43
44 // add a custom set of context conditions
45 checker.addCoCo(new StateNameStartsWithCapitalLetter());
46 checker.addCoCo(new AtLeastOneInitialAndFinalState());
47 checker.addCoCo(new TransitionSourceExists());
48
49 // check the CoCos
50 checker.checkAll(ast);
51
52 // Now we know the model is well-formed and start backend
53
54 // store artifact scope and its symbols
55 AutomataSymbols2Json deser = new AutomataSymbols2Json();
56 deser.store(modelTopScope, args[1]);
57
58 // analyze the model with a visitor
59 CountStates cs = new CountStates();
60 AutomataTraverser traverser = AutomataMill.traverser();
61 traverser.add4Automata(cs);
62 ast.accept(traverser);
63 Log.info("Automaton has " + cs.getCount() + " states.",
64     "AutomataTool");
65

```

3. Architecture of a Model Processor

```
66 // execute a pretty printer
67 PrettyPrinter pp = new PrettyPrinter();
68 AutomataTraverser traverser2 = AutomataMill.traverser();
69 traverser2.setAutomataHandler(pp);
70 ast.accept(traverser2);
71 Log.info("Pretty printing automaton into console:",
72         "AutomataTool");
73 // print the result
74 Log.println(pp.getResult());
75 }
```

Listing 3.4: Example tool for the Automata DSL

Since the parser can only identify *context-free* parsing errors (cf. Chapter 10) additional *context sensitive* constraints have to be validated (e.g., there must exist at least one initial and one final state). For this purpose, an `AutomataCoCoChecker` object is created, which can be configured with concrete context conditions (ll. 42-47). The `checkAll` method checks all registered context conditions (l. 50). After this, a `AutomataScopeDeSer` is added, which is used to store the symbols a scope contains in a specific location (cf. 56). Next, the model is analyzed. Visitors provide an appropriate infrastructure to traverse and operate on the AST (cf. Chapter 8). Here, the number of states is calculated (l. 59).

Finally, the model is pretty printed (l. 67). Pretty printing serves several purposes: Firstly, the resulting model may be easier to maintain or to simply store in form of documentation. Secondly, pretty printing helps to check, whether parsing and the AST construction was complete and correct. Usually, but not in this example, an executable implementation of the automata model would also be generated.

This example only covers the essence of working with an implemented DSL. There are many other possibilities which are covered in their respective chapters.

Chapter 4

MontiCore Grammar for Language and AST Definitions

In MontiCore, grammars are the central notation from which a lot of infrastructure, including the language parser and the internal representation of a model (called the abstract syntax, short AST), are derived.

This chapter will explain the MontiCore grammar format. It discusses productions defining different types of nonterminals and their relations, context conditions that define the well-formedness of a grammar, and additional concepts that allow further configuration of the code generation process from MontiCore grammars.

MontiCore grammars describe the *context-free syntax* of languages using notation based on the Extended Backus-Naur Form (EBNF, [ASU86]) and the ANTLR tool [Par13]. Additionally a MontiCore grammar provides convenient constructs for specifying commonly used *options* and additional *context-sensitive* concepts of languages. As MontiCore creates Adaptive LL(*) parsers, it can also support *semantic predicates*, which make it possible to describe common context free parsable languages. However, MontiCore's main purpose is to develop DSLs, most of which have a rather straightforward syntax. Therefore, MontiCore has been developed with comfort and agility as primary goals.

A MontiCore grammar defines the *concrete syntax* and the *abstract syntax* of a language in one artifact. That is to say, MontiCore derives the following from a grammar:

- The *parser* used for reading the model in the form of concrete syntax and produce the internal representation, i.e., abstract syntax (cf. Chapter 6).
- The *data structure* of the abstract syntax (AST, cf. Chapter 5).
- The transformations that are used to fill an AST when parsing a model.
- The infrastructure needed to manage *symbols* and *scopes*.

The majority of the frontend of the generator tool is generated using a MontiCore grammar. Furthermore, MontiCore provides *inheritance*, *extension* and *overriding* mechanisms for productions, thus allowing an improved reuse of sublanguages (cf. Chapter 7). However, this chapter will focus on the syntax and semantics of MontiCore grammars.

Each grammar in MontiCore consists of a head and a body. The head comprises the package declaration, import statements as well as the name of the grammar. Furthermore,

```
1 grammar MinimalExample extends de.monticore.MCBasics {
2   A = "Hello" B ;
3   B = Name "!";
4 }
```



Listing 4.1: Minimal grammar example

it may mark a grammar as a *component* or define grammars that are extended which will be explained in Chapter 7.

A grammar's body may comprise four kinds of statements that can be specified in any arbitrary order:

- *Productions* (like A and B) are the main elements of a grammar and make up the syntactic specification of the language.
- *Lexer productions* help describe small tokens, such as names, values and atomic keywords, which are later read by the grammar.
- *Grammar directives* which allow configuration of the grammar.
- *Grammar concepts* which further extend the capabilities of language specification in the grammar.

For reference purposes we have included an EBNF version of the MontiCore grammar in Section 4.6. It describes the MontiCore grammar language using a MontiCore grammar. To avoid confusion when describing a grammar language using grammars, we avoid direct references to EBNF-nonterminals in the following explanations, instead using examples.

4.1 Lexical Tokens for the Scanner

The first step for processing a model is to run a *lexer* (also called scanner) for the lexical analysis of the input model. The lexer segments sequences of input characters which describe a model into a sequence of *tokens*. These tokens are passed to the parser to create AST objects [Völ11, ASU86].

Some typical forms of tokens include

- keywords like "if",
- operators like "++", ">>", "*",
- delimiters like "(", ",", "
- values like 3.2, 42,
- names like Person and age,
- qualified names like de.monticore.dex.Person or
- whitespaces that are ignored by the grammar.

Individual tokens can be directly included as terminals in the productions and thus need no explicit rules (like `"if"`). On the other hand, it is possible to introduce *nonterminals* that stand for a class of tokens, like all numbers, strings or names. These simple nonterminals are defined via lexical productions and contribute to the abstract syntax in form of attributes, but are not as AST classes on their own. We sometimes refer to those simple nonterminals as *tokens*.

4.1.1 Definition of Tokens using Regular Expressions

Lexical productions are simplified forms of productions and consist of a left-hand side, i.e. the name of the token, and a right-hand side, i.e. the body of the production. The production body defines the structure of the token by use of a *regular expression*. If not specified through an explicit type in the production (see Listings 4.7 and 4.8), a token results in a value of type `String`. The token is later stored as an attribute of the associated type in the AST named after the token. If the input matches the regular expression defining this token, the lexer recognizes the nonterminal and produces the token.

```

1 token SimpleName = ('a'..'z' | 'A'..'Z')+ ;
2
3 token SimpleString = '"' ('a'..'z' | 'A'..'Z')* '"';

```

Listing 4.2: Lexical productions for `SimpleName` and `SimpleString`

Listing 4.2 shows a standard definition of two nonterminals as tokens called `SimpleName` and `SimpleString`. `SimpleName` is defined as a sequence of upper and lower case letters requiring at least one letter to be present. The second lexical production introduces the nonterminal `SimpleString` defined as a sequence of upper and lower case letters, which are embedded in quotation marks. In this case, the sequence may be empty.

The definitions of lexical rules correspond to the rules of regular expressions as follows:

- Constant strings denote keywords and are surrounded by single or double quotes, like `'st'` or `"st"`.
- A range of characters is given by `lowerChar..upperChar`.
- The `|` character separates alternatives.
- Grouping items is done by use of parentheses `(and)`.
- The character `~` is used for negation, meaning the expression matches what is not part of the negated expression.
- The `+` sign denotes that the previous item may be added to nonterminals or groups one or more times.
- The `*` character denotes the previous item being added zero or more times.
- The `?` character denotes the previous item being added zero or one-time.

4. MontiCore Grammar for Language and AST Definitions

```
1 token NUM_INT =
2   ('0'..'9')+ EXPONENT? SUFFIX? ;
3
4 fragment token SUFFIX =
5   'f'|'F'|'d'|'D' ;
6
7 fragment token EXPONENT =
8   ('e'|'E') ('+'|'-')? ('0'..'9')+ ;
```

Listing 4.3: Lexical productions for Numbers using token fragments

Lexical definitions may be reused in other definitions to make them more readable. Lexical productions marked as `fragments`, as it is the case for `SUFFIX` and `EXPONENT` in Listing 4.3, can only be used in other lexical productions and cannot be nested recursively. Fragments are not passed to the parser but instead allow a modular definition of lexicals. Therefore, only `NUM_INT` results in a `String` that is stored in the AST, however the string will contain the fragments.

Technical Info 4.4: Limited Scanning Capability

For efficiency reasons, a scanner does not backtrack. If a *token is prefix* of another token, e.g. `>` and `>>` then an unfortunate combination, e.g. in `List<List<String>>` will fail. Here grammar directive `splittoken` can help.

When you want to use a *reserved keyword* also as e.g. variable name in another context, then you can use the `nokeyword` directive.

Both are discussed in Section 4.3.

4.1.2 Actions to Process a Token

Actions (i.e., Java code) can be embedded into lexical definitions to modify the lexers results directly. Hence actions do not contribute to the definition of concrete syntax, but extend the parser by mapping concrete syntax to elements, which are stored in the abstract syntax, or to completely different things.

```
1 token WS =
2   ( ' '
3     | '\t'
4     | '\r'      // Macintosh
5     | '\n'      // Unix
6   ) :{ _channel = HIDDEN; };
```

Listing 4.5: Lexical productions for white spaces

To allow an arbitrary number of white spaces and line breaks in a model, MontiCore has a predefined nonterminal `WS` (cf. Listing 4.5). The last line encloses Java code to avoid pass-

ing this token to the parser. Alternatively you can replace the Java code with the following statement `"-> skip"`. Thus, it is not necessary to explicitly place the nonterminal `WS` in grammars. For a more detailed discussion, refer to the ANTLR documentation.

```

1  token STRING = '"'
2      ( ESC
3      | ~('"' | '\\\'' | '\n' | '\r' )
4          )*
5      '"'
6      :{setText(getText().substring(1, getText().length() - 1));};

```

Listing 4.6: Lexical production for strings without quotation marks, which are removed in a Java action

In the example shown in Listing 4.6, the embedding quotation marks are removed from the token before it is passed to the parser. It is allowed to add actions at the end of a lexical production for a free computation of the result.

Listing 4.7 shows how to adapt the different types for the generated attributes in the AST. If the right side of a production is a lexical production like `Name`, the corresponding attribute is of type `String`. We can change the default attribute type by adding a predefined type like `float`, separated by a colon. Some of the supported predefined types are: `float`, `int` and `char`. The corresponding default conversion methods translate the parsed value of type `String` to the derived type.

```

1  // token results can be converted
2  // here NUMBER becomes type float
3  token NUMBER = ('0'..'9')* '.' ('0'..'9')* 'f': float;
4  A2 = b:NUMBER "," c:NUMBER*;
5
6  // while Name is stored as a String
7  A1 = b:Name "," c:Name;

```

Listing 4.7: Changing the result type of lexicals

If the default methods do not work, we can implement our own conversion method as shown in Listing 4.8. The token `CARDINALITY` should be adapted to the type `int`. The declared variable `x` has the default type `String` and we can add a `Java-Block` to convert the `String` to the desired type `int`.

4.1.3 Predefined Tokens in Importable Grammars

MontiCore is shipped with some basic grammars meant for reuse as described in Section 7.4. These are, among others:

- `MCBasics`
- `MCLiteralsBasis`,

4. MontiCore Grammar for Language and AST Definitions

```
1 // token results get adapted:
2 // type conversion to int
3 // by Java code that does the conversion
4 token
5     CARDINALITY = ('0'..'9')+ | '*' :
6     x -> int : { // Java code:
7         if (x.equals("*"))
8             return -1;
9         else
10            return Integer.parseInt(x.getText());
11    };
```

Listing 4.8: Adding a conversion method for lexical types

- `MCCCommonLiterals`, and
- `MCJavaLiterals`.

They provide a set of basic literals useful for almost every parse process (`MCBasics`) or for expressions that Java programmers know quite well. The grammars described in Chapter 18 build full expression and statement sublanguages based on these literals.



Tip 4.9: Predefined Tokens

Predefined tokens can be found in the MontiCore repository, for example in the following component grammars:

```
1 Repository: MontiCore/monticore github
2 Directory: monticore-grammar/src/main/grammars/
3 Files:     de.monticore.MCBasics.mc4
4           de.monticore.literals.MCLiteralsBasis.mc4
5           de.monticore.literals.MCCCommonLiterals.mc4
6           de.monticore.literals.MCJavaLiterals.mc4
```

Some of the ways you can include tokens in a grammar are: use of `extends de.monticore.MCBasics` or inclusion of the directory in the grammar path. A detailed description can be found in Chapter 17.

When including the available basic grammars `MCBasics`, `MCCCommonLiterals` and `MCJavaLiterals`, a number of predefined tokens can be used as described in Table 4.10.⁴ All tokens (cf. Table 4.10, column 1) are stored as `String` but the `MCCCommonLiterals` and `MCJavaLiterals` grammars also provide nonterminals (cf. Table 4.10, column 2)

¹fragment, not a complete token

²delimiters are removed

³delimiters are removed

⁴Please note that the type grammars `MCBasicTypes`, `MCCollectionTypes`, etc. do not define tokens, but only parser nonterminals. The grammars `MCCCommonLiterals` and `MCJavaLiterals` also define additional parser nonterminals.

Table 4.10: Some of the predefined tokens of the `MCBasics`, `MCCCommonLiterals` and `MCJavaLiterals` component grammars.

Token	Value Type	Defined in
<code>WS</code>	-	<code>MCBasics</code>
<code>SL_COMMENT</code>	-	<code>MCBasics</code>
<code>ML_COMMENT</code>	-	<code>MCBasics</code>
<code>NEWLINE</code>	- ¹	<code>MCBasics</code>
<code>Name</code>	-	<code>MCBasics</code>
<code>Digits</code>	-	<code>MCCCommonLiterals</code>
<code>Char</code>	-	<code>MCCCommonLiterals</code>
<code>String</code>	- ²	<code>MCCCommonLiterals</code>
<code>CharLiteral</code>	<code>char</code>	<code>MCCCommonLiterals</code>
<code>BooleanLiteral</code>	<code>boolean</code>	<code>MCCCommonLiterals</code>
<code>StringLiteral</code>	<code>String</code> ³	<code>MCCCommonLiterals</code>
<code>NatLiteral</code>	<code>int</code>	<code>MCCCommonLiterals</code>
<code>BasicLongLiteral</code>	<code>long</code>	<code>MCCCommonLiterals</code>
<code>BasicFloatLiteral</code>	<code>float</code>	<code>MCCCommonLiterals</code>
<code>BasicDoubleLiteral</code>	<code>double</code>	<code>MCCCommonLiterals</code>
<code>Num_Int</code>	-	<code>MCJavaLiterals</code>
<code>Num_Long</code>	-	<code>MCJavaLiterals</code>
<code>Num_Float</code>	-	<code>MCJavaLiterals</code>
<code>Num_Double</code>	-	<code>MCJavaLiterals</code>
<code>IntLiteral</code>	<code>int</code>	<code>MCJavaLiterals</code>
<code>LongLiteral</code>	<code>long</code>	<code>MCJavaLiterals</code>
<code>FloatLiteral</code>	<code>float</code>	<code>MCJavaLiterals</code>
<code>DoubleLiteral</code>	<code>double</code>	<code>MCJavaLiterals</code>

with respective AST classes that provide methods `getValue()` in order to convert the stored `String` to a more usable type. See Chapter 17 for details.

Furthermore, there are four token nonterminals defined in the `MCBasics` grammar that are not passed to the parser:

NEWLINE tokens are not stored, but used as token separator

WS tokens are not stored, but used as token separator

SL_COMMENT describes *single line comments* in Java style, like `// . . .`. Those comments are not passed to the parser, instead they are attached to the AST object which is created by the parser and therefore can be retrieved if necessary, but need not be included in productions explicitly (cf. Section 5.7)

ML_COMMENT like `/* . . . */` do the very same as `//`, but can span over *multiple lines*.

Multi line comments are not nested like in Java (as opposed to C++). In case that this form of comment is not desired, the `MCBasics` grammar should not be used.

The type grammars `MCBasicTypes`, `MCCollectionTypes`, `MCSimpleGenericTypes`, `MCFullGenericTypes`, and `MCArrayTypes` grammars define nonterminals that provide different kinds of data types. Some examples of this are:

- primitives types such as `int` or `boolean`,
- collection types such as `MCListType` or `MCSetType`,
- widely reusables types `MCImportStatement` or `MCQualifiedName`,
- data structures such as generics or arrays.

See Table 4.11 for the list of the most interesting nonterminals.

Table 4.11: Predefined nonterminals of Types grammar.

Nonterminal	Meaning
<code>MCType</code>	Interface for all forms of types
<code>MCQualifiedName</code>	Sequence of Names, separated by a "."
<code>MCPriimitiveType</code>	"boolean","int" etc.
<code>MCGenericType</code>	Interface for all forms of generic types
<code>MCListType</code>	for <code>List<T></code> , but not <code>java.util.List<T></code>
<code>MCTypeArgument</code>	Interface for all forms of type arguments
<code>MCWildcardTypeArgument</code>	Single type argument, using the wildcard "?"

4.2 Productions in the Grammar

A production consists of a left-hand side that defines a new nonterminal (e.g. `A`), and the right-hand side of a production which describes how the nonterminal is defined i.e., its body. Listing 4.12 shows some simple examples of productions, which are rather similar to lexical productions. In addition, productions support recursion (thus becoming context-free and not just regular productions, as per the Chomsky hierarchy) and a number of techniques to control how to map productions to AST classes. The mapping process is partially introduced here but discussed in more detail in Chapter 5.

```

1  A = "Hello" "World" "." ;
2  B = ("Good Morning" Name ) | A ;
3  C = "Hello" (Name || ",")+ ;
4  D = A B* (C | D)
5      | B* A ;

```

Listing 4.12: Some production examples

The body of a production is composed of terminals and nonterminals, both of which may be used as part of alternatives, may themselves be optionals or occur multiple times. The MontiCore grammar provides these constructs for the productions:

- Double quotes to define constant strings e.g. `"state"` and are then used as keywords.
- `LowerChar..upperChar` to define a range of characters (cf. Listing 4.8).
- The `|` character to separate alternatives.
- Parentheses `(and)` to signify grouping.
- `+` to signify the appearance of a group or nonterminal one or more times.
- `*` to signify the appearance of a group or nonterminal zero or more times.
- `?` to signify the appearance of a group or nonterminal zero or one time.
- `[C1 | ... | Cn]` to express an alternate group of constants (terminals), where exactly one occurs.
- Shorthand notations `(NT || T)*` and `(NT || T)+` to define repetitions of the left nonterminal `NT` being separated by the right terminal `T`.
- References to other nonterminals, e.g. `Name@State`, which mean the name references an entity that is defined by a `State` nonterminal.
- The key statement `key("state")` to define a local keyword `state`. This keyword is almost identical to a permanent keyword, but name `"state"` can still be used as a normal name in other places. The argument of the `key(.)` statement must match the `Name` nonterminal or a list of `|` separated `Name`. E.g. `key("F"|"f")` describes the float suffix for numbers.

Furthermore, *nonterminals* allow for structuring of the parsing, including mutual recursion (with only a slight restriction on mutual left recursion). The main differences from nonterminals to lexical productions are the ability to use other arbitrary nonterminals, but also the absence of negation `~`.

Several elements on the right-hand side can be decorated to control the AST:

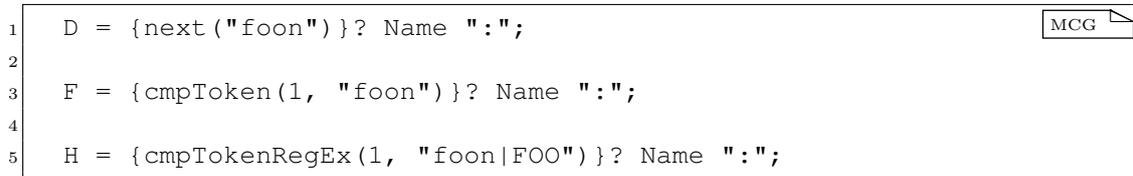
- Names such as `n:NT` and `n:"st"` can be attached to terminals and nonterminals, describing where the attributes will be stored in the AST.
- References to other nonterminals can be attached using `@`, like `Name@NT`. For example, `Name@State` expresses that the name references an entity that is defined by a `State` nonterminal, i.e., elsewhere there will be a `State` defined with this name. This is also called referencing symbols because the nonterminal which is referenced to must also define a symbol. For more information on this check Chapter 9.

MontiCore also provides so called *semantic predicates*, which are in detail discussed in Section 4.5. However, there are a number of standard methods available that allow us to control parsing process:

- Function `next("42")?` is usable in semantic predicates, such as `{next("42","41")}? Digits`, to check if the next token of the model equals to the string. The token is first parsed according to the following non-terminal (here: `Digits`) and then the semantic predicate is checked. For `Names`, the `key(.)` shortcut does the same, but `next` also allows arbitrary many strings to be listed as alternatives.
- the `noSpace(n)` function forbids spaces between consecutive token, e.g. `{noSpace(2)}? "-" "--"` only accepts `"--"` and `{noSpace(2,3)}? ">" ">"` forbids spaces between the angle brackets. This can also be deferred to extending grammars, using the `splitToken` keyword.
- Function `cmpToken(n,s)` can be used to restrict possible consecutive tokens. For example `{cmpToken(1,"st")}? Name` specifies that the name must be `st`. While `next` must be used preceding to the token `cmpToken` allows to look further into the forthcoming token using the distance as its first argument. `cmpToken` allows arbitrary many strings to be compared
- Function `{cmpTokenRegex(n,r)}?` is similar to `cmpToken(n,s)`, but interprets `r` as regular expression.

Listing 4.13 shows how the predefined functions `next`, `cmpToken` and `cmpTokenRegex` can be used in conjunction with arbitrary tokens, e.g., the `Name` nonterminal.

```
1 D = {next("foon")}? Name ":";
2
3 F = {cmpToken(1, "foon")}? Name ":";
4
5 H = {cmpTokenRegex(1, "foon|FOO")}? Name ":";
```



Listing 4.13: `next`, `cmpToken` and `cmpTokenRegex`

The regular expression passed to `cmpTokenRegex` is based on the regular expressions used in the standard libraries of Java. Thus, regular expression according to the syntax provided by Java are possible⁵.

4.2.1 Terminals

Terminals are enclosed in quotation marks (e.g. `"if"` or `"!"`) and are usually not part of the abstract syntax (cf. Chapter 5). Semantically relevant terminals can be marked as such by naming them or surrounding them with square brackets, e.g. a terminal in an alternative (cf. Listing 4.14). In the former, the given name of the terminal is used as the name of the attribute. In the latter, a boolean stores whether the terminal occurred in the model. The mapping of relevant terminals to the AST is discussed in Chapter 5.

Introducing `"Hello"` as explicitly mentioned terminal disallows `"Hello"` to be used as ordinary (variable) name elsewhere. The key statement `key("Hello")` can be used to

⁵<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>

```

1  E = "Hello"
2    (who: "World" | who: "Tom")
3    "!";
4
5  F = ["initial"]?;
```

Listing 4.14: Augmentation of terminals for storage in the AST

define a local keyword `Hello`. This keyword is almost identical to a permanent keyword, but now `"Hello"` can still be used as a normal name in other places. The argument of the `key(.)` statement must match the `Name` nonterminal or a list of `|` separated `Names`. E.g. `key("F"|"f")` describes the float suffix for numbers and `key("World"|"Tom")` is a shortcut usable in Listing 4.14. For an even more convenient alternative see grammar directive `nokeyword` in Section 4.3.

4.2.2 Enumeration

If the language describes alternatives of multiple terminals resulting in boolean attributes, one may instead use alternatives when creating the abstract syntax. Instead of mapping each terminal to a boolean attribute, an integer with constants mimicking an enumeration may be used. To use this variant, the alternatives are enclosed within square brackets (Listing 4.15). If this is the case, it is necessary to add a name in front of the square brackets, separated by `|`. As with square brackets, each terminal can be named, e.g., `PRIVATE:"-"`, which will result in a name stored as a constant. However, as described in Chapter 5, MontiCore will derive a name automatically if none is given explicitly, such as `PUBLIC` for `"public"`. In case the default derivation is not desirable (e.g. `"-"` by default results in `MINUS`) an explicit name can be added. Please note that adding a name equivalent to the automatically derived name is allowed. In the following example (Listing 4.15) the four alternatives would produce only two constants because `PUBLIC:"+"` and `"public"` will coincide.

```

1  G = vis:[ PUBLIC:"+ " | "public" |
2          PRIVATE:"- " | "private"];
```

Listing 4.15: A choice of alternate terminals is stored as integers

Instead of using a list of keywords (i.e. relevant terminals), it is also possible to use an enumeration nonterminal. As shown in Listing 4.16, the MontiCore grammar language allows for the definition of enumerations directly by using the keyword `enum`, followed by a name and a body consisting of a list of constants separated as alternatives. An enumeration nonterminal can be used like any other nonterminal.

```

1  enum VISIBILITY =
2      PUBLIC:"+" | "public" |
3      PRIVATE:"- " | "private" ;
4  H = vis:VISIBILITY;

```

Listing 4.16: Explicit definition of an enumeration

4.2.3 Nonterminals

A nonterminal is defined using a *production*, where the nonterminal on the left-hand side is replaced by the body of the production on the right-hand side. A nonterminal can be part of an alternative ($A|B$), be optional $A?$ or occur multiple times when indicated by a $*$ or a $+$ character (at least one). Nonterminals are always mapped to the AST (cf. Chapter 5). For every nonterminal, there is a class generated for the AST data structure. Nonterminals on the right-hand side of a production result in compositions stored as attributes with access methods.

```

1  Automaton =
2      "automaton" Name "{"
3      ( State | Transition ) *
4      "}";

```

Listing 4.17: Automatic naming of unnamed nonterminals

Both terminals and nonterminals can be explicitly named by adding a name and colon. In case no explicit name is given, the name of the composition is derived from the nonterminal name by lowercasing the first letter. Thus, `state:State` is equivalent to `State` in Listing 4.17, reducing the developer's writing effort. If a nonterminal on the right side is marked with a $*$ or a $+$, a list of this nonterminal is generated. The name of this lists attribute is also either derived from the nonterminals name or, if given, from the explicit name. Additionally, a `s` will be appended to the name. That means a nonterminal `State*` will be stored as a list `List<State>` with the attribute name `states`.

If nonterminals are grouped and marked with cardinalities, then each nonterminal will receive its own list. This means $(A|B)*$ will become two independent list, which store the order within the `As` and `Bs`, but forget the order between them.

A convenient way of defining the frequently appearing pattern of lists with separators is provided by MontiCore in the following construct: use `y:(A || ",")+` to define a *comma separated list*, which is equivalent to `y:A(", " y:A)*`.

On the left (i.e. the `A`), there must be a single nonterminal, whilst the right side should contain a terminal. The definition of the nonterminal `MCQualified Name` is given as `(Name || ".")+` and demonstrates this feature. Accordingly, `y:(A || ",")+` is equivalent to `(y:A(", " y:A)*)?`, as it also allows repetition.

4.2.4 Interface Nonterminals: implements

An unique feature is the ability to define interface nonterminals. An example of an interface nonterminal is given in Listing 4.19. Here, the production `I` defines an interface nonterminal and the productions `A` and `B` implement the interface `I`. In production `C`, the interface `I` is used like a normal nonterminal on the production's right-hand side. Semantically, an interface production can be considered equivalent to a production which has its implementing productions as alternatives on the right hand side, as shown in Listing 4.20

Interfaces can be used to extend languages and thus are a core concepts of language composition (cf. Chapter 7). The important main advantage here is that interface `I` does not refer to the implementation nonterminal `A`, but in the opposite direction. This is especially interesting for language extension, because the extending grammar is defined later and thus the base grammar can be reused black-box without modification.



Tip 4.18: When to use an Interface Nonterminal

Interface nonterminals share a lot of the characteristics of interfaces in programming languages. Using an interface nonterminal allows to decouple the definition of certain structures of a language, while leaving open "holes" (extension points).

It is worth introducing an *interface nonterminal* for important language elements, such as `Expression` or `Statement`, especially when it is intended to extend the language later. Interfaces therefore serve as extension points or potentially also variation points, in case future language extension or embedding is planned.

An interface does not prescribe the possible concrete syntax, while abstract nonterminals (described in Section 4.2.6) do.

```

1  interface I ;
2  A implements I = "...1" ;
3  B implements I = "...2" ;
4  C = I "...";

```

MCG

Listing 4.19: An interface nonterminal and several nonterminals implementing it

```

1  I = A | B ;
2  A = "...1" ;
3  B = "...2" ;
4  C = I "...";

```

MCG

Listing 4.20: Alternative to interface nonterminal in Listing 4.19 accepting the same concrete syntax, but `I` knows `A` and `B`

Furthermore, productions of interface nonterminals can have a body. This body defines which kind of signature all implementing nonterminals need to provide in order to implement the interface nonterminal. Considering the example in Listing 4.21, the interface `I` defines the signature `x:Integer` and `y:Name*` for all nonterminals implementing `I`.

Thus, both entities need to be part of the body of A and B, as they implement I. However, as demonstrated, the concrete syntax and order of the elements is not prescribed by I, which allows for flexible adaptation of the concrete syntax of nonterminals implementing an interface. As such, interface bodies are mainly a mechanism used to add functionality to the abstract syntax nodes described in Chapter 5.

```

1  interface I =      x:Integer      y:Name* ;
2  A implements I =  x:Integer "...1" y:(Name || ",")* ;
3  B implements I =  y:Name*      "...2" x:Integer      "... " ;
4  C = I "... " ;

```

Listing 4.21: Interface nonterminal defining its signature

4.2.5 Extending Nonterminals: extends

The MontiCore grammar format allows extending the production of an already defined nonterminal via addition of alternatives which modify the original definition. Listing 4.22 shows nonterminal B extending the already defined nonterminal A. The production of the nonterminal C uses A on its right-hand side and thus also includes the alternative of B. In the context of parsing, extension B is equivalent to adding the extending nonterminal B as a new alternative to the extended nonterminal A. By choosing a definition inverse to EBNF, we can provide an object-oriented solution where subclasses can extend their superclasses without any changes in the definition of the superclass. This second core mechanism is used to allow for reuse of grammar and language extensions. Listing 4.23 shows an equivalent alternative to Listing 4.22 that accepts the same concrete syntax for nonterminal A.

```

1  A = "...1";
2  B extends A = "...2";
3  C = A;

```

Listing 4.22: Extending the production of a nonterminal

```

1  A = "...1" | B;
2  B = "...2";
3  C = A;

```

Listing 4.23: Equivalent alternative to extension in Listing 4.22 accepting the same concrete syntax for A, but A knows and thus is coupled to B

4.2.6 Abstract Nonterminals

An abstract nonterminal is similar to an interface nonterminal, but is introduced using the keyword `abstract`. Abstract nonterminals can bundle nonterminals together, as shown

**Tip 4.24: When to use Nonterminal Extension**

Extension of nonterminals (like A in Listing 4.22) shares characteristics with class extension. For concrete syntax this means that an already implemented nonterminal A gets additional alternatives (so to say "subclasses" like B). This can also occur if the original nonterminal is extended.

The newly introduced nonterminal B is never usually used explicitly in the concrete syntax, but plays a role in the abstract syntax.

If A is meant for extension, it is advantageous to make it an interface. But if a default version of the concrete syntax should exist, an ordinary nonterminal needs to be defined and then extended in sub-nonterminals.

in Listing 4.25. Here, AutomatonElement is an abstract nonterminal. The nonterminals State and Transition extend the nonterminal AutomatonElement, meaning that both states and transitions are elements of an automaton. As such, the nonterminal AutomatonElement can now be used in a production's right side to allow both states and transitions. Listing 4.26 shows an alternative to Listing 4.25 that accepts the same concrete syntax.

```

1  abstract AutomatonElement;
2
3  State extends AutomatonElement = "...1" ;
4
5  Transition extends AutomatonElement = "...2" ;

```

MCG

Listing 4.25: Abstract production in a grammar

Abstract nonterminals can be extended by other (abstract) nonterminals by using the keyword `extends`. This mechanism can be used to bundle other productions as alternatives and can be used to more clearly mark extension points. This means that the grammar is designed to be extended later by the addition of further nonterminals that extend the abstract ones. As abstract nonterminals are mapped to abstract classes, a production can only extend one abstract production but may implement many interfaces. Similarly to interfaces, an abstract nonterminal having a body means all nonterminals extending the abstract nonterminal must provide one too.

```

1  AutomatonElement = State | Transition;
2
3  State = "...1" ;
4
5  Transition = "...2" ;

```

MCG

Listing 4.26: Alternative to abstract nonterminal in Listing 4.25 accepting the same concrete syntax

4.2.7 Starting Nonterminal

By default, the start (axiom) of a MontiCore grammar is the first nonterminal defined within the grammar. However, in case a different nonterminal should act as the starting nonterminal of the language, it can be explicitly marked as such by introducing it with the keyword `start`, as shown in Listing 4.27. This is especially helpful, when the starting nonterminal is inherited.

```
1 grammar Automata3 extends InvAutomata, Expression {
2
3   start Automaton;
4
5   // ...
6 }
```

Listing 4.27: Explicitly setting a top-level nonterminal that is inherited with `start`

4.2.8 Infix Operations and Priorities

MontiCore can of course define an expression language that uses infix operations. For their convenient description, MontiCore provides the possibility to attach priorities to infix operations and, if necessary, also the keyword `<rightassoc>`, which marks the infix operation as a right associative. Figures 4.28, 4.29, and 4.30 show an example of a typical expression language using an excerpt of Java.

```
1 component grammar ExpressionsBasis
2   extends MCBasics, MCLiteralsBasis {
3   interface Expression;
4
5   NameExpression implements Expression <350>
6     = Name;
7
8   LiteralExpression implements Expression <340>
9     = Literal;
10 }
```

Listing 4.28: Grammar `ExpressionsBasis` which provides an interface `Expression` and basic expressions for the other expression grammars to use

MontiCore allows for definition of nonterminals like `Expression` as an interface and the subsequent additions of alternatives. Because in the case of `Expressions` many alternatives are infix they should get a priority in form of an integer, e.g. `<170>`, which tells the parser how to parenthesize infix expressions.

In our example `a+b*c` would parse as `a+(b*c)`, because `180>170`. Many of the non-infix alternatives do not need explicit priorities assigned to them, but it may be helpful for further extension later on. However, the priority of prefix operators, like `!"` also influences

```

1
2 AssignmentExpression implements Expression <60> = <rightassoc>
3   left:Expression
4   operator: [ "=" | "+=" | "-=" | "*=" | "/=" | "&=" | "|="
5             | "^=" | ">>=" | ">>>=" | "<<=" | "%=" ]
6   right:Expression;

```

Listing 4.29: Excerpt of grammar `AssignmentExpressions` for several forms of assignments

```

1
2 MultExpression implements Expression <180>, InfixExpression =
3   left:Expression operator:"*" right:Expression;
4
5 PlusExpression implements Expression <170>, InfixExpression =
6   left:Expression operator:"+" right:Expression;

```

Listing 4.30: Excerpt of grammar `CommonExpressions` for common expressions like `a+b` including infix operation priorities

the parsing order. For example `!a && b` would wrongly be parsed as `!(a && b)` if `!` has lower precedence than `&&`.

The advantages of denoting expressions in this form is twofold: (1) The grammar is kept small and simple, hence fairly readable. (2) As discussed in Chapter 5 the abstract syntax is structurally rather equivalent to the concrete syntax. For a deeper discussion of how to deal with parsing of infix expressions that naturally occur to be left recursive, we refer to standard literature.

MontiCore is designed for extensibility and thus iteratively allows extending the `Expression` language by repeatedly adding more alternatives. In order to add alternatives with priorities into the middle of already existing alternatives, MontCore uses explicit numbers as priorities instead of approaches like ANTLR, which use the order of occurrence in the grammar. We also defined our `Expression` language with larger priority numbers in steps of 10, allowing for additional infix alternatives to be added where desired. This is even possible when extending languages through composition.

Unfortunately, it is not possible to use a sub-nonterminal, such as `PlusExpression` directly, instead only the main nonterminal, here `Expression` may be included. Solutions are: (1) using CoCo's to restrict other forms of expressions (on the top-level only?), or (2) use a new nonterminal `N = Expression + Expression` that only looks like a `PlusExpression`.

4.2.9 Restricting the Cardinality of a Nonterminal

In order to constrain the number of occurrences of a nonterminal on the right hand side of a production, the `astrule` keyword may be used. Using a statement like the one given



Tip 4.31: Mutual Left-Recursion can be used

The good news is that MontiCore can handle left recursion within its productions. ANTLR can already handle direct left recursion (like $A = A \dots$).

MontiCore has expanded this feature for mutual left recursion which includes an interface `nonterminal` and many implementing alternatives, as shown in Figures 4.28, 4.29, and 4.30.

This is a relevant advantage in MontiCore, because language embedding and extension allow mutual recursion across language components, i.e. when an already defined nonterminal `Expression` is to be extended with more variants of expressions in an extending grammar.

in Listing 4.33 (see Fig. 4.32) will create the option to constrain the occurrences of the nonterminals on the right-hand side by setting a minimum and maximum number of occurrences. Line 6 shows such a constraint. Any natural numbers are allowed. It is also possible to use `+`, `*` and `?` instead of `min` and `max`. The `astrule` keyword does not normally affect the concrete syntax but this is an exception that we wanted to highlight before properly introducing the keyword in Chapter 5.

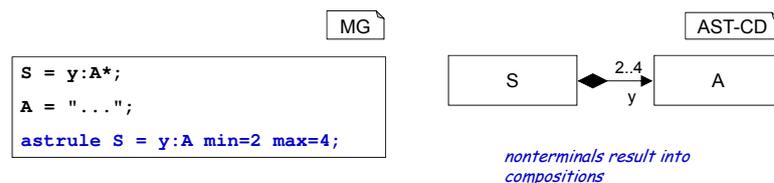


Figure 4.32: Constraining the cardinality of a nonterminal, when parsing

```

1 // nonterminals for the concrete syntax
2 S = y:A*;
3 A = "...";
4
5 // constraining occurrences of A in S:
6 astrule S = y:A min=2 max=4;

```

Listing 4.33: Constraining the cardinality of a nonterminal

4.2.10 Symbols and Scopes

Textual languages use explicit names in order to reference any kind of language entity (methods, classes, attributes, states, signals, etc.) which has been defined elsewhere. For efficient management of names and their referenced entities, symbols and their related management infrastructure is helpful.

For a more detailed explanation of the definition and usage of symbols or scopes using MontiCore's grammar see Chapter 9.

In order to generate a symbol management infrastructure (cf. Chapter 9), which is usually not complete, productions can be marked as symbol definitions or visibility restrictions. The keyword `symbol`, when attached to a production, introduces a new symbol. A symbol must either have the nonterminal `Name` directly on the right-hand-side of the production, otherwise only an abstract symbol class is generated. To use the abstract symbol class, the `getName()` method needs to be implemented manually.

The keyword `scope`, when attached to a production, indicates that the nonterminal introduces a new scope that includes exactly only that AST node and all its child nodes. All symbols defined in these child nodes are restricted in their visibility to this scope.

Furthermore, nonterminal `Name`, when used as the right-hand side of a production, can be marked as a reference to another nonterminal by appending `@` plus the name of the nonterminal that is being referenced (e.g. `Name@State`). See Chapter 9 for details.

The statements `symbolrule` and `scoperule` allow to specify further attributes to store extra information for a symbol or a scope. They are discussed in Section 9.2.3

4.2.11 Passing Code to the ANTLR Parser

It is sometimes necessary to directly specify additional Java code for the underlying ANTLR parser. For this purpose, MontiCore allows adding Java code as shown in the following example (Listing 4.34).

It is possible to add methods, attributes and constants to the generated parser class. Sometimes it is necessary to add variables or methods to be used by the actions defined in the grammar. The variable `ltCounter` defined in Listing 4.34 counts the number of LT (<) used by type parameters or type arguments and can be used for checking the correct number of brackets. The method `incCounter()` enables increasing the counter by the parameter value passed to the method.

```

1 concept antlr {
2   parserjava {
3     public int ltCounter = 0;
4
5     public void incCounter(int i){
6       this.ltCounter = this.ltCounter + i;
7     }
8   }
9 }

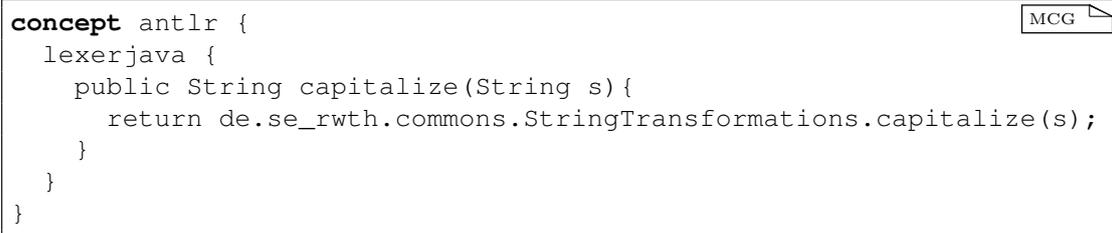
```

Listing 4.34: Add Java code to the parser

Similar to extension of the parser shown above, the lexer can be extended with custom Java code. An example for a lexer extension is shown in Listing 4.35. An additional method `capitalize()` is added to the lexer which takes a `String` and returns it with its first letter capitalized.

Insertions into the lexer have some restrictions. In particular, it is not possible to extend the list of imports, which means that all references to external classes need to be fully

```
1 concept antlr {
2   lexerjava {
3     public String capitalize(String s) {
4       return de.se_rwth.commons.StringTransformations.capitalize(s);
5     }
6   }
7 }
```



Listing 4.35: Add Java code to the lexer

qualified. Furthermore, it has proven useful to keep the amount of Java code in grammars as small as possible by, for example, delegating the execution of complex functionality to other Java classes.

The `parserjava` and `lexerjava` statements will be mapped to two different classes and therefore cannot directly be called from one to the other. However, the function `getCompiler()` can be used from within the lexer code to access the parser code. Please note, however, that the lexer uses a lookahead and thus may have progressed much further than the parser knows. Thus, controlling the lexing mode from the parser is not easy or sometimes impossible.

4.2.12 Annotations for Nonterminals and Grammars

Annotations in programming languages like Java are used to add additional information to classes, fields or methods e.g., whether their deprecated or override functionalities of a super class. MontiCore offers similar annotations as well.

Deprecated Annotation Languages evolve and so do grammars. Similar to the Java programming language, MontiCore allows to mark a production or a whole grammar as deprecated with the `@Deprecated` annotation. It can be added directly before a production or before a grammar itself. A comment can be added (e.g. `@Deprecated("text of the comment")`).

The `@Deprecated` annotation is intended to signal the user of a language that this non-terminal respectively the whole grammar will vanish in a future release and alternatives exist. As a result, generated Java code will also be annotated as `@Deprecated`.

Override Annotation When using language inheritance as explained in Chapter 7 it is possible to redefine nonterminals by overriding existing nonterminals (cf. Chapter 7). To this end, a nonterminal production is marked with the `@Override` annotation. Omitting the annotation when overriding a nonterminal results in a warning while adding an override annotation to a production that does not override an existing nonterminal is considered as an error.

4.2.13 Predefined Nonterminals in Importable Grammars

MontiCore is shipped with a larger number of basic grammars meant for reuse as described in Section 7.4.

These grammars provide a set of sublanguages useful for many forms of languages including expressions, statements, types, etc. that Java programmers know quite well. The grammars are described in Chapter 18.



Tip 4.36: Predefined Nonterminals

Predefined nonterminals building useful sublanguages can be found in the MontCore repository, for example in the following component grammars:

```

1 Repository: Monticore/monticore github
2 Directory:  monticore-grammar/src/main/grammars/
3 Files:     de.monticore.Cardinality.mc4
4           de.monticore.Completeness.mc4
5           de.monticore.JavaLight.mc4
6           de.monticore.UMLModifier.mc4
7           de.monticore.UMLStereotype.mc4

```

Some of the ways you can include tokens in a grammar are: use of `extends de.monticore.MCBasics` or inclusion of the directory in the grammar path. A detailed description can be found in Chapter 17.

Expressions build on the literals that were mentioned in Section 4.1. An expression language is usually difficult to design and has many infix, prefix and postfix operations to be dealt with. Statements again build on expressions. Chapter 18 describes a larger variety of grammars that describe composable expression sublanguages for typed Java-like languages. Statements and Java methods are defined in Chapters 19 and 20

4.3 Additional Control Directives in the MCG Language

Besides the possibility to define tokens and nonterminals, there are a few control directives that allows to adapt the parsing process.

Here, we describe the ones that deal with the parsing of the concrete syntax.

- the `nokeyword` directive is similar to the `key` function, but handles all occurrences of the given keyword list, e.g. `nokeyword "state", "automaton"`; results in all occurrences of `state` and `automaton` keywords in the grammar to be treated as local keywords, but also allows these as names in ordinary places, where names are allowed.
- the `splittoken` directive is similar to the `noSpace` function to prevent whitespaces between individually defined tokens. For example `splittoken ">>>"` handles all occurrences of this token as three separate token but does not allow spaces in between.

It is important to note, that both directives can adapt the way how tokens respectively names are treated even backwards in the grammar inclusion hierarchy.

`splittoken` was introduced to remove any need to preemptively split all tokens into single character tokens by a component grammar developer, just because it may be that some later extension grammar will introduce some constructs that might come into conflict. This can then be done by the extension developer.

`nokeyword` has a similar goal: If a component grammar has not made "state" a local keyword, this can still be achieved in retrospect by an extending grammar without touching the original nonterminal.

The further grammar statements `symbolrule` and `scoperule` allow to specify further attributes and thus allow to store extra information in a symbol or a scope. They are discussed in Section 9.2.3.

4.3.1 Splitting Tokens

If a token consists of multiple special characters (e.g. ">>" in a production like `Shift = ">>"`) then a certain other productions using a prefix of that token will not be recognized correctly anymore, e.g. `List<List<String>>`.

For sake of efficiency Antlr and therefore MontiCore do not backtrack on token level, which is if two token `>` and `>>` are defined, where one is a prefix of another, in a constellation, where both are possible in a sequence, only the second will be recognized. To avoid this, there are three options:

1. Accept that ">>" need spaces inbetween "`>` `>`" for correct recognition,
2. Split the token manually and use the `nospace` function, e.g. `Shift = {nospace(2)}? ">" ">"`. Now, both token are recognized separately and the semantic predicate `{nospace(2)}?` ensures that there is no space before the second token.
3. Use directive `splittoken` followed by a comma separated list of tokens that should be split and a semicolon (cf. Listing 4.37).

Solution 1 is slightly unpleasant to the modeler; solutions 2 and 3 slow down the parsing process. The `splittoken` has the considerable advantage, that the split need not be defined initially already in the component grammar, but can be deferred to later extending grammars, when the problem actually occurs.

In Listing 4.37 the token `:::"` is split into single character tokens in all productions, because of the final `splittoken` directive. Thus, the productions need not be altered to express a split of this token. When composing languages (see Chapter 7), the `splittoken` directive can even be used for tokens inherited from extended grammars such that these grammars need not be altered if a token produces problems after composing.

Splitting is only allowed for tokens consisting of special characters but not letters or numbers. For example `#tag` should use `nospace()` instead.

```

1  A = "::::" Name;
2  B = "::::"* Name;
3  D = foo:"::::" | bar:"---";
4  E = foo:"::::"* ;
5  F =  foo:["::::"];
6
7  splittoken "::::", "---";

```

Listing 4.37: Add a conversion method for lexical types

4.3.2 Local Keywords: Avoid handling Keywords as Tokens

 **Tip 4.38:** Temporary Keyword with `nokeyword "foo";`

If a keyword, like `"state"` is introduced in a production, then `state` cannot be used as name anymore. Not for variables, methods, attributes.

To avoid this, a *temporary keyword* can be introduced using `nokeyword "state";`.

The introduction of a keywords like `"if"` or `kg` disallows to use the characters `if` or `kg` as normal `Name`. When composing languages (see Chapter 7), however, this may lead to a problem, because keywords suddenly influence the allowed names in each other languages. Programming languages are very careful when adding new keywords, e.g. looking at the Java challenges when introducing the `assert` statement.

To introduce a *temporary keyword* `key ("km")` can be used as discussed in Section 4.2. If this concept is overused, it can slow down the parsing process when the terminals are used for selection between alternatives.

MontiCore also offers the possibility to make keywords local using the `nokeyword` directive. Listing 4.39 demonstrates its usage for the `automaton` keyword, but the arguments can also be a comma separated list.

```

1  Automaton =
2     "automaton" Name "{" (State | Transition)* "}" ;
3
4  nokeyword "automaton";

```

Listing 4.39: Using `nokeyword` to define `"automaton"` as a local keyword

Similar to the `splittoken` directive, the `nokeyword` directive expresses that all occurrences of the listed keywords should not be handled as token and it can be used to decide this also retroactively for keywords inherited from extended grammars.

4.4 Context Conditions for the MCG Language

As in any other language, a number of context conditions apply for grammars. For the MontiCore grammar language these rules mainly deal with nonterminals extending or implementing other nonterminals, or with existence and naming conflicts. The following descriptions provide details on the actual context conditions (i.e., error codes and messages). The following notation is used to explain the error messages:

- [A] A is a placeholder for a concrete value during runtime,
- [A]? is an optional output of A,
- [A|B] is an alternative A or B,
- [placeholder=A|B] is a named alternative.

Naming

<i>CoCo</i> <i>Expl.</i>	0xA4003 (error) The grammar name [grammarName] must not differ from the file name of the grammar (without its file extension).
<i>CoCo</i> <i>Expl.</i>	0xA4004 (error) The package declaration [package] of the grammar must not differ from the package of the grammar file.
<i>CoCo</i> <i>Expl.</i>	0xA4005 (warning) The name [name] used for the nonterminal [nonterminalName] referenced by the production [productionName] should start with a lower-case letter.
<i>CoCo</i> <i>Expl.</i>	0xA4090 (error) The prod: [productionName] contains different rule components with the same name: [ruleComponentName] with incompatible types: [firstType] and [secondType].
<i>CoCo</i> <i>Expl.</i> <i>Hint</i>	0xA4018 (error) The production [productionName] must not use the keyword [keyword] without naming it. Keywords may only be used without explicit naming whenever there could be a valid attribute name derived from it.
<i>CoCo</i> <i>Expl.</i>	0xA4019 (error) The production [productionName] must not use a ConstantGroup with more than one element without naming it.

<i>CoCo</i> <i>Expl.</i>	0xA4024 (error) The production [productionName] extending the production [extendedProductionName] must not use the name [name] for the nonterminal [nonterminalName] as [extendedProductionName] already uses this name for the nonterminal [extendedProductionsNonterminalName].
<i>CoCo</i> <i>Expl.</i>	0xA4025 (error) The overriding production [productionName] must not use the name [name] for the nonterminal [nonterminalName] as the overridden production uses this name for the nonterminal [overriddenProductionsNonterminalName].
<i>CoCo</i> <i>Expl.</i>	0xA4031 (error) The nonterminal [name] should not start with a lower-case letter.
<i>CoCo</i> <i>Expl.</i>	0xA4033 (warning) The grammar's name [grammarName] should start with an upper-case letter.
<i>CoCo</i> <i>Expl.</i>	0xA4079 (error) The string [keyword] for splittoken may not contain any letters or digits and must be longer than 2.
<i>CoCo</i> <i>Expl.</i>	0xA4091 (error) The string [keyword] for key() must be compatible to 'Name'.
<i>CoCo</i> <i>Expl.</i>	0xA4093 (error) The string [keyword] for nokeyword must be compatible to 'Name'.
<i>CoCo</i> <i>Expl.</i>	0xA4058 (warning) If the string [digits] is defined as terminal, this string can no longer be part of an expression.
<i>CoCo</i> <i>Expl.</i>	0xA4006 (warning) The package name [packageName] contains uppercase letters!
<i>CoCo</i> <i>Expl.</i>	0xA2008 (error) The production [productionName] contains two list nonterminals that result in the attribute name [name]. But one name is derived from the nonterminal name and one is set manually. This is not allowed.

Implements/Extends

<i>CoCo</i> <i>Expl.</i>	0xA2007 (warning) The production [productionName] does not extend the Rule [superRuleName] in a conservative manner at component [ruleComponentName]. This can lead to problems in the AST.
-----------------------------	---

<i>CoCo</i> <i>Expl.</i>	0xA2106 (error) The abstract nonterminal [name] must not implement the nonterminal [typeName]. Abstract nonterminals may only implement interface nonterminals.
<i>CoCo</i> <i>Expl.</i>	0xA2107 (error) The abstract nonterminal [name] must not extend the interface nonterminal [typeName]. Abstract nonterminals may only extend (abstract) nonterminals.
<i>CoCo</i> <i>Expl.</i>	0xA2102 (error) The nonterminal [name] must not implement the nonterminal [typeName]. Nonterminals may only implement interface nonterminals.
<i>CoCo</i> <i>Expl.</i>	0xA2103 (error) The nonterminal [name] must not extend the interface nonterminal [typeName]. Nonterminals may only extend (abstract) nonterminals.
<i>CoCo</i> <i>Expl.</i>	0xA2116 (error) The interface nonterminal [name] must not extend the [abstract external]? nonterminal [typeName]. Interface nonterminals may only extend interface nonterminals.
<i>CoCo</i> <i>Expl.</i> <i>Hint</i>	0xA4001 (error) The production [productionName] overriding a production of a sublanguage must not extend the production [extendedProductionName]. Overriding productions can only implement interfaces.
<i>CoCo</i> <i>Expl.</i> <i>Hint</i>	0xA4002 (error) The abstract production [productionName] overriding a production of a sublanguage must not extend the production [extendedProductionName]. Overriding productions can only implement interfaces.
<i>CoCo</i> <i>Expl.</i>	0xA4011 (error) The nonterminal [name] must not [extend astextend] more than one [nonterminal class].
<i>CoCo</i> <i>Expl.</i>	0xA4012 (error) The abstract nonterminal [name] must not [extend astextend] more than one [nonterminal class].
<i>CoCo</i> <i>Expl.</i>	0xA4013 (error) The AST rule for [nonterminalName] must not extend the type [typeName] because the production already extends a type.
<i>CoCo</i> <i>Expl.</i>	0xA4029 (error) The nonterminal [name] must not extend and astextend a type.

<i>CoCo</i> <i>Expl.</i>	0xA4030 (error) The abstract nonterminal [name] must not extend and astextend a type.
<i>CoCo</i> <i>Expl.</i>	0xA4047 (error) The production [productionName] must use the component [ruleComponentName] from interface [interfaceName].
<i>CoCo</i> <i>Expl.</i>	0xA4097 (error) It is forbidden to extend the rule [productionName] with the external class [externalName].
<i>CoCo</i> <i>Expl.</i>	0xA4150 (error) A grammar must not extend another grammar multiple times.
<i>CoCo</i> <i>Expl.</i>	0xA0113 (error) The production [productionName] extends or implements a non-existent production.

Existence

<i>CoCo</i> <i>Expl.</i>	0xA2025 (error) The nonterminal [nonterminalName] must not be defined by more than one production.
<i>CoCo</i> <i>Expl.</i>	0xA2026 (error) The nonterminal [nonterminalName] must not be defined by more than one production: nonterminals aren't case-sensitive.
<i>CoCo</i> <i>Expl.</i>	0xA2030 (error) The production [productionName] must not reference the [nonterminal interface nonterminal] [referencedName] because there exists no defining production for [referencedName].
<i>CoCo</i> <i>Expl.</i>	0xA2031 (error) The production [productionName] must not use the nonterminal [referencedName] because there exists no production defining [referencedName].
<i>CoCo</i> <i>Expl.</i>	0xA0276 (error) The external nonterminal [name] must not be used in a grammar not marked as a grammar component.
<i>CoCo</i> <i>Expl.</i>	0xA0277 (error) The abstract nonterminal [name] must not be used without nonterminals extending it in a grammar not marked as a grammar component.
<i>CoCo</i> <i>Expl.</i>	0xA0278 (error) The interface nonterminal [name] must not be used without nonterminals implementing it in a grammar not marked as a grammar component.

<i>CoCo</i>	0xA4014 (error)
<i>Expl.</i>	Duplicate enum constant: [enumConstant].
<i>Hint</i>	The constants of enumerations must be unique within an enumeration.
<i>CoCo</i>	0xA4015 (error)
<i>Expl.</i>	The lexical production [productionName] must not allow the empty token.
<i>CoCo</i>	0xA4016 (error)
<i>Expl.</i>	The lexical production [productionName] must not reference the non-terminal [tokenName] because there exists no lexical production defining [tokenName].
<i>CoCo</i>	0xA4017 (error)
<i>Expl.</i>	The lexical production [productionName] must not reference the nonterminal [tokenName] because [tokenName] is defined by a production of another type than lexical. Lexical productions may only reference nonterminals defined by lexical productions.
<i>CoCo</i>	0xA4020 (error)
<i>Expl.</i>	There must not exist more than one AST rule for the nonterminal [nonterminalName].
<i>CoCo</i>	0xA4021 (error)
<i>Expl.</i>	There must not exist an AST rule for the nonterminal [nonterminalName] because there exists no production defining [referencedName].
<i>CoCo</i>	0xA4032 (error)
<i>Expl.</i>	There must not exist an AST rule for the enum nonterminal [nonterminalName].
<i>CoCo</i>	0xA4028 (error)
<i>Expl.</i>	The AST rule for the nonterminal [nonterminalName] must not use the same attribute name [attributeName] as the corresponding production with the type [typeNameInASTRule] as [typeNameInASTRule] is not identical to or a super type of [typeNameInProduction].
<i>CoCo</i>	0xA4032 (error)
<i>Expl.</i>	There must not exist an AST rule for the enum nonterminal [name].
<i>CoCo</i>	0xA4037 (error)
<i>Expl.</i>	The production for the referenced symbol [symbolName] does not exist as a symbol or not at all.
<i>CoCo</i>	0xA4041 (error)
<i>Expl.</i>	Symbol or scope is mentioned more than once in the declaration of [productionName].
<i>CoCo</i>	0xA4056 (error)
<i>Expl.</i>	The left recursive rule [productionName] is not allowed in blocks, because it is not supported by Antlr.

<i>CoCo</i> <i>Expl.</i>	0xA4101 (error) There is no production defining a token in Grammar : [grammarName]
<i>CoCo</i> <i>Expl.</i>	0xA4102 (error) [astrule symbolrule scoperule] does not allow the definition of nested generics. Problem in grammar [grammarName], rule for [ruleType], with additional attribute: [additionalAttributeName]
<i>CoCo</i> <i>Expl.</i>	0xA4151 (error) A symbolRule must not exist twice for a single nonterminal. Violation by [ruleType].
<i>CoCo</i> <i>Expl.</i>	0xA0112 (error) Grammar [grammarName] contains two productions named [productionName]. Production names must be unique within a grammar.
<i>CoCo</i> <i>Expl.</i>	0xA0125 (error) The rule [productionName] inherits symbols from more than one class.
<i>CoCo</i> <i>Expl.</i>	0xA0117 (error) There is no symbol defining rule that belongs to symbolrule [symbolRuleName].
<i>CoCo</i> <i>Expl.</i>	0xA0135 (error) The rule [productionName] inherits scope properties from more than one class.
<i>CoCo</i> <i>Expl.</i>	0xA4119 (error) The production [productionName] should not use the annotation [Deprecated @Override] twice.
<i>CoCo</i> <i>Expl.</i>	0xA4118 (warning) The external production [productionName] must not have a corresponding ASTRule.

Overriding

Please note that overriding of productions is an essential and smart feature of MontiCore that is explained in Chapter 7.

<i>CoCo</i> <i>Expl.</i>	0xA4007 (error) The production for the interface nonterminal [name] must not be overridden.
<i>CoCo</i> <i>Expl.</i>	0xA4008 (error) The production for the abstract nonterminal [name] must not be overridden by a production for an interface nonterminal.
<i>CoCo</i> <i>Expl.</i>	0xA4009 (error) The production for the nonterminal [name] must not be overridden by a production for an abstract nonterminal.

<i>CoCo</i> <i>Expl.</i>	0xA4010 (error) The production [productionName] must not be overridden because there already exist productions extending it.
<i>CoCo</i> <i>Expl.</i>	0xA4026 (error) The lexical production [productionName] must not use a different type to store the token than the overridden production.
<i>CoCo</i> <i>Expl.</i>	0xA4027 (error) The production for the enum nonterminal [productionName] must not be overridden.
<i>CoCo</i> <i>Expl.</i>	0xA0274 (error) Production [productionName] from grammar [grammarName] is a symbol and overwritten by the prod [productionName] of the grammar [grammarName] that also defines a symbol. Remove the second symbol definition, because the symbol property is inherited anyway.
<i>CoCo</i> <i>Expl.</i>	0xA0275 (error) Production [productionName] from grammar [grammarName] is a scope and overwritten by the prod [productionName] of the grammar [grammarName] that also defines a scope. Remove the second scope definition, because the scope property is inherited anyway.
<i>CoCo</i> <i>Expl.</i>	0xA4094 (error) The production [productionName] does not override any production.
<i>CoCo</i> <i>Expl.</i>	0xA4098 (warning) The production [productionName] overrides production [productionName] without annotation.

References

<i>CoCo</i> <i>Expl.</i>	0xA4022 (error) The production [name] introduces an inheritance cycle. Inheritance may not be cyclic.
<i>CoCo</i> <i>Expl.</i>	0xA4023 (error) The grammar [name] introduces an inheritance cycle.
<i>CoCo</i> <i>Expl.</i>	0xA4039 (error) You can only refer to other symbols on the nonterminal Name.
<i>CoCo</i> <i>Expl.</i>	0xA4100 (error) The attributes with the UsageName [name] cannot reference to the different symbols [symbolName] and [symbolName].

The unique error identifier, like 0xA4023, can also be used to find the source code where the context condition is checked. Most of the context conditions for MCG can be found under the following Java package:

<pre> 1 Repository: Monticore/monticore github 2 Directory: monticore-generator/src/main/java/ 3 Package: de.monticore.grammar.cocos </pre>	
---	---

4.5 Semantic Predicates and Actions

Semantic predicates and actions are Java expressions embedded in curly braces and followed by a question mark. They allow for checking of constraints on passed model elements or for the injection of Java code into the parser, e.g. counters to count opening and closing brackets. The syntax and semantics of semantic predicates, as well as actions have been adopted from ANTLR, therefore we do not excessively discuss their syntax and usage here, but refer to the ANTLR documentation available at [Par13] instead.

However, due to the restricted capability of the token lexer to read its input and select the right token, it is sometimes useful to use semantic predicates to help Monticore in selecting the correct parsing form. See Section 21.4.3 for examples.

Several of the above discussed grammar constructs are actually short semantic predicates:

- `{next("42", "41") }?`
- `{noSpace(2, 3) }? ">" ">" ">"`
- `{cmpToken(1, "st") }?`
- `{cmpTokenRegEx(n, r) }?`

Now, as we know the content of the curly brackets `{ . . . }` is ordinary Java, we also know, that we can combine the provided Java functions with our own code as well as with any other functions.

An additional functions are provided by the Monticore runtime environment:

- the `token(n)` function gives access to forthcoming token with number `n`, e.g. within a semantic predicate `token(1)` returns the next token as a string.

We, however, suggest not to overuse semantic predicates, and in particular not to write too much code in the grammar itself. If complex code is necessary, you may write that in extra methods in ordinary Java classes and call them from there.

To facilitate writing semantic predicates and avoid fully qualified names, a star import for the package `de.monticore.parser.*` is included in the parser which allows all handwritten classes located in this package to be used via their simple name.

4.6 EBNF of the MCG Language

This section contains the EBNF form of the context free syntax of the MCG language. It is usable for understanding the concrete syntax, but not meant for parsing.

```

1  /*
2   EBNF MCG 7, Version April, 6th, 2021.
3  */
4
5  MCGrammar ::=
6    ('package' QualifiedName ';' )?
7    (MCImportStatement)*
8    GrammarAnnotation?
9    'component'? 'grammar' Name
10   ('extends' (GrammarReference || ',' )+ )?
11   '{'
12     (GrammarOption | Prod | StartRule
13      | ASTRule   | SymbolRule | ScopeRule
14      | SplitRule | KeywordRule | Concept )*
15   '}' ;
16
17  GrammarReference ::= QualifiedName ;
18
19
20  ##### Grammar Options
21
22  GrammarOption ::=
23    'options' '{' (FollowOption | AntlrOption | KeywordOption)* '}' ;
24
25  FollowOption ::=
26    'follow' Name Alt ';' ;
27
28  AntlrOption ::=
29    Name ('=' (Name | String) )? ';' ;
30
31  KeywordOption ::=
32    'allkeywords' | 'keywords' (Name)+ ';' ;
33
34  GrammarAnnotation ::=
35    '@Deprecated' Name ('(' String ')')?
36    | '@Override'
37    | '@NonConservative' ;
38
39  StartRule ::=
40    'start' Name ';' ;
41
42
43  ##### Productions
44
45  Prod ::= GrammarAnnotation*
46    ( LexProd | ClassProd | InterfaceProd

```

```

47     | AbstractProd | ExternalProd | EnumProd );
48
49 LexProd ::=
50     ('fragment' | 'comment')*
51     'token' Name
52     LexOption? ActionBlock?
53     '=' (LexAlt || '|' )+
54     (':' ('->' Name)? ActionBlock?
55     (Name ('->' QualifiedName (':' ActionBlock)? )? )? )? ';' ;
56
57 ClassProd ::=
58     SymbolDefinition* Name
59     ( ('extends'|'implements') (RuleReference || ',' )+
60     | ('astextends'|'astimplements') (MCType || ',' )+
61     )*
62     ActionBlock?
63     ('=' Alt ('|' Alt )* )? ';' ;
64
65 InterfaceProd ::=
66     'interface' SymbolDefinition* Name
67     ( 'extends' (RuleReference || ',' )+
68     | 'astextends' (MCType || ',' )+
69     )*
70     ('=' Alt ('|' Alt )* )? ';' ;
71
72 AbstractProd ::=
73     'abstract' SymbolDefinition* Name
74     ( ('extends'|'implements') (RuleReference || ',' )+
75     | ('astextends'|'astimplements') (MCType || ',' )+
76     )*
77     ('=' (Alt || '|')+ )? ';' ;
78
79 ExternalProd ::=
80     'external' SymbolDefinition* Name MCType? ';' ;
81
82 EnumProd ::=
83     'enum' Name '=' (Constant || '|')+ ';' ;
84
85 Card ::=
86     '?' | '*' | '+'
87     | 'min' '=' Digits ('max' '=' (Digits | '*'))?
88     | 'max' '=' (Digits | '*') ;
89
90 RuleReference ::= SemanticpredicateOrAction? Name
91     ('<' Digits '>')? ;
92
93
94 //##### Production body
95
96 Alt ::= '<rightassoc>'? GrammarAnnotation? RuleComponent* ;
97
98 RuleComponent =

```

4. MontiCore Grammar for Language and AST Definitions

```
99     NonTerminalSeparator
100   | Block
101   | NonTerminal
102   | Terminal
103   | KeyTerminal
104   | TokenTerminal
105   | ConstantGroup
106   | Eof
107   | SemanticpredicateOrAction
108   | LexNonTerminal
109   ;
110
111 NonTerminalSeparator ::=
112   (Name ':' )? '(' Name ('@' Name)? '&'? '||' String ')' ('*' | '+') ;
113
114 Block ::=
115   '(' (Option ':' | Option? 'init' ActionBlock ':' )?
116   Alt ('|' Alt )* ')' ('?' | '*' | '+')? ;
117
118 Option ::=
119   'options' '{' OptionValue+ '}' ;
120
121 OptionValue ::=
122   Name '=' Name ';' ;
123
124 NonTerminal ::=
125   (Name ':' )? Name ('@' Name)? ('!!' Name?)? '&'? ('?' | '*' | '+')? ;
126
127 Terminal ::=
128   (Name ':' )? String ('?' | '*' | '+')? ;
129
130 KeyTerminal ::=
131   (Name ':' )? KeyConstant ('?' | '*' | '+')? ;
132
133 KeyConstant ::=
134   'key' '(' (String || '|')+ ')' ;
135
136 TokenTerminal ::=
137   (Name ':' )? TokenConstant ('?' | '*' | '+')? ;
138
139 TokenConstant ::=
140   'token' '(' String ')' ;
141
142 Constant ::=
143   (Name ':' )?
144   ( String |
145     KeyConstant |
146     TokenConstant
147   ) ;
148
149 ConstantGroup ::=
150   (Name ':' )? '[' (Constant || '|')+ ']' ('?' | '*' | '+')? ;
```

```

151
152 Eof ::= 'EOF' ;
153
154 SemanticpredicateOrAction ::=
155   '{' ExpressionPredicate '}' '?' | ActionBlock ;
156
157
158 //##### Concepts and Control Rules
159
160 Concept ::=
161   'concept' Name MConcept ;
162
163 SplitRule ::=
164   'splittoken' (String || ',")+ ';' ;
165
166 KeywordRule ::=
167   'nokeyword' (String || ',')+ ';' ;
168
169
170 //##### AST Rules
171
172 ASTRule ::=
173   'astrule' Name
174   ( 'astextends' (MType || ',')+
175     | 'astimplements' (MType || ',')+
176   ) *
177   ('='
178     (GrammarMethod | AdditionalAttribute) *
179   )? ';' ;
180
181 GrammarMethod ::=
182   'method' ('public' | 'private' | 'protected')?
183   'final'? 'static'?
184   MReturnType Name '(' (MethodParameter || ',') * ')'
185   ('throws' (MType || ',')+ )?
186   ActionBlock ;
187
188 MethodParameter ::= MType Name ;
189
190 AdditionalAttribute ::=
191   (Name ':' )? MType Card? ;
192
193
194 //##### Lexer
195
196 LexAlt ::=
197   LexComponent* ;
198
199 LexComponent ::=
200   LexBlock
201   | LexCharRange
202   | LexChar

```

4. MontiCore Grammar for Language and AST Definitions

```
203 | LexAnyChar
204 | LexString
205 | LexActionOrPredicate
206 | LexNonTerminal
207 | LexSimpleIteration
208 | ;
209
210 LexBlock ::=
211   '~'? '('
212   (LexOption ':' | LexOption? 'init' ActionBlock ':')?
213   (LexAlt || '|' )+ ')' ('?'|'*'|'+')? ;
214
215 LexCharRange ::= '~'? Char '..' Char ;
216
217 LexChar ::= '~'? Char ;
218
219 LexAnyChar ::= '.' ;
220
221 LexString ::= String ;
222
223 LexActionOrPredicate ::=
224   '{' ExpressionPredicate '}' '?' ;
225
226 LexNonTerminal ::= Name ;
227
228 LexSimpleIteration ::=
229   (LexNonTerminal | LexString | LexChar | LexAnyChar)
230   ('?'|'*'|'+') ('?')? ;
231
232 LexOption ::=
233   'options' '{' Name '=' Name ';' '}' ;
234
235
236 //##### Symbol Table
237
238 SymbolDefinition ::=
239   'symbol'
240   | 'scope'
241   | 'scope' '(' ('shadowing' | 'non_exporting' | 'ordered')+ ')' ;
242
243 SymbolRule ::=
244   'symbolrule' Name
245   ('extends' (MCType || ',')+
246   | 'implements' (MCType || ',')+
247   )*
248   ('=' (GrammarMethod | AdditionalAttribute)*)? ';' ;
249
250 ScopeRule ::=
251   'scoperule'
252   ('extends' (MCType || ',')+
253   | 'implements' (MCType || ',')+
254   )*
```

```

255     ('=' (GrammarMethod | AdditionalAttribute)*)? ';' ;
256
257
258     // ##### External Productions
259
260     ActionBlock ::= '{' Action '}' ;
261
262     Action ::= ... ;
263
264     ExpressionPredicate ::= ... ;
265
266     MCGConcept ::= ... ;
267
268     // ##### Types and References
269
270     QualifiedName ::= (Name || '.' )+ ;
271
272     MCGType ::= ... ;

```

Listing 4.40: EBNF of the MontiCore grammar MCG

**Tip 4.41: The MontiCore Grammar**

The full and reusable MontiCore grammar can be found in the MontiCore repository here:

1	Repository: MontiCore/monticore github	Files
2	Directory: monticore-grammar/src/main/grammars/	
3	Files: de.monticore.grammar.Grammar.mc4	
4	de.monticore.grammar.Grammar_WithConcepts.mc4	

The latter grammar also includes several additional concepts that help to control the parsing process and related issues.

Chapter 5

Abstract Syntax Tree

A MontiCore grammar defines the concrete and abstract syntax (AST) of a language in an integrated fashion. Thus, lexer, parser and the classes for the abstract syntax can be automatically derived. This chapter explains the structure of the AST classes derived from a MontiCore grammar.

Besides the AST classes and the parser, MontiCore also generates builders and other helpful classes by deriving them from a grammar. The generated files for any given grammar are summarized in Table 5.1. They can be found in the output folder located in the target package plus the subpackage given in Table 5.1.

Table 5.1: Files generated from a grammar

File(s) for	Explanation	Subpackage
AST Classes	Classes are generated by MontiCore and instantiated during parsing	<code>_ast</code>
Parser and Lexer	Read a file and produce AST	<code>_parser</code>
Node Builders	Used to create AST objects (must also be used to create AST objects by hand). They can be modified to inject handcoded AST subclasses.	<code>_ast</code>
Mill	For providing builders	-
Visitors	For traversing the AST and the symboltable (see Chapter 8)	<code>_visitor</code>
Symboltable	Symbols, scopes, visitors and symboltable serialization (See Chapter 9)	<code>_symboltable</code>
Context Condition Infrastructure	Environment to implement context conditions (See Chapter 10)	<code>_cocos</code>
CD Representation of the Grammar	Help understanding the grammar and AST structure (for documentation)	<code>report</code> (no subpackage, next to the grammar package)

5.1 Mapping Nonterminals to the AST

As described in Chapter 4 a production defines a nonterminal. It comprises a nonterminal on the left-hand side and the production body on its right-hand side. For every nonterminal defined in a grammar an AST class is generated. Nonterminals on the right-hand side of a production result in compositions (cf. Figure 5.2). If a nonterminal occurs more than once or has a cardinality greater than 1, there is an *s* added to the derived name, i.e. `State*` is equivalent to `states:State*` with the exception of some trailing *s* differing at the end of some of the methods. Nonterminals that are marked with a *** or *+* will result in list attributes. Furthermore, several equally named nonterminals are grouped and result in one single list (cf. Figure 5.2). As a consequence, nonterminals that should be distinguishable need to be named explicitly with different names.

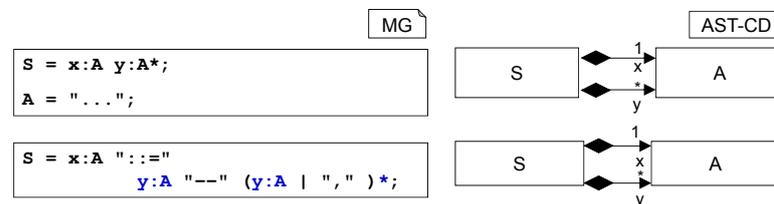


Figure 5.2: Sequences of nonterminals in the AST

Optional nonterminals, i.e., nonterminals marked with a question mark or within an alternative, are mapped to Java `Optionals` of the type of the nonterminal (cf. Figure 5.3). If the `Optional` is present then the nonterminals occurred in the parsed model. If a nonterminal occurs several times (even optionally), e.g. `A? A?`, it is also stored in a list.

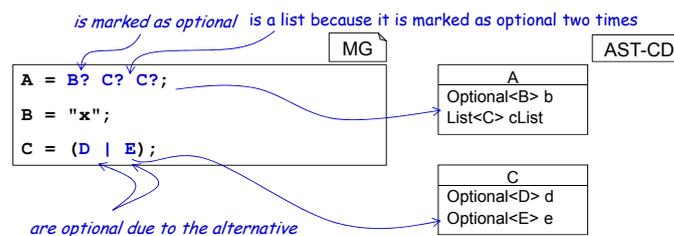


Figure 5.3: Optional nonterminals

Storage in lists is efficient, but forgets some of the information e.g. the order of interleaving. For example in the pathological cases `(A|A)` or `(A|B)*` the alternative taken respectively the order in which *A*s and *B*s occur cannot be fully reconstructed from the AST. In this case, restructuring the grammar helps.

Finally, tokens defined by regular expressions over characters are mapped to `Strings`. Consider the predefined `Name` token, for example, it does not map to an AST class `ASTName` and thus is mapped to a `String` attribute, when used within a production body.

5.2 Interface and Abstract Nonterminals

Interface and abstract nonterminals are mapped differently to the AST. In the AST an interface nonterminal I maps to a Java interface I (actually $ASTI$). If there is a nonterminal A that implements I (cf. Figure 5.4), ASTA will implement $ASTI$. Hence, although the grammars in Listing 4.19 (p. 51) and 4.20 (p. 51) are equivalent regarding the concrete syntax, their AST data structures differ significantly.

Similarly, an abstract nonterminal B maps to an abstract Java class B (actually $ASTB$). Thus, the nonterminal `AutomatonElement` in Figure 5.5 is mapped to an abstract class `AutomatonElement` in Java. Usually other nonterminals implement or extend those nonterminals, which will be explained in Section 5.3.

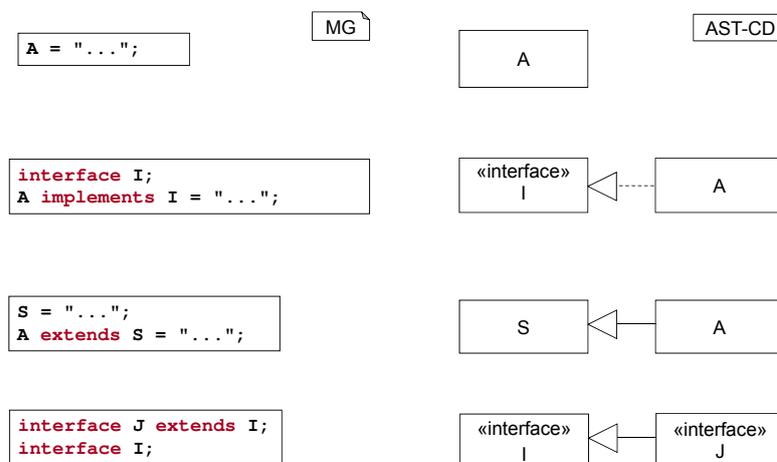


Figure 5.4: How interfaces in a grammar map to the abstract syntax

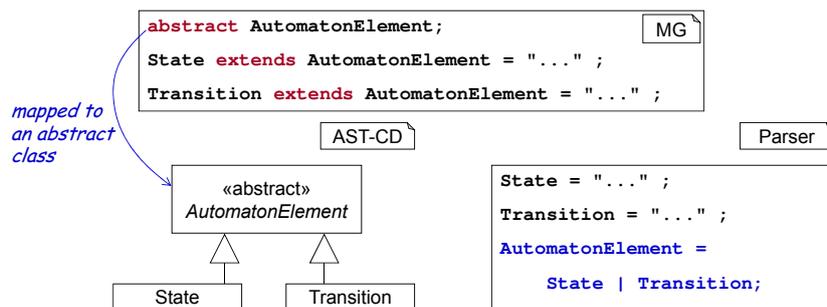


Figure 5.5: How abstract nonterminals in a grammar map to abstract classes

5.3 Extending Nonterminals: `astimplements`, `astextends`

The extension mechanism for the AST is straightforward (cf. Figure 5.7 and 5.4). The keyword `extends` directly maps to an extension on the AST classes. Interface nontermi-



Tip 5.6: Interface and Abstract Nonterminals

Interfaces and abstract nonterminals help in structuring the grammar in order to keep it *readable* and *manageable*.

Even more important is that they are excellent mechanisms to extend a language (cf. Section 5.3 and Chapter 7).

They also help in structuring the abstract syntax, represented in a set of classes and interfaces. Therefore a well structured AST is based on a good structure of the grammar.

nals map to Java interfaces. And thus the nonterminal extension on interfaces also maps to Java interface extension. Therefore, unsurprisingly: Normal productions (cf. Section 4.2.6) can only be extended by normal productions, while abstract productions can be extended by normal and abstract productions. Interfaces (cf. Section 4.2.4) can only be extended by interfaces (cf. Section 4.4), but implemented by normal and abstract nonterminals.

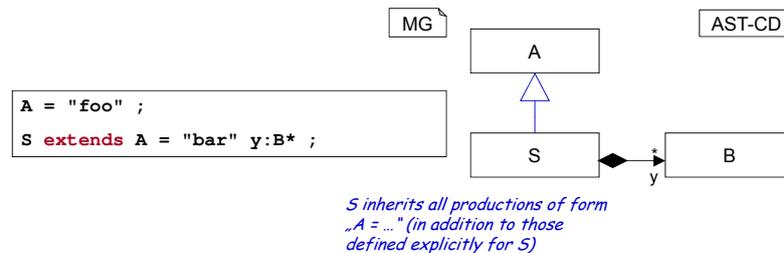


Figure 5.7: Productions extending other productions

The MCG grammar format allows adding interfaces or super classes to the AST classes without any impact on the concrete syntax. To add an interface to an AST class the keyword `astimplements` is used. The keyword `astextends` is used to extend AST classes (i.e. derived from normal nonterminals) by other Java classes and AST interfaces by other Java interfaces.

`astimplements` and `astextends` can only be used when followed by a Java class or interface respectively. They are directly realized in the structure of the generated AST (cf. Figure 5.8 and 5.9).

In the `astimplements` statement any Java interface, e.g. `Observer`, also including interfaces derived from the grammar, e.g. `ASTI`, can be used. However, some restrictions do apply. For instance, given `B astimplements IF`, the Java interface `IF` enforces its methods being implemented by the class `ASTB` (cf. Figure 5.8). There are three options: (1) the nonterminal `B` is declared as abstract, (2) the missing methods are added using the `astrule` statement or (3) a handwritten version of class `ASTB` is provided that implements these methods (see Section 5.10).

The Java class `CL` in the statement `B astextends CL` also obeys restrictions. Because Java only allows single inheritance, the `astextend` statement lets `CL` replace the normal superclass, which is directly or transitively a subclass of `ASTCNode`. Therefore, `CL` must

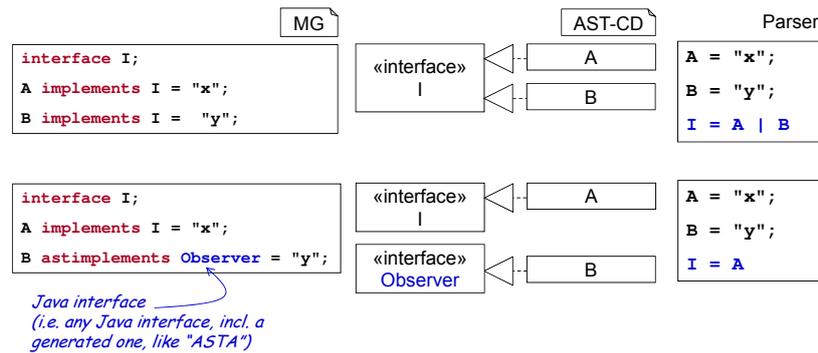


Figure 5.8: Implements in abstract and concrete syntax

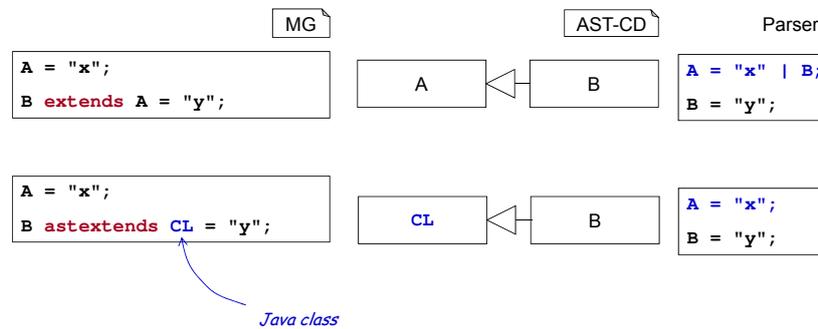


Figure 5.9: Inheritance in abstract and concrete syntax

implement interface `ASTNode`. For example it could be defined as a subclass of `ASTCNode`. Thus, it is not possible to use foreign classes, such as `Observable`. Additionally, in order to use a Java class or interface which is being extended or implemented by an AST class, the used Java class must be fully qualified in the grammar.

Please note, that the attributes inherited from `CL` are ignored by MontiCore's standard functionality, e.g. when parsing. Therefore, the developer is responsible for completing the object data. Again, this can be done by (1) declaring the nonterminal `B` abstract, (2) using the `astrule` statement for `B`, or (3) providing a handwritten version of `ASTB`.

If `B` is declared as an interface, it is allowed to use arbitrary Java interfaces for `IF`. For example in `interface B astextends IF`, Java interface `IF` can be chosen freely.

5.4 Extending the Abstract Syntax Implementation

AST rules start with the keyword `astrule` and enable the definition of additional attributes and methods for the generated AST classes, but do not have an impact on the concrete syntax of the language. The notation is aligned to the grammar format, i.e. AST rules are defined similar to productions but start with the keyword `astrule` (cf. Figure 5.10). Here it is also possible to use the keywords `astextends` or `astimplements` as described above.

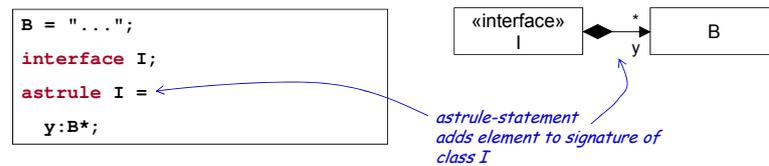


Figure 5.10: Extending the AST structure

The extensions defined with `astrule` will result in additional method signatures in the AST interface and corresponding attributes and methods implementations in the AST classes for all implementing productions.

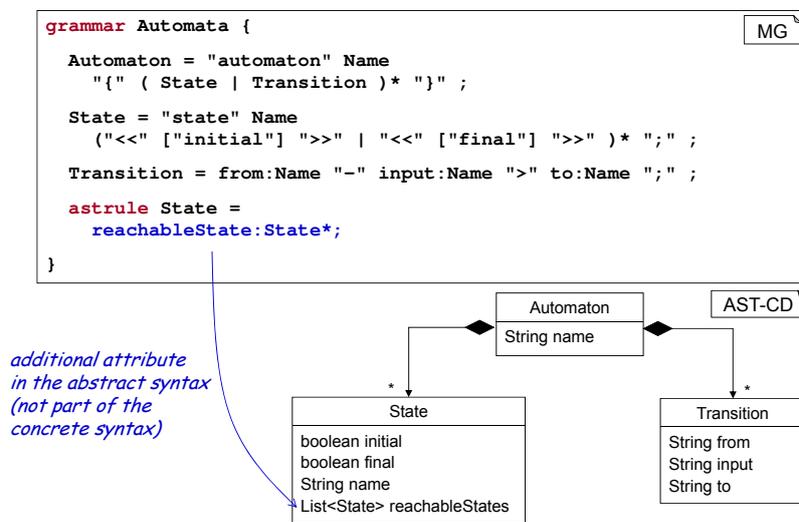


Figure 5.11: Adding attributes in the AST with the `astrule` statement

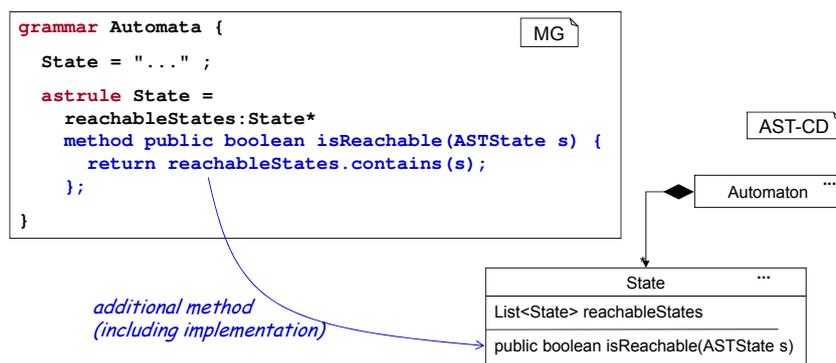


Figure 5.12: Adding methods in the AST with the `astrule` statement

In AST rules, nonterminals can be used in the same way as they are used in productions (cf. Figure 5.11). AST rules produce additional attributes and methods in the generated class. However, this does not affect the concrete syntax and when the parsing is complete, any

newly defined attributes, like `reachableStates`, will not have a defined value directly after parsing.

Additional methods are introduced by the keyword `method` followed by the usual Java syntax for method declarations (cf. Figure 5.12). If an attribute `n` (derived e.g. from nonterminal `n:A`) was already present within the normal production, the type (e.g. `n:B`) specified in the AST rule has precedence. This allows us to override the type chosen by the attribute derivation.



Tip 5.13: AST Rules, like `astextends` and `astrule`

When the resulting abstract syntax (which results in a tree of nodes) does not fully accommodate the developers needs, additional attributes may be added directly in the grammar.

After parsing, the AST objects then, however, need to be completed, for example by using a visitor (see Chapter 8) that fills all the additional attributes.

Methods can also be added using the `astrule` statement, but for complex methods there is a more comfortable approach, using handcoded extensions (see Chapter 14).

Both, methods and getters/setters for attributes that are added using `astrule` are visible in the public signature of a generated AST class.

Please note that the mechanism for a handcoded extension of the AST allows roughly the same effects as adding methods in the product. However, it is usually more comfortable because the handcoded extension can be written directly within a comfortable Java IDE of your choice.

5.5 Terminals in the AST

As stated before, terminals are usually not part of the abstract syntax. In case a terminal is semantically relevant, e.g. like a terminal in an alternative, there are two options to mark it as relevant: (1) Adding an explicit name or (2) surrounding it with square brackets (cf. Figure 5.14). In the first case there will be an attribute with the given name of type `String` in the abstract syntax holding the terminal as value (cf. Figure 5.14). The second variant is usually preferable, as it results in a boolean attribute named like the terminal. In this case the name of the attribute is derived from the terminal. In Figure 5.14 the terminal `"Hello"` is not reflected by the AST, but the attribute `who` is. If the terminal in square brackets is not a suitable attribute name, but e.g. an operator like `"++"`, a name can be added explicitly within the square brackets.

Please note that terminals marked as relevant are usually optional or part of an alternative. Otherwise, these terminals are mandatory in the model and thus the attribute will always hold the same value, providing no information.

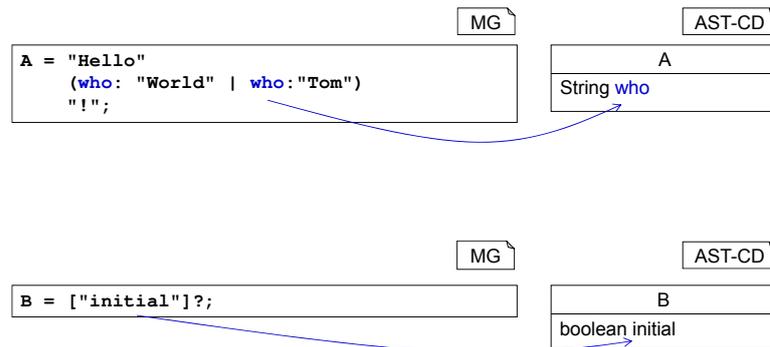


Figure 5.14: Terminals included in the AST

5.6 Enumerations

A list of terminals within square brackets mimics an enumeration and results in an attribute of type `int` in the abstract syntax as shown in Figure 5.15. Furthermore, for each terminal in the alternative there is a constant stored in an extra class created for such constants (e.g. `ASTConstantsGr` for a grammar called `Gr`). If an explicit keyword name is omitted it is derived automatically from the keyword e.g. `PUBLIC` from `public`. Letters are capitalized. For other symbols, such as `+`, there are default names used, e.g. `plus`. It is possible to choose the same name for different keywords meaning that they are semantically equivalent as shown in Figure 5.15 for `+` and `public`. Since no name can be determined for the constant group, the group is given the name `vis`.

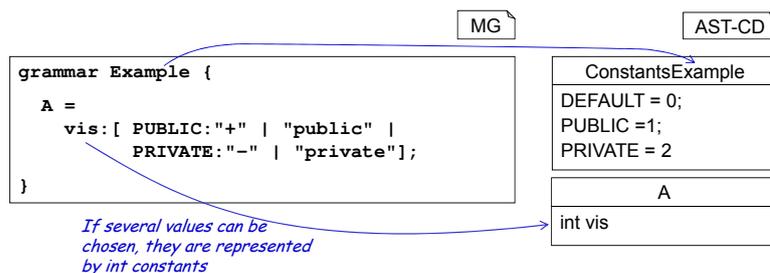


Figure 5.15: Choice of one of several values stored as `int`

In the grammar an enumeration introduced by the `enum` keyword results in an explicit Java enumeration holding the corresponding constants (cf. Figure 5.16). Enumeration nonterminals are used in productions like other nonterminals. Again, it is possible to use a keyword in an enumeration repeatedly, because alternatives with equal names are mapped to the same constant.

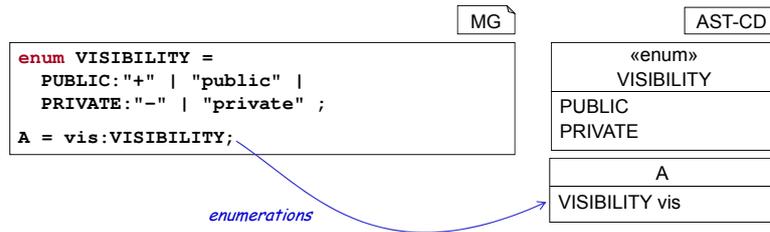


Figure 5.16: Explicit definition of an enumeration

5.7 ASTNode: A Base Interface for AST Classes

All AST classes implement a common interface called `ASTNode` (cf. Figure 5.18). This interface, respectively the default implementations behind it, allows us to compare AST nodes, store comments from the original file and the source position, from where the AST node and its substructure has been parsed.

The `ASTNode` interface shown in Figure 5.17 provides the common signature implemented by all AST classes.

```

1 public interface ASTNode {
2     // Cloning
3     ASTNode deepClone(ASTNode result);
4     ASTNode deepClone();
5
6     // Forms of equalities
7     boolean equalAttributes(Object o);
8     boolean equalsWithComments(Object o);
9     boolean deepEquals(Object o);
10    boolean deepEqualsWithComments(Object o);
11    boolean deepEquals(Object o, boolean forceSameOrder);
12    boolean deepEqualsWithComments(Object o, boolean forceSameOrder);
13
14    // Managing the attached source position (start)
15    boolean isPresent_SourcePositionStart();
16    SourcePosition get_SourcePositionStart();
17    void set_SourcePositionStart(SourcePosition start);
18    void set_SourcePositionStartAbsent();
19
20    // Managing the attached source position (end)
21    boolean isPresent_SourcePositionEnd();
22    SourcePosition get_SourcePositionEnd();
23    void set_SourcePositionEnd(SourcePosition end);
24    void set_SourcePositionEndAbsent();
25
26    // Managing attached comments
27    boolean add_PreComments(Comment precomment);
28    // in total ~30 methods for managing List<Comment> pre
29

```

5. Abstract Syntax Tree

```
30 boolean add_PostComments(Comment postcomment);
31     // in total ~30 methods for managing List<Comment> post
32 }
```

Listing 5.17: Signature of the ASTNode superclass of all AST nodes

There are several groups of methods available in the interface ASTNode: There are methods for cloning and checking the quality of two nodes respectively node hierarchies (ll. 3ff. and ll. 7ff.). By default, the `deepEquals` methods enforce the same order of children occurring in the AST, but the comparison with a boolean stating whether the order should be considered can also be used.

The source position is internally stored as an optional and thus the usual four methods for their management are provided. Because the source position consists of start and end, we have in total eight methods (ll. 15ff. and ll. 21ff.). The source position is usually only set by the parser, which knows where the AST node comes from. The AST also stores comments before and after an AST node. Both may be a list with more than one element. ASTNode thus provides over 30 methods for list management for the comments before and after a node. Figure 5.18 shows a part of the signature and data structure of all ASTNodes.

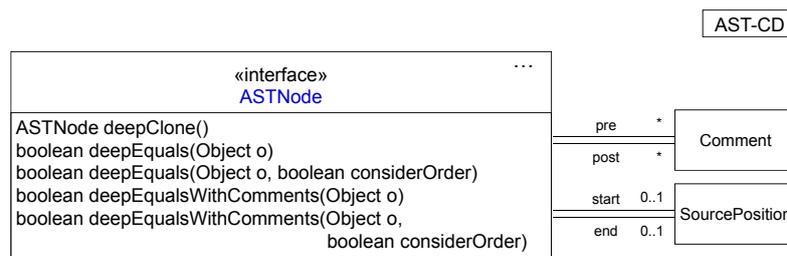


Figure 5.18: Common interfaces of AST classes

For a default implementation of several functions provided by ASTNode objects, the MontiCore runtime environment also provides an abstract subclass ASTCNode that implements `Cloneable`. While it is helpful to know that this standard functionality is provided by all AST objects, in concrete projects they need not be implemented by hand, but the generated subclasses that are derived from the grammar provide already complete implementations.

```
1 Repository: Monticore/monticore github
2 Directory: monticore-runtime/src/main/java/
3 Files:     de.monticore.ast.ASTNode.java
4           de.monticore.ast.ASTCNode.java
```

5.8 Generated ASTNode Subclasses

Given a production, one can schematically infer the interface provided by the subclass of ASTNode that is created. We demonstrate this on the `ASTState` class derived from

the State production (from an enhanced automaton, where states can have priorities, sub-states and transitions enclosed in curly brackets):

```

1 State = "state" Name prio:NatLiteral?
2   ( ["initial"] | ["final"] ) *
3   ( ("{" (State | Transition)* "}") | ";" ) ;

```

The signature of the generated class `ASTState` directly implements all functions to access the parsed information, that is retrieved from the right-hand side of the production. The class furthermore implements all methods from `ASTNode` and it also provides functions that allow to deal with the language specific symbol and scope structure. This will be discussed in Chapter 9. Listing 5.19 and 5.20 show the (beautified) signature without the method bodies.

```

1 package automata3._ast;
2
3 public class ASTState extends ASTCNode implements ASTAutomata3Node
4 {
5   // Storing the parsing result:
6   protected String name;
7   protected Optional<ASTNatLiteral> prio = Optional.empty();
8   protected List<ASTState> states = new ArrayList<>();
9   protected List<ASTTransition> transitions = new ArrayList<>();
10  protected boolean initial;
11  protected boolean r__final;
12
13  // Constructor (but we use builders):
14  protected ASTState();
15
16  // Visitor management:
17  void accept(Automata3Traverser visitor);
18  void accept(MCBasicsTraverser visitor);
19  void accept(MCCommonLiteralsTraverser visitor);

```

Listing 5.19: Signature of the generated AST class to represent states: part 1

Like any other nonterminal, `ASTState` extends the MontiCore RTE class `ASTCNode`. Furthermore, each grammar generates an abstract interface `ASTAutomata3Node` named after the grammar. While this interface is empty it can be used by visitors that want to visit exactly all nodes defined in a certain grammar.

The attributes defined in ll. 5ff. of Listing 5.19 are protected, only accessible and manipulatable through the in Listings 5.20 and 5.21 described get and set functions. There exists only the empty constructor (ll. 13ff.), but any object of that class should be defined by using the generated builders (see Section 5.9).

For each directly or indirectly included grammar the node also provides an appropriate `accept` method (ll. 16ff.), which allows the object to participate in the visitor pattern (see Chapter 8) and also to reuse visitors of the imported languages within the new composed language. Here, the Automata grammar relies on `MCBasics` and `MCommonLiterals`.

When a nonterminal is marked as interface or abstract, then the resulting class is also an interface or abstract. In case of an interface, only the signature of methods is provided and the attributes are deferred to the implementing classes. In Listing 5.20 the second part of class `ASTState` is shown.

```

1 // Treatment of children and semantically relevant terminals:
2 String getName();
3 void setName(String name);
4
5
6 // Optional attributes have four methods:
7 boolean isPresentPrio();
8 ASTNatLiteral getPrio();
9 void setPrio(ASTNatLiteral prio);
10 void setPrioAbsent();
11
12 boolean isInitial();
13 void setInitial(boolean initial);
14
15 boolean isFinal();
16 void setFinal(boolean r__final);
17 }

```

Listing 5.20: Attribute management signature of a generated AST class: part 2

For each nonterminal with a mandatory occurrence one get and one set method is generated to allow access to the attribute. Lines 3ff. show this for nonterminal `Name`. The set function is based on the general principle, that `null` is never used as a value.

For all nonterminals that may occur in cardinality different from 1, a number of additional methods is generated that directly reflect the functionality of the implementation. That means all functions for `Lists` and `Optionals` are directly available within the `ASTNode` preventing that direct access would be necessary. `Optional` attributes have four methods and `List` attributes more than 30 methods in total (see Java's `List` methods).

If the attribute is optional, like `prio` (marked with a "?"), then a method allows us to question the presence (l. 7 in Listing 5.20) and another method allows us to set the value to absent (l. 10). Please also note that there is one get method for an optional attribute: `getPrio` which issues an error message and raises an exception, if the value is absent. `getPrio` thus must be guarded by `isPresentPrio`.

Line 12 shows, how optional, but semantically relevant terminals are managed. And line 15 demonstrates how MontiCore wraps a Java keyword that coincidentally also occurred in the grammar as keyword: it maps `"final"` to `"r__final"`.

For a nonterminal with cardinality greater than one, MontiCore uses a `List` for implementation and provides the full `List` signature consisting of about 30 methods to access and manipulate the list, without having developers to explicitly handle the list. For each attribute a full set of `List` methods is generated. To distinguish the methods they are attached with the nonterminal name and if several objects are involved also with an additional "s". The advantages of these methods are that, on the one hand, writing code

against the AST is more efficient and on the other hand, each of these methods can easily be adapted with handwritten code, e.g. disallowing certain manipulations or enforcing consistency vs. additional objects or back-links that create redundancy. The signature for the `Transition*` attributes is depicted in Listing 5.21.

```

1  void clearTransitions();
2  boolean addTransition(ASTTransition element);
3  boolean addAllTransitions
4      (Collection<? extends ASTTransition> collection);
5  boolean removeTransition(Object element);
6  boolean removeAllTransitions(Collection<?> collection);
7  boolean retainAllTransitions(Collection<?> collection);
8
9  boolean containsTransition(Object element);
10 boolean containsAllTransitions(Collection<?> collection);
11 boolean isEmptyTransitions();
12 int sizeTransitions();
13
14 void addTransition(int index, ASTTransition element);
15 boolean addAllTransitions(int index,
16     Collection<? extends ASTTransition> collection);
17 ASTTransition setTransition(int index, ASTTransition element);
18 ASTTransition getTransition(int index);
19 int indexOfTransition(Object element);
20 int lastIndexOfTransition(Object element);
21 ASTTransition removeTransition(int index);
22 List<ASTTransition> subListTransitions(int start, int end);
23
24 Iterator<ASTTransition> iteratorTransitions();
25 ListIterator<ASTTransition> listIteratorTransitions();
26 ListIterator<ASTTransition> listIteratorTransitions(int index);
27 void forEachTransitions(Consumer<? super ASTTransition> action);
28 Spliterator<ASTTransition> spliteratorTransitions();
29 boolean removeIfTransition
30     (Predicate<? super ASTTransition> filter);
31 void replaceAllTransitions(UnaryOperator<ASTTransition> operator);
32 void sortTransitions
33     (Comparator<? super ASTTransition> comparator);
34
35 ASTTransition[] toArrayTransitions(ASTTransition[] array);
36 Object[] toArrayTransitions();
37 Stream<ASTTransition> streamTransitions();
38 Stream<ASTTransition> parallelStreamTransitions();
39
40 boolean equalsTransitions(Object o);
41 int hashCodeTransitions();
42
43 List<ASTTransition> getTransitionList();
44 void setTransitionList(List<ASTTransition> transitions);

```

Listing 5.21: Signature for a `List` attribute in a generated AST class: part 3

Almost all methods are direct delegators to the internally used `List`. Only the last two (l. 43ff. in Listing 5.21) allow to retrieve the complete list or set a new list. We include the last two methods, because we generally assume that users are experienced, but would suggest to use the first 29 methods only and refer to Java's `List` implementation to understand their effect. Finally, each AST object implements the mechanisms to compare and clone it (cf. Listing 5.22).

```

1
2  boolean deepEquals(Object o);
3  boolean deepEquals(Object o, boolean forceSameOrder);
4  boolean deepEqualsWithComments(Object o);
5  boolean deepEqualsWithComments(Object o, boolean forceSameOrder);
6
7  boolean equalAttributes(Object o);
8  boolean equalsWithComments(Object o);
9
10 ASTState deepClone();
11 ASTState deepClone(ASTState result);
12 }

```

Listing 5.22: Comparison and cloning in a generated AST class: part 4

The set of `deepEquals` in ll. 2f of Listing 5.22 allows to compare entire AST trees including all sub-objects. Its variants allow to control, whether orders in list are relevant, which is the default in `deepEquals(o)`. Variant `deepEqualsWithComments` also checks comments, but none of the variant checks source code positions.

The `deepClone` function in l. 10 of Listing 5.22 produces a copy of the complete AST structure including copies of all sub-objects within the AST. The method `equalAttributes` only checks the attributes with cardinality one, enumerations and simple types, but omits comparison of nonterminal types, `Lists` and `Optionals`.



Tip 5.23: EMF (Eclipse Modeling Framework) Integration is Available

MontiCore is a standalone tool infrastructure.

But if desired, the MontoCore generator can generate signatures for the AST node classes in such a form that they conveniently integrate into the Eclipse Modeling Framework (EMF) [SBPM08] infrastructure.

Among others, this changes the storage of object lists to use `EObjectContainmentEList`, defines functions such as `eGet`, `eSet`, `eUnset`, `eIsSet`, `eBaseStructuralFeatureID`, `eDerivedStructuralFeatureID` and adds notifications using `eNotify` when something changes in the AST e.g. through a setter method.

The following example in Listing 5.24 shows the methods and their signature provided in addition to the extended generation of AST classes, generated to be EMF compatible:

```

1 package automata3._ast;
2
3 public class ASTState extends de.monticore.emf._ast.ASTECNode
4     implements ASTAutomata3Node
5 {
6     // Storing the parsing result:
7     protected String name;
8     protected Optional<ASTNatLiteral> prio = Optional.empty();
9     protected List<ASTState> states =
10         new EObjectContainmentEList<ASTState>(
11             ASTState.class, this,
12             Automata3Package.ASTState_States);
13     protected List<ASTTransition> transitions =
14         new EObjectContainmentEList<ASTTransition>(
15             ASTTransition.class, this,
16             Automata3Package.ASTState_Transitions);
17     protected boolean initial;
18     protected boolean r_final;
19
20     // other methods omitted, because they are not changed
21
22     // EMF ;
23     Object eGet(int featureID, boolean resolve, boolean coreType);
24     void eSet(int featureID, Object newValue);
25     void eUnset(int featureID);
26     boolean eIsSet(int featureID);
27     int eBaseStructuralFeatureID(int featureID, Class<?> baseClass);
28     int eDerivedStructuralFeatureID(int featureID, Class<?> baseClass
29     );
30 }

```

Listing 5.24: EMF version of the ASTState class signature

5.9 Node Construction Using the Node Builder Mill

New AST nodes are constructed using builders that are available through a static delegator pattern (cf. Section 11.1). A *node builder mill* provides builders for each nonterminal that is defined in the grammar of a language. Therefore, the concrete signature varies depending on the nonterminals of a language and their structure.

The following Listing 5.25 shows the signature of the AutomataMill created for the example language for finite automata (cf. Section 21.1). A node builder mill provides a set of static create methods that delegate to internal builder create methods. The two staged builder process is necessary to allow builders to be adaptable in a language composition. That means methods relying on the builder mill of a sublanguage can be applied in a composed language without modification.

```

1 package automata;
2
3 public class AutomataMill {
4
5     static ASTAutomatonBuilder automatonBuilder();
6
7     static ASTStateBuilder stateBuilder();
8
9     static ASTTransitionBuilder transitionBuilder();
10 }

```

Listing 5.25: Signature of the builder mill for all Automaton AST classes

A method, like `stateBuilder`, creates a builder object that is responsible to create a state object, i.e., from class `ASTState` or a subclass thereof.

As usual, the builder class itself provides methods to set the attributes individually before the object is created. Please note that an AST node always has attributes for comments, source position, and potential links to a scope and a symbol that the AST node defines. Therefore, an AST node builder provides methods to manage these as well. Listing 5.26 demonstrates this on a builder for the nonterminal `State`.

In general, these methods do not differ from the methods generated for the class `ASTState`, but have one important difference: Where the method in the class `ASTState` has the return type `void` or is the boolean result of an add operation, the corresponding builder method returns the builder itself. This is helpful for a chaining of calls for a builder `b`, such as `b.setName("Ping").setInitial(true).addTransitions(x)`¹.

A comparison of the `ASTStateBuilder` in Listing 5.26 and the node `ASTState` in Figures 5.20 and 5.21 shows the large overlap of these signatures. Therefore, only the most important methods are repeated in Listing 5.26.

```

1 package automata3._ast;
2
3 public class ASTStateBuilder extends
4     ASTNodeBuilder<ASTStateBuilder> {
5     // Setting an attribute
6     ASTStateBuilder setName(String name);
7
8     // Setting a boolean attribute
9     ASTStateBuilder setInitial(boolean initial);
10    ASTStateBuilder setFinal(boolean r__final);
11
12    // Setting an Optional attribute
13    ASTStateBuilder setPrio(ASTNatLiteral prio);
14    ASTStateBuilder setPrioAbsent();
15 }

```

¹To enable this chaining, we also had to use a generic superclass `ASTNodeBuilder`, which embodies the return type of each setter as `realBuilder` object with the correct type.

```

16 // Setting a List valued attribute
17 ASTStateBuilder setTransitionsList(
18     List<ASTTransition> transitions);
19 ASTStateBuilder clearTransitions();
20 ASTStateBuilder addTransition(ASTTransition element);
21 ASTStateBuilder addAllTransitions(
22     Collection<? extends ASTTransition> collection);
23 ASTStateBuilder removeTransition(Object element);
24 ASTStateBuilder addTransition(int index,ASTTransition element);
25     // ... in total ~30 methods to handle the transition list
26
27 ASTStateBuilder setStatesList(List<ASTState> states);
28 ASTStateBuilder addState(ASTState element);
29     // ... in total ~30 methods to handle the state list
30
31 // Inherited methods for attributes from ASTNodeBuilder
32 // (first for the Optionals)
33 ASTStateBuilder set_SourcePositionStart(SourcePosition start);
34 ASTStateBuilder set_SourcePositionStartAbsent();
35
36 ASTStateBuilder set_SourcePositionEnd(SourcePosition end);
37 ASTStateBuilder set_SourcePositionEndAbsent();
38
39 // Inherited methods for attributes from ASTNodeBuilder
40 // (the Lists of Comments)
41 ASTStateBuilder add_PreComment(Comment element);
42 ASTStateBuilder set_PreCommentList(List<Comment> comments);
43     // ... in total ~30 methods to handle the pre comments
44
45 ASTStateBuilder add_PostComment(Comment element);
46 ASTStateBuilder set_PostCommentList(List<Comment> comments);
47     // ... in total ~30 methods to handle the post comments
48
49 // Is the object contents valid?
50 boolean isValid();
51
52 // Finally constructing the object
53 ASTState build();
54 }

```

Listing 5.26: Signature of the Builder for State objects: part 1

Line 6 of Listing 5.26 shows how a normal attribute defined by the production is treated. Boolean attributes are handled in a similar way (ll. 9f). For optional attributes, such as `prio`, several methods exist (ll. 13f).

List valued attributes, derived from nonterminals with multiplicity higher than 1, can be set as list, but also through the about 30 methods allowing to manipulate the list, e.g. by adding individual new elements. Beginning with l. 17 Listing 5.26 shows an excerpt of the more than 30 methods per list. In the `ASTStateBuilder` case, each attribute `State*` and `Transition*` have their own 30 methods.

Starting in l. 33 the signature for setting and manipulating the inherited attributes is shown. To avoid name clashes, some of the inherited methods have an underscore in their names.

At the end of each building activity the AST object is created using the `build()` method. Please note, that it is possible to use this method several times. Each time a new object is created, but (if not changed) all objects are containing the same attribute values. In particular, the children of such objects are shared. When you want to get a complete copy, please use the `deepClone` method.

Please also note, that the builder may fail with an exception if not all mandatory attributes are provided. This can be checked ahead using the `isValid` method. Internally the `build` methods uses this method as well before constructing an ASTs. In case the method returns false the builder would fail with an error message.

For all optional and list attributes and especially those inherited from `ASTCNode`, the AST builder sets defaults. An `Optional` value is by default absent, a `List` is empty and a `boolean` is false.

While the `set` and `add` methods are the most important, it is also possible to retrieve data stored in the builder. Listing 5.27 shows a small excerpt of the builder for `ASTState` with some `get` methods.

```
1 // Retrieving attribute values
2 String getName();
3 boolean isInitial();
4 boolean isFinal();
5
6
7 // Retrieving an Optional value
8 ASTNatLiteral getPrio(); // partial
9 boolean isPresentPrio();
10
11 // Some inherited retrieval methods for
12 // attributes from ASTNodeBuilder
13 SourcePosition get_SourcePositionStart();
14 boolean isPresent_SourcePositionStart();
15 SourcePosition get_SourcePositionEnd();
16 boolean isPresent_SourcePositionEnd();
17
18 // Some retrievers for the Transition* attribute
19 boolean containsTransition(Object element);
20 boolean isEmptyTransitions();
21 int sizeTransitions();
22 ASTTransition getTransition(int index);
23 int indexOfTransition(Object element);
24 List<ASTTransition> subListTransitions(int start, int end);
25
26 Iterator<ASTTransition> iteratorTransitions();
27 ListIterator<ASTTransition> listIteratorTransitions();
28
```

```

29 // get for the full Transition* list
30 List<ASTTransition> getTransitionList();

```

Listing 5.27: Retrieving methods for a Builder class: part 2

Builder functions for deconstructing an unfinished AST object will rarely be used, but retrieving already added elements or checking whether an element is already in a list is sometimes helpful.

In general, tool developers are responsible to ensure that the attributes of the created nodes will always have correct values. Many MontiCore functions rely on a syntactically correct AST. In particular, MontiCore functions very rarely deal with `null` values, because they assume that absent values are explicitly defined as `Optionals`.

The validity of the AST node can be pre-checked by the above mentioned `isValid` method and generally corresponds to the definition of the production. However, there are configurations definable by productions that are not checked. For example in alternatives (e.g., A and B in $N = A \mid B$;) it is not checked that exactly one alternative exists, but both are optional. Furthermore, minimal and maximal values for lists of attributes (e.g., `ast rule N = A min = 3 max = 5`;) are not checked by the builder and only considered during parsing. The builder can be manually adapted using the TOP-mechanism to add this functionality if desired.

Tool developers are strongly encouraged to use builders to create new AST objects to ensure that the creation process can be replaced, e.g., to use custom node classes. No initialization is needed for a builder mill; a direct call is possible and encouraged.

However, if a completely new AST is to be built from scratch, then it is sometimes more efficient to define a string that contains the concrete model and use a parser to parse the string for building the AST.



Tip 5.28: Use Node Builders to Enable Reuse

When you manipulate the AST and create new nodes, then you are strongly advised to use the provided node builders.

This greatly helps to keep the actual implementation of the nodes hidden from the usage. This is a prerequisite for reusing handwritten code and for a language component to be embedded in the composite language.

Furthermore, when you want to provide your own handwritten extension and do not want to use the mechanism described in Section 5.10, then you can adapt the node builder mill for producing your own version of nodes and others can use your extra functionality without having to notice. New functions could also take advantage of the extra functionality.

5.10 Handwritten Extension of AST Classes and Node Builders

If the generated AST node classes and builders do not completely fulfill the needs and, therefore, often should be extended or overridden with a handwritten implementation. The following approach (similar but not exactly equal to the approach described in Chapter 14) is the best.

5.10.1 Handwritten Extension of AST Classes: TOP-Mechanism

To extend e.g. a class `ASTState` (that would be generated):

1. Create (empty) class `ASTState` in an arbitrary directory `dir` (but not in a directory where generated classes are).
2. Add `dir` to `handcodedPath` (of the MontiCore generator).
3. Run the MontiCore generator (again).
4. Let your own class `ASTState` extend the now existing and newly generated `ASTStateTOP`.
5. Adapt `ASTState` at will.
 - Don't forget to initialize additional attributes and to adapt cloning and comparison methods.
6. If necessary, also adapt the classes that rely on the changed signature such as builders or mills, e.g., by using the same TOP mechanism. This usually is necessary, when attributes have been added.

MontiCore uses a trick here: During generation, it checks in the `handcodedPath`, whether the class `ASTState` that should be generated, has been defined by a developer by hand and therefore already exists. If so, MontiCore does not produce the class `ASTState`, but an abstract superclass called `ASTStateTOP` (cf. Figure 5.29). Figure 5.29 shows an example of a handwritten AST class `State` with manually added attribute `_reachable` and a method `isReachable`.



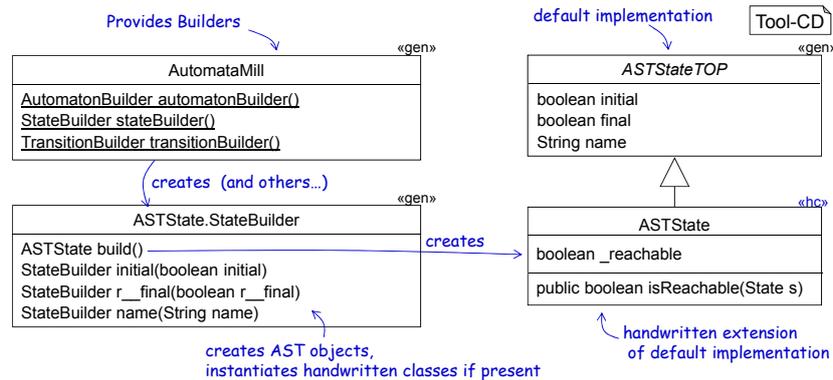
Tip 5.30: Handcoded Extension is easy using the TOP Mechanism

All generated classes `Cls`, including AST classes, node builder mills, etc., can be easily extended with handwritten code using the TOP mechanism. When a class shall be handcoded, then we add it in the `handCoded` path to tell MontiCore to include it and also to generate an abstract superclass of the handcoded class instead.

The handcoded class `Cls` replaces the generated class `Cls`, which now becomes generated as `ClsTOP`. `Cls` (normally) must inherit from `ClsTOP`.

There is nothing more to do. See Section 14.3 for details.

This approach brings numerous advantages:

Figure 5.29: Example: Handwritten AST class `ASTState` injected into the parsing process

- The generated code is still available and can be used.
- The handwritten code is directly integrated into the generated parts, because for example the node builder mill was not changed and still creates `ASTState` objects.
- The handwritten code can inject both, new attributes and method implementations, but also extend the signature of that class making additional functionality externally usable.
- The AST classes that use `ASTState` also do not need to be changed, but interact with the handwritten class.
- The parser directly uses the handwritten class to construct the AST when parsing.

There is a limitation of this approach: After a new handwritten class has been added to the project, a re-generation is necessary. Incremental approaches do not easily detect that, but MontiCore keeps track of the classes it has been looking up when generating. If necessary, cleaning up all generated code before re-generating is radical, but robust.

Please note that we strictly separate generated and handwritten classes in different directory substructures. This also holds when they belong to the same package. This gives us the serious advantage of being able to clean up generated code, or re-generate as often as desired. See also our considerations about agile methodology in Section 1.4.

5.10.2 Handwritten Extension of AST Builders and Mills

For adjusting the AST classes created, it is possible to create a subclass of a node builder mill and override the instance methods to register different (handcoded) mills for specific kinds of nonterminals. This enables using adaptive mills for the creation of AST objects. The builder mills use the static delegator pattern (Section 11.1). That means it has a protected static variable containing an instance of itself to realize the delegation. There is one builder mill attribute for each nonterminal defined in the grammar and one general builder mill attribute that is used for all missing specialized mills. Therefore, a coarse-grained and

also a fine-grained overriding for each type of node is possible. During standard initialization of a node builder mill all these attributes are initialized with the same instance, but a replacement by a custom version is possible. For completeness, we include the protected elements a builder mill provides and that can be used for overriding:

```

1 package automata;
2
3 public class AutomataMill {
4 // ... only the protected elements
5
6 // attributes store individual builder mills for each node
7 // (but all may be the same instance)
8 protected static AutomataMill mill;
9 protected static AutomataMill millASTAutomatonBuilder;
10 protected static AutomataMill millASTStateBuilder;
11 protected static AutomataMill millASTTransitionBuilder;
12 protected AutomataMill();
13
14 protected ASTAutomatonBuilder _automatonBuilder();
15
16 protected ASTStateBuilder _stateBuilder();
17
18 protected ASTTransitionBuilder _transitionBuilder();
19
20 }

```

Listing 5.31: Internal structure of the AutomataMill

The same approach can also be applied in combination with the TOP generation mechanism, i.e., the developer provides a handwritten class AutomataMill inheriting from the then generated AutomataMillTOP (cf. Listing 5.31 and 5.32). In such a hand-coded AutomataMill class, only the creator methods need to be overridden. In Listing 5.32, we assume that MyTransitionBuilder has been implemented accordingly.

```

1 package automata;
2
3 public class AutomataMill extends AutomataMillTOP {
4
5 @Override
6 protected ASTTransitionBuilder _transitionBuilder() {
7     return new MyTransitionBuilder();
8 }
9
10 }

```

Listing 5.32: Handcoded extension of the AutomataMill

Please note, that a manually created static AST mill can still be used by developers, when the language it has been written for is embedded in another language. In a language composition, the concrete mill is internally used and adapted in such a way, that it creates

objects of appropriate subclasses from the composed language. Hence, the user of an AST from a sublanguage is not affected. This is a core technique to enable reuse of functionality of sublanguages on composed languages, because the reused algorithm itself needs no adaptation, not even when creating objects.

Chapter 6

Parser Generation and Use

co-authored by Marita Breuer

This chapter explains how to derive a parser from a given grammar and how to integrate the resulting parser into a DSL tool to read in models. This is mainly done by

1. defining a MontiCore grammar as described in Chapter 4,
2. running the MontiCore generator to generate the parser for the language (cf. Section 6.1) and
3. using the resulting code, which includes the generated parser, AST, builders, visitors etc. and the MontiCore runtime in your DSL tool (cf. Section 6.3).

The parser generated for a language contains a general method for parsing models of the language, i.e., starting with the dedicated *start nonterminal* (cf. Section 6.3). Furthermore, it also provides specific methods to parse each of the sublanguages defined by the other nonterminals. If not explicitly stated otherwise, the first nonterminal defined in a grammar is the start nonterminal.

Sections 6.2 and 6.3 are mainly dedicated to tool developers to get some overview on how to embed a generated parser in your own tool. In contrast, Section 6.1 is mainly for someone who wants to use the generator API directly. This is actually on a meta-meta level (i.e. the level, where the meta-tool resp. the language workbench itself is adapted). Normally it should be sufficient to call the MontiCore language workbench as a closed tool, for example using the command line interface (CLI) or the Gradle integration, which are both explained in Chapter 16.

6.1 Generating a Parser and a Lexer, as done in MontiCore

This section explains the usage of the generator API to generate a parser (i.e. meta-meta level). Thus, this section explains how the language workbench itself can be adapted.

As said, usually it should be sufficient to call the MontiCore language workbench as a closed tool, for example using the command line interface (CLI) or the Gradle integration,

which are both explained in Chapter 16. Both provide the same functionality, wrapped into the externally usable MontiCore tool.

The MontiCore parser generator (see Listing 6.1) is used to generate a parser for a given language. It can be used as a black box tool as described in Chapter 2. However, the generation can also be tailored to individual needs. The rest of this section addresses experienced developers interested in understanding or even adapting the MontiCore meta-meta-tool itself.

For generating a complete parser for a given grammar, that is already available as an internal AST, the class `ParserGenerator` is used (see Listing 6.1).

```
1 Repository: MontiCore/monticore github
2 Directory: monticore-generator/src/main/java/
3 File:      de.monticore.codegen.parser.ParserGenerator.java
```

Listing 6.1: Location of the MontiCore parser generator

```
1 public static void generateParser(
2     GlobalExtensionManagement glex,
3     ASTMCGrammar astGrammar,
4     IGrammar_WithConceptsGlobalScope symbolTable,
5     IterablePath handcodedPath,
6     IterablePath templatePath,
7     File targetDir)
```

Listing 6.2: Method signature used to generate a parser

If the parser generator is used within Java, the method `generateParser` of the class `ParserGenerator` is applicable. It generates a complete parser for the defined language. The method signature is depicted in Listing 6.2. The method accepts the following parameters:

1. the infrastructure for generating files (see Section 13),
2. the AST representation of the grammar that describes a modeling language whose models should be parsed (see Tip 6.7),
3. the symboltable (see Section 9),
4. a list of paths where handwritten files are located, e.g., handwritten AST and other classes (see Section 5.10) meant to be integrated into the generated code,
5. a list of paths for additional FreeMarker templates (see Section 12.1) to customize the generation process, and
6. the directory in which the generated parser will be created, which is freely selectable. For instance, it can be `"gen/"` or `"target/"`.

The parser generator is an important part of the MontiCore language workbench. Listing 6.3 demonstrates how the parser generator can be executed.

```

1                                     Java «hw» GenerateAutomataParser
2 // String args[0] contains the name of the input grammar
3 // String args[1] the path for the output directory
4 // Create the AST
5 String filename = args[0];
6 ASTMCGrammar ast = Grammar_WithConceptsMill.parser()
7     .parse(filename).get();
8
9 // Initialize symbol table
10 // (using imported grammars from the model path)
11 ModelPath modelPath = new ModelPath(Paths.get(
12     "target/monticore-grammar-grammars.jar"));
13 IGrammar_WithConceptsGlobalScope gs = Grammar_WithConceptsMill
14     .globalScope();
15 gs.setModelPath(modelPath);
16
17 Grammar_WithConceptsMill.scopesGenitorDelegator()
18     .createFromAST(ast);
19
20 // Hand coded path
21 IterablePath handcodedPath = IterablePath.empty();
22
23 // Template path
24 IterablePath templatePath = IterablePath.empty();
25
26 // Target directory
27 File outputDir = new File(args[1]);
28
29 // Generate the parser
30 GlobalExtensionManagement glex = new GlobalExtensionManagement();
31 ParserGenerator.generateParser(
32     glex, ast, gs, handcodedPath, templatePath, outputDir);

```

Listing 6.3: Java code creates a parser for automata (using its grammar)

The code block in line 5ff. loads the grammar that describes the language. When the AST is built, it represents only the currently processed grammar without grammars it extends. Information of these grammars is added in subsequent steps.

The statements in line 12 defines where additional grammars are located. In this example only the MontiCore jar is used and thus only the standard grammars provided by MontiCore are available. As a next step, the symbol table is built in line 17ff. While constructing the symbol table, all grammars are loaded and included that our grammar depends on by the global scope `gs`.

The statement in line 32 produces the classes for the desired parser in a subdirectory of the `outputDir` path. The execution of this block includes the generation of a grammar in ANTLR [Par13] format (i.e., a `.g4`-file) and triggering the parser generator ANTLR to create a parser for it. But only the created parse algorithm of ANTLR is used. The generated parser constructs an AST by instantiating AST classes generated by MontiCore.

This is achieved by injecting corresponding Java code into the generated parser. This Java code is complete; the created parser classes just need to be compiled. The `.g4-grammar` and a `.tokens-file`¹ are produced as input for ANTLR. They are just a byproduct of the generation process and only serve as potential documentation.

The Parser and Lexer Generation Process

Generating a parser internally consists of three consecutive steps producing the files listed in Listing 6.4. The output directory, `gen` in this example, is passed as a parameter while the package (in this example: `a.b.XY`) is derived from the grammars package with the grammars name appended.

```
1 Input parameters:
2   grammar file           -- e.g. "a/b/XY.mc4"
3   hardcoded path        -- list of directories
4   output directory      -- e.g. "gen"
5 Inputfile:
6   a/b/XY.mc4            -- a grammar for language XY
7 Output (excerpt):
8   gen/a/b/xy/_parser/
9     XYParser.java       -- generated parser
10    XYAntlr.g4          -- intermediate file for ANTLR 4
11    XYAntlrParser.java  -- internal parts of parser and lexer
12    XYAntlrLexer.java
```

Listing 6.4: List of files produced during the generation of a parser

The following three steps produce the parser:

Step one: In the first step an ANTLR file is created that is used in the following steps as an input for the ANTLR tool to create a parser and a lexer.

Step two: In the second step the ANTLR tool is executed, which produces a parser and a lexer consisting of two classes for parsing and lexical analysis. The parser uses the lexer to tokenize the input (cf. Chapter 4).

Step three: The third step is the generation of a class that provides parser methods for each nonterminal of the grammar. This class encapsulates the functionality of the parser generated by ANTLR and should be used for parsing and constructing the AST objects. It provides methods for the full language as well as for each sublanguage defined by a nonterminal (cf. Section 6.3).

6.2 Interface of the Generated Parser Classes

When the grammar has been processed (either directly calling the API, or the MontiCore CLI or the Gradle plugin, see 16), then a number of classes provide the interfaces described below.

¹This file is not detailed here, please refer to [Par13] for further information

First, there is the already known `XYMill` that allows to retrieve the parser using the `parse()` method. The generated parser class `XYParser` is instantiated as usual through the `XYMill` and contains three main methods (cf. lines 2ff in Listing 6.5). Those three methods are used for parsing a complete model. In addition three parsing methods are created for each nonterminal (cf. lines 7ff in Listing 6.5). The methods for nonterminals are recognizable by their suffix which corresponds to the nonterminal. The methods for the start nonterminal are equivalent to the ones for the complete model (e.g., called `Ax`).

```

1  // Parsers for the language:
2  Optional<ASTAx> parse(Reader reader);
3  Optional<ASTAx> parse(String filename);
4  Optional<ASTAx> parse_String(String text);
5
6  // and for each nonterminal NT furthermore:
7  Optional<ASTNT> parseNT(Reader reader);
8  Optional<ASTNT> parseNT(String filename);
9  Optional<ASTNT> parse_StringNT(String text);

```

Listing 6.5: Methods that can be used for parsing

The three types of parsing methods do the following:

1. Method `parse(Reader reader)` expects the input to be in form of a `Reader` (e.g., `StringReader`). The method processes the reader content.
2. Method `parse(String filename)` expects an existing file given as the parameter `filename` of type `String`. The file contains the model to be parsed.
3. Method `parse_String(String text)` parses the content of the given `String` directly interpreting the string as a model.

Each of the methods returns an `Optional` AST representation. In case of parsing errors, parsing either completely terminates or the method returns an empty `Optional` (see Section 15.3 for error management).

As described above, the mill of a language provides a dedicated static method called `parser()` to retrieve the parser for the language. Section 11.5 describes the Mill Pattern that was also already used in Chapter 5 for AST node instantiation. We highly recommend to use this method whenever a parser for a language is needed such that a mill reconfiguration in extended and composed languages can provide instances of subclasses of the parser without that the instantiating functionality (for the old, embedded language) notices this. This is highly relevant, when the languages are composed as explained in Chapter 7.

The parse methods are called as normal methods. Internally, a parser also uses the mill and builders to create the AST objects. Thus, when composing languages in a conservative way, i.e., extending the language such that all models of the old language are models of the new language, the old parser can still be used, to parse the old models, but internally the AST of the new, extended language is created. That means, in case the language `XY` is extended, handwritten code that uses the `XYParser` for parsing will still be functional for the old `XY` models. It thus allows us to process the old models with the old parser and delivers the new AST through the old `XYParser` methods.

6.3 Executing a Generated Parser

Executing a generated parser usually happens on the meta-level, i.e., within the tool that helps to create the product. The MontiCore parser generator is not needed for this purpose, because the MontiCore parser is used at the meta-meta-level. However, the MontiCore runtime environment (RTE, marked as «RTE») is needed at the meta-level. The MontiCore runtime environment is, therefore, packaged in the provided MontiCore jar as well.

Following the previous sections, we now use the generated parser for the language Automata language (cf. Chapter 4).

As a first example, the Automata parser is applied to an input file in line 5. The parser returns an `Optional` value holding the resulting AST if the model is parsed successfully, or otherwise an absent optional. In line 11, the parser is used to parse another automaton provided as a `StringReader`. Line 14 demonstrates how the parser can be used to parse the content of a `String`. Finally, line 17 demonstrates how to parse a model part, e.g., a `State`, only.

```
1 String filename = "example/PingPong.aut";
2 AutomataParser p = new AutomataParser();
3
4 // parse from a file
5 Optional<ASTAutomaton> at = p.parse(filename);
6
7 // parse from a Reader object
8 String aut = "automaton PingPong {"
9             + "state Ping;"
10            + "}";
11 at = p.parse(new StringReader(aut));
12
13 // another parse from a String
14 at = p.parse_String(aut);
15
16 // parse for a sublanguage, here: a State
17 Optional<ASTState> s = p.parse_StringState("state Ping;");
```

Listing 6.6: Various forms of parsing

The Automata parser reads a file (or string) and constructs the AST corresponding to the model of the Automata language. As described before, if the input model was not syntactically well-formed, the result is absent and at least one error message is issued through the standard error message channel. See Section 15.3 for a detailed description and an explanation on how to configure the error handler. In case of unexpected, internal errors, an exception is thrown in addition to a message to the error handler and immediate, erroneous exit of the tooling is triggered. Please note that the Automata parser neither checks context conditions nor resolves references to other models (see Chapter 9 and 10).



Tip 6.7: MontiCore Grammar Parsing

Caution: Here we are entering circular meta-meta levels.

The MontiCore tool parses grammars. All grammars belong to the Grammar language, which itself is defined as grammar. The MontiCore parser, therefore, is itself a parser generated by MontiCore, using the grammar defined in

```

1 Repository: MontiCore/monticore github
2 Directory: monticore-grammar/src/main/grammars/
3 File: -- grammar describing how MontiCore grammars look like
4     de.monticore.grammar.Grammar_WithConcepts.mc4
5 Directory: monticore-grammar/target/generated-sources
6           /monticore/sourcecode
7 File: -- MontiCore uses this Parser
8     de.monticore.grammar.grammar_withconcepts._parser.
9           Grammar_WithConceptsParser.java

```

Listing 6.8: Where to find the MontiCore grammar grammar

One meta-level down: If needed, a grammar can be parsed as shown in the listing below. We use the Automaton grammar as example (cf. Chapter 7).

```

1 String model = "Automaton.mc4";
2 Grammar_WithConceptsParser parser =
3     Grammar_WithConceptsMill.parser();
4 Optional<ASTMCGrammar> result = parser.parse(model);
5 ASTMCGrammar grammar = result.get();

```

The grammar parser reads the file (or String) and constructs the *grammar AST* that contains all essential information present in the source. If the grammar was not syntactically well-formed, the result is absent and at least one error message has been issued through the standard error message channel.

Chapter 7

Language Composition

Language composition is one of MontiCore’s key concepts. In this chapter we first discuss the motivation for language composition and give a high-level overview of how MontiCore achieves this. Then we discuss in detail how MontiCore composes grammars and which effects this has on the concrete and the abstract syntax that the grammars define.

Further techniques for composition can be found in the respective chapters, namely visitors (Chapter 8), symbol management infrastructure (Chapter 9), context conditions (Chapter 10), and a generator backend (Chapter 13).

7.1 Introduction to Language Composition

From best practices in Software Engineering, we know that the monolithic definition of large artifacts leads to many problems in maintaining, evolving, and reusing assets that have been developed. Programming became more productive when the languages started to support encapsulated implementations and to provide these to other developers through explicitly specified interfaces. *Modularity is a key technique for reuse.* In modern object-oriented programming, it is a key technique to encapsulate a piece of data structure together with the functions operating on it within classes. This considerably enhances black-box reuse as well as evolution of programs where changes can be better localized within a smaller part of the program.

It is generally predicted that a wider spread of software languages will occur for many different areas far beyond software development. Domain specific languages (DSLs), e.g., are used to model brains [PBI⁺16] as well as many other simulations of complex domains with a multitude of different aspects being described in different languages. DSLs are used to specify products, production workflows, scientific artifacts, economically usable data sets, and much more.

To be able to effectively engineer an appropriate software language, reuse on the language level is very important [CFJ⁺16, CBCR15]. Therefore, MontiCore offers an extensive set of mechanisms to define modular artifacts and reuse these either as black-box or in an enhanced and refined form within larger languages.

Language components are made for language composition. A language component is a reusable encapsulation of a, possibly incomplete, language [CBCR15]. A language component usually includes a grammar to define the language. The (normal) nonterminals

provided in this grammar and the additional infrastructure for symbol management and code generation act as "provided language interfaces", and the external nonterminals act as "required language interfaces". Interface nonterminals, interestingly, can be used as provided language interfaces, but also may be acting as extension points and therefore as required languages interfaces.

The MontiCore language workbench supports four language composition mechanisms – details follow after this overview in the rest of the chapter:

Language aggregation means that several artifacts of different languages are used together to describe aspects of the target domain. While the processed artifacts remain separated and can thus individually be edited, compiled etc., they describe a common target and thus need to be consistent and sometimes also need to be mapped to integrated simulation or code artifacts. This imposes restrictions on the artifacts, which can only be understood if the tooling allows an integrated understanding on the abstract syntax, context conditions, symbols, etc. (Example: class diagrams and Java).

Language embedding combines the languages into integrated model artifacts in the frontend, but is otherwise on the backend very similar to language aggregation. That means one single model, which is stored in one artifact, may consist of several sublanguages, which have been developed independently, but now define the overall model together (Example: Expressions in automata). Language embedding composes the languages even more tightly because it also composes the concrete syntax, which enforces a tighter composition of the tooling, including the editors.

Language inheritance is a technique to reuse a language while allowing to modify some language elements. The amount of modifications within the reused language affects reusability of the original models, etc. The new language is defined while reusing knowledge and implementation of the old language.

Language extension is a conservative form of language inheritance, which leads to a higher degree of black-box reuse. Basically, it abstains from dangerous forms of overriding of existing nonterminals. Therefore, it is discussed in the forthcoming chapters together with language inheritance (Example: Java code also compiles with new versions of the compilers.)

In a language aggregate and a language embedding, we also speak of *sublanguages* instead of only language components that are embedded or participate in an aggregation.

It is important to notice that the key focus is to reuse languages and the functionalities operating on these languages that have been *developed* and even *compiled independently*. The reused languages are now embedded as components or aggregated with a minimal set of external adaptations while no changes of the reused language components themselves are necessary.

The focus on reusability is not only on the generated parts, but also on handcoded extensions of the generated language infrastructure and in particular on algorithms coded against that language infrastructure. Algorithms transforming and extending the AST of



Tip 7.1: Wording for Language Composition

MontiCore's language composition techniques are strongly inspired by object orientation. However, due to the composition on two levels, namely grammars and nonterminals, the terms and wording used for extended and extending languages needs clarification. The four grammars A, B, C, D below with the nonterminals N, M, O, P serve as an example.

- A and C are *atomic grammars* resp. describe *basic languages*.
- B and D are *composed grammars* resp. describe *composed languages*.
- B is then also a *part language* of D
- A composed grammar, like D, is composed of one or multiple *subgrammars*.
- Composed languages have one or multiple *sublanguages*. A is also a sublanguage of D, because it is transitively included.
- A nonterminals defined in a sublanguage is *inherited* by the composed language.

```
1 grammar A { N = "n"; }
```

MCG A

```
1 grammar B extends A { M = "m"; }
```

MCG B

```
1 grammar C { O = "o"; }
```

MCG C

```
1 grammar D extends B, C { P = "p"; }
```

MCG D

To avoid confusion with object orientation the terms "super grammar" or "super language" are not used.

a language component can be fully reused on language aggregates, extensions, and embeddings because they still operate on the AST they know of, the builders are still operable, and so on.

MontiCore achieves a most crucial aspect in language composition: the actual language composition is deferred to a late binding point. This is very similar to object-oriented programming techniques, where the composition of classes is intellectually (semantically) understood at development time, but the compiler compiles independent artifacts and only the compiled result needs to be shipped. This enables (1) incremental compilation and thus more efficiency and (2) a "market" of black-box reusable language components, similar to the frameworks in today's programming languages.

Hence, each language component can be mapped to code and compiled independently of all other components. Neither a re-generation of the component is necessary, when the component is embedded, nor need the sources of the component to be shipped together with the language component implementation. Only the grammars need to be shipped

together with the class files of the generated implementation. This is a crucial prerequisite for truly modular language composition and for building libraries of languages or even families of language variants.



Tip 7.2: Composing Languages

We took much effort in the MontiCore language workbench to understand how to define languages from modular components:

- *Component grammars* can be defined explicitly.
- *Import* of grammars allows us to reuse languages as *components*.
- *Language inheritance* is achieved by import plus overriding of nonterminals.
- *Language extension* is a conservative form of language inheritance.
- *Language embedding* is a form of reuse of at least one language as a sublanguage of a new one. The models of the new language comprise sub-models of the embedded language.
- *Language aggregation* allows us to compose multiple languages into a new language, without embedding them into the same models.

These techniques for composition of languages in the large and a controlled modification of the reused languages are possible because MontiCore's grammar language provides *interface nonterminals*, *abstract nonterminals* and *external nonterminals*.

The composition of languages does not only affect *concrete syntax*, but also *abstract syntax*, *builders*, *visitors*, *context conditions*, and *symbol management infrastructure* can be composed.

Most important: The actual language composition is deferred to a late binding point. That means each language component can be generated and compiled independently. Neither is a re-generation necessary, when a language component is embedded, nor need the sources of a component to be shipped together with the language component implementation. This is a crucial achievement for language composition and for building libraries of languages.

The smart combination of these mechanisms allows us to address various forms of language composition and modification of existing components. Some of the composition forms are *conservative* (also called *safe*), while in general manipulations do allow to freely modify nonterminals deeply integrated in the language. This corresponds to the situation in object-oriented programming where inheritance provides some assistance for conservative modifications, but in general developers that modify inherited classes can do harmful things. On the other hand a controlled, methodically careful form of inheritance is a key power of modern object-oriented languages. In this spirit, the MontiCore language workbench offers powerful, to some extent conservative, but partially also dangerous techniques

transferring some of the burden to the developers.¹

The four techniques for language composition mentioned above of course affect the concrete syntax of the languages we define. However, language composition affects many aspects that we need to take into consideration:

- *concrete syntax*,
- *abstract syntax (AST)*,
- *AST creation (e.g. through builders and their mills)*,
- *navigation infrastructure (e.g. through visitors)*,
- *symbol management infrastructure*,
- *context conditions*,
- *handcoded extensions of all these generated parts*, and
- *analytical or generative backend*, implemented against all these generated parts.

This results into a two-dimensional list of issues to discuss, because many of the language aspects need to be discussed together with most of the composition techniques. We do this in the following chapters and sections, where each of these includes reuse of the generated parts as well as handcoded extensions:

	Basics	Inheritance	Embedding	Aggregation
Concrete Syntax	4	7.4	7.5	7.3-7.5
AST	5	7.4	7.5	7.3-7.5
Builder	5.9	7.6	7.6	7.6
Visitors	8	8.2.1	8.2.3	8.2
Symbol Management	9	9.10.1	9.10	9.10.2
Context Conditions	10	-	-	-
Backend (Generator)	(13)	-	-	-

More details can be found in the respective research results, such as [MSN17, Rot17, Wei12, Sch12, Völ11, Kra10, HMSNRW16, MSNRR16, HLMSN⁺15b, RRRW15, HMSNR15].

7.2 Language Composition at a Glance

While the concepts, techniques and methods to deal with language composition are spread over several chapters, we give an overview of the core mechanism in this section. Consequently, the following is a high-level overview:

Concrete syntax: In language aggregation, concrete syntax is not affected at all. For language embedding, e.g. Java expressions in automata, we define a new grammar that

¹How much power vs. restrictive guidance for a development tool is needed strongly depends on the skills of the educated developer and can only be understood when using such a tool. This is ongoing research.

simply imports all nonterminals from the embedded grammars. Then there are three possibilities: (1) Use of the nonterminals allows us to directly reuse the sublanguages that are available; (2) it is also possible to extend nonterminals of the original grammars; and (3) to override productions that have been defined for the original nonterminals.

Extension (2) allows language developers to add additional alternatives, for example new operators for expressions. Abstract nonterminals, interface nonterminals and external nonterminals have especially been designed in the MontiCore grammar infrastructure to facilitate these forms of extensions. Depending on the choice of the new language starting nonterminal, the original language may be extended (e.g. SQL statements in Java) or the original language is embedded (e.g. Java expressions in automata).

Overriding (3) allows to freely modify the original language. It is very powerful, but also dangerous and may reduce reusability of already existing software components operating on a language.

The composition of the concrete syntax through grammars is also used for the development of the new parser for the composed language. In an earlier version of MontiCore [Kra10], we even developed compositional parsers, but in practice it turned out that it is sufficient and efficient enough to generate a complete new parser. As a drawback, however, it is necessary to ship the grammar of the language component together with the language component implementation. The grammar is composed on the source level (actually their AST within MontiCore) and the parser is generated completely afresh from the composed grammar, but all other language component constituents remain untouched.

Parser: The parser itself is generated completely from scratch. That means there is no reuse of the parsers of the sublanguages. However, the parser facade of each sublanguage can still be used because a static delegator is silently redirecting to the parser of the composed language. This leads to a parsing into the composed AST, but since both are implemented using subtypes, all code written against the sublanguage parser is still usable.

Abstract syntax: From Chapter 5 we know that the definition of a grammar not only describes the concrete syntax, but is also a blueprint for the abstract syntax. To be able to use algorithms that have been defined on the abstract syntax classes, such as context conditions, symbol infrastructure, and any form of constructive or analytical handwritten code, it is important that the AST classes of the originally used grammars are directly reusable and not generated anew.

This leads directly to a new composed AST that integrates all AST classes from the original grammars. However, if the production for a nonterminal is modified, a new AST class is generated that inherits from the original. Both classes then have the same name, as they are derived from the same nonterminal, but reside in different packages.

Builder for the abstract syntax: The infrastructure to create new AST objects is adapted accordingly, such that the builder mill (see Section 5.9) is also composed. From a developers point of view, who only knows the new language there is one composed builder mill creating objects for all AST nodes. However, for a reused functionality it is still possible to rely on the old builder mill of the embedded language, which has now internally been modified in such a way that it produces AST objects of the extended language. This is done transparently, such that algorithms on the original embedded

language including functionality that creates new objects are completely reusable. For that purpose, we invented the static delegator pattern (see Section 11.1) and designed it in such a way that it can be adapted through subclassing.

Navigation infrastructure through visitors: Visitors are a core element to navigate through an AST once it has been created through the parser. Because the AST classes are completely reused and potentially only modified through subclassing, visitors on sublanguages may also be reused. Therefore, it is possible to reuse a visitor of a sublanguage out of the box as well as to modify the behavior of the visitor by building a handwritten subclass and overriding certain `visit` methods.

However, when composing several languages, technically such a visitor can only be applied to the elements of one sublanguage and runs into a type-induced matching problem, when a node from a foreign language component appears in the AST. For that purpose, we have created the `Traverser` (see Chapter 8), which is available for each language and allows to compose visitors that have been individually developed for sublanguages. The `Traverser` manages full traversal over the newly defined language nodes and delegates only to appropriate sub-visitors, which it is composed of.

Symbols and Scopes: Symbols and their visibility within artifacts, but especially between artifacts that import each other, are the core binding mechanism to integrate sets of artifacts into a consistent description. Therefore, it is inevitable to provide a compositional infrastructure for symbol management.

Language aggregation, therefore, needs efficient mechanisms to define externally visible symbols from one artifact and allow to use these symbols within another artifact. In language embedding, symbols defined in one part of the artifact should be used in another part of the artifact, even if defined in another sublanguage. So even within the same artifact, symbols cross the borders of languages. The symbol management infrastructure therefore provides a unified mechanism that allows to cross borders of languages as well as of artifacts.

As a speciality in a heterogeneous modeling world, it is necessary to understand how symbols are represented in different languages. This, for example, applies to the Unified Modeling Language (UML), where over 13 languages have been aggregated and are used to describe products together. For example, a method in a class can become a message in a Statechart, or a state in the Statechart may be represented as enumeration value (standard approach) or as subclass (state design pattern, [GHJV94]). Neither is the mapping always the same, as the mappings of states show, nor is the mapping always simple because the symbol may have restrictions or gets additional information along the mapping, e.g. Java methods need a certain signature to be usable as messages in Statecharts.

To manage this heterogeneous set of symbols, the symbol management infrastructure on the one side provides concepts for visibility, import, and export. On the other side, it also provides infrastructure for heterogeneous mappings between different kinds of symbols, which becomes relevant when a composition of the symbol management infrastructure together with their languages is necessary.

Context conditions are highly diverse. Therefore, a concrete context condition usually only applies for an individual language. However, a context condition depends only on a

small part of the language respectively certain forms of symbols and if that part of the language has not been modified during composition, the context condition conceptually still applies. From a technical point of view, context conditions are usually defined using visitors and as discussed above, visitors can directly be reused and composed in various forms. So context conditions naturally compose. If the language is extended in a conservative way, they can be reused easily.

It remains an open question, what happens with context conditions that are defined over a composed language. However, we think that many of these conditions can be reformulated in a decomposed form and then be implemented on the sublanguages. As a main technique for this decomposition, we have developed our symbol management infrastructure in such a way that it allows to map symbols defined in one sublanguage into the symbols of another sublanguage [MSN17, MSNRR16, HMSNR15, MSNRR15, Völ11], which naturally applies, when aggregating or embedding languages.

The backend: The backend of a tool consists of a generator, an interpreter, or analytical algorithms that retrieve interesting information of a larger and very detailed set of models or associated data. These techniques are usually highly specific to the domain and the intended use of the tools. We, therefore, do not believe that their composition is an easy task. At the moment we do not even really know how to compose generators that target the same platform. We have ideas, but this remains future research.

7.3 Grammar Constructs for Language Composition

Chapter 4 has introduced all grammar constructs that deal with monolithic definitions of a grammar in a single artifact. Chapter 5 furthermore discusses the derivation of the abstract syntax from a monolithic grammar. In this section, we discuss the following additional grammar constructs and mechanisms that allow language developers to build language components that import each other:

component is a keyword that allows to mark a grammar as incomplete, which means that no parser, but everything else, such as AST classes, visitors, or builder mills are created.

external is a keyword that, attached to a nonterminal, marks that nonterminal as not defined here, but as an extension point. This nonterminal needs to be bound, when the grammar is used. Thus, external nonterminals are only allowed in component grammars. An external nonterminal is therefore a mandatory extension point of a grammar.

import allows to refer to other grammars and especially component grammars that are imported and can be extended. Like in Java `import` may refer to all grammars of a package using the `*` extension.

interface nonterminals (and to some extent `abstract nonterminals`) enable to structure monolithic grammars, but can also be used to mark extension points of a grammar component. When importing a grammar component, additional alternatives

can be added to imported interfaces. Interfaces can have predefined bodies that are meant for AST-conservative extension (see Section 7.9.2).

overriding of nonterminals is a technique to redefine a nonterminal that can also be applied to imported nonterminals and thus modify imported languages. It is possible to override nonterminals of all forms, including tokens and fragment tokens. Furthermore, it is possible to override a nonterminal and keep the nonterminals body but add a new interface that is implemented.

extending nonterminals is technically an overriding of nonterminals, but when applied carefully and conservatively, the nonterminals are just extended. Depending on the form of overriding, the extension may affect only the concrete syntax or concrete and abstract syntax.

7.3.1 Component Grammar

To assist component-based reusability of grammars, we can define *grammar components* by using the keyword `component` as shown in Listing 7.3. A grammar component defines a sublanguage, i.e., a yet incomplete language that is meant to be extended to form a complete language. For a grammar component, MontiCore produces AST classes, node builder mills, context condition infrastructure, and visitors, but does not produce the parser for models of the defined language. Therefore, a grammar component is allowed to use interface nonterminals, abstract nonterminals as well as external nonterminals without any production body.

```

1 component grammar InvAutomata MCG InvAutomata
2           extends de.monticore.MCBasics {
3   external Invariant;
4
5   State = "state" Name
6           Invariant
7           ( "<<" ["initial"] ">>" | "<<" ["final"] ">>" ) * ";" ;
8 }
```

Listing 7.3: Example of a grammar component with an external nonterminal

A grammar component either is a collection of basic nonterminals that are meant for reuse, quite like a library, or it is an extensible – and therefore incomplete – language with explicitly marked extension points, namely `external` nonterminals, like a framework. The concept of grammar components reflects the concepts that object-oriented programming languages provide to extend classes.

7.3.2 External Nonterminals

An external nonterminal is introduced by the keyword `external`. It has a name but no production body that defines its structure (cf. Listing 7.3). An external nonterminal defines an extension point in the grammar, i.e., it can be used like all other nonterminals

7. Language Composition

in the body of a production, but its syntax is defined in another grammar. In fact, external nonterminals need to be bound when defining a complete grammar. In Listing 7.3, `Invariant` is an external nonterminal and is used in the body of production for the nonterminal `State`. It essentially only introduces the nonterminal `Invariant` and explicitly defines an extension point in the grammar. This extension point has to be filled to complete the language.

For the external nonterminal called `Invariant`, `MontiCore` creates an AST representation as an (empty) interface that needs to be implemented when the language is completed:

```
1 package invautomata._ast; Java «gen» ASTInvariantExt
2
3 public interface ASTInvariantExt
4             extends ASTNode, ASTInvAutomataNode {
5
6     // ... only clone, equals and scope signatures are given
7 }
```

Listing 7.4: External nonterminals are mapped to interfaces in the AST

Please note that for external nonterminals, we translate the nonterminal name into a class by also attaching an "Ext" suffix. Therefore, Listing 7.3 leads to the Java interface `ASTInvariantExt` shown in Listing 7.4.

External nonterminals can only be defined in component grammars, as explained in Section 7.3.1. They cannot be combined with `abstract` or `interface` keywords, and cannot have a right-hand side, but it is possible to bind them to any form of nonterminals in a composition. It is also possible to declare an external nonterminal as scope.

If an extension point shall be bound, language embedding is used.

```
1 grammar Automata2 extends InvAutomata { MCG Automata2
2
3     start Automaton;
4
5     // use this production as Invariant in Automata
6     Invariant = LogicExpr | ["-"] ;
7
8     interface LogicExpr;
9     Truth implements LogicExpr = tt:["true"] | ff:["false"] ;
10    And implements LogicExpr = LogicExpr "&&" LogicExpr ;
11    Not implements LogicExpr = "!" LogicExpr ;
12    Var implements LogicExpr = Name ;
13 }
```

Listing 7.5: Language embedding with binding the external nonterminal

In line 6, the grammar combines the newly defined nonterminal `LogicExpr` with the imported, but not yet bound nonterminal `Invariant`. The new grammar is complete

and a parser is generated. Furthermore, new AST classes are generated and `Invariant` leads to an implementation as a class in Listing 7.6.

```

1 package automata2._ast;
2
3 public class ASTInvariant extends ASTCNode
4     implements ASTInvariantExt, ASTAutomata2Node {
5
6     protected Optional<ASTLogicExpr>
7         logicExpr = Optional.empty();
8     protected boolean mINUS;
9 }

```

Java «gen» ASTInvariant

Listing 7.6: Implementation of the `Invariant` nonterminal

The nonterminal `Invariant` is now mapped to a normal AST class with the specialty that it also has to implement the interface `ASTInvariantExt` which allows all of its objects to be included in the AST from the original grammar and thus integrates ASTs on the object level.



Tip 7.7: External Nonterminal or Interface as Explicit Grammar Extension Hook Point

There are two main forms of hook points for explicit grammar extension:

- using an external nonterminal `NT1`, and
- defining a reusable interface nonterminal `NT2` in a commonly available, separate grammar `G2`.

When composing grammars an external nonterminal `NT1` needs to be explicitly filled as shown in line 6 in Listing 7.5. This approach leads to an extra AST class `EXTNT1` with additionally instantiated objects implementing the delegated pattern.

The second approach needs to define the generally available and commonly known extra grammar `G2` with a reusable nonterminal `NT2`, which is usually an interface nonterminal. All potential extensions can then be implemented in new additional grammars and upon composing these grammars, the interface automatically gets its alternatives composed. The second approach simplifies composition, but leads to a tighter coupling because grammar `G2` is commonly shared.

7.3.3 Importing and Extending Grammars

A modular definition of grammars is based on references between grammars. This can be achieved either by explicit import statements and then using the unqualified grammar name or by a fully qualified grammar name in the `extends` clause. This was, e.g., used for basic grammar components that `MontiCore` provides, such as `de.monticore.MCBasics`.

When using the MontiCore command line interface, the source path of grammars are specified using the the "-mp" options.



Technical Info 7.8: How Grammars are Imported, Extended and Composed

When a grammar shall be extended, the compiler searches the grammar path (specified with the "-mp" options). The provided path list needs to contain all imported (extended) grammars. No other files of the imported grammars need to exist at the time MontiCore is executed.

The result is that all newly defined nonterminals are mapped to code accordingly and a full parser is generated.

However, the imported grammars are not(!) mapped to code because MontiCore assumes that a generator management system, such as Gradle or make will organize redundancy free, incremental generation more efficiently.

As a consequence, the order of processing grammars is not relevant, but each grammar needs to be processed by MontiCore individually.

7.4 Language Inheritance

Language inheritance is a rather powerful feature of MontiCore grammars. Language inheritance allows to extend already existing languages by new nonterminals as well as to redefine existing nonterminals. This way it allows a modular definition of languages by reusing language components. To extend a grammar, a new grammar is created that uses the keyword `extends` after the name of the grammar followed by the name of the original grammar (cf. Listing 7.9). In this example, the language `HierarchicalAutomata` extends the language `Automata1`. This way, the new grammar `HierarchicalAutomata` inherits all productions defined in the original grammar `Automata1`. All nonterminals of the original grammar can be used the same way as nonterminals defined directly in the new grammar. Furthermore, inherited nonterminals can be redefined.

```

1 grammar HierarchicalAutomata extends Automata1 {
2
3     // keep the old start
4     start Automaton;
5
6     // redefine a nonterminal
7     @Override
8     State = "state" Name
9         ( "<<" ["initial"] ">>" | "<<" ["final"] ">>" ) *
10        ( ";" | "{" (State | Transition)* "}" );
11 }

```

Listing 7.9: Language inheritance: One grammar extending another and redefining an inherited nonterminal

By default, the new parser uses the first nonterminal that is defined in the grammar as start and assumes that this is the starting point for describing the overall language. If we want to preserve the original language, but modify it in some of its concepts, we have to explicitly define the start by using the `start` statement as shown above.

7.4.1 Redefining / Overriding Productions of Grammars

A production for nonterminal `NT` is redefined by (a) either defining a new production for the same nonterminal `NT`, thus *overriding* the nonterminal (cf. Listing 7.9) or (b) by *extending* the nonterminal `NT` in a production for a new nonterminal `NT2`. In the first case the new production for `NT` in the new grammar overrides and thus shadows the original production for `NT`. The original is completely replaced. This is the case for the nonterminal `State` as the grammar `Automata1` already defined the nonterminal `State`, but the grammar `HierarchicalAutomata` has a production redefining this nonterminal.

For the abstract syntax, there will be a new `ASTState` class that realizes the new production body. To be a useful replacement, the new class is a subclass of the old one. Both classes have the same name, but are located in different packages (see Listing 7.10).

```

1                                     Java «gen» ASTState
2 package hierarchicalautomata._ast;
3
4 public class ASTState extends automata1._ast.ASTState
5         implements ASTHierarchicalAutomataNode {
6
7     protected String name;
8     protected java.util.List<ASTState> states;
9     protected java.util.List<ASTTransition> transitions;
10    protected boolean initial;
11    protected boolean r_final;
12 }

```

Listing 7.10: The new `ASTState` class extends the old `ASTState` class and serves as a substitute

Through the extension mechanism in the AST classes, it is ensured that the new `ASTState` nodes can be used in all places, where the old ones are expected. This, however, also imposes that the functionality of the new class subsumes the functionality of the old one. This in particular means that the body of the production may be extended, but not completely changed. We may introduce new language entities, but are not allowed to remove nonterminals on the right-hand side nor change their cardinality. As a remark: it is possible, but not recommended, to omit nonterminals or adapt their cardinalities. See Section 7.9 for the discussion about conservative extension.

However, the abstract syntax is not affected by introducing additional terminals, rearranging the order in the production and similar modifications. This of course affects the concrete syntax and that leads to the situation that models of the original language are not models of the adapted language.

We therefore distinguish these forms of redefinition of a nonterminal:

Free modification of the production with the risk that some of the functionality of the original language does not work anymore.

Conservative extension of the AST preserves all nonterminals in their cardinalities as well as semantically relevant terminals. It, however, is allowed to extend the AST by additional semantically relevant entities.

The goal that is achieved by *AST-conservation* is that all functionalities for the old AST still can be used on the new AST.

Conservative extension of the concrete syntax preserves and only extends the concrete syntax. This means that basically the production needs to be preserved as is and can only be extended by optional entities (A?) or lists (A*).

Goal of *CS-conservation* is that the old models are also models of the new language and all models can be reused.

Concrete and abstract syntax compliance. It may be that both, concrete and abstract syntax are preserved, but the AST representation of the same model differs in the original and the extended languages. One minimal example would be a production $A = n:\text{Name } t:\text{Name}$ that is overridden by $A = t:\text{Name } n:\text{Name}$. *CS-AST-compliance* enforces that the same model results in the same tree structure.

For the process of overriding productions a couple of context conditions apply (cf. Section 4.4). For example, a nonterminal can only be overridden by a production of the same kind (except external nonterminals). Thus, productions of abstract nonterminals can only be overridden by productions of abstract nonterminals.

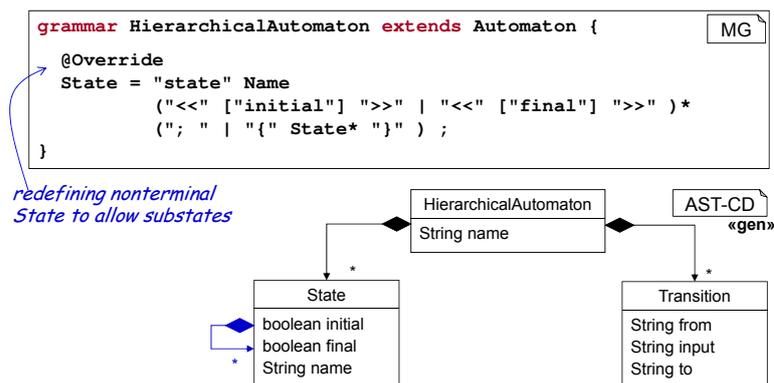


Figure 7.11: Language inheritance

As a final remark, it should be noted that it is formally impossible to completely remove a nonterminal, but overriding the nonterminal by a secret production has the same effect because when the user does not know this secret production, no instance of the respective nonterminal is parsed anymore. For example:

```
1 Transition = "THIS-IS-A-SECRET-77616E636B";
```

MCG fragment

would forbid transitions. However, this can only be applied to a nonterminal that occurs only in lists, alternatives, or optional clauses because if it would be mandatory, no valid model would exist anymore. For example, certain nonterminals implementing an interface such as `Expression` can be ruled out this way. While in principle this effect could also be achieved by using the formally cleaner form of context conditions, the latter does not work, when at the same time a new and similar alternative should be added because that could lead to parsing problems.

7.4.2 Extending the Implementation Structure of a Nonterminal

If the body of an imported nonterminal `NT1` is already defined in perfect shape, but the nonterminal should implement a given interface `NT2` then it is possible to simply declare this additional interface implementation in form of a production without a right hand side, i.e., `NT1 implements NT2` does not change the body of `NT1`, which leaves the concrete syntax unaffected, but adds the desired `implements` relation on the AST.

In the `Automata4` grammar shown in Listing 7.12 the production for the nonterminal `State` that is originally defined in `Automata1` is overridden by the production in `Automata4`. The new production now implements the interface `nonterminal AutElement`, which expresses that the `State` nonterminal should additionally implement the specified interface nonterminal. As shown there is no right hand side of the production, thus no body is defined for the state production. The omitted body indicates that the original body should continue to be used. As a result, the concrete syntax of the state production is not changed. For the abstract syntax a subclass of the original `ASTState` is generated which is also called `ASTState` but located in the package `automata4._ast` and additionally implements the interface generated for `AutElement`. Thus, this extension is conservative in both aspects concrete and abstract syntax as models of `Automata1` are valid models of `Automata4` and also the infrastructure for `Automata1` can still be used for models of `Automata4`.

```
1 grammar Automata4 extends Automata1 {
2   interface AutElement;
3
4   //Override and add interface but keep original body
5   @Override
6   State implements AutElement;
7 }
```

MCG Automata4

Listing 7.12: Language inheritance: One grammar extending another and redefining an inherited nonterminals inheritance structure without modifying the body

In case the body of the nonterminal should be altered, this is possible as well. To eliminate the body an empty body can be defined as shown in Listing 7.13. In contrast to the grammar `Automata4` shown in Listing 7.12 the grammar `Automata5` in Listing 7.13

overrides a nonterminal and does eliminate its body. As shown the two cases differ such that Automata5 explicitly defines an empty body `=;` for the overridden Automaton nonterminal while Automata4 only redefined the State productions head by implementing the interface AutElement and no body is defined, i.e. no equals sign is used. Overriding the nonterminal State with an empty body however would lead to an error as the Automaton production uses the state nonterminal with a *-cardinality. In this case a parser generation is not possible. Eliminating the body is in general a non-conservative extension except for the case the original production defined a body without concrete or abstract syntax as well.

```
1 grammar Automata5 extends Automata1 { MCG Automata5  
2   //Override and eliminate the body  
3   @Override  
4   Automaton = ;  
5 }
```

Listing 7.13: Language inheritance: One grammar extending another and redefining an inherited nonterminals by eliminating the body

Of course both options, i.e. overwriting a nonterminal to change the body and adding an interface can also be used in combination. Listing 7.14 demonstrates this case. Here the State production is supplemented by an interface and a new body is defined. The adaptation shown here is not conservative with respect to the concrete syntax, because the markers for final and initial states were moved to the front, while the abstract syntax is conservatively extended.

```
1 grammar Automata6 extends Automata1 { MCG Automata6  
2 interface AutElement;  
3  
4   //Override and define a new body  
5   @Override  
6   State implements AutElement =  
7     ( ["initial" | "final" ])* "state" Name;  
8 }
```

Listing 7.14: Language inheritance: One grammar extending another and redefining an inherited nonterminals inheritance structure as well as the body

7.4.3 Extending Multiple Inherited Grammars

It is possible to extend multiple grammars, which strongly corresponds to multiple inheritance in OOP. This is done by defining a comma separated list of grammars after the extends keyword (cf. Listing 7.15). Here, the grammar Automata3 extends the grammars InvAutomata and Expression. Unlike the example in Listing 7.5, this time we reuse an existing language for invariants.

```

1 grammar Automata3 extends InvAutomata, Expression {
2
3   // LogicExpr is defined in grammar Expression and now
4   // bound to the external NT
5   @Override
6   Invariant = LogicExpr;
7
8 }

```

Listing 7.15: Language embedding: Filling extension points

As before, all productions of the original grammars are inherited. If two nonterminals have the same name in the original grammars the order in which the original grammars are listed after the `extends` keyword is relevant. The definition of the first (leftmost) grammar defining the nonterminal is taken as the valid definition, while all following ones are ignored. In our example, if both grammars `InvAutomata` and `Expression` define the same nonterminal, the definition in grammar `InvAutomata` would be taken.

If a grammar extends several other grammars, those grammars may share common sub-grammars. Thus, diamond extension works. A nonterminal imported through two different extension paths is imported only once. If a nonterminal is imported through different paths, having different definitions, then the first imported definition takes precedence. This even plays a role, when the body of the defining production is the same in both imported grammars because the objects that are instantiated belong to the package of the first grammar.

7.5 Language Embedding

With the presented mechanisms of language extension and overriding of nonterminals, we can achieve specific effects, such as embedding one language into another. This is especially interesting when both languages have been independently developed and there was no joint definition of the abstract syntax, the context conditions, or any other elements of these languages before.

We can assume that this is the case for the languages `Expression` and `InvAutomata` already used and composed in Section 7.4. The nonterminals of both languages now co-exist. Through the selection of the new starting nonterminal (using the `start` statement) the master language can be defined. For a combination of the languages it would then be necessary to either fill external nonterminals or override already implemented nonterminals in such a way that nonterminals of both languages refer to each other.

Because language `InvAutomata` (Listing 7.3) has an extension point `Invariant`, we can embed the expression language for those invariants by mapping the extension point to the existing nonterminal. In line 6 of Listing 7.15, the extension point `Invariant` of the imported grammar `InvAutomata` is bound by the nonterminal `LogicExpr` of the imported grammar `Expression`.

The above example has shown that it is possible to embed one language into another. It is of course also possible to define a completely new start and reuse nonterminals from both

sublanguages in its production body. Or we can also embed given languages in our newly defined ones, what we regularly do, when importing `MCBasics`, etc.

There are, however, limitations that a developer should consider:

1. A keyword defined as token in one language remains a keyword in all parts of the composed language, even if the other languages regarded that as a normal name.
2. Whitespaces and forms of comments must be identical in all sublanguages, because they are processed by the composed lexer in a uniform way.
3. Token definitions should also be shared, for instance, if two languages define the same integers using different token nonterminal names then the parser generator has to announce an ambiguity because tokens are parsed free of the context of their use.
4. Grammars yield a flat namespace for nonterminals, which means that all nonterminals of an imported grammar are merged into the new version. As already mentioned, when a nonterminal is defined in two grammars, then the first one takes precedence. This may lead to unintended changes of the languages because this unintended form of overriding could affect the second language that normally also uses that nonterminal.

7.6 Composing the Builder Infrastructure

A builder for any AST object belonging to a grammar `G1` is created using the *language mill*, called `G1Mill`, as discussed in Section 5.9. The language mill uses the static delegator pattern (see Section 11.1) to map the static call to retrieve a builder to an internal mill object. This enables MontiCore to also compose the language mills in all forms of language composition, aggregation, and inheritance.

Consequently, for a developer knowing the composed grammar `G2`, for which we assume it extends `G1`, builders for all nonterminals can be retrieved from `G2Mill`. However, for old functionality, which was developed only for grammar `G1`, all `G1Mill` builders still work and thus functionality can be reused without changes.



Tip 7.16: Builder in Composed Languages provide Excellent Reusability

One of the big advantages of the MontiCore language composition technique is the possibility to reuse functionality that has been developed on sublanguages.

This is assisted by the code generator using various techniques, including reusability of the builders. I.e. functionality that creates new objects through the provided builder mills can normally be reused in black-box form, without any need of source code adaptation, even though it will then operate on the extended and composed languages and even create AST nodes of the composed language.

Furthermore, this functionality can be extended when relying on the visitor composition techniques as described in Chapter 8.

In case a nonterminal S of $G1$ has been redefined or extended, then $G1Mill$ now produces new $g2_ast.ASTS$ objects instead of old $g1_ast.ASTS$ objects. As $g2_ast.ASTS$ is a subclass of $g1_ast.ASTS$, the old functionality does not recognize anything.

To ensure this, the language mill $G1Mill$ has to be initialized accordingly. For this purpose, each language mill is equipped with a static initialization method `init` that initializes the mill (here $G2$) and all mills of the languages it depends on (here: $G1$) to deliver objects of that language. So an `G2Mill.init()` statement ensures that all calls of the form `G1Builder.sBuilder()` deliver `g2_ast.ASTSBUILDER` objects. The mill pattern is further explained in Section 11.5.

Please note that a mill initialization can be overridden by initializing another mill that depends on the mill. So only one language mill should be initialized at the program start.

If the extension of nonterminal S is AST-conservative (see Section 7.9), then the new builder completes the build process with exactly the same attributes being set as by the old builder. This works because in an AST-conservative extension all new attributes are optional or lists.

If the extension is not AST-conservative, then either (1) the $G1Mill$ methods may not be used anymore or (2) the builder needs to predefine values for the new, hidden attributes. In the latter case, handwritten extensions of the $G1Mill$ class of the builders, e.g., using the TOP mechanism are recommended.

To complete the picture, it is worth mentioning that overriding of nonterminals, e.g., S , not only leads to a subclassing relationship between $g2_ast.ASTS$ and $g1_ast.ASTS$, but also the builder `g2_ast.ASTSBUILDER` becomes a subclass of `g1_ast.ASTSBUILDER`. This is helpful and necessary, to inject the subclass builders into the functionality knowing the superclass only.

7.7 Composing Parsers

The language mill offers a method called `parser` to get the parser of a language. Each sublanguage $G1$ has its own parser in class $G1Parser$. If $G2$ extends $G1$, then a class $G2Parser$ exists that provides the parsing methods for the nonterminals of $G2$.

When languages are composed or extended, MontiCore actually does not compose the parsers, but creates a new complete parser for each composition of languages. $G2Parser$ is independent of $G1Parser$'s functionality. This is why language composition needs the subgrammars as sources (i.e., the `.mc4`-files). However, this is hidden for developers.

For language composition, in addition to the complete $G2Parser$ a subclass of the $G1Parser$ called $G2ForG1Parser$ is generated. This subclass overrides the parsing methods of $G1Parser$ that are impacted by $G2$. The mill pattern ensures that when initializing the mill of $G2$ a subclass $G2ForG1Mill$ of the $G1Mill$ is injected in the $G1Mill$ that overrides the internal `_parser()` method of $G1Mill$. The new implementation of `_parser()` returns an instance of the $G2ForG1Parser$. So if the $G1Mill$ is used to get a parser, the instance of a subclass of the parser is provided, which delegates to the parser

of the composed language for nonterminals which have been changed in the composed language. Thus the functionality for G1 can be reused and parses nonterminals changed by the composition with the parser of the composed language.

Consequently, for a nonterminal NT from a sublanguage G1 of a composed language G2, the method `G1Parser.parseNT` has the same effect as using `G2Parser.parseNT`. Furthermore, when restricting to standard use of visitors etc., functions developed against the sublanguage G1 will never experience differences in the AST even though the resulting AST may contain additional AST objects belonging to G2 only. This also holds if NT itself is conservatively redefined. Hence, composition remains fully transparent and parsers, builders, and the later discussed visitors of a sublanguage need not to be aware of their embeddings in a composition. They can be fully reused and do not even need to be recompiled.

There is, however, some caution necessary: The parsing only works well, if the concrete syntax is conservatively extended (see Section 7.4.1). Section 7.4.1 also discusses some precaution necessary to ensure that type incompatibilities do not prevent compilation of the composed AST classes.

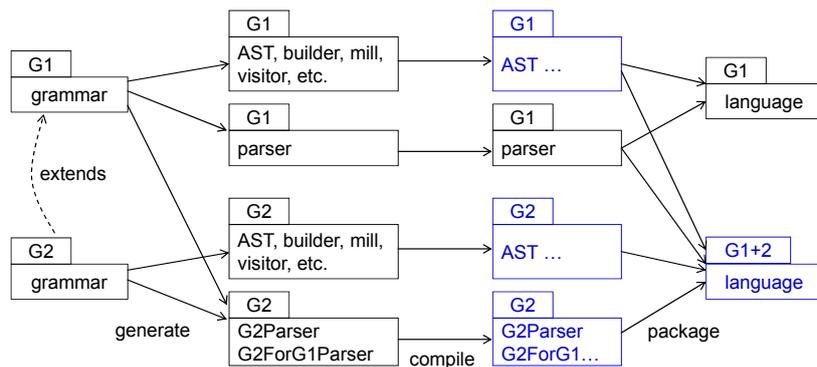


Figure 7.17: Composition of a language is executed as late as possible: late binding

Figure 7.17 shows that both, the generation and the compile processes of sublanguages are decoupled and, thus, it is possible to ship language components as pre-compiled libraries. In MontiCore 3 even the parsers were compositional and decoupled. However that did not work too well because the scanners had limited capabilities for composition. We therefore decided to early compose grammars and create monolithic parsers. This is acceptable for several reasons (1) MontiCore leaves parser frontends intact, while allowing to use the composed parser, (2) the parser does have a very limited signature (namely the parsing methods) and a well encapsulated functionality, and (3) the parser is not supposed to be manually adapted e.g. through subclassing.

7.8 Composition of Visitors and Context Conditions

When composing a language, it is relevant that every element of the language can be composed. This does not only include concrete and abstract subjects, but especially also

technical infrastructure that allows to define functionality on the language models.

A good composition means that the original functionality can be reused without having to touch the source code, even though it may be useful and necessary to extend the functionality.

This is why we have realized our visitor infrastructure in a compositional form. It is described in the following Chapter 8. That means the visitor even for a specific sublanguage can easily be extended with additional functionality on the new nonterminals, without adapting the source code or even recompiling the sources.

Many context conditions built on visitors and are therefore easily reusable and composed languages as well.

7.9 Conservative Extension

It is worth examining the conservative extension properties defined in Section 7.4.1 for ASTs as well as concrete syntax in greater detail. Conservative extension is generally interesting when a nonterminal already has a definition that shall be extended, but its properties shall be conserved. This, therefore, does not apply to an external nonterminal, which can be implemented freely, but to normal and abstract nonterminals.

Let us for the following discussion assume that we have a number of languages LG^* extending language $LG1$ (where "*" stands for any number). For a better understanding, we attach a suffix to each nonterminal, to describe where it comes from. Please be aware that this suffix is not present in the grammar themselves.

```

1  grammar LG1 {
2      MLG1 = Decimal;
3      NLG1 = "one" MLG1;
4      PLG1 = "some" MLG1*;
5      QLG1 = "optional" MLG1?;
6  }
7  grammar LG* extends LG1 {
8      MLG* = ...
9  }

```

MCG fragments

7.9.1 Conservative Extension of the Concrete Syntax

Conservative extension of the CS is a property of the set of models parsed by a grammar. It means²: $Sem(LG1) \subseteq Sem(LG2)$. Such a property on grammars is generally undecidable [HMU06]. However, a set of sufficient criteria can be defined that ensures this property. To assist the developer, we thus describe these criteria in the following.

²Let us denote the language, i.e. the set of words, of a grammar L by $Sem(L)$

7. Language Composition

```
1 grammar LG* extends LG1 { MCG fragments
2   // we describe a bunch of grammars LG* here,
3   // each grammar has one nonterminal M
4   MLG02 = Decimal;           // CS-conservative
5   MLG03 = Decimal P?;       // CS-conservative
6   MLG04 = "-"? Decimal;     // CS-conservative
7   MLG05 = P* Decimal P?;    // CS-conservative
8   MLG06 = Name;           // not cons.
9   MLG07 = Decimal*;        // CS-conservative
10  MLG08 = Decimal?;        // CS-conservative
11  MLG09 = Decimal | Name;   // CS-conservative
12  MLG10 = Decimal d:Decimal?; // CS-conservative
13  MLG11 = d:Decimal;       // CS-conservative
14 }
```

The above examples of redefining nonterminal M demonstrate what is allowed. The Decimal nonterminal needs to be retained, although it might be given another name (LG11), which affects only the AST.

If Decimal changes its cardinality, the cardinality may only be widened. That means from mandatory (N) to optional (N?) or nonempty list (N+), and from all three to list (N*). Separators may be added, like in (N || ",")*. Alternatives like in LG09 count as a switch to optional.

Before, between, and after the existing terminals and nonterminals it is allowed to add optional and list nonterminals because their omission is generally allowed (LG03-LG05).

It is, however, not allowed to omit nonterminals (LG06) or rearrange their order (LG21). But, it is allowed to add more optional variants of an already existing nonterminal (LG22, LG23).

```
1 grammar LG* extends LG1 { MCG fragments
2   NLG20 = "one" M?;         // CS-conservative
3   NLG21 = M "one";         // not cons.
4   NLG22 = M? "one" M;     // CS-conservative
5   NLG23 = x:M? "one" M;   // CS-conservative
6 }
```

Many of those conservative extensions on the CS are, however, no conservative extensions on the AST.

Please note that obviously a conservative extension also needs to keep the original starting nonterminal. By default, MontyCore takes the first explicitly defined nonterminal as start. Thus, a `start` statement is usually needed to set the starting nonterminal correctly and retain conservative extensions of the CS.

7.9.2 Access-Conservative Extension of the Abstract Syntax

Conservative extension of the AST is a property of the data structures and it comes in two important variants:

- The AST data structures that a programmer expects under LG1 are still valid under LG2. That means when navigating an AST, e.g. with a visitor and accessing children, no surprises occur. We call that *AST-access-conservation*.
- All operations on an AST that a programmer might use to manipulate an LG1 AST are having the same effect under LG2. We call that *AST-modification-conservation*.

Fortunately, some conservation properties are already ensured by the extensible OO type system. Unfortunately, the Java type system is not powerful enough to fully support all potentially interesting forms of language extension, such that the AST is conserved. We discuss the problems and some workarounds with a few examples in Section 7.9.4.

AST-access-conservation basically means that given an AST object with a certain type information, the object may be from a subtype, but behaves like the known type. That means all getters and value retrievers work as from then known type, navigation to children works as normal and applying a visitor works. This is generally the case, when the cardinality of a nonterminal stays untouched. For the examples from above, this is as follows:

```

1 grammar LG* extends LG1 { MCG fragments
2   MLG02 = Decimal;           // AST-conservative
3   MLG03 = Decimal P?;       // AST-conservative
4   MLG04 = "-"? Decimal;     // AST-conservative
5   MLG05 = P* Decimal P?;   // AST-conservative
6   MLG06 = Name;           // not cons.
7   MLG07 = Decimal*;        // not cons.
8   MLG08 = Decimal?;       // not cons.
9   MLG09 = Decimal | Name;  // not cons.
10  MLG10 = Decimal d:Decimal?; // AST-conservative
11  MLG11 = d:Decimal;      // not cons.
12
13  NLG20 = "one" M?;        // not cons.
14  NLG21 = M "one";        // AST-conservative
15  NLG22 = M? "one" M;    // not cons.
16  NLG23 = x:M? "one" M;  // AST-conservative
17 }
```

The modifications of the languages LG02 to LG05 are AST-access-conservative because they basically leave the inherited nonterminal(s) unchanged. Any changes of the cardinality of the nonterminal, such as in LG07 to LG09, LG20, LG22, omitting the nonterminal (LG08), or renaming the nonterminal (LG11) is not AST-access-conservative.

LG22 ist not access-conservative because it extends the cardinality of M to M* (even though it is restricted to 1-2). In contrast, LG23 is access-conservative, because the new instance of M has a different name x.

Relaxing the cardinality is generally not access-conservative because the accessor could rely on an assumption that there is exactly one, at most one, or potentially a nonempty list. This also holds, but will in practice not necessarily be a problem, if the cardinality is only relaxed through a different `min` and `max` definition (see Section 4.2.9).

On the other hand, strengthening the cardinality would in principle be access-conservative. It works, when using the `min`/`max` definitions or restricting from M^* to M^+ , but does not generally work, when adapting the cardinality from M^* to $M^?$ or M and also not when restricting $M^?$ to M because the access signature changes. See Section 7.9.4, how that can be handled.

Changing the order of the nonterminals, like in `LG21`, also is generally access-conservative because the AST does not remember the order of the nonterminals.



Tip 7.18: Common Access Functions, but Different Concrete Syntax

It may happen that two variants A and B of a language construct exist that share the same abstract syntax, but differ in the concrete syntax. E.g. an association has a left and a right end with identical content. To allow common access (and modification) functions, but ensure AST-conservative realizations, it is possible to use a common interface C, e.g. as in:

```

1
2  interface C      = x:Number y:String ;      // order irrelevant
3  A implements C = x:Number ", " y:String ;
4  B implements C = y:String ":" x:Number ;

```

7.9.3 Modification-Conservative Extension of the Abstract Syntax

Preserving all abilities of modification for an AST includes that it is allowed to freely manipulate the AST nodes. This includes setting new children for single, mandatory nonterminals (N) as well as full list and optional manipulation (for N^* and $N^?$). It, however, also includes to create new AST nodes through the `LG1` mill and builder interfaces.

Then functionality implemented against a language component `LG1` can be reused as a precompiled, black-box functionality for any language `LG*` that is based on `LG1`.

AST-modification-conservation is almost equivalent to AST-access-conservation, because the appropriate sets of methods are generated and thus work closely together. However, there are two important differences: Builders and cardinalities of nonterminals.

Enums cannot be overridden or extended. Thus, non-conservative extension of enums is also not possible.

The builder for an overwritten nonterminal M_{LG02} is a subclass of the builder for the original M_{LG01} and thus inherits all methods. Because optional nonterminals and list nonterminals do have the defaults *absent* and *empty list*, if only nonterminals with these cardinalities are

added in a redefined production, then the builder of M_{LG02} behaves like the one of M_{LG01} and delivers the appropriate AST object. It delivers always an instance of class M_{LG02} , but through the mill of $LG1$ it looks like a superclass object of type M_{LG01} .

As a consequence: it is inevitable that all functionality uses builders for the AST instead of the direct constructor.

If the language extension is not conservative, the builder needs to be adapted accordingly by a handcoded version (e.g. using the TOP mechanism) to be reusable within functionality of a sublanguage.

The second difference regards cardinalities: While access allows strengthening, manipulation generally allows relaxation of cardinalities in subclasses to remain substitutability. Together that means that a fully AST-conservative language extension may neither strengthen nor relax cardinalities and thus not adapt the cardinalities at all. Otherwise, some functions can not be used anymore, or special measurements need to be taken to handle that.

7.9.4 AST Signatures Causing Java Type Errors

The considerations in Sections 7.9.1 and 7.9.2 do hold irrespectively, whether the nonterminal under consideration (e.g. M and N) itself is adapted. However, in some cases even though the modification would be AST-access-conservative, the Java type system produces a compilation error. These are in particular the following cases:

```

1  grammar LG2 extends LG1 {
2      MLG2 = Decimal;
3      NLG2 = "one" MLG2;           // ok
4      PLG2 = "some" MLG2*;       // does not compile
5      QLG2 = "optional" MLG2?;   // does not compile
6  }
```

MCG fragments

The problem is that when nonterminal M is modified, we actually get a second AST class called $lg2_ast.ASTM$. This is a subclass of $lg1_ast.ASTM$ that is fully behavioral conform. The redefinition of N is fine because it uses only a single, mandatory M .

But the adaptation of P does not compile because some of the generated methods in $lg2_ast.ASTM$ return $List<lg2_ast.ASTM>$, but the inherited signature forces them to return $List<lg1_ast.ASTM>$ objects. Java does not accept these types as compatible. While Java is correct in general, we have ensured through a consequent use of builders and the configurable static delegator pattern that actually no object of class $lg1_ast.ASTM$ is instantiated in the tool. All objects are of subclass $lg2_ast.ASTM$ and thus all list manipulations would be safe, even though the type system does not accept that.

There are two circumventions to the problem: (1) If it is possible prevent redefinitions of both nonterminals, M and N , in parallel, if M occurs in N 's body with a cardinality other than one. This can, e.g., be achieved by adapting the base language $LG1$:

7. Language Composition

```
1  grammar LG1 { MCG fragments
2    M = Decimal;
3    N = "one" M;
4    P = "some" MStar;
5    MStar = M*;
6    Q = "optional" MOpt;
7    MOpt = M?;
8  }
```

As an alternative (2), which is especially suited, if the base language cannot be adapted, we might give the nonterminal M a different name:

```
1  grammar LG2 extends LG1 { MCG fragments
2    M = Decimal;
3    P = "some" m2:M*;
4    Q = "optional" m2:M?;
5  }
```

Then everything compiles and is CS-conservative, but not AST-conservative because the methods to handle attribute *m* inherited from P_{LG1} (and also Q_{LG1}) are still there, but do not provide a useful implementation because the attribute *m* is not set anymore when parsing. A handcoded adaptation might just delegate the methods for *m* to the methods for attribute *m2* (while adapting the types using casts).

Using handcoded adapters to integrate the ASTs generated from a base language LG1 and an extended language LG2 also can be applied if the inheritance for some reason shall not be AST-conservative. This includes, e.g., omitting a nonterminal in LG2, which leaves the inherited functions useless. One might still allow those in the AST, therefore blending the AST of LG2 with LG1, which might be a helpful intermediate structure when transforming from LG2 to LG1. There is much potential in this blending of AST structures, when transforming between languages.

Blending also can be applied when the cardinality of a nonterminal has changed. Due to the almost disjointness of the method sets generated for N , $N?$ and N^* these methods can live together in an AST class. For access functions, narrowing of the cardinality is feasible, e.g. a stored single object can be regarded a one element list. For manipulation functions relaxing the cardinality is allowed, e.g., setting an optional to absent can lead to an empty stored list. However, in both directions the opposite does not work and the developer of the handwritten code can implement appropriate exceptions or handlers for all the problems that arise if the language extension is not conservative.

In summary, a *conservative extension* that means an extension that preserves concrete syntax and abstract syntax is relatively restricted in the adaptation of inherited productions, but allows developers to rely on a certain robustness that eases development.

Chapter 8

Visitors for AST Traversal

co-authored with Robert Heim, Nico Jansen

Processing a model requires to implement operations on the AST of the model and often also its symbol table (cf. Chapter 9). Since many operations share the same traversal algorithm it is favorable to separate the traversal algorithm and the actual operations on individual nodes. Thus, the traversal algorithm becomes reusable.

The *visitor pattern* [GHJV94] separates operations on complex data structures from the structure itself. It provides *visit* methods that act as hook points during a predefined traversal of the data structure. Thereby, it is easy to *add new operations* without touching the implementation of the data structure itself or its traversal algorithm.



Tip 8.1: Modularity and Reuse of Visitors

The visitor infrastructure enables modularity and reuse along several criteria:

1. Visitors can be defined for individual grammars and reused in composed languages.
2. Navigation and visiting functions are decoupled.
3. Larger parts of the infrastructure, e.g. the navigation, is generated with defaults.
4. Handwritten code on a sublanguage can be directly reused without changes in extended languages.

The original visitor pattern [GHJV94] describes the traversal algorithm as part of the data structure. For AST or symbol table processing this prohibits adjusting the traversal in specific visitor realizations as they all share a common (generated) implementation. Hence, MontiCore provides a combination of classical *internal visitors* that define traversal within the data structure and *external visitors* that define it in the visitors [Oli07]. Furthermore, the original visitor pattern is *imperative* since it stores the result of a visitor run as state in the visitor. This approach is distinguished from *functional visitors* [Oli07]. The latter

approach utilizes return values of methods to prevent stateful calculations. MontiCore's visitor infrastructure is external and allows imperative and functional use.

The following sections describe MontiCore's visitor infrastructure and how it enables agile development of concrete visitors to process ASTs. The same visitor infrastructure also allows to access the symbol table as extension to an AST, which together with the symbol table is explained in Section 9.9.

8.1 Visitor Infrastructure for a Language

Given a grammar, MontiCore generates the AST classes (cf. Chapter 5) and a parser to translate a textual model into its AST representation (cf. Chapter 6). Additionally, MontiCore generates a visitor infrastructure that consists of a default traversal algorithm for the AST and the symbol table (explained in Section 9.9) as well as `visit` methods serving as hook points for the processing of specific AST nodes.

Generally, the *visitor infrastructure* of a language L supports a depth first traversal algorithm on the AST of language L providing the following four methods for each nonterminal C, which results in an AST class ASTC of L:

- `handle(ASTC node)` defines the iteration algorithm of ASTC (default: depth first). By default `handle` calls `visit`, `traverse` (the children), and `endVisit` on `node`.
- `traverse(ASTC node)` defines a climbdown strategy (i.e., along the children; no order is guaranteed by the default implementation).
- `visit(ASTC node)` is called when entering `node`.
- `endVisit(ASTC node)` is called when leaving `node`.

In the following, we explain MontiCore's realization of the visitor pattern, which further extends the basic concept to support seamless traversal of all nodes for composed languages. For each language L, the generator produces the interfaces `LTraverser`, `LVisitor2`, and `LHandler` as well as the class `LTraverserImplementation`.

8.1.1 Traverser Interface and Implementing Class

```
1 package l._visitor; Java «gen» LTraverser
2
3 public interface LTraverser {
4     // ... simplified list of methods
5
6     // Hooks, to be adapted for concrete functionality:
7     default public void visit    (ASTC node)
8     default public void endVisit (ASTC node)
9 }
```

```

10 // provides default implementation to handle a node
11 default public void handle (ASTC node)
12
13 // provides default implementation to
14 // manage traversal though all sub-nodes
15 default public void traverse(ASTC node)
16
17 // Same for each other type of nodes (here D):
18 default public void visit (ASTD node)
19 default public void endVisit(ASTD node)
20 default public void handle (ASTD node)
21 default public void traverse(ASTD node)
22 }

```

Listing 8.2: Signature of a Traverser for language L

The traverser is the conceptual entry point for every action within the visitor infrastructure. It comes with the default strategy for handling and traversing the AST, thus implementing the respective methods for all types of nodes. Furthermore, the traverser manages visitors for the different sublanguages, which contain the implementations for the `visit` and `endVisit` methods. The traverser itself contains default implementations for these methods as well, delegating this call to all visitor implementations that have been added. If a special traversal is required that differs from the default, it is possible to add handlers to the traverser that realize the alternative behavior. Listing 8.2 shows an excerpt of the generated traverser interface for language L with nonterminals C and D, providing the corresponding visitor-related methods. While `visit` and `endVisit` calls are always delegated to the corresponding `LVisitor` implementations, `handle` and `traverse` can be managed directly and are only redirected if an `LHandler` is added.



Tip 8.3: Standard Traversal of Language L

For a given language L, it is common to use the `LTraverser` to process the AST.

It comes with a default traversal algorithm following a depth-first strategy on the AST. It is rarely required to adjust the default depth-first traversal. Usually, it is sufficient to provide the `visit` methods by implementing the `LVisitor2` interface.

Attaching an `LHandler` is only recommended when explicitly customizing the handling or the traversal of nodes.

Conceptually, the `LTraverser` implements the `visit`, `endVisit`, `handle`, and `traverse` methods for each AST node of a language, including all sublanguages. The `LTraverserImplementation` manages attributes of visitors and handlers, which can be attached for delegating the corresponding method calls. A language developer always implements against the API of the traverser interface, when instantiating it using the language-specific mill (cf. Section 11.5). As the implementing class only manages the attributes, it must not be adapted, as all changes will not be transferred to inheriting languages. Figure 8.6 shows the generated infrastructure for a language L. The class


Tip 8.4: Visitor Infrastructure Generated for each Language L

MontiCore generates three interfaces and two classes for the visitor infrastructure of each language L. They are explained in the following and are generated to these locations:

```

1 Directory: target/.../sourcecode/
2 Files:    l._visitor.LTraverser.java
3           l._visitor.LTraverserImplementation.java
4           l._visitor.LVisitor2.java
5           l._visitor.LHandler.java
6           l._visitor.LInheritanceHandler.java

```

LVisitor2 and LHandler are meant for extension by subclassing. They are realized as interfaces, providing default implementations for their methods. The LInheritanceHandler is a specific realization of an LHandler provided by MontiCore covering a particular traversal strategy that also calls all `visit` and `endVisit` methods of super types of a handled node (cf. Section 8.1.4).

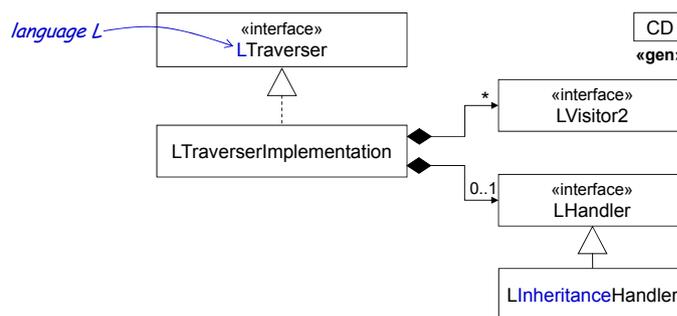


Figure 8.6: Generated visitor infrastructure for a language L

LTraverserImplementation implements its corresponding interface and contains a list of LVisitor2s and a at most one LHandler. While it only makes sense to adapt the default traversal once for each traverser, we can realize multiple visitors for visiting a node. This mechanism can be used to parallelize multiple visitors that share the same traversal algorithm, resulting in more efficient computations, while retaining the modularity of the visitor implementations. Furthermore, MontiCore generates the LInheritanceHandler as a predefined, yet customized handler with a traversal strategy covering all super types of a handled node (cf. Section 8.1.4).

Figure 8.7 shows an example of the traverser for the Questionnaire language described in Section 21.7. A concrete class (cf. QuestionnaireTool at the bottom of Figure 8.7) encapsulates the traverser and hooks in a corresponding visitor instance (cf. QuestionnairePrettyPrinter). The traverser provides the default methods, including the traversal algorithms. Thus, during runtime, the traverser checks for `handle` and `traverse` methods if a customized handler is provided. In this case, it delegates the call. Otherwise, it follows the default depth-first traversal strategy. Similarly,

Technical Info 8.5: Mechanisms used by the Visitor Pattern

The *static type* of a variable or parameter is the type information known at compile time (i.e., the type written within the source code). The *dynamic type* is the actual type of the stored object. For example, a variable could be typed by an interface type, but its value might be of a class type that implements the interface.

Calling a method implementation based on the dynamic type of an argument is called *dynamic dispatching*. Java is a single dispatch language since it only provides dynamic dispatching on the `this`-reference. This enables overriding methods since the more specific overridden version is executed during runtime. However, the method selection does not take into account the dynamic type of the other arguments. This is also different from *method overloading*, which is based on the static type information of arguments.

To realize an extensible and flexible visitor pattern, *double dispatching* is needed to call the correct, dynamically selected method.

This is realized by adding `accept(LTraverser)` methods within each AST node of a language L. Instead of directly calling the `handle` method of a node's child, the child's `accept` method is called with the traverser instance as argument (cf. Figure 8.7). The `accept` method then simply calls back the appropriate `handle` method by using the child's specific type. Through these two calls the double dispatch is realized in a typesafe, efficient way and the correct visit method is called.

This pattern is extended to also assist language composition where children might be extended, but the original visitor interface does not statically provide `handle` methods for the new child-types during (its former) compile time. This problem is known as the *expression problem* and, therefore, MontiCore provides extended infrastructure. More details can be found e.g. in [HMSNRW16].

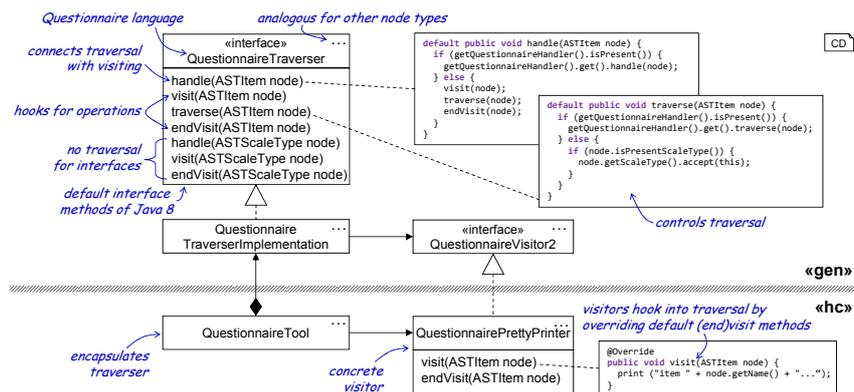


Figure 8.7: Traverser for the Questionnaire language and an handcoded usage

the traverser delegates visiting the specific AST nodes to the hooked visitors (in this case, the `QuestionnairePrettyPrinter`). The visitors must implement the interface

QuestionnaireVisitor2 where the functionality for visiting nodes must be implemented by overriding the methods `visit` and `endVisit`. Executing the traversal and the corresponding concrete visitors is as simple as calling the traverser's `handle` method for a given AST node. The default implementation of the traversal is based on simulated double dispatching for determining the dynamic runtime types of a node's children (cf. Technical Info 8.5).

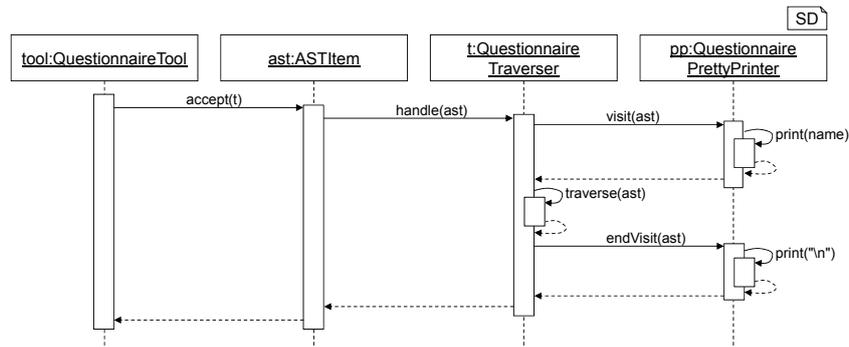


Figure 8.8: Control flow of a traverser with an attached visitor implementation

The control flow of the pretty printing example for the Questionnaire language is shown in Figure 8.8. The pretty printer `pp` carries the executed `visit` and `endVisit` operations. With respects to the interaction the interesting aspect is the `handle`, `visit`, `traverse`, `endVisit` call sequence.

In the `traverse` method, the traversing of sub-nodes results in a double dispatching call that consecutively calls `accept(t)` on all sub-nodes, which in turn again trigger the `handle` method of the traverser. This behavior is omitted in the diagram, as it is only a repetition of the already represented traversal pattern.

The provided visitor infrastructure is a powerful mechanism to traverse the whole AST and perform distinct actions on the different nodes, while also retaining all advantages of available type information.

8.1.2 Visitor2 Interface

The visitor interface provides `visit` and `endVisit` methods for each AST node of a language `L`. When traversing the AST, these methods are called per delegation using a traverser with corresponding hooked in visitors. Thus, following a separation of concerns approach, the visitor interface itself does only contains methods for visiting an AST node. Listing 8.9 shows an example of the `LVisitor2` interface providing corresponding methods for the AST nodes `C` and `D`. By default, the `visit` method is called directly when reaching the respective node during traversal (i.e., at the beginning of the `handle` method of the delegating traverser). Analogously, the `endVisit` method is called when leaving the corresponding node, i.e., after handling all sub-nodes in the given AST. Both methods

```

1 public interface LVisitor2 {
2   // for each kind of nodes:
3   default public void visit(ASTC node) { }
4   default public void endVisit(ASTC node) { }
5   default public void visit(ASTD node) { }
6   default public void endVisit(ASTD node) { }
7 }

```

Listing 8.9: Simplified presentation of `visit` and `endVisit` operations in the visitor interface of language L

provide an empty default implementation, enabling adding operations by overriding only the required methods.

For adding visitor implementations, the traverser offers the method `add4L(LVisitor2)`, providing a hook point for multiple instances. The visitor interfaces are language-specific, providing only `visit` and `endVisit` methods for AST nodes directly defined within the corresponding grammar. For composed languages, the traverser supports adding partial visitors for each sublanguage X by offering the same `add4X` methods (cf. Section 8.2). This fosters reusability across language boundaries.

To enable simultaneous traversing for independent visitor operations, MontiCore allows adding multiple visitors to a traverser, which are then processed consecutively at each node. Managing lists for each type of visitor interface allows encapsulating side-effect-free visitors together within a single traverser. However, when using the mechanism, a language developer must ensure that the effects of the hooked instances are not conflicting.

8.1.3 Handler Interface

As the traverser comes with a default strategy for traversing and handling the AST nodes of a language, it generally suffices to only use a traverser in combination with several added visitors. However, in some cases, it is necessary to adapt the default depth-first strategy towards a more sophisticated or purpose-oriented approach. Therefore, MontiCore introduces for each language L the `LHandler` interface that can be set in a traverser for providing customized `handle` or `traverse` methods.

Listing 8.10 shows an example of these methods with default implementations. These are generated for all AST nodes of a language L and provide the same standard behavior as the traverser. This enables overriding specific methods only while obtaining the default behavior for the others. As the traverser is the conceptual entry point that actually operates on the AST, every interaction must be delegated back towards the traverser. Thus, the `handle` method uses the `getTraverser` method to perform the consecutive `visit`, `traverse`, and `endVisit` operations. This approach is a variation of the `realThis` pattern described in Section 11.2. The main difference is that traverser and handler do not have any inheritance relationship to each other. This has the effect that entities, such as the AST nodes, only need to be linked to traversers and not to handlers. Further, avoiding complex inheritance relationships retains an efficient compile time.

```

1 public interface LHandler {
2     // for each kind of nodes:
3     default public void handle(ASTC node) {
4         getTraverser().visit(node);
5         getTraverser().traverse(node);
6         getTraverser().endVisit(node);
7     }
8     default public void traverse(ASTC node) {
9         // here: empty method body, as ASTC has no sub-nodes
10    }
11 }

```

Listing 8.10: Simplified presentation of `handle` and `traverse` operations in the handler interface of language L

To add a handler, a language developer can use the corresponding setter of the traverser. As it is only useful to override the traversal once, it is possible to add a handler using the method `setLHandler`, which is provided for each sublanguage L. This mechanism not only adds the handler to the traverser but also vice versa. As described, the handler needs to delegate interacting calls with the AST back to the traverser. Thus, the `setLHandler` method also automatically adds the traverser to the implementation of the handler. This requires implementing the `getTraverser` and `setTraverser` methods. The signatures of these methods are also prescribed by the interface.



Tip 8.11: Configuring a Traverser

To configure a traverser for language L, perform the following steps:

1. Create a class that implements the language-specific `LVisitor2` interface overriding the required `visit` and `endVisit` methods.
2. If customized handling or traversal of nodes is needed, also create a class that implements `LHandler`.
3. Instantiate the (predefined) traverser using the language-specific mill. The traverser is not overwritten, but configured.
4. Instantiate all visitor and handler realizations and add them to the traverser as configuration.

8.1.4 Inheritance Handler for Explicit Visit of Supertypes

The default traversal strategy visits nodes only in their most specific type. In case a language definition includes productions that extend or implement others (especially when interface or abstract productions occur) a more sophisticated traversal is required to enable hooking into the intermediate node types. This, for example, enables implementing a

common operation for similar specific node types that share a super type (e.g., an interface) by overriding the `visit` hook of the shared super type. This of course is only possible if the super type provides all relevant information for the operation. A simple example use case is counting the occurrences of AST nodes of the super type.



Tip 8.12: Inheritance Handlers should be Preferred

It is common that a concrete handler implements the `LInheritanceHandler` since a language `L` often includes interfaces or abstract productions or at least some productions that extend others. Even if those do not exist, the inheritance handler is generated and we recommend to use it, because as the language evolves, newly introduced nodes that extend existing ones (or those that implement an interface) can then automatically adopt the behavior of its supertypes.

The generated mill also provides a preconfigured traverser (i.e., the `inheritanceTraverser`) that always uses the inheritance handlers for the nodes of each sublanguage. This allows inheriting visitor behavior across language boundaries without any configuration in the composing language.

To this effect, MontiCore generates the `InheritanceHandler` as a predefined handler implementation that does not only visit nodes in their most specific type, but also calls the `visit` methods of all their super types. An example is the provided class `ASTRange` depicted in Listing 8.13, where the `handle` method for `ASTRange` also visits the node with its super type `ASTScaleType` and the other supertypes as defined by the derived AST data structure of the language.

```

1                                     Java «gen» QuestionnaireInheritanceHandler
2 public class QuestionnaireInheritanceHandler {
3     // handle for ASTRange calls four visit methods because
4     // ASTRange extends/implements
5     // ASTScaleType, ASTQuestionnaireNode, ASTNode:
6     public void handle(ASTRange node) {
7         getTraverser().visit((ASTNode) node);
8         getTraverser().visit((ASTQuestionnaireNode) node);
9         getTraverser().visit((ASTScaleType) node);
10        getTraverser().visit(node);
11        getTraverser().traverse(node);
12        getTraverser().endVisit(node);
13        getTraverser().endVisit((ASTScaleType) node);
14        getTraverser().endVisit((ASTQuestionnaireNode) node);
15        getTraverser().endVisit((ASTNode) node);
16    }
17 }
```

Listing 8.13: Implementation of an inheritance handler's `handle` method

8.2 Visitors for Composed Languages

Chapter 7 has described the various forms of language composition and how they affect the concrete and the abstract syntax. The following sections explain how the visitor infrastructure supports composition.

For *language aggregation* the visitors of the involved languages remain separated since the models remain separated as well. Therefore, in case of language aggregation nothing needs to be done. This is different from *language embedding* and *inheritance* where the language elements of the embedded/inherited language become part of the host language. Therefore, these cases are discussed below based on examples.

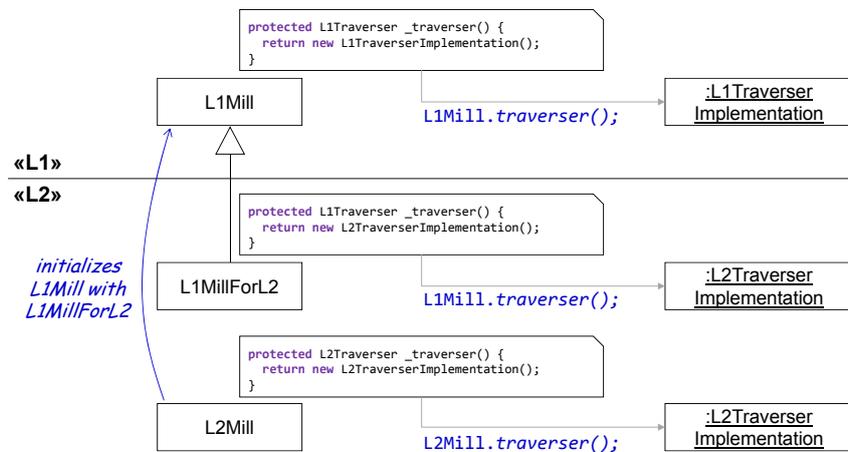


Figure 8.14: The traverser instantiation via mill always guarantees the most specific traverser type in composed languages

In general, the proposed infrastructure is specifically designed to support reusability of visitors in composed languages. A traverser always provides `visit`, `endVisit`, `handle`, and `traverse` methods for all AST node types of the included sublanguages. These methods delegate to the added, language-specific visitor and handler implementations. Thus, a traverser is aware of all traversable nodes of all sublanguages (in contrast to a visitor or a handler that only knows the nodes specific to the respective language). The traverser can always handle the incoming AST by delegating the method calls to the corresponding visitors or handlers. Therefore, it is essential to use the most specific traverser.

Because it cannot be known, in general, how a language is composed with other languages in the future, the developer of a visitor has to program exactly against the current language. To be able to reuse such code in compositions, MontiCore uses a mill (cf. Section 11.5), which transparently provides the required traverser type of the composed language. Figure 8.14 gives a brief overview for two languages L1 and L2, where the latter extends the first. Implementing a reusable traverser from L1 is as easy as calling the static method `L1Mill.traverser()`. As a mill in a composed language is initialized with the most specific instance, it can always provide the correct traverser. Thus, in the context of L1, the mill provides an `L1Traverser` (respectively its implementing class), while in the context of L2, the mill is exchanged with an `L1MillForL2` instance, resulting in a dynamic

provision of an `L2Traverser`. Thus, the reusability of traversers from sublanguages is guaranteed, as long as the mill is used for the instantiation of traversers.

8.2.1 Visitor Infrastructure for Language Inheritance and Extension

If a language `LG2` is an AST-conservative extension of `LG1`, then a visitor `V1` of the original language `LG1` can be reused directly in the context of the language `LG2` by attaching it to a corresponding traverser for the language `LG2`. The implementations of the AST classes remain the same and thus the visitor `V1` supports these classes.

There are two challenges to be discussed: First, the extension `LG2` of the language `LG1` might not be AST-conservative. That means that the production of at least one non-terminal `N` overrides the original production in such a manner that some elements of the original body are omitted or their cardinality changed. Hence, some of the assumptions of the original visitor of the original language may be violated. This needs to be carefully clarified by examining the manually implemented behavior of `V1`.

Second, a visitor `V1` for the language `LG1` usually can only be applied to the new language `LG2` with reservation because new nonterminals occur that the old visitor `V1` is not aware of. While traversing the new nodes is covered by the traverser, `V1` has no specific `visit` methods for those new nonterminals, resulting in handling these without any specified behavior. Therefore, either visitor `V1` needs to be subclassed by the new visitor `V2`, or the language developer needs to add a second visitor `V2` to the traverser. Adding visitors to the traverser is more general because it allows composing several base language visitors instead of only extending a single one [HMSNRW16] and further follows the separation of concerns principle for the dedicated sublanguages.

We demonstrate visitor extension in this subsection on the following example for automata and give a variant in the following subsection. In Listing 8.16, l. 1 and 2 the `Automata6` language conservatively extends `Automata5` from Listing 8.15. Listing 8.16 then introduces a new form of transition that has an additional output (l. 4). The new transition implements the interface `AutElement`, enabling to use complex transitions whenever an instance of `AutElement` is required.

The resulting AST data structure is illustrated in Figure 8.17. The AST node `ASTAutElement` is implemented by the new `ASTTransitionWithOutput` node type of `Automata6`. `MontiCore` also produces the interface `Automata6Visitor` for the new language. Here, the hand-coded visitor implementations extend their respective generated language-specific visitor interfaces. `Automata5PrettyPrinter` provides the visiting behavior for the basic automaton elements, while `Automata6PrettyPrinter` adds default implementations for the new node type `ASTTransitionWithOutput`. Using both functionalities, a language developer must add the implementations to the traverser, using the `add4Automata5` and `add4Automata6` methods. This way, all implementations of the original language's visitor (and its interface) are reused. Hence, when implementing a visitor for the new language, the aggregated default implementations are available. Consequently, a complete pretty printer for the new language can be implemented by reusing the `Automata5PrettyPrinter` class of the original language without modification and


```

1 public class Automata6PrettyPrinter
2     implements Automata6Visitor2 {
3     @Override
4     public void visit (ASTTransitionWithOutput node) {
5         print (node.getFrom());
6         print (" - " + node.getInput () + " / " + node.getOutput () + " > ");
7         print (node.getTo());
8         println ("");
9     }
10 }

```

Listing 8.18: A visitor implementation for the new language Automata6

printer of the Automata5 language is able to visit `ASTTransition` nodes but does not know about the class `ASTTransitionWithOutput`.

As MontiCore demands retrieving a concrete traverser from the mill, a valid traversal is always ensured. Thus, when reusing the `Automata5PrettyPrinter` without adding any `Automata6Visitor2` implementation, the AST can still be traversed and each node originating from Automata5 is visited. Nodes of the type `ASTTransitionWithOutput` would be handled but without visiting the nodes explicitly, as by default, there is no visitor attached that deals with Automata6 nodes. Furthermore, as the traverser knows how to handle these nodes, all children will be traversed as well, always resulting in a complete traversal of the entire AST. This means that a concrete visitor implementation can easily be reused in an inheriting language by adding it to a traverser. Considering that the mill dynamically sets a traverser to its most specific type at runtime, even a preconfigured traverser from a base language can be reused in an inherited one with no additional effort.

Please note that defining visitors for *language inheritance* should not be confused with the inheritance visitor. These mechanisms are orthogonal.

8.2.2 Visitor for Language Inheritance with Overriding Nonterminal

It is worth to note that visitors can also be reused when some of the nonterminals of the original language are overridden. As already said, this should be done in a conservative way, such that the body of the nonterminal does not provide surprises (e.g., unsets attributes or semantically relevant terminals) for visitors of the original language.

The following example (cf. Listing 8.19) is very similar to the previous one. `Automata15` accepts the same models as `Automata5`, but because `Automata15` has not been prepared for explicitly providing an interface for extension, we redefine the `Transition` nonterminal, such that transitions with output become possible (cf. Listing 8.20). The new production body conservatively extends the old one, as it still allows to parse all old transitions, but it also has an extension in the form of an optional output.

The resulting AST is different from the previous example `Automata6`. In particular, two versions of `ASTTransition` classes exist:

8. Visitors for AST Traversal

```
1 grammar Automata15 extends de.monticore.MCBasics { MCG Automata15
2
3   Automaton = "automaton" Name "{" (State|Transition)* "}";
4
5   State = "state" Name ";";
6
7   Transition =
8     from:Name "-" input:Name ">" to:Name ";";
9 }
```

Listing 8.19: Automaton language without explicit extension point

```
1 grammar Automata16 extends Automata15 { MCG Automata16
2
3   start Automaton;
4
5   @Override
6   Transition =
7     from:Name "-" input:Name ("/" output:Name)? ">" to:Name ";";
8
9 }
```

Listing 8.20: Conservative extension of transitions from Automata15 of Listing 8.19

```
1 Directory: out/target/.../sourcecode/ Files
2 Generated Files: automata15/_ast/ASTTransition.java
3                  automata16/_ast/ASTTransition.java
```

The version '16 of `ASTTransition` (cf. Listing 8.20) is furthermore a subclass of version '15. Thus, the pretty printer for `Automata16` is again implemented as a visitor, which can be hooked in a traverser together with version '15 of the pretty printer.

```
1 public class Automata16PrettyPrinter Java «hw» Automata16PrettyPrinter
2     implements Automata16Visitor2 {
3   @Override
4   public void visit(automata16._ast.ASTTransition node) {
5     print(node.getFrom() + " - " + node.getInput());
6     if(node.isPresentOutput()) {
7       print(" / " + node.getOutput());
8     }
9     print(" > " + node.getTo());
10    println(";");
11  }
12 }
```

Listing 8.21: A visitor of the new language for an overridden nonterminal

The main difference can be seen in line 6 of Listing 8.21, which accounts for the additional optional attribute `output`. Line 4 of Listing 8.21 shows an important detail; namely, it is a `visit` method for the new version of the `ASTTransition` class. To avoid ambiguities the fully qualified class name is used.

Which form of extension the user actually wants to use is potentially a matter of taste but often depends on the choice of the designer of the original language, who may have explicitly included extensibility by providing interfaces.

8.2.3 Visitors for Compositional Language Embedding

Language embedding, especially when multiple languages are involved, enforces composition of existing and reusable visitors. Compositional visitors are often used when composing context conditions that have been defined for individual sublanguages.

By construction, MontiCore supports visitor composition by combining visitors and handlers of different sublanguages into a single traverser that delegates the distinctive method calls. As said earlier, the generated infrastructure uses a variation of the `realThis` object composition pattern described in Section 11.2. This leads to a powerful but also relatively complex object structure. It is described in the following based on the pretty printing of a composed language `Automata3`, which is already defined in Listing 7.15 on page 125.

We assume that two pretty printers are given in the classes `ExpressionSublangPP` and `InvAutomataSublangPP`. Both implement the usual visitor interface. However, as the printer for the expressions comes with a customized `handle` method, it also implements the corresponding handler interface. These are intentionally prepared for composition. That means the handlers must delegate each operation with the AST back to the traverser, as only traverser instances navigate through the AST. Therefore, they implement the corresponding `getTraverser` and `setTraverser` methods (see Listing 8.22):

```

1                                     Java «hw» ExpressionSublangPP
2 public class ExpressionSublangPP
3     implements ExpressionVisitor2, ExpressionHandler {
4
5     protected ExpressionTraverser traverser;
6
7     @Override
8     public void setTraverser(ExpressionTraverser traverser) {
9         this.traverser = traverser;
10    }
11
12    @Override
13    public ExpressionTraverser getTraverser() {
14        return traverser;
15    }
16    // ... more methods
17 }
```

Listing 8.22: Prepare a handler implementation for compositional use

Furthermore, to become compositional, all methods of a handler are implemented in such a way that they do not use `this` (neither explicitly or implicitly), but use `getTraverser` instead. For example, if attributes need to be shared, `getTraverser().getAttribute()` is used. Thus, implementing these methods is mandatory for handlers (and thus, also enforced by the generated infrastructure) and optional for visitors. By default `visit` and `endVisit` methods only operate on the given AST node. However, if they require shared state information, they can also interact via the traverser using the same pattern. As an alternative, one can externalize all states of the visitors into a separate shared object. In the example, the object is an `IndentPrinter` instance, which does the printing and manages indentation. The class `IndentPrinter` is independent of any AST and can thus be reused in all pretty printers. It is noteworthy that all generated handler classes and interfaces implement the signatures of `getTraverser` and `setTraverser` by default.

The example of language embedding for visitors in Figure 8.23 is based on the example in Listing 7.15. The `ASTInvariant` of the `Automata3` language implements the `ASTInvariantExt` interface of the `InvAutomata` language and stores an object of class `ASTLogicExpr`. The traverser class of the new language knows both original language visitor and handler interfaces for integration (here, only the handler for `Expression` is shown, as the others are not used in this example).

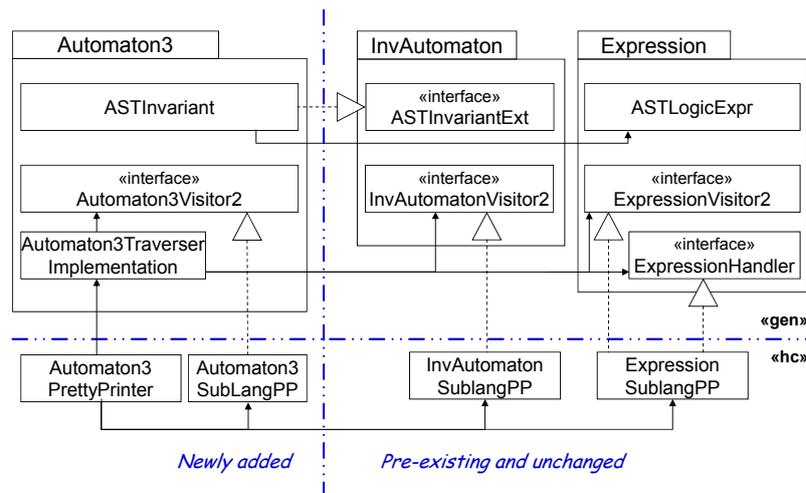


Figure 8.23: Overview of visitor infrastructure for `Automata3`

Figure 8.23 also shows that neither the original languages nor their handwritten pretty printers have to be adapted. The composition of two languages in a *third* grammar `Automata3`, however, leads to the definition of *four* classes:

- Two original pretty printers are reused directly.
- A rather small pretty printer for the new nonterminals of grammar `Automata3` is manually defined in `Automata3SublangPP`.
- The composition is manually carried out in `Automata3PrettyPrinter` that instantiates the `Automata3TraverserImplementation`, whose only purpose is to

delegate to the correct pretty printer while traversing the AST.

This complexity is necessary because Java does not allow multi-inheritance of classes. Hence, it is not possible to extend both languages' pretty printers. Instead, MontiCore uses delegation.



Tip 8.24: How to Compose Visitors according to their Languages

Composition is technically challenging but does not enforce much encoding. For a new visitor the following needs to be done:

1. If using custom handlers, implement their `get-/setTraverser` methods to enable composition
2. Don't care about `handle` or `traversal` at language borders. The composite will do.
3. If visitors have a local state that should be shared, use an external object and share its reference in all visitors.
4. Build a method (potentially in another class) encapsulating the instantiation and connection of the generated traverser of the composed language and the needed visitors of the sublanguages.

This composition mechanism does not rely on reflection and is type preserving.

The class `Automata3TraverserImplementation` provides setters for visitors and handlers of all (potentially transitively inherited) original languages. MontiCore uses knowledge of the AST model by generating this traverser in such a way that it delegates all `handle` and `traverse` calls to the original concrete handler as well as all `visit` and `endVisit` calls to all attached original visitors of the language that the current node belongs to. So the traverser makes heavy reuse of all these methods. On the other hand, it is important that handlers delegate back to the composite.

In the example, the `Automata3SublangPP` visitor in Listing 8.25 visits `ASTInvariant` nodes (lines 5 and 10) to print additional comments:

```

1                                     Java «hw» Automata3SublangPP
2 public class Automata3SublangPP implements Automata3Visitor2 {
3
4     @Override
5     public void visit (ASTInvariant node) {
6         out.print ("/*[*/* " );
7     }
8
9     @Override
10    public void endVisit (ASTInvariant node) {
11        out.print ("/*[*/* " );
12    }

```

8. Visitors for AST Traversal

```
13 |
14 |     // ... more methods
15 | }
```

Listing 8.25: The implementation of the pretty printer for the Automata3 sublanguage (with only one nonterminal)

Please note that the traverser only delegates the `visit` and `endVisit` methods with the `ASTInvariant` signature to this visitor. The same is true for other visitors: Only the nonterminals that are explicitly defined in that language are delegated to the respective visitor or handler).

Next, all visitors and handlers are composed using the traverser like in Listing 8.26.

```
1 | protected IndentPrinter out;
2 | protected Automata3Traverser traverser;
3 |
4 | public Automata3PrettyPrinter(IndentPrinter o) {
5 |     out = o;
6 |     traverser = Automata3Mill.traverser();
7 |
8 |     // ... configured with three sublanguage visitors
9 |     traverser.add4InvAutomata(new InvAutomataSublangPP(o));
10 |     ExpressionSublangPP espp = new ExpressionSublangPP(o);
11 |     traverser.add4Expression(espp);
12 |     traverser.add4Automata3(new Automata3SublangPP(o));
13 |
14 |     // add expression sublanguage visitor also as handler
15 |     // as it provides a custom handle strategy
16 |     traverser.setExpressionHandler(espp);
17 | }
```

Listing 8.26: Composing the three visitors through delegation and giving them the same shared state

The statements from lines 9-12 instantiate the sublanguage visitors and directly add them to the composite. The object `o` is handed over to all sub-visitors to share the same state. Furthermore, as the expression's pretty printer also implements the corresponding handler interface, it is also added as such in line 16.

The composition always needs to compose visitors of all participating sublanguages. If we omitted a statement, then all nodes of that sublanguage are not visited anymore and therefore just ignored in a composition. For example, omitting the statement in line 11 would lead to a disappearance of the expression part. Omitting the statement in line 9 would omit everything related to printing the automata as most of the contents are provided by `InvAutomata` and thus would be ignored.

Finally, Listing 8.27 demonstrates how to use the composed visitor.

```

1 // Common storage for all pretty printers
2 IndentPrinter ppi = new IndentPrinter();
3
4 // The composite visitor
5 Automata3PrettyPrinter acpp = new Automata3PrettyPrinter(ppi);
6
7 // run the visitor
8 acpp.print(ast);
9 Log.println(ppi.getContent());

```

Listing 8.27: The composed visitors can be used as if it is only one monolithic component

In line 2, the shared state is instantiated. Line 5 creates the composite. In l. 8, the traverser is applied to the model by internally calling `ast.accept(traverser)`; and in l. 9 the result is retrieved from the object containing the shared state.

As mentioned, this mechanism acts as a blueprint for composition and can be reused in many forms. If no shared state is necessary, the `ppi` object can, of course, be omitted. The shared state could also be inside the traverser, when each visitor refers to the traverser for the state as well.

The blueprint shows that, in generally, for n languages that are composed by another language, $n + 1$ visitor objects for sublanguages are composed by a traverser for the overall language. Including the traverser this sums up to $n + 2$ objects in total if all kinds of nodes need to be addressed. In practice, n of these visitors can usually be reused and only one new visitor needs to be newly defined.

The configuration of a traverser from a sublanguage can be reused by simply reusing the corresponding traverser from the composing language. Retrieving traverser instances via a mill automatically ensures that instances of the traverser of the most specific composing language are used.

Note that it is not forbidden to write a single visitor class for several sublanguages, which means that it only has to be instantiated once and can be added several times. On the other hand, we also allow lists of visitors to be handed to a traverser to parallelize calls during traversal.

The realizations discussed in this chapter provide a lot of flexibility in various forms of compositions. However, when restructuring an extended subgrammar, for example, by letting it extend a new grammar, then, unfortunately, the visitor configurations also have to be adapted. Therefore, we recommend stabilizing the extended grammars first.

Chapter 9

Symbol Management Infrastructure

co-authored with Arvid Butting and Pedram Mir Seyed Nazari

This chapter gives an introduction to the management of symbols within models, respectively the generated infrastructure for symbol management. Further, it explains the process of resolving symbols.

Symbol tables are tightly integrated with the AST, but because parsing is based on a context-free grammar, the symbol table is constructed in an extra pass after parsing. However, the symbol table should be regarded as part of the AST, even if we will see, it extends the AST to a graph with an embedded spanning tree.

The symbol management infrastructure (SMI) is based on [MSN17] and [Völ11].

The SMI, as described in the following, is inspired by the general-purpose programming language Java. Java has a typical complex type system including extensibility through inheritance, private, protected, and public visibility, and generics. The SMI is designed to handle all these concepts and especially applicable to modeling languages. But the SMI is also defined in such a way that it assumes reasonable defaults for common cases and the SMI is, therefore, also usable in simple cases. The TOP mechanism also allows to adapt the SMI with handwritten code (cf. Chapter 14).

Overview: Defining Symbols directly in the Grammar

MontiCore generates default symbol handling. This covers a larger set of cases, but not everything. If someone is happy with the standard cases, this section will help, because it gives an overview. For a deeper understanding of how the mechanism works and how it can be adapted, the rest of the chapter is helpful.

There are three main concepts available in the grammar language itself:

- The keyword `symbol` attached to a nonterminal `K` defines that the Name occurring on the right side introduces a new symbol, i.e. is *symbol defining*. The symbol is of kind `K` and MontCore generates infrastructure to manage symbols of this kind in the class `KSymbol`.



Tip 9.1: How to read this chapter

This chapter addresses all aspects of the SMI:

- Foundational topics are discussed in Sections 9.1 (introduction, containing especially definitions for the terms used in this chapter), 9.4 (collaboration between AST, symbol, and scope),
- Tool designers interested in using symbols read Sections 9.5 (using symbols), 9.6 (instantiating symbol tables), 9.8 (resolving symbols), and 9.9 (the visitor design pattern for symbol tables).
- Language designers should read in addition Sections 9.2 and 9.3 (defining symbols and scopes).
- For language aggregation with existing SMIs in the sublanguages read Sections 9.7 (loading and storing symbols) and 9.10 (symbol tables and composed languages).

- When a name is *using a symbol*, this is modelled by `Name@K`, which tells MontiCore that here an existing symbol name of kind `K` is used. MontiCore generates infrastructure to efficiently connect symbol uses with their definitions.
- The keyword `scope` attached to a nonterminal tells MontiCore that the nonterminal opens a new scope and all symbols defined within the scope are managed there.

Scopes are configurable among others with: whether they shadow external symbols, and whether they encapsulate their symbols or allow external and qualified access.

Figure 9.2 shows how the grammar constructs to define symbol table infrastructure may be used.

```

1 // define a new form of symbols
2 symbol K = "... " Name "... "
3
4 // usage of a symbol
5 NT = "... " Name@K "... "
6
7 // a nonterminal defines a scope with local symbols
8 scope M = "... " "{" Body "}"

```

MCG

Listing 9.2: Example use of symbol defining grammar constructs

MontiCore uses the information given in Figure 9.2 to generate a bunch of additional classes and methods:

- Symbol classes, here `KSymbol` to carry symbol information.
- Scope managing classes for the language managing symbols defined in models.

- Loading and storage functionality for scopes and symbols.
- The AST for nonterminal K is extended to link to the symbol defined in K .
- The AST for NT is extended to allow efficient navigation to the symbol definition (and lookup).
- The AST for M is extended to link with the scope.
- All AST nodes know their *enclosing scope*, which allows to resolve for any kind of symbol within each part of a model.
- Methods are generated that build the scope and symbol structure after the AST structure has been created.

And all of the above mentioned classes can be adapted e.g. through the TOP mechanism, which is in many cases relatively straightforward because appropriate hook points exist.

9.1 Introduction to Symbol Table Concepts

Every textual software language has names to (1) identify and reference entities defined in the language and (2) use entities via these names. In most cases, names serve as identifiers consisting of a character sequence complying with some specific rules. Typical examples are names for Java classes or methods that may only contain letters, numbers, dollar signs and underscores. Furthermore, they may not begin with a number.

9.1.1 Symbols

Names occur in two forms: name definition and name usage. In some cases a new name is defined and used at the same time, e.g., attributes in Java are declared and initialized in a single statement. As a variant, entities may be introduced and thus (implicitly) defined with its first use – untyped languages often allow to introduce variables that way.

A *name definition* introduces a new name for a (new) model entity. In other words, a name definition defines a new model entity having some specific kind, such as state, class, or method. *Kinds* can enforce additional information, e.g., (method) signature or visibility.

A *name usage* refers to an entity defined elsewhere, either in the same or a different model. Depending on where in the source code the same name is used, it can refer to different entities, because of their restricted *visibilities*.

Rarely, it may happen that a symbol is defined along with its first use. These symbols typically do not have an additional body or carry extra information, but are only needed to connect structures. E.g., states in simple, flat automata or features in feature diagrams could be handled this way.

A *symbol* is an abstraction of the model entity. It contains the name that is defined and relevant information about the model entity that is needed to use it. For example, a *method*

definition in the model introduces the *method name* and leads to the introduction of the *method symbol*, which contains the method signature, but not the method body.

In order to easily and efficiently retrieve the relevant information that a name usage refers to, so-called *symbol tables* are employed. By definition, a *symbol table* is a data structure that maps a name to the corresponding symbol. It allows to effectively find declarations, types, signatures, implementation details etc. for a name. Due to restricted visibilities, *scopes* divide the mapping into smaller, typically hierarchically composed structures.

The *visibility* of a symbol is the region in a model where the symbol is accessible by its name. A symbol can be *shadowed* by another symbol. In Java, e.g., an attribute can be shadowed by a local variable within a method. In addition, an *access modifier* directly attached to the definition of the symbol controls visibility, e.g., public, private, and protected in Java.

Each symbol belongs to a specific *kind* depending on what kind of model element it denotes, e.g., variable, method, state, action, port, label, class, etc. The symbol contains useful information depending on the kind, e.g., for methods, the method name, the modifiers, the return type, and the parameter types.



Tip 9.3: What is a Symbol?

Symbols are introduced in the models by giving a certain model entity a name.

Every *symbol* has a name. A model entity that does not have a name cannot define a symbol.

Neither anonymous symbols, nor symbols with constructed complex names (that nobody knows of) are helpful and thus do not exist.

Each symbol has a *kind*. Depending on the kind, the symbol carries different essential information. For example, a state can be initial or final (booleans). A variable has a type. A class has a signature, consisting of attributes, methods, superclasses, and interfaces. A signal in the Internet-of-things (IOT) domain usually has a value range and a frequency.

A symbol is an abstraction of its defining model entity. It does not repeat the whole information from the AST. Otherwise, the AST itself could be used.

When symbols are exchanged between models in heterogeneous language environments, the symbols might occur with different kinds, names or other adapted essential information across a language respectively scope border. For example, a state `Ping` of an automaton might become an enum value `Ping` or, alternatively, a boolean method `isInPing()` in Java. An attribute `age` of a class diagram may be mapped to a private attribute `age` and two methods `getAge()` and `setAge(.)` in an Automata expression sublanguage.

9.1.2 Scopes

A *scope* holds a collection of symbols and impacts their visibility.

A *scope* typically is defined by a nonterminal with a body, which is enclosed by some kind of brackets and contains nonterminals that may introduce names. The brackets define the

visible containment structure and the nonterminals in the body may introduce new names that are then potentially restricted to be visible in their enclosing scope. Please note that a nonterminal may introduce several scopes, but MontiCore only assists one scope per nonterminal. If needed a simple refactoring of the grammar is possible.

Scopes are hierarchically nested, reflecting the nested visibilities of the symbols they define.

A *shadowing scope* allows its symbols to shadow other symbols that are already defined outside (e.g., in enclosing scopes). A *visibility scope* does not allow shadowing. E.g. a Java method spans a shadowing scope, while a statement block in Java only spans a visibility scope.

A management infrastructure for symbols, where the `SymbolTable` is a vital part of, serves three main purposes:

1. It allows a *quick navigation* from the usage of a symbol (via its name) to its definition.
2. It *collects relevant information* about a symbol, which may be spread among one or several sources. For example, for a class, the infrastructure may collect the list containing the attributes defined in the class and the attributes inherited by the class.
3. It acts as a *surrogate* for the entity definition in other models that use the symbol (via its name). Thus, it enables that only the symbol tables of other models can be loaded instead of loading (parsing) the complete models.

The result of a name search is a *symbol* that represents essential information about a named model entity. If the entity is defined in the loaded model, a link to the definition is also provided. Different symbols may have the same name, when defined in different scopes or if they are otherwise differentiable, e.g., like methods through argument types.

A symbol table consists of a *scope tree* (or scope graph, see Section 9.3) with associated lists of symbols at each scope to manage the visibility respective accessibility of symbols. Scopes contain symbol definitions, but also import and export symbols. Each symbol is defined in exactly one scope.

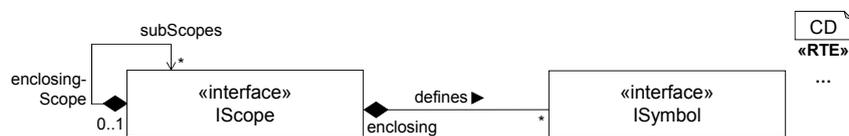


Figure 9.4: Overview of the main concepts of SMI

The SMI of MontiCore provides reasonable defaults to facilitate the development of language-specific symbol tables. Figure 9.4 gives an overview of the two main elements of the SMI, which are extended by generated, language-specific classes. Both are introduced in the remainder of this chapter. This chapter uses the Automata language (introduced

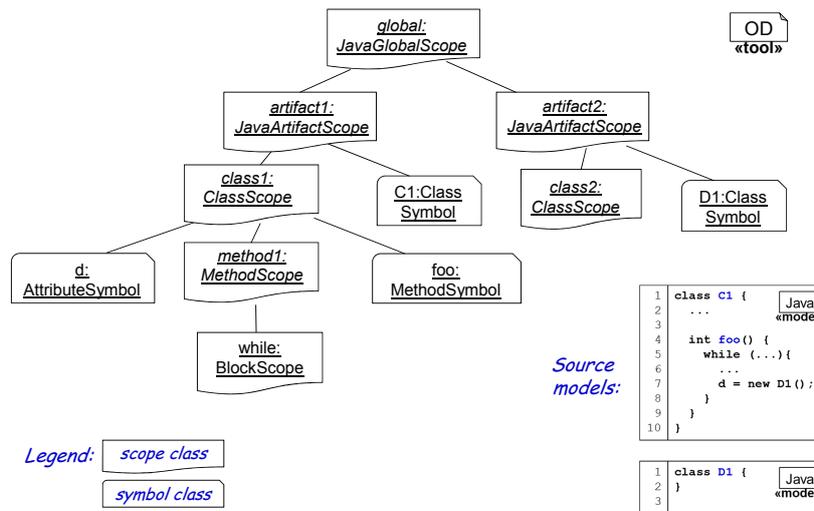


Figure 9.5: An example structure of a hierarchy of scopes

in Section 21.1) as well as concepts of the Java programming language for the illustrations of the generated parts.

Figure 9.5 shows an example scope structure for Java. In the object diagram, the global scope manages two artifact scopes, which internally define the class symbols C1 and D1. Each class has also a body and therefore a scope. The scope of C1 is furthermore containing two symbols `d` and `foo`. Java distinguishes the class scope from the artifact scope because in the same Java file (i.e. artifact) additional classes may be defined.

The various tool classes, such as `AttributeSymbol`, `MethodSymbol`, `ClassSymbol`, `BlockScope`, `MethodScope`, `ClassScope`, `JavaArtifactScope`, and `JavaGlobalScope` are generated by MontiCore for the specific language and their purposes and mechanisms are explained in the rest of the chapter.

It is noteworthy that the method `foo` has a body, in which local variables and other symbols can be defined, but the method symbol `foo` itself is defined outside and belongs to the enclosing class scope. The same holds for the class symbol C1.

The figure also shows that we represent symbols and scopes in a slightly modified form compared to ordinary objects and classes for better readability.

9.2 Defining Symbols

The keyword `symbol` was already mentioned in Chapter 4. It can be attached to nonterminals that are defined in the grammar. The keyword `symbol` indicates that a nonterminal's body defines a new symbol. We demonstrate the effect on an essential part of an Automata grammar, shown in Listing 9.6 that also contains a reduced form of expressions and statements.

```

1 grammar Automata
2   extends de.monticore.literals.MCCommonLiterals {
3
4   symbol scope Automaton =
5     "automaton" Name "{" (State | Transition)* "}" ;
6
7   symbol scope State =
8     "state" Name
9     (("("<<" ["initial"] ">>" ) | ("("<<" ["final"] ">>" ) ))*
10    ( ("{" (State | Transition | Counter)* "}") | ";" ) ;
11
12   Transition =
13     source:Name@State "-" input:Name ("|" Statement)? ">"
14     target:Name@State ";" ;
15
16   symbol Counter = "counter" Name "=" NatLiteral ";" ;
17
18   interface Statement ;
19   Print implements Statement = "!" Name ;
20   Increment implements Statement = "++" Name ;
21 }

```

Listing 9.6: Automata with counters and transition statements

The grammar in Listing 9.6 defines three kinds of symbols along with the three nonterminals `State` (l. 7), `Counter` (l. 16), and `Automaton` (l. 4). The grammar introduces a symbol kind for each nonterminal tagged with the keyword `symbol`, which leads to a number of additionally generated classes as described below in Section 9.2.2. For example, the nonterminal `State` introduces a symbol of the kind `StateSymbol` (and thus also a Java class of this name) and in automata models, each state actually introduces a corresponding symbol instance.

The keyword `symbol` can be attached to nonterminals that introduce a `Name` on the right-hand side.

If the `Name` is mandatory, exactly one symbol is created. If `Name?` is optional, a symbol is only created, if the modeler has actually given the model entity a name.

The `symbol` keyword cannot be added to a production of the form `A=Name*`, with a cardinality larger than 1. If this is desired, a restructuring of the grammar can help, e.g. by using `A=B* ; symbol B = Name` instead.

If the `symbol` keyword is added to a nonterminal that does not have a name on the right-hand side or a named `Name` nonterminal (e.g., `b:Name`), MontiCore produces an abstract symbol class. The abstract class contains an abstract method `getName()` that language engineers have to extend by applying the TOP mechanism.

The keyword `symbol` can also be assigned to interface nonterminals by adding it before the `interface` keyword. The effect of this is that all nonterminals that implement the interface define a symbol, even if these are not explicitly marked with the `symbol` keyword.

The keyword `symbol` is inherited both, from interface nonterminals to their implementing nonterminals, but also from a nonterminal A to its extending nonterminal B. The keyword `symbol` may only be "inherited" once ("single inheritance").

If the keyword `symbol` is inherited to B, then B nonterminals define symbols of the "inherited" kind `ASymbol` as well. This changes if the nonterminal B also explicitly has the `symbol` keyword attached because then it defines symbols of its own kind `BSymbol`, which is then a subclass of `ASymbol`.

9.2.1 Runtime (RTE) Classes For Symbols

Figure 9.7 (top part) depicts two interfaces that are provided by the SMI runtime environment (RTE).

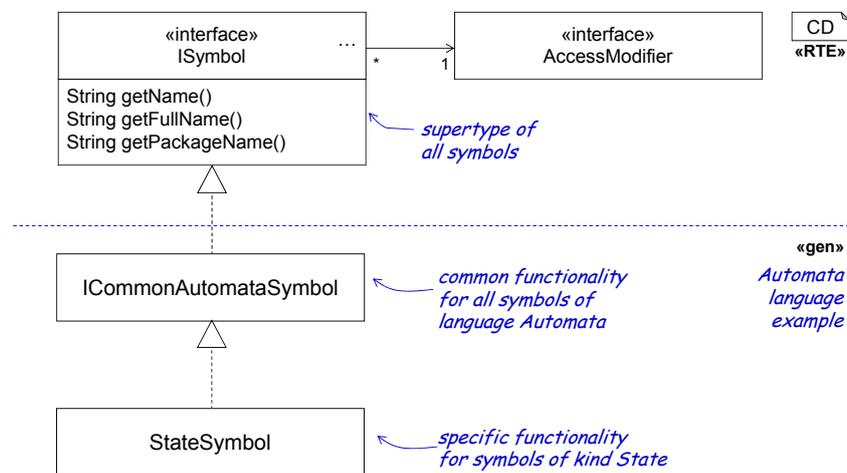


Figure 9.7: Symbol interfaces provided by SMI RTE and a language specific class

The `ISymbol` interface is the supertype of all symbols and groups information that every symbol has, such as the symbol's simple name (e.g., type name `Integer`, class name `Person` or state name `Ping`) via `getName`, its package name (e.g., `java.lang`) via `getPackageName`, and its fully qualified name (e.g., `java.lang.Integer`) via `getFullName`. As introduced before, every symbol has a kind (e.g., kind `method`, `state`, etc.), which is realized through the type of the symbol class (e.g., `StateSymbol`) that is generated for each symbol kind. As depicted in Listing 9.8, the generated classes extend the RTE classes accordingly.

```

1 public interface ISymbol {
2     // the name:
3     String getName();
4     String getPackageName();
5     String getFullName();
6
7     // connection to the AST defining node:

```

```

8  boolean isPresentAstNode();
9  ASTNode  getAstNode();
10 SourcePosition  getSourcePosition();
11
12 // a symbol knows the scope it is defined in:
13 IScope  getEnclosingScope();
14
15 // connection to access modifier
16 AccessModifier  getAccessModifier();
17 setAccessModifier(AccessModifier  accessModifier);
18 }

```

Listing 9.8: Interface of all Symbol classes

Each *symbol* has a *name*. Additionally, it has an optional *package name*. If the AST node defining the symbol is available, a navigation from symbol to the AST is possible through the respective methods.

Furthermore, each symbol is defined within its *enclosing scope* (see discussion in Section 9.3). The methods for obtaining the AST node and the enclosing scope defined in `ISymbol` will be overwritten in the generated subclasses by methods with more specific return types and additional methods for setting the AST node or the enclosing scope will be provided.

Figure 9.7 illustrates how concrete, generated symbol classes inherit from `ISymbol`. `StateSymbol` realizes the state symbol kind and will be explained in the following Section 9.2.2.

Only some AST nodes define a symbol. Thus, the `ASTNode` interface has no connection to the `ISymbol` interface. If a concrete AST node defines a symbol, such as the `ASTState`, then the AST class has generated methods for realizing the connection to the symbol.

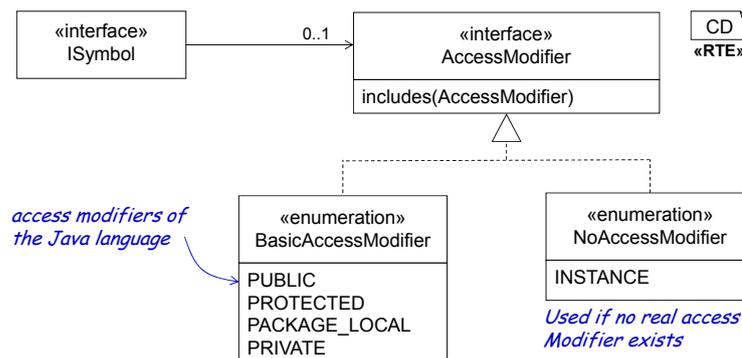


Figure 9.9: Symbols may carry access modifiers defining their visibility

Because a substantial number of languages provide the possibility that symbols define their own visibility using access modifiers, such as `public`, `protected`, or `private`, MontiCore supports a possibility to attach accessibility information already with the RTE class `BasicAccessModifier` as shown in Figure 9.9.

9.2.2 Generated Classes For Symbols

For each grammar, the generator creates an additional subdirectory (package) `_symboltable` with the following classes that allow to manage symbols and scopes. For each kind of symbol that is introduced with a `symbol` keyword in a language's grammar, the MontiCore generator produces four classes and an interface (here demonstrated by the example of the nonterminal `State`):

StateSymbol.java is generated to contain all relevant information about state symbols. The generator itself includes the *name* as well as links to the defining *AST node* and the *enclosing scope*. In case that a symbol spans a scope, the class also contains a link to the spanned scope.

If the default attributes are not enough, an extension of the symbol class with the `symbolrule` keyword or the TOP mechanism is possible.

StateSymbolBuilder.java builds instances of the class `StateSymbol`. The instance of the builder should be obtained through the language's `Mill`. It is mandatory to set the name of each symbol in the symbol builder.

StateSymbolDeSer.java realizes the serialization and deserialization strategy of `StateSymbols` for loading and storing symbol tables of the language. It is explained in more detail in Section 9.7.

IStateSymbolResolver.java realizes the resolution algorithm for adapted `StateSymbols`. This algorithm is factored into a class on its own so that language designers can provide their own algorithm, e.g. for adapting symbols of a foreign kind to `StateSymbols`.

These classes are typically added to the global scope to integrate the resolving for adapted symbols into the resolution process.

Symbol resolvers are explained in more detail in the context of resolving for adapted symbols in Section 9.10.4.

StateSymbolSurrogate.java is a subclass of the class `StateSymbol` that realizes loading on demand. This class must only be used by the symbol loader if additional symbols are stored elsewhere and potentially do not need to be loaded. This prevents transitive loading cascades and mainly applies for type / class symbols.

In addition, the following class is generated to aggregate language specific, but symbol common functionality:

ICommonAutomataSymbol.java is in addition generated as a common super-interface of all symbols that are specific in the language automata.

Its main purpose is to allow manual extension e.g. via the TOP mechanism if needed.

```

1 package automata._symboltable;
2
3 public class StateSymbol implements ICommonAutomataSymbol {
4
5     // symbol name
6     public StateSymbol(String name);
7
8     // signatures of some important methods:
9     public String getName ();
10    public String getFullName ();
11    public String getPackageName ();
12
13    public IAutomataScope getEnclosingScope ();
14
15    public ASTState getAstNode ();
16    public boolean isPresentAstNode ();
17
18    public AccessModifier getAccessModifier ();
19
20    public void accept (AutomataVisitor visitor);
21
22 }

```

Listing 9.10: Excerpt of the generated implementation of class `StateSymbol`

An excerpt of the method signatures that a generated symbol class contains is depicted by the example of the class `StateSymbol` in Listing 9.10. Each generated symbol class implements the language-specific symbol interface (here: the interface `ICommonAutomataSymbol` depicted in l. 3).

Because symbols always have a name, the constructor of the symbol class (l. 6) requires to set the symbol's name. The symbol class further contains methods for obtaining the symbol's name (l. 9), enclosing scope (l. 13), AST node (l. 15), and access modifier (l. 18).

The AST node is internally realized as an optional attribute because the AST node may not be present if the symbol has been loaded from a stored symbol table (cf. 9.7).

The method `getAstNode` either directly returns the AST or throws an error, if the AST is not present. The presence of the AST can be checked by calling the method `isPresentAstNode` (l. 16).

Symbol classes are traversible with visitors. Thus, all symbol classes realize an `accept` method (l. 20) for the language's visitor interface.

Extensions of these generated classes can again be achieved via the TOP mechanism described in Section 14.3 or by building subclasses. In the example, the `StateSymbol` class additionally should carry useful information, e.g., if the state is initial (`isInitial`) or final (`isFinal`). We might also be interested to know, which input the state can react to. In general, symbols should only carry the information that are relevant for correct usage of the symbol. What we regard as relevant information, however, depends on the form of use of the symbols.

9.2.3 Defining Additional Symbol Attributes via `symbolrule`

Symbol rules enable defining extra attributes and methods for symbol kinds, which results in additional fields and the methods of the respective symbol class. Examples symbol rules are given in Listing 9.11.

Symbol rules begin with the keyword `symbolrule`, followed by the symbol-defining non-terminal. They are realized similar to the `astrule` for AST classes. The body of a `symbolrule` contains attribute and method definitions. Attributes can be marked optional ('?') or as iterations ('*', etc.).

```

1 symbolrule S1 = a:AType b:java.lang.String? ;
2
3 symbolrule S2 = c:int*
4
5         method public int sumCs() {
6             int result = 0;
7             for(int i = 0 ; i < c.size() ; i++) {
8                 result += c.get(i);
9             }
10            return result;
11        }
12        ;

```

Listing 9.11: `symbolrule` that defines three symbol attributes and a symbol method

The symbol rule in l. 1 in Listing 9.11. defines two attributes for `S1Symbol`. The first attribute has the name `a` and is of the type `AType`. The second attribute demonstrates that qualified names may be used and that an attribute can be optional. From this, MontiCore generates the additional attributes `AType a` and `Optional<java.lang.String> b` into the Java class `S1Symbol`. The cardinality `*` shown in l. 3 leads to an attribute of type `List<Integer>` plus all appropriate access and modification methods as usual. MontiCore handles boxing and unboxing properly.

The second symbol rule also defines the method `sumCs()` for the class `S2Symbol`. As usual, a method definition in the body of a symbol rule begins with the keyword `method`.

Obviously, complex methods should better be integrated into symbol classes through the TOP mechanism. However, for simple methods the symbol rule can be an alternative that reduces the number of handwritten artifacts.

A benefit of defining fields via symbol rules is that MontiCore generates more than only the fields in the symbol class: Symbol attributes defined through symbol rules are included in the generated load/storage mechanisms for the symbol as described in Section 9.7 and Section 9.7.3 if the type of the attribute is a default Java type (`int`, `String`, etc.). When adding an attribute through the TOP mechanism, its persistence must be added manually as well.

9.3 Defining Scopes

As defined in Subsection 9.1.2 a scope holds a group of symbol definitions and limits their visibility, but also manages import from other scopes and export to other scopes. For example, local variables in a Java method are grouped together by the scope of that method. Thus, local variable symbols are *defined* in a *method scope*.

Scopes define the visibility of symbols. The exact visibility and accessibility of a symbol outside its own scope highly depends on the language that should be realized. MontiCore provides a general infrastructure for symbol management and offer the option to adjust it for language-specific mechanism if needed (1) by scope configuration mechanisms and (2) with handwritten code. This allows, e.g., realizing individual forms of symbol visibility concepts in scopes.

The keyword `scope` attached to a nonterminal production introduces a scope instance at each AST node of the nonterminal. We identify two scopes in the example grammar in Listing 9.6 (p. 161), the scope for `Automata` (l. 4) and scopes for `State` (l. 7). The keyword `scope` can be attached to any nonterminal, but it is mainly useful, if there actually are symbols that can be enclosed in a subscope, i.e. there are nonterminals on the right-hand side of the production that introduce symbols in their bodies.

Whether a nonterminal defines a scope is completely orthogonal to whether this nonterminal also provides a symbol. However, a *named scope*, for example, allows qualified access to internal names, while an *anonymous scope* (introduced via a nonterminal production without `symbol` keyword) usually hides the names defined internally. Examples are Java classes and methods versus statement blocks `{ ... }`.

9.3.1 Artifact Scope and Global Scope

Due to a modular structure of the many artifacts that are typically involved in a development project, the symbol management infrastructure provides two standard scopes: the *artifact scope* and the *global scope*.

The *artifact scope* represents the scope of the whole artifact and thus is the top scope of a model. When storing and loading symbols, these symbols are aggregated in artifact scopes.

MontiCore generates a language-specific artifact scope class for every grammar. For example, the class `AutomataArtifactScope` is generated for `Automata` language.

If no other local scopes are established, MontiCore adds all symbols defined in a model to its artifact scope, thus exhibiting a flat visibility structure. If a language introduces additional scopes, these scopes become subsopes of the artifact scope. The scopes within a model are typically arranged as a tree, which is structurally homomorphic to the AST. Each scope, thus, contains a potentially empty list of subsopes. This creates a tree of scope instances for each model, where the root scope is always an artifact scope.

Many modeling languages can benefit from a concept of structuring models into packages. Artifact scopes, thus, have a built-in mechanism for realizing packages. An artifact scope

holds an attribute capturing the package as `String` as well as a getter and a setter method. The attribute can be used to qualify and unqualify symbol and scope names with the package name.

Artifact scopes contain language-specific information for managing model artifacts and realize parts of the language-specific inter-model symbol resolution algorithm (described in Section 9.8).

The *global scope* of a language is the root of the scope graph of a language. Thus it is the direct or indirect enclosing scope of all scopes of the language. Its direct subscopes are usually artifact scopes since these are the top scopes of the models. That way, the global scope “connects” models and enables inter-model symbol resolution. The global scope usually also contains globally available information, such as global types like `int` and `boolean` in Java. These types can be used in every model without explicitly being imported. The global scope further is the basis for realizing language aggregation in MontiCore (cf. Section 9.10).

MontiCore generates language-specific global scope classes for every grammar, such as `AutomataGlobalScope` for the `Automata` grammar.

Because the global scope is responsible for loading stored symbols on demand from the file system, MontiCore uses the concept of *model path*. As the `classpath` in Java, the model path contains a list of paths containing the files to be loaded. To manage model path entries, the global scope has an attribute typed with the MontiCore RTE class `ModelPath`. Details are given in Section 9.7.

To be able to find symbol table files that should be loaded, these have to be stored at a location following the guidelines described in Section 9.7.5.

An instance of the global scope can be obtained via the mill by using the `globalScope()` method. In this form, the global scope comes fully configured and ready to use. The model path is by default empty. The default file extension of stored symbol tables is `sym`. The generated `DeSers` are used for serialization and deserialization and no resolvers for adapted symbols are hooked in.

Listing 9.12 serves as a teaser for the various configuration options of the global scope. Line 2 shows how to set the model path, which is described in more detail in Section 9.7.4. The file extension for looking up symbol tables can be changed from the default suffix `"sym"` using the `setFileExt` method as demonstrated in line 3 (cf. Section 9.3.2). Line 4 adds a `StateResolver` to resolve adapted symbols. This is detailed in Section 9.10.4. Finally, line 6 shows how to add custom symbol `DeSers` for serialization and deserialization, which is described in Section 9.7.

9.3.2 Runtime Environment Classes for Scopes

Since MontiCore 6, the symbol table infrastructure is strongly typed. Thus, the MontiCore RTE only contains general scope interfaces and generated, language-specific scope interfaces add language-specific methods. Further, the MontiCore generator produces specific

```

1 IAutomataGlobalScope gs = AutomataMill.globalScope();
2 gs.setModelPath(new ModelPath(Paths.get("src/models")));
3 gs.setFileExt("autsym");
4 gs.addAdaptedStateSymbolResolver(new MyStateResolver());
5 gs.putSymbolDeSer("automata._symboltable.StateSymbol",
6                 new MyStateDeSer());

```

Listing 9.12: Configuring global scope attributes

symbol resolution algorithms for each symbol kind. Thus, the MontiCore RTE scope-interfaces do not provide any common realization and are rather small.

Listing 9.13 shows the most important signatures that the RTE class `IScope` provides and that are implemented by all generated scope classes.

```

1 public interface IScope {
2
3     boolean isShadowing();
4     void setShadowing(boolean b);
5
6     boolean isExportingSymbols();
7     void setExportingSymbols(boolean b);
8
9     boolean isOrdered();
10    void setOrdered(boolean b);
11
12    void setName(String name);
13    void setNameAbsent();
14    String getName();
15    boolean isPresentName();
16
17    IScope getEnclosingScope();
18
19    int getSymbolsSize();
20
21    void setAstNode(ASTNode node);
22    void setAstNodeAbsent();
23    boolean isPresentAstNode();
24    ASTNode getAstNode();
25
26    void setSpanningSymbol(IScopeSpanningSymbol symbol);
27    void setSpanningSymbolAbsent();
28    boolean isPresentSpanningSymbol();
29    IScopeSpanningSymbol getSpanningSymbol();
30 }

```

Listing 9.13: Signature of all scope classes that implement `IScope`

All scopes have the properties `shadowing` (l. 3), `exportingSymbols` (l. 6), and `ordered` (l. 9) for which the interface provides getters and setters.

Scopes can be configured with three Boolean switches, each leading to different behaviors regarding the visibilities of the symbols contained in the scopes.

isShadowingScope is true when it is a shadowing scope, where local names shadow imported names. E.g. in Java the method body scope is shadowing, and therefore local variables shadow attributes (but not other local variables in the same method).

isOrdered defines, whether symbols can be used within a scope before they are actually defined. This is e.g. the case in Java classes, but not within method scopes.

isExportingSymbols indicates whether the scope exports symbols to the environment or keeps the locally defined elements encapsulated.

By default, scopes do export symbols because, for example, a static field `f` defined in a class (scope) `C` can be accessed through `C.f`. For that purpose, a scope has an optional name in the form of a `String` defined already in the `IScope` interface. Navigating into the scope can be achieved through the use of that `String` as qualifier part (e.g. `C`) of the symbol name `C.f`.

Each scope except the global scope has an enclosing scope. The methods `getEnclosingScope` and `getSubScopes` (see association in Figure 9.4) allow to access the enclosing scope and the subscopes, respectively. The interface `IScope` defines the signature for the method `getEnclosingScope()` (l. 17) with the return type `IScope`. Language-specific scopes implement this method and concretize the return type to language-specific scope types. `IScope` does not contain a method for setting the enclosing scope, as concretization of the argument type for the scope would not be possible in language-specific scopes. Due to similar reasons, the `IScope` interface does not define any method for handling subscopes. Such methods are generated in a language-specific form instead.

For convenience, each scope implements the method `getSymbolsSize()` (l. 19) to obtain the number of all symbols that are contained locally in this scope. Due to the hierarchical structure of scopes, the real number of resolvable symbols usually is larger.

The interface further provides methods for the connection of a scope to an AST node (l. 21) and the connection to a symbol that spans the scope (l. 26). The access and manipulation methods handle both connections as optional because neither an AST node nor a spanning symbol is always present. Unlike the connection from a scope to its enclosing scope, the types for the connections to AST nodes and spanning symbols are not concretized in scope implementations, as both can be of various types.

```

1 public interface IArtifactScope {
2
3     String getPackageName();
4     void setPackageName(String packageName);
5
6     String getFullName();
7 }

```

Java «RTE» IArtifactScope

Listing 9.14: `IArtifactScope` dealing with scopes for full artifacts and thus manages package information

Listing 9.14 shows the MontiCore RTE interface `IArtifactScope` that is implemented by all generated artifact scopes. It provides the method `getPackageName` that returns the name of the package that the artifact described by the scope is located in. Further, it contains the method `getFullName` for obtaining the full name of an artifact scope, which is the name of the artifact scope preceded by a qualifier that indicates the package if it is non-empty.

```

1 public interface IGlobalScope {
2
3     ModelPath getModelPath();
4     void setModelPath(ModelPath modelPath);
5
6     String getFileExt();
7     void setFileExt(String fileExt);
8
9     void addLoadedFile(String name);
10    void clearLoadedFiles();
11    boolean isFileLoaded(String name);
12
13    void init();
14    void clear();
15
16    Map<String, ISymbolDeSer> getSymbolDeSers();
17
18    void setSymbolDeSers(Map<String, ISymbolDeSer> symbolDeSers);
19    void putSymbolDeSer(String key, ISymbolDeSer value);
20
21    ISymbolDeSer getSymbolDeSer(String key);
22
23    IDeSer getDeSer();
24    void setDeSer(IDeSer deSer);
25 }

```

Listing 9.15: `IGlobalScope` describes the interface of the global scope

Listing 9.15 shows the MontiCore RTE interface `IGlobalScope`. It is also implemented by generated, language-specific global scopes.

The interface for global scopes introduces the methods `getModelPath` and `setModelPath` for obtaining and setting the model path, in which symbol tables are looked for, if external symbols need to be loaded.

The global scope provides the methods `init` and `clear` (repeatedly usable) for its initialization and for clearing the global scope. The `clear` method (cf. Section 9.7) resets the global scope and thus unloads all artifact scopes, empties the model path entries, and sets all `DeSers` that have been reconfigured to the initial default configuration.

Global scopes have the two methods `getFileExt` and `setFileExt` to access and modify a regular expression for file extension of symbol table files that the global scope should consider for loading symbol tables. This is explained in more detail in Section 9.7.

Global scopes manage a list of names with candidates for symbol table file names that have been attempted to load during symbol resolution. This prevents loading the same artifact scope more than once. Global scopes do not distinguish successful or failed attempts. Whenever a file is attempted to be loaded, the symbol resolution first checks whether a file with this name has been loaded before by calling the method name `isFileLoaded`. If this is not the case, the file name added to the list with the global scope method `addLoadedFile` and the file is attempted to be loaded. The list can be cleared with the method `clearLoadedFiles`. All three methods for loaded files are used internally by the symbol resolution algorithm and are typically not called by handwritten code. This is explained in more detail in Section 9.7.

Additional methods of the `IGlobalScope` interface deal with loading and storing symbols and are therefore discussed in Section 9.8.

Generated, language-specific artifact and global scopes contain additional methods. These are introduced in generated interfaces and classes described in Section 9.3.3.

9.3.3 Generated Classes For Scopes

The MontiCore RTE scopes described in Section 9.3.2 are the basis for generated, language-specific scope interfaces and scope classes. These language-specific scopes enable realizing type-safe handling of symbols within each scope as well as type-safe connections of a scope to its environment, such as the AST node, enclosing scope, and subscopes.

MontiCore provides exactly one generated scope kind for a language, which is attached to all nonterminals marked with the keyword `scope`. However, the scope kind can be parametrized with different properties (e.g., whether it is ordered or not) per scope instance individually. MontiCore separates generated scope interfaces from generated scope classes to accustom to the potential multiple inheritance between a language and the languages it inherits from during language composition. This is explained in more detail in Section 9.10.

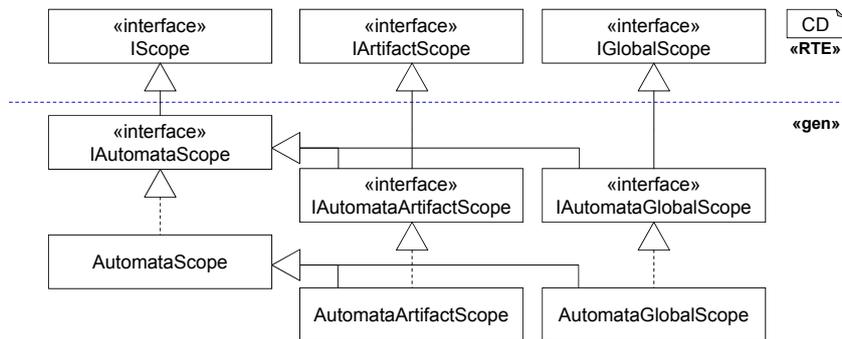


Figure 9.16: Scope classes generated for the Automata language

An overview of the scopes that MontiCore generates for each language is depicted by example of the Automata language in Figure 9.16.

IAutomataScope extends the class `IScope` of the MontiCore RTE and mainly introduces the language-specific method signatures and default implementations for some of these methods.

IAutomataArtifactScope extends both the language-specific scope and the MontiCore RTE `IArtifactScope`. Through inheriting from the language-specific scope interface, the signatures and default implementations of the scope methods can be reused. However, the language-specific artifact scope interface overrides some method default implementations to add behavior specific to artifact scopes. This especially includes handling of artifact names during symbol resolution.

IAutomataGlobalScope reuses some of the language-specific signatures and default implementations from `IAutomataScope`. However, the language-specific global scope interface also overrides some methods to add behavior specific to global scopes. This includes inter-model symbol resolution as well as loading of stored symbol tables.

AutomataScope implements `IAutomataScope` and realizes all attributes and methods for which no default implementation in the scope interface exists.

AutomataArtifactScope again provides the implementation for its interface.

AutomataGlobalScope again provides the implementation for its interface. The global scope class reuses most fields from the scope class but adds fields and methods for handling model path entries, resolvers, `DeSers`, and the file extensions of symbol files. The model path and resolvers are required for realizing inter-model resolution (cf. Section 9.8). `DeSers` and symbol file extensions are required for realizing loading of stored symbol tables (cf. Section 9.7).

As a general remark, the diamond inheritance structure of many of the implementation classes is resolved through adding additional interfaces and providing default implementations in these interfaces. The default implementations are inherited across the inheritance hierarchy. Some methods, however, cannot be provided as defaults and do have to be generated into the implementing classes. Thus, they are not be inherited across the interfaces. This becomes relevant when looking at language composition in Section 9.10.

The scope methods that are generated for realizing the connection of a scope to its environment are explained in more detail in Section 9.4 and the scope methods that realize symbol resolution are explained in Section 9.8.

MontiCore generates scope classes and scope interface for all languages regardless whether these define symbols through the grammar or not. This is a deliberate decision that enables language engineers to use the TOP mechanism, e.g., to add handwritten symbols or adapt resolution processes, for any language. A further advantage of generating scopes for each language is that inheritance relationships for languages can be applied to scopes coherently. This is explained in more detail in Section 9.10. For similar reasons, MontiCore always generates artifact scope and global scope classes and interfaces for all languages.

Symbol management in a scope is relatively simple: Figure 9.17 shows the two methods `add` and `remove` for managing symbols. `getLocalStateSymbols` retrieves all local symbols as a multimap that maps a symbol name to a list of all `StateSymbols` defined

9. Symbol Management Infrastructure

in the scope that have this name. Usually, an entry in the map contains only a single symbol for a given name. In some cases, e.g., in Java method signatures, it is allowed that multiple symbols with the same name co-exist and are thus stored in the same entry of the multimap. These symbols must be distinguishable by other means. Java methods, e.g., are distinguishable by their parameters.

`getStateSymbols` returns a list of all `StateSymbols` defined in the scope. This is a flattened list of all values of the respective symbol map, which is the result of the `getLocalStateSymbols` method. Please note that in the example both `getLocalStateSymbols` and `getStateSymbols` do not respect visibilities and deliver all symbols. Neither of the methods returns symbols defined in subscopes.

For resolving a given symbol name the most relevant methods are discussed in Section 9.8. They respect visibilities and also examine subscopes.

```
1 public interface IAutomataScope {
2
3     // provided for each kind of symbols (here: "State")
4     public Multimap<String, StateSymbol> getStateSymbols() ;
5     public List<StateSymbol> getLocalStateSymbols();
6
7     public void add(StateSymbol);
8     public void remove(StateSymbol symbol);
9
10    // resolving methods are not shown here
11 }
```

Java «gen» IAutomataScope

Listing 9.17: IAutomataScope core functions

`IAutomataArtifactScope` and its implementation `AutomataArtifactScope` mainly concentrate on the storage of the list of imports that the artifact uses. These imports are needed to tell the global scope where to look for foreign symbol tables when their loading is needed.

```
1 public class AutomataGlobalScope {
2
3     public class AutomataGlobalScope extends AutomataScope
4         implements IAutomataGlobalScope {
5
6         public AutomataSymbols2Json getSymbols2Json()
7         public void setSymbols2Json(AutomataSymbols2Json s2j)
8
9         public void loadFileForModelName(String modelName)
10
11        public void loadState(String name)
12
13        public List<IStateSymbolResolver>
14            getAdaptedStateSymbolResolverList()
15        public void setAdaptedStateSymbolResolverList(
16            List<IStateSymbolResolver> r)
```

Java «gen» AutomataGlobalScope

```

17 |
18 | // methods for automaton symbols and implementations
19 | // of methods of IGlobalScope are not shown here
20 |
21 | }

```

Listing 9.18: Method signatures of the AutomataGlobalScope class

Global scopes extend the scope class and implement the global scope interface of the language, as described in Listing 9.18. As the global scope interface inherits from the interface `IGlobalScope`, global scope classes contain the implementation for the methods described in Listing 9.15 as well.

Each global scope has a getter and a setter method for the `Symbols2Json` class of the language. This is used by the method `loadFileForModelName` for loading symbol tables from a given name of a symbol table file. Global scopes further have a loading method, such as `loadState`, for each symbol kind of the language. The method `loadState` uses the method `calculateModelNamesForState` of the language-specific global scope interface to calculate candidates for symbol table files that contain state symbols. Afterwards, it loads the symbol table with the method `loadFileForModelName`. Loading of symbol tables is described in more detail in Section 9.7.

For language composition, global scopes manage a list of symbol resolvers. For each symbol kind of the language, the global scope has an individual attribute that enables resolving for adapted symbols. Symbol resolvers are explained in more detail in Section 9.10.4.

All instances of scopes, artifact scopes, and global scopes must be obtained through the mill (cf. Section 11.5) of a language because this enables language compositionality with black-box reuse of functionality.

A tool may only use a single instance of the global scope (hence, the name “global”). This global scope instance is obtained from the mill with the `globalScope` method. The global scope object has to be handled with care: The global scope is not stateless, it has to be initialized (`init()`) and reset (`reset()`) properly even during unit testing. This clears the loaded artifact scopes of a global scope, model path entries that might have been set, and the `DeSers` that might have been reconfigured.

9.3.4 Defining Scope Attributes and Methods via `scoperule`

Similar to AST rules and symbol rules, scope rules are grammar rules that enable defining extra attributes and methods for the scopes of a language, which results in fields of the generated scope class. The left-hand side of scope rules is only the keyword `scoperule`, i.e., the name of the scope type is omitted as there is only one scope type. Like symbol rules, the body of a scope rule contains attribute and method definitions. Attributes can be marked optional (`'?'`) or as iterations (`'*'`). Using scope rules has the same benefits over using the TOP mechanism on the scope class that are described for symbol rules.

```

1  scoperule = a:ISymbol* b:boolean
2  method public boolean isEmpty() {
3      return this.getSymbolsSize() == 0 ;
4  } ;

```

MCG

Listing 9.19: A scoperule that defines two attributes and a method of the scope

An example for the usage of scope rules is given in Listing 9.19. The scope rule in l. 1 defines two attributes and a method of the language's scope. The first attribute has the name `a` and is of the iterated type `ISymbol`. In the generated scope class, this produces an attribute of type `List<ISymbol>` and list-type access and mutation methods. The second attribute `b` is of the built-in primitive type `boolean` and is translated to an attribute of the same name and type in the scope class as well as a getter and a setter method in the scope interface. The scope rule further defines a method `isEmpty()` that returns `true` if no local symbol is contained in the scope and `false` otherwise. Through the scope rule, the method will be available as Java method in the scope class.

9.4 Collaboration between AST, Symbol, and Scope

Figure 9.20 illustrates the relation between AST nodes, symbols and scopes by example of an excerpt from the automata language introduced in Listing 9.6.

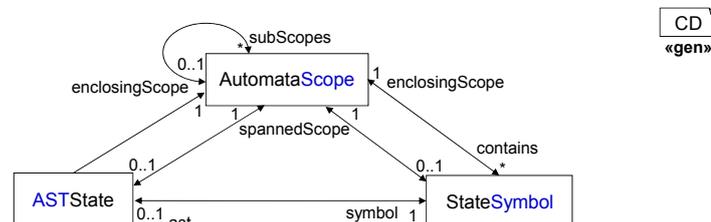
CD
«gen»

Figure 9.20: Relationship between AST, symbol, and scope by example

An AST node and a symbol are bidirectionally linked together if they represent the same model element. For example, a state is represented by the node `ASTState` and the symbol `StateSymbol`. That way, we group the information contained by these two classes and use the information as needed. Not every type of AST node has a corresponding symbol, e.g., the AST for an import statement has no symbol as it does not define a name. Although we usually create the symbols from the AST, this is not necessarily the case.

When an external symbol table is loaded a symbol exists without a corresponding AST node. Thus, the association from `StateSymbol` to `ASTState` has the cardinality `0..1`. The opposite direction has the cardinality `1`, even though an AST node is typically instantiated in a much earlier model processing phase.

Both, AST nodes and symbols, are defined in the enclosing scope. For example, `ASTState` and `StateSymbol` are defined in an `AutomataScope`. Each type of AST node, regardless

whether it defines a symbol or not, has an enclosing scope, which is the scope defining the namespace of the location of the AST node instance.

A scope has a map with symbols for each symbol kind it can potentially contain. While in many languages it is forbidden to have multiple symbols with the same name and kind in a scope, it is intended in some cases in which the symbols can be distinguished based on further information. For instance, the scope of a Java class can contain multiple method symbols with the same name, if these are distinguishable based on their arguments. To realize this, the symbol maps in scopes are realized as multimaps and allow multiple symbols with the same name and kind in a scope.

Scopes are usually arranged in trees. Thus, each scope except the global scope, which is the root of the tree, has one enclosing scope. MontiCore realizes this with association with the cardinality `0..1`. For efficient symbol resolution, the reverse direction of the association exists, too and each scope has a collection of `subScopes`.

Some symbol kinds also span scopes. This can be indicated in the grammar if the respective nonterminal has both keywords `symbol` and `scope` and is reflected in the abstract syntax. For example, hierarchical states define a symbol and span a scope. Therefore, the `StateSymbol` has an association with the `AutomataScope`. Each state symbol spans exactly one scope. However, as scope kinds are defined per language, not all scopes are spanned by a state symbol. In fact, there may be other nonterminals in the automata language that span scopes. To this end, the type of attribute `spannedSymbol` in the `AutomataScope` is `ISymbol` and not `StateSymbol`. If a nonterminal spans a scope, the respective AST node is in a bidirectional association with the scope. In the example, the `ASTState` has an association with the `AutomataScope`. Similar to the association between symbol and spanned scopes, the reverse direction of the association is less strictly typed. This is due to the fact that a scope can be spanned by different nonterminals, therefore the concrete type of the AST node cannot be specified in the scope.

The associations shown in Figure 9.20 are realized through Java attributes with the usual access and manipulation methods specific to their cardinality. The signature of access and manipulation methods in the scope is also contained in the scope interface.

9.5 Using Symbols

When a model element shall be used at another place in the model, then it is *used* by its name. For example, in our Automata language `State` names are used in transitions. Actually in complex situations additional information may be necessary. Method overloading, e.g., can only be resolved, because method name and argument types are available.

In a grammar the nonterminal `Name` can be extended by a suffix when it is used on the right-hand side of a grammar rule. This suffix explains to which kind of symbol the name refers to. Listing 9.6 repeats an excerpt of Listing 9.6 (p. 161), lines 12ff, where a transition is equipped with a `source State` and a `target State`.

`source:Name@State` instead of only `source:Name` leads to the generation of additional methods to retrieve a symbol which a name is referring to in Listing 9.23. Internally, the



Tip 9.21: Navigation from Symbol Usage to Symbol Definition

There are several ways to establish navigation from symbol usage to definition of a symbol. When this navigation is taken sporadically or the condensed information in the symbol is already sufficient, it is usually sufficient to calculate the symbol table infrastructure and then do all navigation through its lookup.

However, when navigation will happen often, it is efficient to establish a direct link in the form of an additional attribute in the using AST node linking to the defining AST node. After the symbol table infrastructure has been used to establish the links once, it may be bypassed through that direct link.

This is for example quite helpful, when interpreting an automaton, where the transition node should be directly linked to the node of the target state and the source states should be directly linked to the outgoing transitions (via a map that takes the input into consideration).

Additional attributes can alternatively be added to a generated AST via the grammar (see Section 5.4) or in the form of handwritten Java code (see Chapter 14).

```

1 Transition =
2   source:Name@State "-" input:Name ("|" Statement)? ">"
3   target:Name@State ";" ;

```

Listing 9.22: Repeated excerpt of Automata grammar

method implementation relies on symbol resolution that is explained in more detail in Section 9.8. Based on the name symbol and the desired kind (in the example, `StateSymbol`), it is resolved by the enclosing scope of the AST node. Once this has been resolved successfully, the symbol is cached and subsequent requests do not have to execute resolution for the same symbol again.

```

1 public class ASTTransition ... {
2   // additionally generated methods, because of
3   // Transition = source:Name@State ... target:Name@State
4
5   // Retrieving the symbol:
6   public StateSymbol getSourceSymbol();
7   public boolean isPresentSourceSymbol();
8   public StateSymbol getTargetSymbol();
9   public boolean isPresentTargetSymbol();
10  // Directly navigate to the definition of the symbol in the AST
11  // (if the defining model is loaded):
12  public ASTState getSourceDefinition();
13  public boolean isPresentSourceDefinition();
14  public ASTState getTargetDefinition();
15  public boolean isPresentTargetDefinition();
16 }

```

Listing 9.23: Extended signature of ASTTransitions

As a convenient shortcut, it is also possible to directly retrieve the AST node (in the example, of type `ASTState`), where the mentioned symbol is defined, in case the defining model is loaded. In our `Automata` example, states are defined within the same model and, therefore, navigation is always successful once the symbol table has been created from the AST node.

In general, `Name@K` is allowed for any kind of symbols `K` known in a grammar. It is also possible to apply this kind of reference to optional names, such as `g:Name@A?` or withing alternatives, such as `g:Name@A | h:Name@A`, but not to lists. I.e, `g:Name@A*` is forbidden. The same names can also not hint toward different kinds of symbols. I.e. `g:Name@A | g:Name@C` would be forbidden because of the same "g" for both occurrences.

9.6 Instantiating Symbol Tables

For the actual instantiation of the symbols as well as the establishment of all links described above additional code that can be executed after the parsing process is generated. Furthermore, if names are used in a model, where the corresponding referenced symbols neither exist within the model nor within imported symbol tables, executable context condition checkers with appropriate error messages are needed.

Because of the wide variety of possibilities of how to define and look up symbols, it will at least for complex cases be necessary to add handwritten extensions to realize proper symbol table instantiation. However, in straightforward cases, such as flat scopes, or hierarchical scopes with standard visibility, the generated scope classes can be used directly. For detailed discussions, see [MSN17].

As the symbol table instantiation for the typed symbol table infrastructure largely depends on the symbol and scope types of the language, all parts of the symbol table instantiation are generated.

Conceptually, the symbol table is instantiated with the basis of the AST and therefore, has to be performed after a model has been parsed. The instantiation of symbol tables is itself separated into several phases.

9.6.1 Phase 1: Symbols and Scope Skeletons

In the first phase, the AST is traversed by a language-specific *scope genitor* that is realized as a Java class, e.g. `AutomataScopesGenitor` implementing a language visitor. The scope genitor creates the *skeleton* of the scope tree for the model and instantiates all its symbols. Additionally, it connects the scope and symbol objects with their AST nodes as described in Section 9.4. The scope genitor *does not initialize* any additional attributes of the scope and symbol objects it created. These attributes have to be initialized manually.

The reason for this separation of skeleton creation and attribute instantiation is that symbol attributes can require symbols from other parts of the symbol table, which could be created after the first phase. This is, for example, the case when a variable has to refer to its state.

```

1                                     Java «gen» AutomataScopesGenitor
2 public class AutomataScopesGenitor implements AutomataVisitor2,
3                                             AutomataHandler {
4
5     protected Deque<IAutomataScope> scopeStack = new ArrayDeque<>();
6     protected AutomataTraverser traverser;
7
8     // creates the scope structure as skeleton
9     // and adds all symbols
10    public IAutomataArtifactScope createFromAST(ASTAutomaton node);
11
12    // predefined visit / endVisit methods for all
13    // nonterminals (for scopes and symbols)
14    public void visit(ASTAutomaton node);
15    public void endVisit(ASTAutomaton node);
16
17    public void visit(ASTState node);
18    public void endVisit(ASTState node);
19
20    public void visit(ASTTransition node);
21    public void endVisit(ASTTransition node);
22
23    public AutomataTraverser getTraverser();
24    public void setTraverser(AutomataTraverser traverser);
25
26    // when a new scope is needed:
27    public IAutomataScope createScope(boolean shadowing);
28
29    public void putOnStack(IAutomataScope scope);
30
31    // hook points to set scope attributes
32    protected void initScopeHP1(IAutomataScope scope);
33    protected void initScopeHP2(IAutomataScope scope);
34
35    // hook points to set scope attributes in the artifact scope
36    protected void initArtifactScopeHP1(IAutomataArtifactScope scope);
37    protected void initArtifactScopeHP2(IAutomataArtifactScope scope);
38
39    // hook points to set symbol attributes in the StateSymbol
40    protected void initStateHP1(StateSymbol s);
41    protected void initStateHP2(StateSymbol s);
42
43 }

```

Listing 9.24: Methods of the AutomataScopesGenitor

Listing 9.24 shows the signature of a scope genitor using the AutomataScopesGenitor as an example. The class implements the visitor and handler interfaces. The creation of the symbols and scopes is done in the visit methods. Many of the generated methods, especially the visit and the endVisit methods, are completely realized. The methods containing HP in their names are explicitly dedicated as hook points for manual adaptations.

As Listing 9.24 shows, a generated scope genitor has a method `createFromAST` enabling to instantiate a symbol table for an AST. This method creates all scopes and symbols and links them accordingly and with the AST nodes as described above. It also returns the top level scope as result.

The access methods for the `traverser` are prescribed by the handler interface and are used by the `createFromAST` method to start the traversal.

Artifact scopes have prepared `package` and `import` attributes that are well suited to be used in combination with the nonterminals `MCPackageDeclaration` and `MCImportStatement` from the grammar `MCBasicTypes`. In certain cases, the `createFromAST` method can set the package and import of an artifact scope automatically. In other cases, however, the developer has to care manually because the package and the imports are used for resolving symbols (cf. Section 9.8).

The genitor manages the stack of scopes `scopeStack` corresponding to the scope hierarchy in which it currently acts.

During the traversal of each AST node, the *scope on top of the scope stack* is set as the enclosing scope of symbols and AST nodes. Whenever the AST traversal encounters a nonterminal that spans a scope, a new scope instance is obtained from the language mill and put on top of the scope in the implementation of the `visit` method for the AST node of the scope-spanning nonterminal. In the respective `endVisit` method, the scope is removed from the stack. In a genitor, the `visit` and `endVisit` methods are not meant to be overridden, but can be adapted using the hook points described below.

In case a nonterminal defines a *symbol*, a symbol object is instantiated via the mill in the `visit` method of the nonterminal. The symbol is also linked with the enclosing scope.

In case a nonterminal spans a *scope*, the created scope instance is set as a subscope of the current scope on the top of the scope stack.

If a nonterminal both *defines a symbol and spans a scope*, the created symbol is added to the enclosing scope before the new subscope is created. That means a symbol that spans a new subscope does itself not belong to this subscope but to the parent.

The predefined `createScope` and `putOnStack` methods create a scope respectively put a scope on the stack. The methods are used within the `visit` methods. Even though they are public, they are not directly intended for use by the language developer.

Furthermore, there are *hook points* to further initialize scopes and symbols. By default, the implementation of these methods are empty. `initScopeHP1` is called as the last step of the scope creating `visit` method. `initScopeHP2` is called even later in the corresponding `endVisit` method, where the skeletons of subsopes and also the symbols have been created. The methods `initArtifactScopeHP1` and `initArtifactScopeHP2` in analogy allow to initialize the created artifact scope. `initArtifactScopeHP1` is called before the traversal of the AST and `initArtifactScopeHP2` after the traversal.

For each symbol kind `Sym` there are also the two methods `initSymHP1` and `initSymHP2` which serve as hook point. Again, method `initSymHP1` is called at the end of the `visit` method in which a symbol was created. The `initSymHP2` method is called in the corresponding `endVisit` method.

Please note that a full initialization of the scope and the symbols may not be feasible with these hook points because, for example, typing information may not yet be completed and thus several relevant symbols may not yet be available. This is why the first pass only defines the skeleton and some attributes of symbols and scopes are to be added in later phases. However, the hook points allow to initialize some attribute already along with the skeleton creation in the first phase.

9.6.2 Phase 2+: Filling Symbols with Value

After the first phase, symbols only carry their name and are appropriately linked with the AST and scopes by default, but any other attribute still needs to be filled.

In a second phase, which may also be realized in a *visitor traversal*, the above mentioned additional symbol and scope attributes are filled. Section 9.9 discusses how the generated visitor infrastructure navigates over AST, symbol, and scope objects to simplify this.

Because the nature of such attribute initializations largely depends on the symbol or scope attributes that are initialized, MontiCore does not generate further infrastructure. Instead, it is recommended to define one or more visitors that run in parallel, i.e. in a single phase, in a composed visitor, or that are executed subsequently in several phases.

In complex languages, such as Java, it may be that more than two phases are needed. This is typically the case when generic type systems are used (e.g. Java) or type classes are allowed (e.g. Haskell).

9.7 Loading and Storing Symbol Tables

Symbol tables can be persisted and loaded again to improve the language infrastructure for several purposes:

- As the scopes and symbols of a model contain the *information required for type checking*, only symbol tables are needed to check the consistency of symbol uses, type consistency and similar context conditions between models.
- Symbol tables contain the externally visible essence of a model only and they are persisted in a form that is well accessible. Thus, loading a stored symbol table is more *efficient* than parsing the model, creating the symbol table from the AST of the model, and then using this symbol table instead.
- The symbol table is a surrogate for the actual model. In model-driven development, models are the central development artifacts. For *integrating models with other models* of foreign stakeholders, it can therefore be sufficient to communicate the symbol table of the models only whereas the actual models remain private.
- Storing symbol tables fosters *efficient language aggregation*. With persisted symbol tables, novel forms of adapters can be realized between the persisted representations of symbol tables.

9.7.1 Stored Symbol Tables

Serialization is the process of translating an object structure into a character sequence and *deserialization* is the opposite process that translates a characters sequence into an object structure. To realize this, serialized objects are encoded in a *serialization format* such as JSON, XML, or other textual forms.

MontiCore symbol tables are serialized in JSON as it is a commonly used, relatively compact, yet human readable notation for which efficient and sophisticated tool support exists. The infrastructure for realizing serialization and deserialization of an object structure depends on the type of the objects. In Monticore, the serialization strategy for a type is contained in a `DeSer` class for this type.

Figure 9.25 shows an excerpt of the symbol table, which is generated to `PingPong.autsym` and comes from the well-known automaton `PingPong` e.g. defined in Section 21.1.



```

1 {
2   "generated-using": "www.MontiCore.de technology",
3   "name": "PingPong",
4   "symbols": [
5     {
6       "kind": "automata._symboltable.AutomatonSymbol",
7       "name": "PingPong",
8       "spannedScope": {
9         "symbols": [
10          {
11            "kind": "automata._symboltable.StateSymbol",
12            "name": "NoGame"
13          },
14          {
15            "kind": "automata._symboltable.StateSymbol",
16            "name": "Ping"
17          },
18          {
19            "kind": "automata._symboltable.StateSymbol",
20            "name": "Pong"
21          }
22        ]
23      }
24    }
25  ]
26 }

```

Listing 9.25: Content of an Example Symbol Table `PingPong.autsym`

The top element represents the artifact scope of the model. Line 3 contains the name of the model, which is also used to qualify the symbols if needed. From the artifact scope, all locally defined symbols are serialized via containment, i.e., in attribute `"symbols"` of the artifact scope JSON object. Symbol `"PingPong"` in l. 7 describes the automaton, which

in this case (and often) is identical to the model name. The other symbols, e.g. "Ping" in l. 16, describe individual states.

Each symbol and scope object is defined by its "kind", which is actually a fully qualified Java class. A symbol always contains its "name".

If a symbol spans a scope, the spanned scope is serialized as a JSON object contained in the symbol. However, the spanned scope is only serialized if it exports at least one symbol beyond the artifact scope.

MontiCore symbol tables are complex data structures encoding e.g. bidirectional associations between scopes and subscopes, symbols and enclosing scopes, and symbols and spanned scopes (cf. Section 9.4). For the serialization, only one direction of the association is stored through containment to avoid redundancy and loops.

The opposite direction has to be re-established during the deserialization. Monticore's generated classes do that by default, but manual adaptations have to bear this in mind.

For reasons of compactness, stored symbol tables are serialized without any unnecessary *whitespace* by default. Further, Monticore has built-in strategies for *reducing redundant information* by omitting the serialization of *default values* for some data types. Thus, Monticore omits the serialization of object members if they have values as follows:

- List, if it is empty
- Optional, if its value is absent
- Boolean, if its value is false
- String, if its value is the empty string
- int, etc.: numeric values, if it is equal to 0 (or 0L, 0.0f, etc.).
- scopes, if they do not contain any local symbols or if they do not export symbols beyond the artifact scope.

To assure symmetry, the deserialization of built-in data types reconstruct absent values with respect to the defaults accordingly.

9.7.2 RTE Classes For Symbol Table Persistence

Again, the Monticore RTE and the generated classes work together to support the loading and storing of symbol tables. The RTE classes include a common interface for all Desers of scopes, a common interface for all Desers of symbols, and the infrastructure to parse and print JSON. The classes of the JSON infrastructure are depicted in Figure 9.26 and explained in the following.

IDeSer is the interface that all generated DeSer classes (cf. Section 9.7.3) for scopes implement. The global scope manages the scope DeSer of the language to enable symbol Desers to use it for deserializing scopes spanned by symbols. The interface is shown in Listing 9.27.

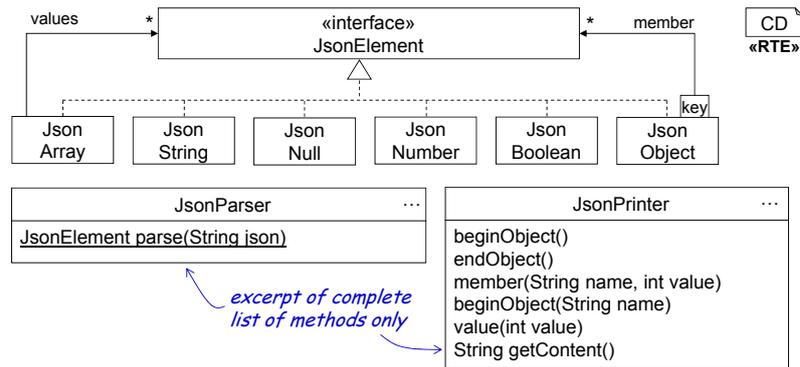


Figure 9.26: Infrastructure for parsing and printing JSON

ISymbolDeSer is the interface that all generated `DeSer` classes (cf. Section 9.7.3) for symbols implement. All implementing `DeSers` are added as configuration to the global scope for loading symbols and used directly for their storage. The interface is shown in Listing 9.28.

JsonDeSers is a class containing a collection of methods supporting the realization of concrete `DeSers`. The class further provides constant `Strings` for commonly used member names in serialized symbol tables.

JsonPrinter is a class that contains an API for printing JSON elements. It contains methods for printing individual JSON numeric, boolean, or `String` values as well as methods for printing JSON objects and arrays. For objects and arrays, the printer contains methods indicating the begin and the end of an object or array as well as methods for printing array values or object members. For optimal results of nested JSON structures, all `DeSers` should use the same instance of the `JsonPrinter`.

JsonParser is a class realizing a parser for JSON that creates instances of the abstract syntax classes implementing the `JsonElement` interface. The parser is used through the static method `parse` that accepts a JSON-encoded `String` passed as a method argument and returns an instance of the interface `JsonElement`.

JsonElement is the common interface for all classes of the abstract syntax of JSON. This interface is required to form a common super type for values of a JSON array or members of a JSON object. The interface contains methods to check the actual type of an abstract syntax element. This relieves language engineers from using `instanceof` checks and down casts.

JsonObject realizes the abstract syntax of a JSON object. For the deserialization, this class offers a variety of methods for accessing the members of the serialized object.

JsonArray realizes the abstract syntax of a JSON array. The method `getValues` returns a list of `JsonElements` contained in the array. It can be used for traversal in a `for-loop` to handle the individual elements of the array one after another.

JsonString realizes the abstract syntax of a JSON `String`. The value of the `String` can be obtained through the method `getValue`.

JsonNumber realizes the abstract syntax of a JSON number. The value of the number can be obtained as different numeric Java types with individual methods, such as, e.g., through the methods `getNumberAsInt` or `getNumberAsDouble`.

JsonBoolean realizes the abstract syntax of a JSON Boolean. The value of the Boolean can be obtained through the method `getValue`.

```

1                                     Java «RTE» IDeSer
2 public interface IDeSer<S extends IScope,
3     A extends IArtifactScope,
4     J> {
5
6     String serialize(A toSerialize, J symbol2json);
7     String serialize(S toSerialize, J symbol2json);
8
9     // Hookpoints to serialize additional attributes
10    default void serializeAddons(A toSerialize, J symbol2json) {}
11    default void serializeAddons(S toSerialize, J symbol2json) {}
12
13    default A deserialize(String serialized) { ...
14
15    A deserializeArtifactScope(JsonObject scopeJson);
16    S deserializeScope(JsonObject scopeJson);
17
18    // Hookpoints to deserialize additional attributes
19    default void deserializeAddons(A artifactScope,
20        JsonObject scopeJson) { ...
21    default void deserializeAddons(S scope,
22        JsonObject scopeJson) { ...
23 }

```

Listing 9.27: Signature of the IDeSer interface

The IDeSer interface shown in Listing 9.27 is generic with the types of the scope and the artifact scope that the DeSer is able to (de)serialize and the type of the Symbols2Json class of the language. The latter is required as an argument for the `serialize` method.

IDeSer defines the methods `serialize` and `deserialize` that realize the (de)serialization strategy for either the artifact scope type or the scope type that the DeSer handles. The argument object of the `serialize` method is serialized and returned as `String`. The second argument is an instance of the language-specific Symbols2Json class to traverse the symbol table that is being serialized. This argument is required as a symbol that spans a scope contains the serialized spanned scope as an object member.

Methods to handle additional attributes (addons) are explained below.

```

1                                     Java «RTE» ISymbolDeSer
2 public interface ISymbolDeSer<S extends ISymbol, J> {
3
4     String serialize (S toSerialize, J symbol2json);

```

```

5 |
6 | default S deserialize (String serialized) {
7 |     JsonObject symbol = JsonParser.parseJsonObject(serialized);
8 |     return deserialize(symbol);
9 | }
10 |
11 | S deserialize (JsonObject serialized);
12 |
13 | String getSerializedKind();
14 | }

```

Listing 9.28: Signature of the ISymbolDeSer interface

The ISymbolDeSer interface shown in Listing 9.28 is generic with the class of the symbol kind that the DeSer is able to (de)serialize and the type of the Symbols2Json class of the language. The latter, again, is required as an argument for the `serialize` method.

ISymbolDeSer defines the methods `serialize` and `deserialize` that realize the (de)serialization strategy for the symbol class that the DeSer handles. The argument object of the `serialize` method is serialized and returned as a `String`. The second argument is an instance of the language-specific Symbols2Json class to traverse the symbol table that is being serialized. This argument is required as a symbol that spans a scope contains the serialized spanned scope as an object member. A further method returns the `String` indicating the symbol kind that the DeSer handles.

9.7.3 Generated Classes for Symbol Storage and Their Adaptation

For each language, MontiCore generates a DeSer class that is capable of serializing and deserializing scopes and artifact scopes of the language. Furthermore, MontiCore generates DeSer classes for each symbol kind that is defined in the language.

This also includes the additional attributes introduced via a `symbolrule` or a `scoperule` with a distinction of several cases:

- Basic built-in types, such as `int`, `boolean`, and also `String` are directly translated into a primitive type in `Json`. The (de-)serialization of an attribute `A` is realized in the `serializeA` and `deserializeA` methods of the DeSers.
- `Optional` or `List` types with built-in typed arguments are also handled fully automatic.
- For object types and e.g. generic types, MontiCore does not generate a complete serialization strategy, but instead, produces abstract `serialize` and `deserialize` methods for the attribute. The presence of abstract methods in a DeSer enforces that the DeSer class itself is generated as an abstract class.

To support serialization, the symbol tables of a language have to be traversed in a dedicated order. The language-specific Symbols2Json class implements a language's visitor interface and realizes this traversal.

AutomataDeSer is the DeSer that serializes artifact scopes and scopes of the Automata language. The class contains methods for both the scope interface and the artifact scope interface. The methods of the DeSer are depicted in Listing 9.29.

StateSymbolDeSer is one of the DeSer classes serializing symbols, here for StateSymbols. The class has `serialize` and `deserialize` methods similar to scope DeSers. Further, the hook methods `serializeAddons` and `deserializeAddons` are available in symbol DeSers in the same way they are for the DeSers of scopes.

AutomataSymbols2Json is the interaction point for language engineers with the serialization infrastructure as it provides `load` and `store` methods. While the loading of symbol tables is realized as part of the symbol resolution algorithm, the storing of symbol tables has to be performed by the language tool.

Internally the `AutomataSymbols2Json` realizes an Automata visitor. The serialization of scopes and symbols is delegated to the respective DeSers. To enable the reconfiguration of DeSers in the global scope, the `Symbols2Json` classes obtain the DeSers through the DeSer map in the global scope. Usually, it is not necessary to extend this class with handwritten code. Instead, the scope and symbol DeSers should be customized.

```

1 public class AutomataDeSer implements Java «gen» AutomataDeSer
2     IDeSer<IAutomataScope,
3         IAutomataArtifactScope,
4         AutomataSymbols2Json> {
5     public String serialize(IAutomataScope s)
6     public String serialize(IAutomataScope s, AutomataSymbols2Json s2j)
7     public void serializeAddons(IAutomataScope s,
8         AutomataSymbols2Json s2j)
9
10    public String serialize(IAutomataArtifactScope s)
11    public String serialize(IAutomataArtifactScope s,
12        AutomataSymbols2Json s2j)
13    public void serializeAddons(IAutomataArtifactScope s,
14        AutomataSymbols2Json s2j)
15
16    public IAutomataArtifactScope deserialize(String s)
17    public IAutomataArtifactScope deserializeArtifactScope(
18        JsonObject j)
19    public void deserializeAddons(IAutomataArtifactScope s,
20        JsonObject j)
21
22    public IAutomataScope deserializeScope(JsonObject j)
23    public void deserializeAddons(IAutomataScope s, JsonObject j)
24 }

```

Listing 9.29: AutomataDeSer for scopes and artifact scope of the Automata language

Listing 9.29 shows the signature of the generated `AutomataDeSer` class. It provides the following methods:

serialize methods serialize an (artifact) scope object passed as the method argument and return the serialized String. There are further serialize methods with an additional Symbols2Json class as argument, which perform the actual serialization. The Symbols2Json class is employed for traversing the contained symbol and sub-scope structure of the scope.

serializeAddons methods are *hook points* for the serialization of additional attributes of the scope (or artifact scope). By default, the methods have an empty implementation but can be overridden if the DeSer is customized with the TOP mechanism. These methods are invoked after the serialization of all other attributes of the (artifact) scope, i.e., after the contained symbols are serialized.

deserialize methods deserialize a serialized (artifact) scope to an object of the respective type. There is only a single method with the actual name deserialize that returns a deserialized artifact scope. To avoid clashes of return types, there are the two methods deserializeScope and deserializeArtifactScope for deserializing both types individually. To enable delegation of the deserialization of a Json object from the Symbols2Json class to the respective DeSer, the latter two methods have a JSON object as argument.

deserializeAddons methods are the counterparts for the serializeAddons hook points and should be handled in parallel. The body, which is empty by default, can be overridden. The methods are invoked after the deserialization of all attributes of the (artifact) scope, i.e., after the contained symbols are deserialized.

```

1                                     Java «gen» StateSymbolDeSer
2 public class StateSymbolDeSer implements ISymbolDeSer<StateSymbol,
3                                     AutomataSymbols2Json> {
4 public String getSerializedKind()
5
6 public String serialize(StateSymbol s, AutomataSymbols2Json s2j)
7 protected void serializeAdjacentStates(List<String> adjacentStates,
8                                     AutomataSymbols2Json s2j)
9 protected void serializeAddons(StateSymbol s,
10                                AutomataSymbols2Json s2j)
11
12 public StateSymbol deserialize(JsonObject j)
13 protected List<String> deserializeAdjacentStates(JsonObject j)
14 protected void deserializeAddons(StateSymbol s, j symbolJson)
15 }

```

Listing 9.30: StateSymbolDeSer for State symbols with addon attribute adjacentStates of type List<String>

Listing 9.30 shows the signature of the class StateSymbolDeSer by the example of State symbols and under the assumption that a symbol rule was defined that adds the additional attribute adjacentStates of type List<String>. The other methods have similar purposes as in scope DeSers.

With attributes defined in scope rules (cf. Section 9.3.4) and symbol rules (cf. Section 9.2.3) of a grammar, symbols and scopes receive additional attributes that have to be considered for the serialization and deserialization. For each such attribute, the generated DeSer of the scope or symbol has two additional methods, as can be seen with attribute `adjacentStates` above.

This attribute leads to the methods `serializeAdjacentStates` and `deserializeAdjacentStates`. The first method is invoked by the `serialize` method of the symbol to serialize instances of the attribute. The second method `deserializeAdjacentStates` deserializes a `JsonObject` of the `StateSymbol` that is passed as the method argument.

MontiCore generates implementations for attribute serialization and deserialization only in case it has a built-in strategy for the attribute's type. Otherwise, the methods are generated as abstract methods and, thus, the DeSer classes are also generated as abstract classes, which need to be handled using the TOP mechanism.

If and only if a DeSer or its a TOP-extension are instantiatable, it is automatically added to the map of DeSers in the global scope, where the DeSers reside to load symbols.

The control flow for a serialization follows the typical visitor pattern, where the `visit/endVisit` methods start and end the serialization of a scope or symbol object, while the standard `traverse` algorithm recursively includes e.g. symbols in a scope or also nested scopes, if desired. Internally, the result is built by a `JsonPrinter` stored in an `Symbols2Json` visitor.

9.7.4 Loading Symbol Tables

If models may use symbols of other models, then the loading of symbol tables becomes part of the model processing pipeline for the language. Loading and storing symbol tables is realized within the `load` and `store` methods of the generated `Symbols2Json` class.

Loading a symbol table begins with loading the content of a file as a `String`, followed by deserializing the `Json`-encoded `String` containing the stored artifact scope by using the global scope's DeSers responsible for each serialized type.

The loading usually must not be used by the language engineer directly, but is embedded in the global scope, when looking up a symbol. Loading of symbol tables is integrated into the generated inter-model symbol resolution algorithm (cf. Section 9.8) that is realized in the global scope.

A model path in the global scope describes a set of directories that contain stored symbol tables. Package names allow to look deeper into subdirectories.

Therefore, to find stored symbol tables, the directories that contain symbol files have to be added to the model path in the global scope. In appropriate tools this is often done using an external parameter, e.g. `-modelPath` or `-mp` in command line interfaces.

Other than that, no manual effort is required to load symbol tables. However, loading can be adjusted by applying the TOP mechanism to the global scope interface. Typical spots for manual adjustments are:

- The global scope interface method `calculateModelNamesForC` that takes a qualified symbol name of symbol kind `C` as argument and calculates a set of candidates for model names that contain this symbol.

MontiCore's default implementation only considers symbols that are directly contained in an artifact scope and does not look inside nested subsopes. If symbols of nested scopes should be considered in inter-model resolution as well, this method must be overridden.

- The global scope class method `loadC` iterates over the candidates for model names of symbols with the kind `C` calculated by the method `calculateModelNamesForX`, calculates symbol file names, and uses the `Symbols2Json` class to load the correct artifact scope.

This method should be overridden, e.g. if not only the symbol table, but as an alternative the original model should be loaded. Loading a model is useful if the AST of a loaded symbol table should be available, but otherwise this is not recommended.

- The global scope map contains a `DeSer` for each type that could be serialized. It can be modified to use different `DeSers` for a specific type, e.g. for a specific symbol kind.

9.7.5 Storing Symbol Tables

When *storing a symbol table*, the symbol table is traversed by the visitor-based `Symbols2Json` class, which serializes each type with the respective `DeSers` managed by the global scope. The resulting serialized `String` is stored into a file.

Storing a symbol table, however, requires some explicit decision by the language engineer regarding where it has to be integrated into the pipeline of processing models.

The locally built symbol table can be stored after the initial model processing is finished, and all potential additional symbol attributes are calculated. We recommend to only store a symbol table, if the model is checked to be well-formed.

To store a symbol table, the language tool requires an instance of the generated language-specific `Symbols2Json` class. The `store` method of this class takes two arguments, namely the artifact scope to be stored and the name of the file in which the symbol table is stored in. The file has to be stated as a `String` and must include the full path name. This path name usually includes an absolute or relative path to a general symbol table location in combination with additional folders that resemble a package structure, a model name and a standardized file extension (if a Java like approach is used).

To enable an efficient identification of symbol tables with the built-in loading functionality, it is viable that the naming guidelines for files containing stored symbol tables are followed:

- Symbols are stored in a clearly identified *symbol output folder*. As usual for generated files, it should be located in the `target` folder of a project.
- If a language supports *package declarations*, the package is translated to a folder structure as in Java. For example, if a model `M` is contained in the package `a.b.c`,

its qualified name is `a.b.c.M` and the folder structure relative to the symbol output folder should be `a/b/c`.

- The *name of the file* `M` that contains the symbol table of a model must match the name of the artifact scope. Usually, the name of the artifact scope also matches the name of the model (similar to Java).
- We suggest the convention to use `sym` as the suffix of any file extension of any file storing a symbol table.

If a distinction between symbol table files of different languages is necessary, it may be useful to use language-specific file endings for symbol table files, e.g. `autsym` for automata and `sym` for other models. In the example of the automata language, the symbol table of a model `PingPong.aut` located in the package `game` could be stored to the file `target/game/PingPong.autsym`. However, this has the disadvantage that the importing language tool needs to be aware of different exporting languages, which is not necessarily the case, if the exporting models are from different languages, but they offer the same kinds of symbols and thus allow the importing tool to be agnostic regarding the source of the symbols. For that specific reason, we also do not store the kind of the artifact scope in the symbol table.

9.7.6 Realizing Custom Serialization Strategies

Sometimes, the default serialization strategy generated by MontiCore is not satisfactory for language engineers. The reasons for this can be manifold, but include that MontiCore cannot generate a serialization strategy for a symbol rule attribute, that an additional symbol attribute is to be serialized, or that language engineers intend to omit that some symbols are exported in stored symbol tables.

Customization of the serialization strategy can be handled either by extending the deserialization to accept more variants of symbols or as joint adaption of the serialization and deserialization. An example for realizing a custom serialization strategy for a data type for which no built-in serialization strategy is available is explained in Section 21.6. The following describes two further exemplary scenarios for modifications.

Adapting `AutomatonSymbolDeSer`

In the scenario, we customize the generated default serialization for symbol tables of the Automata language. Listing 9.25 on pg. 183 shows an example, where artifact scope contains a the automaton symbol, which has an attribute of the serialized spanned scope, which again contains serialized state symbols.

To realize a more compact form of serialization, the language engineer intends to serialize state symbols as a list of state names that are an attribute of a serialized automaton symbol rather than serializing the scope spanned by an automaton. This should not have any effects on the data structure of symbol table classes of the automata language, but results in a file similar to Listing 9.31.

```

1 {
2   "generated-using": "www.MontiCore.de technology",
3   "name": "PingPong",
4   "symbols": [
5     {
6       "kind": "automata._symboltable.AutomatonSymbol",
7       "name": "PingPong",
8       "spannedScope": {
9         "isShadowingScope": false
10      },
11      "states": [
12        "NoGame",
13        "Ping",
14        "Pong"
15      ]
16    }
17  ]
18 }

```

Listing 9.31: Optimized symbol table with state list only

This is feasible, if no additional symbol information is needed. In the example, we extend the generated AutomatonSymbolDeSer with the TOP mechanism (cf. Listing 9.32).

```

1 public class AutomatonSymbolDeSer
2     extends AutomatonSymbolDeSerTOP {
3   // store states as array of names
4   protected void serializeAddons (AutomatonSymbol a,
5     AutomataSymbols2Json s2j) {
6     JsonPrinter p = s2j.getJsonPrinter();
7     p.beginArray("states");
8     for (StateSymbol s : a.getSpannedScope().getLocalStateSymbols()) {
9       p.value(s.getName());
10    }
11    p.endArray();
12  }
13
14  // load states from such an array and instantiate symbols
15  protected void deserializeAddons (AutomatonSymbol a, JsonObject j) {
16    IAutomataScope s = a.getSpannedScope();
17    for (JsonElement e : j.getArrayMember("states")) {
18      String name = e.getAsJsonString().getValue();
19      StateSymbol state = AutomataMill.stateSymbolBuilder().
20        setName(name).build();
21      s.add(state);
22    }
23  }
24 }

```

Listing 9.32: Customization of the AutomatonSymbolDeSer

The `serializeAddons` method uses the `JsonPrinter` of the `AutomataSymbols2Json` to produce the JSON syntax. The additional serialization must only serialize one or more new JSON object members. In the example, a new member with name "states" of the JSON array type is added using `beginArray` and `endArray`. Inside the array all state symbols contained in the scope of the automaton symbol are printed.

During the *deserialization*, additionally serialized symbol or scope attributes are usually added to the symbol or scope object passed to the `deserializeAddons` method as argument. In this example, however, the loaded state names should not be added to the automaton symbol object. Instead, they define new symbols that are added to the automaton scope. As the method for deserializing addons is invoked after all other parts, the spanned scope object has already been set. The iteration creates symbol objects using the mill and adds the new symbols to the spanned scope.

Adapting `AutomataSymbols2Json`

As a result of the above modification, state symbols are serialized twice, in the state list and as symbols. To overcome this, we adapt the generated `AutomataSymbols2Json` class with the TOP mechanism as depicted in Listing 9.33.

```
1 public class AutomataSymbols2Json Java «hw» AutomataSymbols2Json
2                                     extends AutomataSymbols2JsonTOP
3                                     implements AutomataHandler {
4
5     public AutomataSymbols2Json() {
6         super();
7         getTraverser().setAutomataHandler(this);
8     }
9
10    // adapt the traversal, such that state symbols are not visited
11    @Override public void traverse(IAutomataScope s) {
12        for (AutomatonSymbol aut : s.getLocalAutomatonSymbols()) {
13            aut.accept(getTraverser());
14        }
15    }
16 }
```

Listing 9.33: Customization of the `AutomataSymbols2Json`

`AutomataSymbols2Json` extends its generated TOP class and implements the `AutomataHandler` interface, so that it can be added as a handler to the traverser. This can be realized, e.g., in a constructor of the class.

By overriding the `traverse` method for automata scopes, the traversal of scopes during the symbol table serialization is adjusted. In this example, the traversal of all state symbols is omitted and only the automaton symbols contained in a scope are visited.

9.8 Resolving Symbols in Scopes

Symbol resolution is the main purpose of the whole symbol table infrastructure. Given a name, the resolution mechanism has to identify the symbol (or symbols) fitting to that name or return the information that no appropriate symbol exists. The symbols are then used e.g. to check the well-formedness of models or to navigate from the usage of a symbol to its definition.

The scopes introduced in Section 9.3.3 provide several `resolve` methods. A scope manages individual maps of symbols for each symbol kind by encoding the symbol kind into the name of the resolve methods. For example, the Automata language has the scope `IAutomataScope` that provides resolve methods named `resolveState` for `State` symbols.

Section 9.8.3 explains how the four variants of the `resolveState` as well as the six variants of their generalizations `resolveStateMany` work in detail.

9.8.1 How to Use Symbol Resolution

The application is simple: Each AST node knows its enclosing scope, which can be asked to resolve a symbol. For convenience, MontiCore also generates direct navigation from the AST node to the symbol, when indicated in the grammar. For example `to:Name@State` in a production for `Transition` generates to a method `getToSymbol` allowing to navigate from `ASTTransition` directly to the destination `StateSymbol`.

The following lists an excerpt of the available resolve methods by the example of a state symbol, the full list of methods is explained in Section 9.8.3. Each resolve method has a `String` argument with the name of the symbol(s) to resolve for.

`resolveState(String)`: Simple, standard search for a state symbol. If no matching symbol is found in the current scope, the search will continue, e.g., in the enclosing scope.

`resolveState(String, AccessModifier)`: Searches for a state symbol with the given name (first argument) and with the given access modifier (second argument). `AccessModifier` contain e.g. `public`, `protected`, and `private`.

`resolveState(String, AccessModifier, Predicate<StateSymbol>)`: In certain cases, additional information is used to identify the correct symbol. For example methods are not only resolved due to their name, but also due to the signature of the arguments. The filter is defined by a predicate over `StateSymbols`.

The above methods all return an `Optional<StateSymbol>` object. This form of resolution algorithms is based on the assumption that for a given name (and predicate) at most one symbol is visible in the respective scope. Resolving of course respects visibilities and shadowing. If more than one would be resolved, the algorithm issues an error.

The symbol table infrastructure also provides `resolveStateMany(...)` methods that resolve exactly as the `resolveState` methods, but return a collection of *all symbols*

(e.g., `List<StateSymbol>`) that match the search criteria. The resolution mechanism is rather complex and elaborated in detail in [MSN17] and only summarized in the following. Each generated scope interface has four variants of the `resolveStateMany` methods for each symbol kind. For each variant of the `resolveState` method as described above, there is a corresponding `resolveStateMany` method with the same list of arguments and a list of symbols as the return type. The signatures of the methods are depicted in Listing 9.40.

The resolve methods described so far execute the resolve algorithm explained in Section 9.8.2, i.e., begin resolving in the local scope, continue resolving in enclosing scopes until the global scope, and proceed with resolving top-down into foreign artifact scopes. There are further resolve methods that realize parts of the resolution algorithm:

`resolveStateDown(..)`: These methods contain the top-down resolving for a single state symbol and exist in variants with and without access modifiers and predicates as arguments. Top-down resolution begins in the current scope, and if no matching symbol was found, continues the resolution in all subscopes.

`resolveStateDownMany(..)`: These methods implement the top-down resolving for multiple state symbols and exist in variants with and without access modifiers and predicates as arguments. The result is a list of state symbols.

`resolveStateLocally(String)`: Resolve for a single state symbol with the given name only in the current scope. The resolution does not consider any enclosing scopes or subscopes.

`resolveStateLocallyMany(..)`: Resolve for all state symbols with a given name in the current scope, do not consider any enclosing scopes or subscopes.

9.8.2 Concept Of Symbol Resolution

The standard symbol resolution mechanism that MontiCore generates comprises three consecutive phases:

- Bottom-up intra model resolution
- Inter model resolution
- Top-down intra model resolution

These phases correspond to the search in the tree structure of the parsed ASTs. Figure 9.36 shows the tree structure reaching over the global scope based on the tree already shown in Figure 9.5 (p. 160).

If the scope structure and resolution algorithm shall behave different from the default presented here, it can be customized in various forms, as described in Section 9.8.4.

 Tip 9.34: Example: Resolving a State Symbol

An example for the resolution of the state symbol `Ping` in the automaton model `PingPong.aut` is depicted in Listing 9.35.

The artifact scope `modelTopScope` is asked by invoking the `resolveState` method with the argument `"Ping"`. The result is optional to indicate the absence of a fitting symbol.

If more than a single symbol has been found, the resolution terminates with an error.

```

1 // parse the model and create the AST representation
2 ASTAutomaton ast = parse("PingPong.aut");
3
4 // setup the symbol table
5 IAutomataArtifactScope modelTopScope =
6     createSymbolTable(ast);
7
8 // resolving a name in the model
9 Optional<StateSymbol> aSymbol =
10    modelTopScope.resolveState("Ping");

```

Listing 9.35: Example for resolving a state symbol

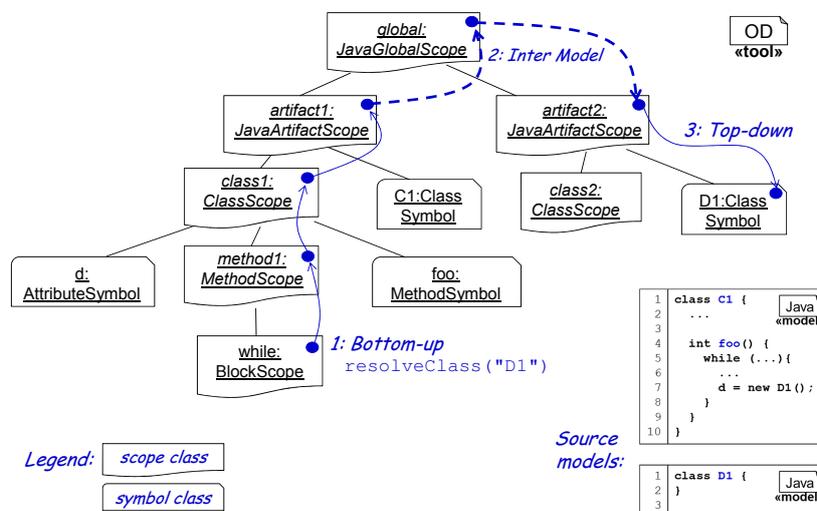


Figure 9.36: Principle of resolution in a hierarchy of scopes

Phase 1: Bottom-up intra model resolution

Bottom-up intra model resolution searches a symbol in the local scope. If a symbol with a suitable name and kind is found and the current scope is a shadowing scope, the symbol is returned and the algorithm terminates. Otherwise, the resolution continues the search in the enclosing scope. This is iterated until the enclosing scope is the global scope. Then,

the search proceeds in the global scope with inter model resolution.

Consider, for example, the usage of the variable `x` of Listing 9.37. A variable with this name is defined in the same model (or same class). When resolving `x`, the search starts in the innermost scope, i.e., the scope of the method `m`. Since `x` is not defined in `m`, the resolution mechanism continues in the enclosing class scope, finds `x` and returns it.

```
1 public class Example {
2     private int x = 0;
3
4     public void m() {
5         boolean b = x > 0;
6     }
7 }
```



Listing 9.37: Example class for bottom-up intra model resolution

Phase 2: Inter model resolution

Inter model resolution resolves for symbols beyond an artifact scope of an individual model. Therefore, inter model resolution is realized in the global scope because global scopes know where to look for symbol tables of foreign artifacts.

The resolution begins with identifying candidates for artifact names that are prefixes of the given symbol name and loads possible symbol tables on demand. Within the identified and now loaded symbol tables the symbol is resolved via top-down intra model resolution.

For instance, in Listing 9.38 the local variable `d` is of type `D`, which is defined outside the current artifact and thus needs inter model resolution.

```
1 package q.e;
2 import t.*;
3
4 public class Example {
5     public void m() {
6         D d;
7     }
8 }
```



Listing 9.38: Example class for inter model resolution

Considering the `package` and the `import` statements, `D` could be defined in the packages `q.e` or `t`, having the qualified name candidates `q.e.D` and `t.D`, respectively.

Hence, inter model symbol resolution is used to check, which of the candidates exist. Inter-model resolution in the global scope at first identifies candidates for qualified names of artifact scopes. By default MontiCore assumes that (1) the full qualifier of the symbol is the name of the artifact and (2) that the qualified symbol itself may be an artifact name. Therefore, in this example, the file name candidates are `q.e` and `t` and `q.e.D` and `t.D`.

The qualified symbol itself is added because in Java-inspired languages the name of the artifact typically matches the name of the top-level symbol in the artifact scope. In Java, class `D` is contained in file `D`.

The global scope loads missing artifact scopes and their symbols on demand as explained in Section 9.7 and manages them for further use.

The symbol resolution then proceeds with top-down inter model resolution in all artifact scopes that are candidates for containing the searched symbol. Ambiguities that arise from multiple symbols that are found are reported as errors, which in turn means that always all relevant scopes are used and the resolution does not stop with the first symbol found. This is slower but also safer because accidentally using the wrong symbol is detected and directly prevented.

Phase 3: Top-down intra model resolution

The third phase is implemented in the methods of the sort `resolveStateDown` and `resolveStateDownMany`.

Top-down intra model resolution begins with searching a symbol in the local scope. If the symbol is found in the current scope, it is returned and the algorithm terminates. If the symbol is not defined locally and the symbol name is unqualified, no further downward resolution is applied.

If the symbol is not defined locally but the symbol name is qualified, the resolution uses the qualifier to identify a subscope with a fitting name. If such a subscope exists, the first part of the qualified name is cut off and the top-down intra model resolution continues searching for the remaining name in the scope spanned by the symbol.

For example, consider the assignment of `Foo.bar` to variable `f` in line 2 of Listing 9.39. When resolving top-down for `Foo.bar`, no symbol is found in the scope of the class `Example`. The resolution then identifies symbol `Foo` that spans a scope for an enumeration. The resolution algorithm cuts off "`Foo.`") and resolves the remaining name `bar` in the spanned subscope, where it is actually found.

```

1 public class Example {
2     Foo f = Foo.bar;
3
4     enum Foo {
5         bar;
6     }
7 }

```

Listing 9.39: Example class for top-down inter model resolution

9.8.3 Generated Implementation for Symbol Resolution

As described above, a resolution is initiated through the `resolve` methods contained in the scopes. In normal scopes, `resolve` methods realize the *bottom-up intra model*

9. Symbol Management Infrastructure

resolution algorithm. Global scopes realize the *inter model resolution* in these and call artifact scopes again for *top-down intra model resolution* using the `resolveDown` methods.

Language engineers therefore mainly use the `resolve` or `resolveMany` methods shown in Listing 9.40, while the other offered methods are typically used internally.

```
1 import java.util.function.Predicate;
2
3 public interface IAutomataScope {
4     Optional<StateSymbol> resolveState (String name)
5     Optional<StateSymbol> resolveState (String name,
6                                         AccessModifier modifier)
7     Optional<StateSymbol> resolveState (String name,
8                                         AccessModifier modifier,
9                                         Predicate<StateSymbol> p)
10    Optional<StateSymbol> resolveState (boolean foundSymbols,
11                                       String name,
12                                       AccessModifier modifier)
13
14    List<StateSymbol> resolveStateMany (String name)
15    List<StateSymbol> resolveStateMany (String name,
16                                       AccessModifier modifier)
17    List<StateSymbol> resolveStateMany (String name,
18                                       AccessModifier modifier,
19                                       Predicate<StateSymbol> p)
20    List<StateSymbol> resolveStateMany (String name,
21                                       Predicate<StateSymbol> p)
22    List<StateSymbol> resolveStateMany (boolean foundSymbols,
23                                       String name,
24                                       AccessModifier modifier)
25    List<StateSymbol> resolveStateMany (boolean foundSymbols,
26                                       String name,
27                                       AccessModifier modifier,
28                                       Predicate<StateSymbol> p)
29 }
```

Listing 9.40: Standard resolving method signatures for `StateSymbols` in the `IAutomataScope` interface

As described above, `resolve` and `resolveMany` follow the standard procedure. Possible arguments are symbol name, `AccessModifier` to retrieve also private, or only publicly accessible symbols, a filtering `Predicate` that allows searching for additional information, such as method signatures, and `foundSymbols`, which is a boolean flag indicating whether the resolution call has found symbols already. The latter is for internal use in combination with non-shadowing scopes only.

Additionally generated methods are shown in Listing 9.41. `resolveLocally` methods resolve symbols only in the local scope, without proceeding to resolve in enclosing scopes or subscopes.

In the combination with symbol adapters (cf. Section 9.10.3), MontiCore prepares hook

methods for resolving adapted symbols with empty defaults. They allow to adapt the resolution mechanism using adapters as explained in Section 9.8.4.

```

1 public interface IAutomataScope {
2     Optional<StateSymbol> resolveStateDown(..)
3     List<StateSymbol>     resolveStateDownMany(..)
4
5     Optional<StateSymbol> resolveStateLocally(..)
6     List<StateSymbol>     resolveStateLocallyMany(..)
7
8     List<StateSymbol>     resolveAdaptedStateLocallyMany(..)
9
10    List<StateSymbol>     continueAsStateSubScope(..)
11    List<StateSymbol>     continueStateWithEnclosingScope(..)
12    boolean isStateSymbolAlreadyResolved()
13    setStateSymbolAlreadyResolved(..)
14 }

```

Listing 9.41: More generated resolving method signatures for StateSymbols in the IAutomataScope interface

The resolution algorithm is encoded into a set of collaborating resolve methods to allow adaptation, but also enable efficient lookup along the scope hierarchy. For example, the method `continueAsStateSubScope` is used as part of the *top-down intra model resolution*. It matches a prefix of the symbol name with the name of the symbol that spans the scope. If the names match, it invokes the `resolveStateDown` method with the remaining symbol name. Artifact scopes have a separate implementation of these methods as part of the *inter model resolution*.

The method `continueStateWithEnclosingScope` is part of the *bottom-up intra model resolution*. It checks whether the resolution algorithm has to continue resolving in the enclosing scope and if this is the case, proceeds resolving in the enclosing scope. The resolution proceeds either (1) if no symbols are found in the current resolution yet or (2) if the current scope is a non-shadowing scope.

The methods `isStateSymbolAlreadyResolved` and `setStateSymbolAlreadyResolved` are part of a mechanism that prevents resolve method call loops in case that symbol adapters can be chained to produce cyclic symbol adaptations.

9.8.4 Customizing Symbol Resolution

The generated symbol resolution mechanism handles many standard cases inspired by Java's typing mechanisms and UML's many different kinds of symbols. It should therefore also be suitable for languages with simpler or similar symbol concepts. For specific customizations, all resolving methods can be overridden with the TOP mechanism. This section presents two exemplary use cases for customization of the symbol resolution to demonstrate some possibilities.

Use Case 1: Finding Symbols in Foreign Artifact Scopes with Hierarchical Names

The default resolution calculates candidates for model names such that only symbols that are directly contained in the artifact scopes of the corresponding models are found. Therefore, symbols contained in nested scopes of foreign artifacts cannot be resolved by the default implementation. For example, in a hierarchical automata language, a state S contained in a state R that is itself contained in a state Q of the automaton P , by default, can not be resolved from a scope outside of the automaton with the qualified name $P.Q.R.S$.

In our example, the language shall allow a *qualified name* to identify the *package*, here: empty, model name, here P , named scopes within the model, here $Q.R$, and finally the name of the model element S .

To enable addressing foreign symbols contained in named subsopes, the calculation for model name candidates has to be customized. This realizes a *hierarchical namespace* of state symbols in a way similar to static inner types in Java. To achieve this, language engineers have to apply the TOP mechanism to the `IAutomataGlobalScope` interface and adapt the method `calculateModelNamesForState`, which calculates model name candidates to consider all prefixes as potential names for models that contain the symbol.

For example, a resolving for a state symbol with a qualified name $P.Q.R.S$ by default considers only $P.Q.R.S$ and $P.Q.R$ as potential names for a model that contains the symbol S . To realize a hierarchical namespace of symbols within automata, also the model name candidates $P.Q$ and P have to be considered.

The generated SMI on the other hand is already capable of resolving qualified symbols within an artifact. In particular, the artifact scope can by default resolve the remaining name parts pointing to the symbols $R.S$ (in artifact $P.Q$) and $Q.R.S$ (in artifact P) in addition to S (in artifacts $P.Q.R$ and $P.Q.R.S$).

```

1 @Override
2 default Set<String> calculateModelNamesForState(String name) {
3     Set<String> names = new HashSet<>();
4     // calculate all prefixes
5     while (name.contains(".")) {
6         name = Names.getQualifier(name);
7         names.add(name);
8     }
9     return names;
10 }

```

Listing 9.42: Handwritten adjustment of the method `calculateModelNamesForState`

For resolving for deeply nested states in an automaton during inter model resolution, the `calculateModelNamesForState` method in the interface `IAutomataGlobalScope` can be overridden as depicted in Listing 9.42. With the extended search of artifacts, symbols in nested scopes can be found.

Use Case 2: Selection of Symbol Table Files by their File Extension

By default, all files that may contain artifact scopes must be located relative to a model path entry. The qualifier of the model name candidate, e.g. `P.Q.R.S`, is translated into a path in the same way as packages in Java and the calculated model name equals the name of the symbol table file, e.g. file `S` extended by suffix `sym` in directory `P.Q.R`.

With the method `setFileExt` of global scopes, a regular expression for the file extension of symbol table files can be set changing the default `sym` to a language-specific file ending of models (cf. Section 9.7).

To take into account symbol tables provided by foreign languages, the default file endings during loading of symbol tables is `*sym`. Sometimes this is too general and only symbol tables of specific languages should be taken into account.

The file ending can be set to, e.g., `autsym` for including only symbol tables of the specific automata language.

9.9 Visitors Also Handle Symbol Tables

The visitors that MontiCore generates for each language (cf. Chapter 8) are not only able to traverse the AST but also the symbol tables of a language. As for each AST node, there are `visit` and `endVisit` methods for each symbol of the symbol table as well as for the scope interface and the artifact scope interface of the language. Furthermore, visitors contain methods for visiting the MontiCore RTE interfaces `ISymbol`, `IScope`, and `IArtifactScope` that are the super types of language-specific symbols and scopes. Given a language `L` and a symbol kind `C`, the signatures of all methods that the generated visitor interfaces provide for visiting symbol tables are depicted in Listing 9.43.

The traverser and handler classes that MontiCore generates for each language include `traverse` and `handle` methods for symbols and scopes of the symbol tables, too.

The *default traversal algorithm* as explained in Chapter 8 follows a *depth-first strategy on the AST*. Whenever handling an AST node that spans a new scope, MontiCore additionally traverses this scope and its symbols shallowly before returning to the AST. Thus, the traversal algorithm follows the overall AST structure but also covers the symbol table at its respective hook points, but does not directly traverse from a super-scope to any of the super-scope's subscopes.

Since the AST and the symbol table together define a graph structure, in general, loops can occur when traversing, resulting in infinite computations. The default AST and symbol table structures ensure that these loops do not occur by traversing the above described spanning tree only. However, when adapting the algorithm or manipulating the AST, the language developer has to take this into account.

```

1 public interface LVisitor {
2   // ... simplified list of methods
3
4   // Hooks, to be adapted for concrete functionality:
5   // (here for symbol kind C and language L)
6   default public void visit    (CSymbol s)
7   default public void endVisit (CSymbol s)
8   default public void visit    (ILScope s)
9   default public void endVisit (ILScope s)
10  default public void visit    (ILArtifactScope s)
11  default public void endVisit (ILArtifactScope s)
12
13  // language independent super interfaces
14  default public void visit    (ISymbol s)
15  default public void endVisit (ISymbol s)
16  default public void visit    (IScope s)
17  default public void endVisit (IScope s)
18  default public void visit    (IArtifactScope s)
19  default public void endVisit (IArtifactScope s)
20 }

```

Listing 9.43: Symbol table method signatures of a Visitor of a language L

9.10 Symbol Tables in Composed Languages

Language composition affects the symbol tables of the individual languages. In fact, symbol tables are of central importance for all kinds of language composition that MontiCore supports as described in Chapter 7.

Language inheritance, language extension, and language embedding rely on inheritance between the grammars of the individual languages. In all of these forms of language composition, an integrated language infrastructure is generated (cf. Chapter 7). This includes an integrated symbol table infrastructure that is explained in Section 9.10.1.

In *language aggregation*, no integrated language infrastructure is generated. Instead, the symbol table infrastructures of the languages that are aggregated are configured to realize the language aggregation through the exchange of stored symbol tables. This is explained in Section 9.10.2.

When composing languages, a symbol of a foreign, unknown *symbol kind* might be imported into a language. In such a case, a *symbol adapter* translates one symbol kind into another symbol kind. Symbol adapters are explained in Section 9.10.3.

These symbol adaptation mechanism can be applied between aggregated as well as between composed languages, where symbols *cross the language border* within one model.

9.10.1 Symbol Management Infrastructure for Language Inheritance

We explain the effect of language inheritance by example of the grammars A, B, C, and D shown in Figure 9.44.

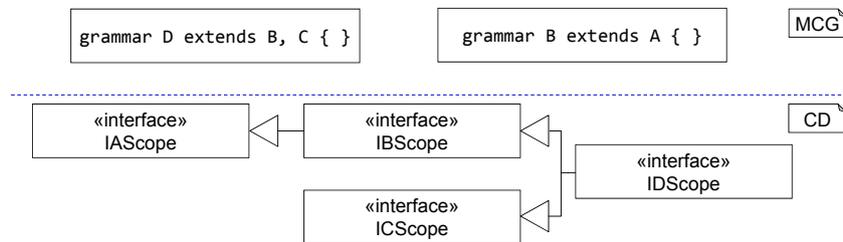


Figure 9.44: Effect of grammar inheritance on inheritance between scope interfaces

Language inheritance has no effect on the *symbols* because a symbol is defined by a concrete production and thus independent of the rest of the language

For scopes, the situation is different. As described in Section 9.3, the *scope* interface and the *scope* class in general, but also the *artifact scope* interface and class, as well as the *global scope* interface and the *global scope* class depend on the language.

Thus, a composed new language leads to new scopes. But again, the general principle of reuse is essential: To foster *reusability* of (1) handwritten extensions in the sublanguages as well as (2) code written for and against the API of a sublanguage, the generated classes must have a relatively complex structure:

Figure 9.44 shows that the scopes exhibit the same inheritance relations as the languages themselves. This allows to reuse TOP-extended scope functionality from scopes. Scope interface IDScope thus extends the scope interfaces IBScope and ICScope. Unfortunately, this does not hold for scope implementations because of the issues arising from multiple inheritance.

The inheritance structure is also retained for the interfaces of the global and the artifact scopes as visualized in Figure 9.45

Through this forms of diamond inheritance, the scopes of a language can contain symbols of symbol kinds that are defined in inherited languages and resolve methods for inherited symbol kinds exist as well.

Please remember that all scope objects are instantiated through mills and that the mill extension mechanism automatically delivers the correct scope object of the composed language, even if a mill of a sublanguage is requested. This allows to reuse code written for sublanguages.

The ScopeGenitors that instantiate symbol tables and the Symbols2Json classes that load and store symbol tables are both realized as visitors and connected via Traversers (cf. Chapter 8).

For instantiating symbol tables in the context of language composition, MontiCore generates a ScopesGenitorDelegator class for each language. This class instantiates a

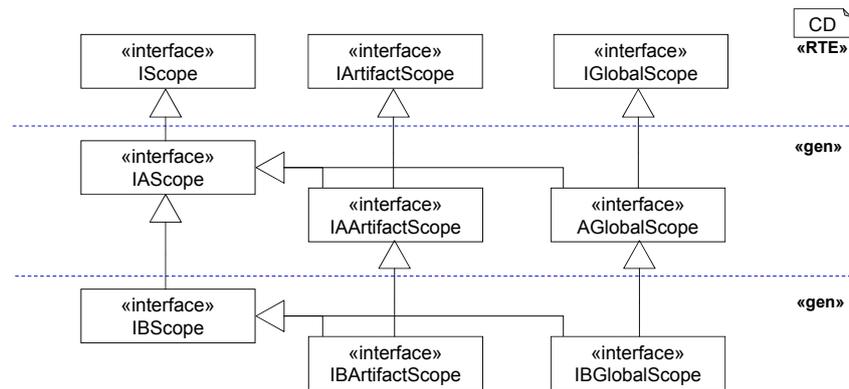


Figure 9.45: Effect of grammar inheritance on artifact and global scope interfaces

traverser of a language and adds scope genitors of the current language and all inherited languages. It further provides a `createFromAST` method that delegates to the respective method from the scopes genitor. The scopes genitor delegator is instantiated through the method `scopesGenitorDelegator()` of a language's mill. Therefore, if a language inherits from other languages, language engineers should use the scopes genitor delegator to instantiate symbol tables instead of the scopes genitor.

Language inheritance does not produce inheritance in `DeSer` classes.

For symbols, language composition does not have an influence and thus, the `SymbolDeSers` for these classes are not subject to change due to language composition.

`DeSers` for *scopes*, however, use symbol `DeSers`, which they obtain from the global scope. In composed languages, this includes `DeSers` for all *symbols kinds* that are defined in sublanguages.

9.10.2 Symbol Management Infrastructure for Language Aggregation

As said earlier, in *language aggregation* neither concrete, nor the abstract syntax of the individual languages are affected. Thus also scopes and symbols remain unaffected.

However, when using symbols between models then often the symbol kind does not fit. E.g. a state symbol `Ping` of an automaton needs to be mapped into method, enumeration or variable symbols in Java to be accessible from there. Or vice versa, *Java methods* (with empty signatures) might apply as *message* symbols in automata.

The support for language aggregation in the symbol management infrastructure, thus, concentrates on *symbol adapters*.

9.10.3 Symbol Adapters

We explain the symbol adaptation principle on the example of the Automata language that imports symbols from the Class Diagram (CD) language, where the method names of a class are used as message stimuli to trigger transitions and thus imported from the CD.

For realizing this, an automaton model explicitly contains an `import` statement that refers to the CD, where the symbols come from. There are three options to where to adapt symbols, when adaptation is needed:

1. The CD tool stores a symbol table with state symbols in it. Drawbacks are: the CD tool needs to know which kinds of symbols could be expected in downstream tools and has to store several formats.
2. The Automaton tool executes the adaptation when loading the foreign CD symbol table. Drawbacks here: Basically the same, but this time on the other side: The Automaton tool needs to know the kinds of symbols which it draws from.
3. Execute a stand alone adapter that reads a symbol table, adapts the symbols and stores the symbols ready for use in the target language. Drawback is that there is one additional tool to be executed in correct order in the tool pipeline and the overall execution time increases.

Usually the upstream tools are defined earlier and more stable, so that option one is probably rather rare. The other two options, however, can actually be chosen.

In any case an adaptation of symbols is needed, which manifests in a mapping from symbols of one symbol kind into symbols of another kind and must not necessarily be only a 1-to-1 mapping. For example an association symbol in a CD may manifest into a potentially larger number of access and manipulation function symbols for a logic language, like OCL. It is also possible that the symbol name changes.

To keep the individual tool modular and reusable, a tool internally only resolves for symbols of their own kind. Only when explicitly crossing a language border adapters can be added to map between the symbol kinds. This happens in composite languages at scopes that represent a language border and can be observed between artifact and global scopes.

Symbol adapters realize the adapter pattern and translate between symbols of a source kind to symbols of a target kind. For this, a symbol adapter inherits from the symbol class of the target kind and has an attribute `delegate` of the source symbol kind (also referred to as “adaptee”). Our naming convention for symbol adapters translating from a source symbol kind `S` to a target symbol kind `T` is `S2TAdapter`.

Listing 9.46 shows an example for a symbol adapter that adapts class symbols of the CD language to stimulus symbols of the Automata language, which is a (not so elegant) connection of CDs with automata using the State Design Pattern [GHJV94]. The adapter extends the class of the target adapter kind, i.e., `StimulusSymbol` (l. 2) and has an attribute of the source symbol kind, i.e., `CDClassSymbol` (l. 4). The source symbol object is handed to the class as the constructor argument (l. 6). Relevant methods of the super class, such as the method `getName` (l. 11), are overridden and delegate to the corresponding methods in the source symbol object.

```

1                                     Java «gen» CDClass2StimulusAdapter
2 public class CDClass2StimulusAdapter extends StimulusSymbol {
3
4     protected CDClassSymbol original;
5
6     public CDClass2StimulusAdapter(CDClassSymbol o) {
7         super(o.getName());
8         this.original = o;
9     }
10
11     @Override public String getName() {
12         return original.getName();
13     }
14 }

```

Listing 9.46: Symbol adapter for CDClassSymbols to StimulusSymbols

The symbol adapter delegates most methods to the original symbol, but may implement certain methods individually. For example, usually the method `getName()` is just delegated. Methods like `getEnclosingScope()` may need specific solutions ranging from (1) raising an exception, over (2) resulting in a pseudo scope object in which the externally loaded symbols are collected, to (3) adapting the entire scope graph data structure.

9.10.4 Resolving for Adapted Symbols

MontiCore provides two mechanisms for integrating symbol adapters into the symbol resolution process.

The first mechanism is for *language inheritance* and enables adapting symbols between a source and a target kind, which are both available within the scope. The `resolveAdaptedTLocallyMany` methods for each available symbol kind `T` in the scope interface act as hook points with empty default implementation and can e.g. be overridden using the TOP mechanism. The handwritten interface can provide a strategy for finding adapted symbols.

Listing 9.47 shows an example that resolves `StimulusSymbols` by looking for `CDClassSymbols` locally. For each suitable `CDClassSymbol` that is found, a `CDClass2StimulusAdapter` is instantiated and added to the scope.

```

1                                     Java «hw» ICDAutomataScope
2 public interface ICDAutomataScope extends ICDAutomataScopeTOP {
3
4     @Override
5     default List<StimulusSymbol> resolveAdaptedStimulusLocallyMany(
6         boolean foundSymbols, String name, AccessModifier m,
7         Predicate<StimulusSymbol> p)
8     {
9         // resolve source kind

```

```

10     List<CDClassSymbol> cdClasses = resolveCDClassLocallyMany(
11         foundSymbols, name, m, x -> true);
12
13     List<StimulusSymbol> adapters = new ArrayList<>();
14
15     for (CDClassSymbol s : cdClasses) {
16         // instantiate adapter
17         CDClass2StimulusAdapter c2s = new CDClass2StimulusAdapter(s);
18         if (p.test(c2s)) { // check predicate
19             adapters.add(c2s);
20             this.add(c2s); // add adapter to scope
21         }
22     }
23     return adapters;
24 }
25 }

```

Listing 9.47: Resolving symbols with added adapters if the scope knows both symbol kinds

The resolution from the user perspective starts with the known `resolve` methods. All the adaptation happens under the hood using the following typical strategy:

1. Resolve the symbol of the source kind *S* locally with `resolveSLocallyMany`.
2. If a symbol was found, instantiate a class `S2TAdapter` adapting the *S* symbol to the target kind *T*.
3. To deliver the same object in repeated requests, the adapter symbol is added to the list of symbols in the scope.

The second mechanism that MontiCore provides for integrating symbol adapters is useful for *language aggregation*, as it does not require both source and target kind to be available in a single type of scope.

The *global scope* manages a list of `SymbolResolver` hook points for each symbol kind. During resolution of a symbol of kind *S*, the global scope iterates over all managed symbol resolvers for the kind *S*.

For each symbol kind *T*, MontiCore generates the interface `ITSymbolResolver` that defines the abstract method `resolveAdaptedTSymbol`. In this example, a subclass `S2TSymbolResolver` is defined and hooked into the global scope. When the global scopes looks for a symbol of kind *T* the hook is called and may now look for a symbol of kind *S* in the same potential source artifacts (which are defined by the qualified package names). If found, the adapter is added and returned.

In our example in Figure 9.48 the resolver `CDClass2StimulusAdapters` can be added to the global scope of the automata language. The effect of this is that the method `resolveAdaptedStimulusSymbol` of the resolver implementation is called during resolution for a `StimulusSymbol`. Internally, the method then resolves for a `ClassSymbol`. If such a symbol is found, an adapter is instantiated and added to a list of adapted symbols, which is then returned by the method.

```
1 public class CDClass2StimulusResolver
2     implements IStimulusSymbolResolver {
3
4     @Override
5     public List<StimulusSymbol> resolveAdaptedStimulusSymbol(
6         boolean foundSymbols, String name, AccessModifier m,
7         Predicate<StimulusSymbol> p)
8     {
9         List<StimulusSymbol> r = new ArrayList<>();
10        Optional<CDClassSymbol> s = BasicCDMill.globalScope()
11            .resolveCDClass(name, m);
12        if(s.isPresent()){
13            CDClass2StimulusAdapter a
14                = new CDClass2StimulusAdapter(s.get());
15            if(p.test(a)){
16                r.add(a);
17            }
18        }
19        return r;
20    }
21 }
```

Listing 9.48: Example resolver for CDClass2StimulusAdapters

As usual, the generated classes can be extended by handwritten code to adapt the symbol infrastructure management. All artifacts that MontiCore generates for the symbol table realize a specific default solution for the management of symbols. Language engineers can extend this to realize, e.g., sophisticated symbol visibility concepts, various forms of import mechanisms specific to a symbol kind, flat name spaces of symbols in a model, symbols for built-in types, or import functionality for symbols defined externally to MontiCore.

Chapter 10

Realizing Context Conditions

co-authored with Robert Heim

A language definition in MontiCore is based on a context-free grammar (CFG). Such a grammar only defines the language features in general and does not support context-sensitive restrictions (e.g., that a specific entity of a model must exist when used elsewhere). In addition, some restrictions are much easier to express in a context-sensitive way, while a context-free representation of the same constraint would be cumbersome (see below for an example). *Context conditions* enable such context-sensitive restrictions. They are predicates that further restrict the set of models described by a CFG and determine the set of *correct* – also called *well-formed* – models of a language.

A *context condition* (CoCo) is a predicate on a CFG-correct sentence where the context of a word is used to determine the total correctness, also called *well-formedness*.

A model/sentence of a language is *well-formed* if it fulfills all context conditions. Well-formedness is the basis to define semantics, to generate code, etc.

Some typical forms of context conditions are:

- A variable must be declared before it is used.
- The type of a variable must exist.
- Only compatible values can be assigned to a variable.
- A method call must fit to its signature.
- A deterministic automaton must have exactly one start state.
- A class hierarchy does not have cycles.

There exist multiple kinds of context conditions, such as, conventions (e.g., names must start with a capital letter), unreachable statements and many more.

Context conditions can be checked at different times, but should always be checked before a model is used for its designated purpose. Useful times for checking are:

1. After parsing and building the AST (see Chapter 6).

2. After creating the symbol table from the AST (see Chapter 9).
3. When a modification was applied on the AST, it may be worth to check all or some context conditions again.

Many context conditions are and can only be checked during or after the symbol table is created, since the symbol table provides helpful information and enables an efficient way of implementing them. However, if the symbol table is not needed, context conditions can be checked against the pure AST. Also, context conditions can again be checked after applying transformations on the AST to ensure that the AST still represents a valid model.

For example, the Automata language (cf. Section 21.1) requires context conditions such as the following:

- State names must be unique.
- Source and target of a transition refer to an existing state.
- State names must start in uppercase.

The first one cannot be checked in a context-free way as names can be defined throughout the model. The uniqueness of state names ensures that references to states (e.g., when defining transitions) are unambiguously resolvable. The second restriction complements the first one, since state names not only must be unique, but referenced states must be defined in the model. This is a context-sensitive restriction, because names cannot be resolved in a context-free way. While it is possible to formulate the requirement of capitalized state names in a context-free manner, this would require some tedious amounts of token definitions. Here, a context condition can easily check that state names are capitalized. The following sections describe MontiCore's context condition infrastructure.

10.1 Context Condition Infrastructure

This section describes MontiCores infrastructure to implement context conditions for a DSL. Given a grammar (such as the Automata grammar in Section 21.1), MontiCore generates a context condition infrastructure. Context conditions are predicates concerning specific model elements, such as states, classes, fields etc. The implementation of a context condition relies on the internal representation of a model element, which is the respective AST node (and its sub-tree). Hence, MontiCore generates a set of interfaces each providing a `check` method for a specific AST node type. In case of the Automata language this results in three generated interfaces, namely `AutomataASTStateCoCo`, `AutomataASTTransitionCoCo` and `AutomataASTAutomatonCoCo`, each defining a check signature for the corresponding AST node.

For example, MontiCore generates the interface `AutomataASTStateCoCo` that defines the method signature of the `check` method for nodes of type `ASTState` (cf. Figure 10.1).

Implementing a context condition on an AST element is as simple as implementing the corresponding interface. As a best practice, each interface implementation should only implement one context condition at a time. For example, the context condition

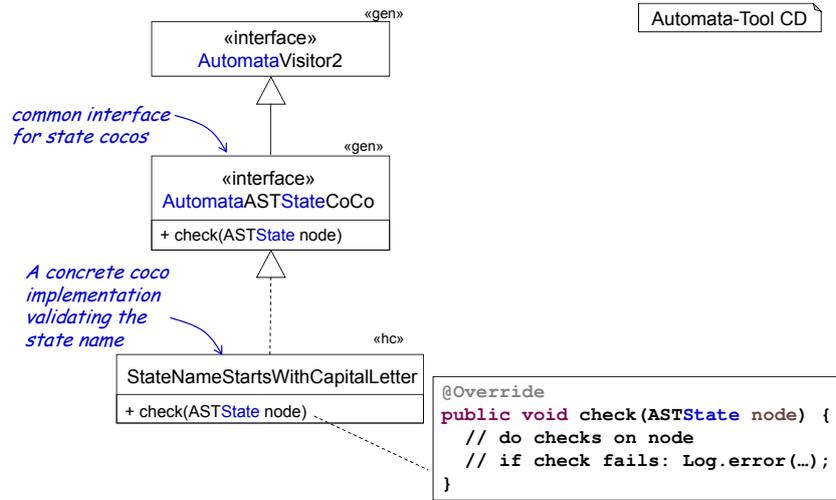


Figure 10.1: CoCo infrastructure for nonterminals

StateNameStartsWithCapitalLetter implements the former mentioned interface for state nodes (cf. Figure 10.1). Listing 10.6 depicts the complete implementation of this context condition.

To check the context conditions on a model MontiCore generates a so called *checker* for a given grammar. For a language L the checker is called LCoCoChecker and provides an addCoCo method for each generated coco interface (i.e. for each nonterminal). This enables developers to register their implemented context conditions at the checker. Additionally, a checker provides a checkAll method that can handle any AST node of the language. The latter executes all registered context conditions on the given AST node and its children. Typically, one would hand the root node of an AST to the method to check a complete model for well-formedness.

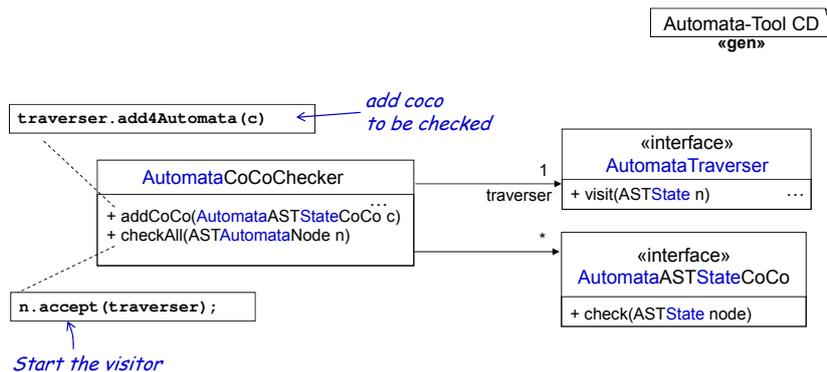


Figure 10.2: The generated CoCo checker

As an example, the Automata language’s checker AutomataCoCoChecker is shown in Figure 10.2. Internally, the checker uses a traverser (cf. Chapter 8) to traverse a given AST node. Hence, in case of the Automata language the AutomataCoCoChecker uses the

10. Realizing Context Conditions

```
1 public class AutomataCoCoChecker {
2
3     private automata._visitor.AutomataTraverser traverser ;
4
5     public void checkAll (ASTAutomataNode node)  {
6         node.accept (getTraverser ());
7     }
8
9     public void addCoCo (AutomataASTAutomatonCoCo coco)  {
10        traverser.add4Automata (coco);
11    }
12
13    // ... analog infrastructure for other AST elements ...
14 }
```

Listing 10.3: Implementation of the AutomataCoCoChecker class

```
1 // setup context condition infrastructure
2 AutomataCoCoChecker checker = new AutomataCoCoChecker ();
3
4 // add a custom set of context conditions
5 checker.addCoCo (new StateNameStartsWithCapitalLetter ());
6 checker.addCoCo (new AtLeastOneInitialAndFinalState ());
7 checker.addCoCo (new TransitionSourceExists ());
8
9 // check the CoCos
10 checker.checkAll (ast);
```

Listing 10.4: Configure the AutomataCoCoChecker and check the context conditions

AutomataTraverser interface. Listing 10.3 shows the implementation of the `checkAll` method and the method regarding the states.

Listing 10.4 shows a usage example of the `AutomataCoCoChecker`. First it is instantiated (l. 2) and then configured by adding some cocos (ll. 5f). Assuming that a model is available in the variable `ast` the checker is executed to check all registered context conditions (l. 10) on the parsed model by handing the AST to the `checkAll` method.

Usually, context conditions as well as checkers are implemented in a state free form and thus a single instance can be reused on many models.

10.2 Implementation of Context Conditions

Besides using the described infrastructure there are some best practices for implementing context conditions. Every context condition should have a *unique error code* that makes it easier to communicate the error (for example when asking developers for help) or identify the error's source position. It also facilitates writing dedicated tests for a context condition

**Tip 10.5: Fixed Sets of Context Conditions**

A language typically has a fixed set of standard context conditions. A best practice is that language implementations should provide configured coco checkers to language users.

In case of the Automata language, the handwritten class named `AutomataCoCos` implements the method `getCheckerForAllCoCos` that creates an `AutomataCoCoChecker` object and configures it by adding the common context conditions. Language users can use this method to obtain a configured checker to check a model for well-formedness.

It is of course possible to add new CoCos to the checker or execute some afterwards if desired.

(cf. Section 10.3 for details). An *error message* should be a human-readable, compact explanation of why a specific context condition is not fulfilled and should contain the exact source position. Both support the modeler in fixing a violation.

With the whole context condition infrastructure provided by MontiCore, it is easy to develop own context conditions. Listing 10.6 shows the full implementation of the context condition `StateNameStartsWithCapitalLetter`. The handwritten class implements the generated interface for state context conditions (ll. 2f). Hence, its `check` method has an `ASTState` node as parameter (l. 5). In case of a violation of the context condition (ll. 6ff.), the implementation uses MontiCore's logging infrastructure (cf. Chapter 15) to issue a warning (ll. 12f). In a productive environment such an error terminates the program and shows the error message. However, for testing purposes this behavior can be adjusted by disabling the fail quick feature of the logger. Section 10.3 describes more details on testing context conditions and the test setup.

**Tip 10.7: Using the Symbol Table in Context Conditions**

Normally the symbol table is used to understand, which symbols are defined, what kind and what extra information they carry.

To use the symbol table within context condition implementations, the symbol table must be created before executing the context conditions. Then, the links from AST nodes to the corresponding symbols and scopes are set (cf. Chapter 9).

Often, context conditions use the symbol table to obtain required information. For example, there are two options to check whether the source state of a transition exists. First, one can iterate over all state children of the `ASTAutomaton` node and try to find the correct state's AST node. Alternatively and more efficiently, one can make use of the resolving mechanism that is provided by the symbol table. Listing 10.8 shows the solution using the symbol table. It retrieves the enclosing scope of the AST node through `node.getEnclosingScope()` (l. 5). Having the enclosing scope, one can try to resolve the source state using its name (`node.getFrom()`, ll. 7f). If a state with the particular name does not exist, this leads to an error (ll. 10f).

10. Realizing Context Conditions

```
1 Java «hw» StateNameStartsWithCapitalLetter
2 public class StateNameStartsWithCapitalLetter
3     implements AutomataASTStateCoCo {
4     @Override
5     public void check(ASTState state) {
6         String stateName = state.getName();
7         boolean startsWithUpperCase =
8             Character.isUpperCase(stateName.charAt(0));
9
10        if (!startsWithUpperCase) {
11            // Issue warning...
12            Log.warn(
13                String.format(
14                    "0xADD02 State name '%s' is not capitalized.",
15                    stateName),
16                state.getSourcePositionStart());
17        }
18    }
19 }
```

Listing 10.6: Implementation of a context condition for State objects

```
1 public class TransitionSourceExists Java «hw» TransitionSourceExists
2     implements AutomataASTTransitionCoCo {
3     @Override
4     public void check(ASTTransition node) {
5         IAutomataScope enclosingScope = node.getEnclosingScope();
6         Optional<StateSymbol> sourceState =
7             enclosingScope.resolveState(node.getFrom());
8
9         if (!sourceState.isPresent()) {
10            // Issue error...
11            Log.error(
12                "0xADD03 Source state of transition missing.",
13                node.getSourcePositionStart());
14        }
15    }
16 }
```

Listing 10.8: Using the symbol table in a context condition

10.3 Testing Context Conditions

As a best practice for testing context conditions, one should test both, valid and invalid models. The former make sure that *valid models do not violate the context condition* (i.e. true positives and no false negatives), whereas the latter ensure that *invalid models do violate the context condition* (i.e. true negatives and no false positives). Consequently, two

```

1 public class TransitionSourceExistsTest {
2
3     // setup the parser infrastructure
4     AutomataParser parser = new AutomataParser() ;
5
6     @BeforeClass
7     public static void init() {
8         LogStub.init();
9     }
10
11    @Before
12    public void setUp() throws RecognitionException, IOException {
13        Log.getFindings().clear();
14    }
15 }

```

Listing 10.9: Initial setup to test a context condition

different kinds of tests should exist for every context condition.

In MontiCore tests for context conditions are implemented using the test-framework JUnit. Listing 10.9 shows a best practice to initialize a context condition test. A reusable parser object is initialized and stored as attribute in l. 4f, because it behaves as if stateless. The `init` method (ll. 7) changes the behavior of the log by using a stub `LogStub` that does not have side effects (no output) and disables the fail quick feature of MontiCore's logger (cf. Chapter 15). This ensures that the test is further executed when the error occurs. The test can then assert expected errors. Since this initialization is required only once before all specific tests, the `init` method is annotated with `@BeforeClass`. Before every test, the `setUp` method (ll. 11) – annotated with `@Before` – clears all findings (of potential previous tests) to ensure a clean test setup.

10.3.1 Testing a Context Condition on a Valid Model

Testing a context condition on a *valid model* consists of the following three steps:

- Parse the model, obtain the AST, and create its symbol table.
- Check the context condition on that AST.
- Verify that no errors occurred.

```

1 @Test
2 public void testOnValidModel() throws IOException {
3     ASTAutomaton ast = parser.parse_String(
4         "automaton Simple { state A; state B; A -x> A; B -y> A; }"
5     ).get();
6
7     // setup the symbol table

```

10. Realizing Context Conditions

```
8     IAutomataArtifactScope modelTopScope = createSymbolTable(ast);
9
10    // setup context condition infrastructure & check
11    AutomataCoCoChecker checker = new AutomataCoCoChecker();
12    checker.addCoCo(new TransitionSourceExists());
13
14    checker.checkAll(ast);
15
16    assertTrue(Log.getFindings().isEmpty());
17 }
```

Listing 10.10: Testing a context condition on a valid model

Listing 10.10 demonstrates this by testing the context condition `TransitionSourceExists` (cf. page 216). First of all, the model is specified in ll. 3f. An `ArtifactScope` that contains the symboltable of the model is created from this model (cf. l. 8). For more information about `ArtifactScopes`, see Chapter 9. The context condition is instantiated in and added to a checker. Next, the checker is executed on the model (cf. ll. 11f). Finally, the test verifies that no errors occurred in ll. 16f.

10.3.2 Testing a Context Condition on an Invalid Model

Testing a context condition on an *invalid model* is similar to the above check, but at the end checks for the expected errors.

Listing 10.11 shows a test on an invalid model that does not define the source state of a transition. Again, the model is specified (cf. l. 3) and the symbol table for this model is created in l. 9. This model uses a state that has not been defined. A checker is configured with the context condition under test and executed on the invalid model (cf. ll. 12f). This example expects exactly one error with a given text. Checking that all expected findings occurred (cf. ll. 18f) ensures that the context condition identifies the invalid model as such.

```
1 @Test
2 public void testOnInvalidModel() throws IOException {
3     ASTAutomaton ast = parser.parse_String(
4         "automaton Simple { " +
5         " state A; state B; A - x > A; Blubb - y > A; }"
6     ).get();
7
8     // setup the symbol table
9     IAutomataArtifactScope modelTopScope = createSymbolTable(ast);
10
11    // setup context condition infrastructure & check
12    AutomataCoCoChecker checker = new AutomataCoCoChecker();
13    checker.addCoCo(new TransitionSourceExists());
14
15    checker.checkAll(ast);
16
17    // we expect one error in the findings
```

```
18 | assertEquals(1, Log.getFindings().size());
19 | assertEquals("0xADD03 Source state of transition missing.",
20 | Log.getFindings().get(0).getMsg());
21 | }
```

Listing 10.11: Testing a context condition on an invalid model

Please note that the use of `LogStub` prevents that the error is actually printed and the program terminates. Instead the error message is only stored in the findings and continues execution.

It is possible to check the source position of the error in the invalid model as well. However, it is often useful to reduce the assertion to checking the error code (`0xADD03`), because error messages are relatively often modified.

Chapter 11

Design Patterns Used and Invented for MontiCore

co-authored with Nico Jansen

Design patterns [GHJV94] are helpful concepts as they provide reusable solutions to commonly occurring problems. They can be used to structure a generated product as well as the generator itself. This is in particular useful for handwritten extensions that need to integrate with generated parts. While MontiCore uses quite a number of standard design patterns such as template-hooks, visitors, adapters, factories, and builders, some design patterns consistently used in MontiCore's context have been either substantially adapted or even newly created.

The visitor pattern is a prominent example and is thus described in its own Chapter 8. The builders used to create AST and symbols objects are also refined, and described in Section 5.9. Their composition and adaptation is subject to Section 14.2. Other slightly adapted design patterns follow in this chapter.

11.1 Static Delegator Design Pattern

The *static delegator* is a design pattern that combines the advantages of publicly accessible static methods with the possibility to redefine them. For that purpose the pattern introduces a hidden delegate object that can be replaced on demand for customization. We demonstrate this on the method `info(String, String)` that is part of the logging API described in Section 15.3 and shown in Listing 11.1.

The public *static method*, that is meant for external use and is therefore publicly available, is eponymous for the pattern. The *static host class* provides one or more such public static methods. Internally, the static method delegates to an object, which provides the actual implementation in a so called *do-method*. Considering the example in Listing 11.1, the static method `info` delegates the method call to its internally used object `log`, which is an instance of the `Log` class. The static method calls the instance method `doInfo` that provides the actual implementation. Static delegators may have many such pairs.

The static *delegate object* (`log`) in line 3 is kept hidden but exchangeable and is used as a delegate for the static methods through the static `getLog()` method (ll. 6ff.). The

11. Design Patterns Used and Invented for MontiCore

```
1 public class Log {
2     // the single static delegator target
3     protected static Log log;
4
5     // Getter for the underlying Log.
6     protected static Log getLog() {
7         if (log == null) {
8             setLog(new Log());
9         }
10        return log;
11    }
12
13    // Allows to set an individually defined Log instance
14    protected static final void setLog(Log log) {
15        Log.log = log;
16    }
17
18    public static final void info(String msg, String logName) {
19        getLog().doInfo(msg, logName);
20    }
21
22    protected void doInfo(String msg, String logName) {
23        // a default implementation, but can be overridden
24    }
25 }
```

Listing 11.1: A static delegator method

method `getLog()` initializes automatically on its first use (l. 8), such that no external initialization is required. However, the behavior can be changed by redefining the hidden static instance in a new subclass as shown in Listing 11.2.

```
1 public class LogStub extends Log {
2
3     protected LogStub() { }
4
5     // Initialize the LogStub as Log
6     public static void init() {
7         LogStub l = new LogStub();
8         l.isNonZeroExit = false;
9         Log.setLog(l);
10    }
11
12    // The customized behaviour
13    protected void doInfo(String msg, String logName) {
14        // adapted implementation
15    }
16 }
```

Listing 11.2: Customized static delegator method

After an explicit invocation of the method `LogStub.init()` shown in Listing 11.2, line 6ff., the static delegate object in `Log` will be an instance of `LogStub` and thus provide its customized behavior of the `info` method. That is because the public static method `info` of the class `Log` (cf. line 18 of Listing 11.1) now delegates to the protected method `doInfo` (cf. line 13ff.) of an instance of the class `LogStub`.

In many cases, e.g., in the shown logging, the method `LogStub.init()` has to be invoked as early as possible to ensure proper initialization from the beginning.

The *static delegator* design pattern can be used in many circumstances, for example, builder mills, protocol objects etc. It may come in variations, for example:

- Many static methods delegating to the same instance.
- Many static methods where each of the static methods internally has its own instance object, thus allowing high configurability (cf. the generated AST builder mills described in Section 5.9).
- The host class of the static methods and the delegate class can be decoupled (thus having separate classes).
- Several subclasses may be defined allowing configuration or even dynamic reconfiguration during runtime.

The main benefit of this design pattern is that one method or a certain set of methods is available uniquely throughout all pieces of code, such that it is still possible to redefine the methods even though they have a static externally visible interface. This also assists mocking side effects of a static delegator (like protocols, database access, or GUI) when testing the system, e.g., by replacing the static delegator by a dummy.

Limitations are also coming with the use of static methods. For instance, web frameworks forbid static methods. If parallel processes want their own individual instantiations, then conflicts between otherwise independent processes may occur and must be managed like described in [Rum17].

In contrast to the delegation pattern in [GHJV94], this pattern is a conjunction of a singleton and delegating methods. Thus, the static methods do not need to use the same object to delegate to. Instead, there can be multiple objects stored and handled internally.

11.2 RealThis Object Composition Pattern

The central idea of this pattern is that several objects are composed in such a way that they behave like one single object to the external world. For this purpose, this section introduces the *RealThis object composition pattern*, which is essentially a combination of the *composite pattern*, the *callback pattern* and the *delegation pattern*, where the composite delegates to its components. For a tight integration, the components then callback to the composite using the `realThis` link.

As in the composite pattern, there is a single class that is visible to the outside and whose instantiated object acts as the central, externally known object. In the example in

Figure 11.3, this is class A. Furthermore, there are several subclasses that realize partial functionality. Objects of these subclasses can be freely combined as components of the composite to define the overall functionality. In Figure 11.3, these are the classes B and C. In this example, A provides the method signatures `foo` and `bar`, which are not implemented by A directly but by subclasses. Here B implements the method `foo` and C the method `bar`. Note, that arbitrary many methods are allowed.

A internally holds for each method a link to an object that realizes the method. In the example, there exist two links to the objects `forFoo` and `forBar` of type A. A valid configuration of A objects must set these links to appropriate objects of the subclasses. In the example, `forFoo` is assigned with an instance of B and `forBar` is assigned with an instance of C. Note that this is just one option, other configurations could be attached only to B objects, etc. Each method can be configured using an individual subclass object to delegate to.

Whenever A receives a call of `foo` or `bar`, it is delegated to `forFoo` or `forBar` respectively. Thus in this example A uses the `foo` realization of B and the `bar` realization of C, but externally acts as a single entity.

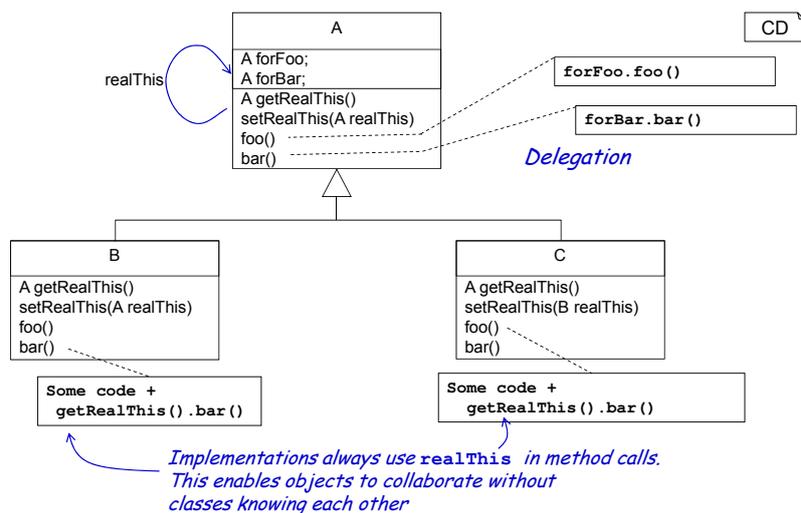


Figure 11.3: Components use `realThis` instead of `this` to enable close collaboration

To ensure a tight collaboration between the objects in that composition, the callback to `realThis` is used.

For this, A has two methods `setRealThis` and `getRealThis` to manage a `realThis` instance of itself. The `realThis` instance is used instead of `this`, i.e., all calls of the form `this.bar()` are made to `realThis.bar()` instead. This enables the interaction of the individual objects. B and C inherit the `realThis` methods as well and thus collaborate – potentially without even knowing each other syntactically.

In all objects B and C of that composition, however, the `realThis` is set to the instance of A. This way, if something is called on the same object (but of course via the indirection `realThis`), the control is given back to A again.

An exemplary flow where `foo` is called on `a:A` is shown in Figure 11.4. Let us assume `B.foo()` calls `bar()`, which by configuration shall be from class `C`. As can be seen, in this configuration `a:A` delegates the call to its subobject `forFoo:B`. In `B`, instead of calling `bar()` directly, `getRealThis().bar()` is called. This, in effect, calls `bar` again on `a:A` and `a:A` delegates to `c:C`.

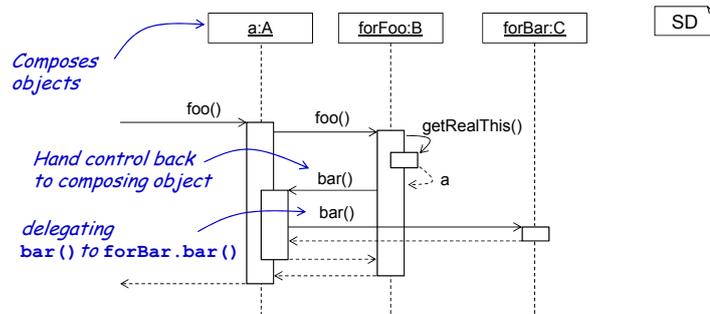


Figure 11.4: The runtime program flow in a `realThis` composition

When generating code, it is often convenient to generate independent artifacts in different generation phases or by different generators. However, during runtime, a tight integration may be necessary. This is, in particular, the case when parts of classes originate from different generators or a class functionality shall be partially handcoded and generated. To allow separated artifacts (files) but integrated runtime objects (actually an object group that acts like a single object) in Java, individual classes need to be defined, but their instantiated objects need to be closely composed. In MontiCore, this is achieved by using the described `RealThis` pattern.

In general, the `RealThis` pattern enables late composition of functionality while users would not recognize that they work with a composed object. A drawback is the overhead required during the implementation since every explicit and implicit `this` usage must be correctly identified and replaced by `realThis` in all the participating subclasses. Also, during runtime, such composition incorporates many object instances and an increased method delegation stack occurs (cf. Figure 11.4).

In practice, it is possible to add as many subclasses as desired and select each time only the exactly desired subset of functions. If a link is not set, an empty default behavior may be implemented in the composite `A` itself.

If attributes need to be shared among the objects, these attributes should be located in the composite class `A` and made available through appropriate access methods.

It is also possible to operate with subclasses like `B` in an isolated form because then the `realThis` link just points to the `B` object itself. Furthermore, it is possible to even nest the structures, i.e. `A` may contain and delegate to another `A` object or `B` might use delegation as well.

In C#, partial classes serve a similar purpose. Compared to the `realThis` approach, they have the advantage that native language support is provided, but the disadvantage that

the partial classes need to be compiled together and are statically fixed (thus no individual use or configurable variability in the composition is possible).

Variant of the RealThis Pattern without Common Superclass

To overcome the single inheritance limitation of Java, interfaces can be used. That means method signatures are defined in an interface of the class and the `realThis` attribute is of the interface type (cf. Figure 11.3). This enables a composing object to extend all interfaces of the composed objects (cf. Figure 11.6) and does not enforce a common superclass anymore.

This is the case, for example, when a language is composed of sublanguages. In this case, the sublanguages already provide parts of the functionality required by the composed language. Users of the composed language then only interact with the composed object and not with its participating objects. Since Java does not support multiple inheritance, a strict separation into interface and implementing class is needed. The interfaces offer all methods including `getRealThis` and `setRealThis`. The example in Figure 11.5 shows the interfaces B and C and their implementations BImpl and CImpl. B offers the method `foo` while C offers the method `bar`.

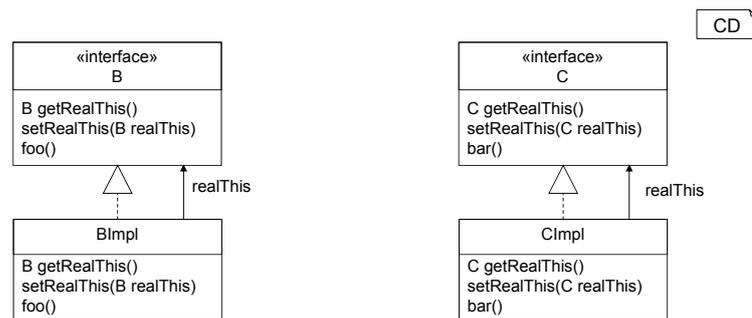


Figure 11.5: Splitting Classes in Interface and Implementation

As before, within classes, no call may use `this`, but call are made through a call through the `getRealThis` accessor method.

These classes and interfaces can now be combined by a third class AImpl so that they behave as a single object, as shown in Figure 11.6.

The implementations of these methods in A again delegate to the respective object. As before, AImpl sets itself as `realThis` of the two internal objects `b` and `c`.

Thus the object as well as the method call structure of this variant of the pattern are identical to the first variant, but the class structures differ.

As an advantage, the composite AImpl offers the method signatures and implementations of B and C, which is an advantage over the first variant, where the offered methods are fixed by the superclass and cannot be extended. As a disadvantage, a new subclass D in the first variant can be directly included, while in the second variant, the interface A and the class AImpl have to be adapted to make D usable.

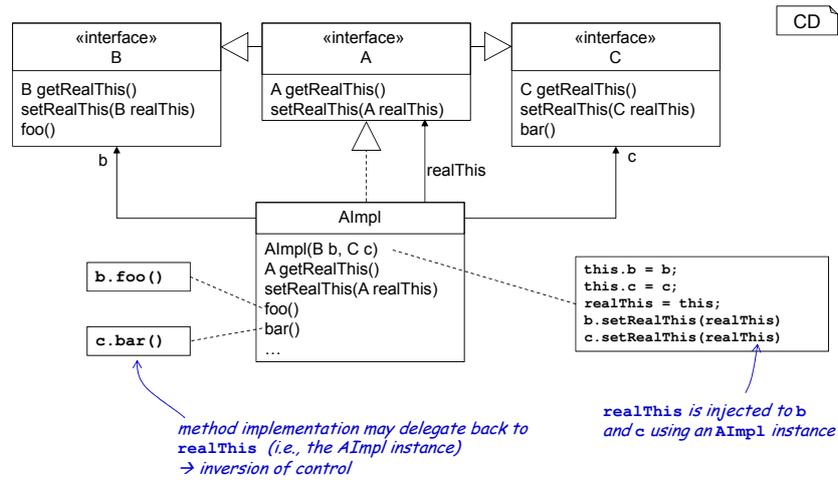


Figure 11.6: Composing objects using the realThis approach

The visitor infrastructure of MontiCore is built on the second variant of this pattern to enable visitor composition (see Chapter 8), where the integrative composite classes are newly generated for each language composition.

11.3 Attribute and Association Access Pattern

When discussing the architecture of a piece of software, we often only look at the data structure, namely classes, their attributes, and if explicitly modeled their associations. Developers using that data structures, however, need to know the access and modification methods that allow modifying attributes as well as the implementation of associations. Using a systematic derivation from attributes to their access methods has two big advantages. First, developers know the names and signatures of the methods by only looking at the data structure. And second, generators can be used to actually execute that derivation.

MontiCore uses the following patterns consistently. We also recommend that generators in general use these patterns, but of course are free to add additional functionality dependent on the desired realizations.

11.3.1 Attribute Access Pattern

Ordinary Attribute. An ordinary attribute `a` of type `Person`, which is stored in a class `Group`, needs exactly the two classic `get/set` methods, one that allows to retrieve the value and one that allows to set the value (see Listing 11.7).

We generally assume that `null` is not part of the values in the system and thus also do not check for or against `null`. If the absence of a value shall be modeled, optionals shall be used.

```
1 class Group {
2     void setA(Person a);
3     Person getA();
4 }
```

Listing 11.7: Classic get and set method signature for an attribute

We could use the same simple generation mechanism for all kinds of attributes, but at least for the container attributes `List`, `Set`, `Map`, and `Optional`, it is useful to not fully expose the container, but offer methods for manipulation of the container content directly. Writing all those methods manually is usually tedious and often error-prone, but when a generator is available, it is relatively simple to generate appropriate access functionality in a systematic way.

Optional Attribute. For optional attributes, it is convenient to have some additional methods for checking if a value is present and for setting the value absent. For convenience, we therefore provide the four methods depicted in Listing 11.8 for each optional attribute `b`, which here is of type `Optional<Person>`.

```
1     Person getB();
2     boolean isPresentB();
3     void setB(Person a);
4     void setBAbsent();
```

Listing 11.8: Methods for an optional attribute

Method `isPresentB` allows to directly understand whether the real value is present. Method `setBAbsent` directly sets the value absent and with `setB`, the unwrapped, but existing real value can be stored in the optional attribute. The normal `getB` method can fail, because the stored value may be absent. So the method may only be used when the developer is sure that the real value is stored, e.g., after asking with `isPresentB`.

Dependent on the context, the partiality of the `getB` method is implemented by *fail quick*, i.e., the message issues an exception which in a generator (like MontiCore) normally leads to erroneous termination of the generation process.

The provided methods are systematically derived from the underlying container class `java.util.Optional`: (1) The name of a method is composed of the name of the underlying method, e.g., "isPresent", and a suffix derived from the name of the respective attribute in a capitalized form, e.g., "B". (2) The method is just delegating to the respective method in the container class. This principle also is applied systematically to the other containers described below.

List Attribute. The `java.util.List` interface provides more than 30 methods for retrieving data from the list or for manipulating this data. When interested in hiding the

list attribute and for comfort, it is indeed useful to systematically translate the list methods into a signature for access and modification of the list attribute under consideration. Therefore, Listing 11.9 shows all 30+ methods that are directly mapped into the signature of a class containing a list attribute `c` here of type `List<Person>`.

```

1  void clearC();
2  boolean addC(Person element);
3  boolean addAllC(Collection<? extends Person> collection);
4  boolean removeC(Object element);
5  boolean removeAllC(Collection<Object> collection);
6  boolean retainAllC(Collection<Object> collection);
7
8  boolean containsC(Object element);
9  boolean containsAllC(Collection<Object> collection);
10 boolean isEmptyC();
11 int sizeC();
12
13 void addC(int index, Person element);
14 boolean addAllC(int index,
15                 Collection<? extends Person> collection);
16 Person setC(int index, Person element)
17 Person getC(int index);
18 int indexOfC(Object element);
19 int lastIndexOfC(Object element);
20 Person removeC(int index);
21 List<Person> subListC(int start, int end);
22
23 Iterator<Person> iteratorC();
24 ListIterator<Person> listIteratorC();
25 ListIterator<Person> listIteratorC(int index);
26 void forEachC(Consumer<? super Person> action);
27 Spliterator<Person> spliteratorC();
28 boolean removeIfC(Predicate<? super Person> filter);
29 void replaceAllC(UnaryOperator<Person> operator);
30 void sortC(Comparator<? super Person> comparator);
31
32 Person[] toArrayC(Person[] array);
33 Object[] toArrayC();
34 Stream<Person> streamC();
35 Stream<Person> parallelStreamC();
36
37 boolean equalsC(Object o);
38 int hashCodeC();
39
40 List<Person> getCList();
41 void setCList(List<Person> c);

```

Listing 11.9: Methods for a List attribute

Again, all names are derived by adding the attribute name after the name of the underlying method taken from the List interface.

However, for readability, sometimes an additional trailing `s` is added to a method name, especially when the argument is itself a list of several objects.

Because all functions are directly available on that interface, it will not be necessary to retrieve the list object directly. Still, with the last two methods `getList` and `setList`, it is also possible to directly handle the list.

11.3.2 Association Access Pattern

Unidirectional Association. If an association is unidirectional, then it is implemented as an attribute of appropriate type. Dependent on the cardinality, it is

- a normal (mandatory) attribute with normal `get/set` methods,
- wrapped in an `Optional` attribute with the same access methods as described above, or
- its values are collected in a `List`, `Set`, or `Map` attribute which can be accessed and manipulated using a generated set of functions which is systematically derived from the `java.util.List/Set/Map` classes.

The signature to access and modify these attributes has been discussed above, and especially Listings 11.8 and 11.9 show how the signature for a given unidirectional association looks like. Therefore, from the implementation view, unidirectional associations and attributes of the above types have the same access and manipulation signatures.

Bidirectional Association. If an association is bidirectional, then a natural form of implementation is to have attributes and access/manipulation functionality on both sides. So from a usage point of view bidirectional associations behave like unidirectional associations and it is especially relevant that they provide the same access and manipulation methods.

While storing an association on both sides is an efficient and also well usable effective implementation, it also introduces redundancy. Encapsulating this redundancy appropriately allows preventing the association from becoming inconsistent. As described in [Rum16], this means that each manipulation function needs to call the appropriate manipulation on the other side of the association to keep it consistent. I.e., for developers one call of a manipulation function is sufficient and the association remains consistent on both ends. As a consequence, the developers still have the same interface available and do not have to care about consistency issues.

However, it is necessary to note that associations with cardinality restrictions could need even more restrictive manipulation methods. It might also be interesting to restrict the ability to manipulate an association to one side and allow to navigate only on the other side of the association.

11.3.3 The Extended Builder Pattern

MontiCore uses an extended version of the *builder pattern* originated from [GHJV94]. While the classic builder pattern concentrates on methods to set values for their attributes even in the building process (1) the already defined values need to be retrievable and (2) especially containers like lists and sets need methods for incremental construction. A classic builder pattern does not provide any assistance for these two problems and thus forces the developer to maintain values in addition.

MontiCore's *extended builder pattern* uses the same method signature as shown above for respective attributes, with one slight difference. While above the manipulation functions often have void as a result, the respective manipulation functions of a builder return the builder object itself and thus allow chaining of manipulation calls as usual. Based on attributes a, b, and c in class Group, the GroupBuilder has access functions exactly like the Group class, but the manipulators look like shown in Listing 11.10.

```

1 class GroupBuilder {
2     // manipulating attribute: Person a
3     GroupBuilder setA(Person a);
4
5     // manipulating attribute: Optional<Person> b
6     GroupBuilder setBAbsent();
7     GroupBuilder setB(Person a);
8
9     // manipulating attribute: List<Person> c
10    GroupBuilder clearC();
11    GroupBuilder addC(Person element);
12    GroupBuilder addAllC(Collection<? extends Person> collection);
13    GroupBuilder removeC(Object element);
14    GroupBuilder removeAllC(Collection<Object> collection);
15    GroupBuilder retainAllC(Collection<Object> collection);
16
17    GroupBuilder addC(int index, Person element);
18    GroupBuilder addAllC(int index,
19                        Collection<? extends Person> collection);
20    GroupBuilder setC(int index, Person element);
21    GroupBuilder removeC(int index);
22    GroupBuilder forEachC(Consumer<? super Person> action);
23    GroupBuilder removeIfC(Predicate<? super Person> filter);
24    GroupBuilder replaceAllC(UnaryOperator<Person> operator);
25    GroupBuilder sortC(Comparator<? super Person> comparator);
26
27    GroupBuilder setCList(List<Person> c);
28 }

```

Listing 11.10: Manipulation methods provided by builders

Because manipulators return the GroupBuilder itself, chaining becomes possible: `b.setA(p1).setB(p2).addC(p3).addC(p4)`. For this convenience, however, several other results, such as boolean for a successful adding or the added object itself, are

sacrificed. Unlike the real object, a builder does not enforce consistency when attributes are changed, only during the `build()` consistency is enforced.

This mechanism for attribute access is implemented in many data classes of the RTE, especially into class `ASTNode` and its generated descendants as shown in Section 5.7 and discussed in Sections 5.8 and 5.9.

11.4 Template Hook Pattern

The *template-hook* pattern (often also called *template method* [GHJV94]) is one of the simplest and most basic patterns. We explicitly mention it here because it exhibits its strengths in object-oriented programming languages. It is not only useful for frameworks, but especially also for the integration of handwritten and generated code, while keeping both sorts of code separated in individual artifacts.

Please note that "template" in this section does not refer to FreeMarker templates, but to Java methods.

The pattern in its simplest form consists of two methods: the *template* method and the *hook* method. The template contains a predefined algorithm and calls the hook for executing more primitive or specific actions. The hook, however, is empty and meant for redefinition in subclasses. The pattern may be applied several times, using a method sometimes as template and sometimes as hook. Chains of hooks may also occur. See e.g. [Pre95, FPR01, GHJV94] for a more detailed discussion. Two variants are partially shown in Figure 11.11:

- Defaults for the hooks exists vs. keeping the hook abstract,
- Template and hook are implemented in the same class vs. the template delegates to the hook in a different class (object).

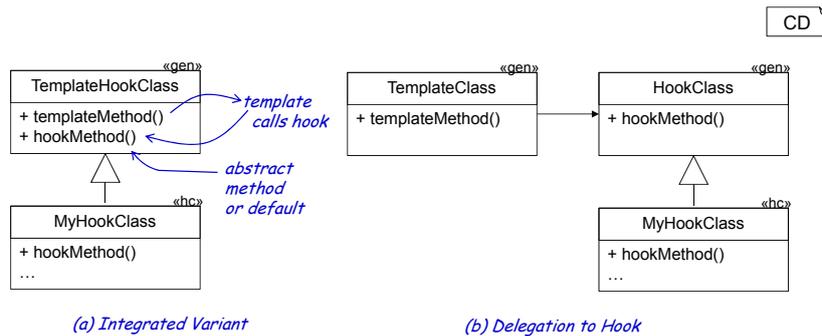


Figure 11.11: Template hook pattern variants that can be used for integrating handwritten and generated code

A conjunction of core functionality and delegates of some basic actions to hook methods is a key mechanism in frameworks, where the template method belongs to the framework and the hook method is meant to be defined by the individual application via subclassing respective framework classes (or implementing framework interfaces).

In some cases generated code is intended to be extended by handwritten code or at least an option to extend generated by handwritten code should be provided. In this case, providing such hooks leads to flexibly extensible generated code. That means, a generated piece of code is designed similar to a framework providing various hook methods.

MontiCore extensively uses the TOP mechanism to integrate handwritten code with generated parts of the program (see Section 14.3). The created TOP-classes are subclassed with handwritten classes and thus allow to override parts. Thus, almost all methods in a generated class can act as hook methods. This is an advancement compared to the *generation gap* pattern [Vli98]. The TOP mechanism also uses subclassing for handwritten code, but the generator is sensitive to the code and renames the generated superclass, such that the handwritten class not only adapts the implementation, but also extends the signature.

MontiCore also applies the template-hook design pattern to FreeMarker templates using the hook point mechanisms explained in Section 13.5.

11.5 Mill Pattern to Assist Composition

MontiCore extensively assists composition of language components. To enable composition as well as extensibility and reuse of code for components, the handwritten or generated code needs to ensure several properties. Because this problem is not only relevant for a language workbench, but in general for composable components, this section describes the *mill design pattern* that has been created to solve the following problems:

1. The data structure needs to be extensible, which means subclasses can be injected allowing the extension of the data structure, without any need for change of the provided functionality of a component.
2. Component functionality must still be usable in a compositional setting, even if it creates new objects, e.g., when manipulating the AST or the symbol table.
3. Black-box reuse of component functionality must be the standard case. That means that component functionality normally has not to be adapted for the extended, composed case, neither does it have to be recompiled, but can be reused as is.
4. Simple access to relevant functionality.

These requirements, and especially 2, are tricky because they prevent any component functionality to directly use a static creation method of an object, i.e., a constructor, because in a composed setting, an appropriate subclass has to be instantiated instead of the statically known class. The appropriate solution is to use a builder (respectively a factory) object. However, the builder also needs to be created and because of the compositionality requirement, not only the data objects but also the builders must be extensible and replaceable through subclasses. As a consequence, the builders must also not be created through a static constructor call. We need a *factory for the builders*, which we call *mill*.

In the following, we explain the mill pattern and mill composition on the example of the language mills that MontCore uses. Figure 11.12 shows the mill of a composed language G2 and the mill of one of its components G1, where we assume nonterminal Foo is defined in G1 and adapted in G2 and Bar is added in G2.

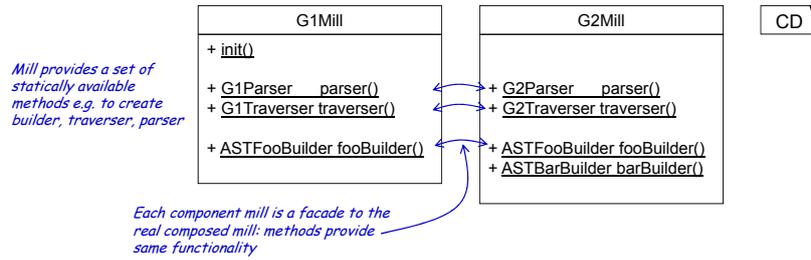


Figure 11.12: Publicly available interfaces of the mill pattern for object creation

Mill provides functionality. In the example, the `mill` is used as a source to create *builders* as described in the above Section 11.3.3, e.g., for `ASTFoo`, which in turn then creates the real `ASTFoo` objects. The mill also creates builders for symbol and scope classes.

And the mill also creates functional objects from a component that may change behavior when composed. In MontiCore, these are, e.g., the parser, traversers, and builders of a language. In MontiCore, the *mill* of a language component is therefore the general access point for the infrastructure generated for the language component as well as all subcomponents.

Mill as static access point. The mill pattern combines the generally available static access to functionality while retaining their adaptivity, because it is based on the *static delegator pattern* described in Section 11.1.

Mill allows overriding of the static functionality. The static delegator pattern internally delegates from the static method to a protected instance method that actually executes the desired creation function. In our example, `G1Mill.parser()` actually delegates to method `_parser()` of the `G1Mill` object. This instance is protected and managed by the mill itself. So from the outside, developers do not have to cope with this instance at all. However, if desired, developers may inject their own version using the `initMe(.)` as described below.

Mills in a Composed Setting. The mill pattern is designed in such a way that it is particularly suitable for composition of components. For this purpose, it uses the facility of the static delegator pattern to inject mill objects of mill subclasses to override functionality behind the statically available functions.

That means in the example that all static mill functions of `G1` are still available and can be used. This is particularly useful for functionality that is programmed against a component, like `G1`. In MontiCore, this is used, e.g., for `AST` nodes like `ASTFoo` in Figure 11.13 that is defined in language `G1` and overridden in language `G2`. Thus, `G1Mill.fooBuilder()` actually instantiates a builder for class `G2.ASTFoo`. The parser retrieved by `G1Mill.parser()` now delivers a `G2Parser` object. As discussed in Section 7.9, these extension mechanisms, however, only work safely in MontiCore, if the composed language `G2` is a conservative extension of component language `G1`.

For example, `G1Mill.parser()` and `G2Mill.parser()` deliver the same resulting object because all the static methods in the various `*Mill` classes are only facades to the same composed mill behavior.

Initialization of the Mills in a Composed Setting. To ensure the appropriate mill to be in operation, it is in a *composed setting necessary to instantiate the composition mill exactly once*. For this, the composed mill offers an `init()` method and none of the component mills need to be instantiated in addition. The `init()` method initializes the internal instances of the mill and all its component mills.

This is a strong form of initialization because it really configures all mills appropriately to produce only objects that belong to the composed language even if accessed through a mill facade of a component. Please note that in MontiCore, `init()` will always override all initializations of all components. So really, only one initialization is useful at the very beginning of the program. In another realization of such a pattern, `init()` could recognize that there was already an initialization and behave appropriately.

Mills in a Composed Setting. Because composition may involve multiple components, the mill mechanism cannot be realized by using inheritance but needs the typical application of the delegation pattern to establish a kind of multiple inheritance from several components.

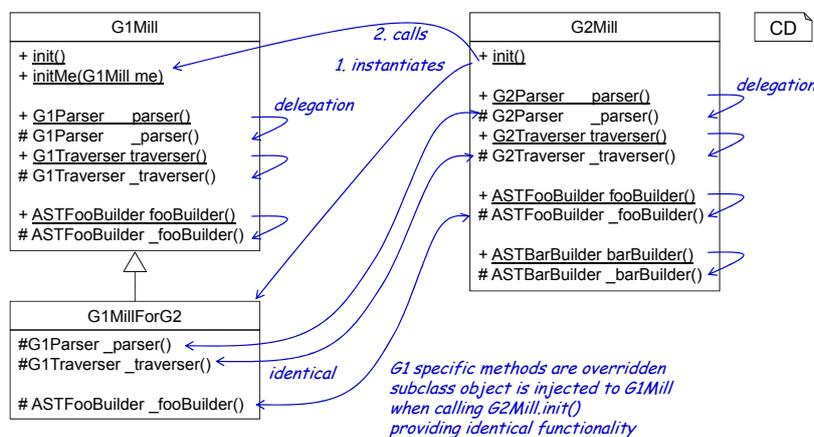


Figure 11.13: Internal Structure of the Mill Pattern

A composed mill does include all functions of its component mills. That means the composed mill (object) is the same behind all the component mills (objects). Internally the composed mill object is still instantiated as a singleton and all component mills are replaced by delegating objects. The MontiCore generator, however, slightly optimizes this by expanding the function content directly, which means that `G1MillForG2` and `G2Mill` share a number of identical method implementations as shown in Figure 11.13.

11.6 Multiple Interface Composition Pattern

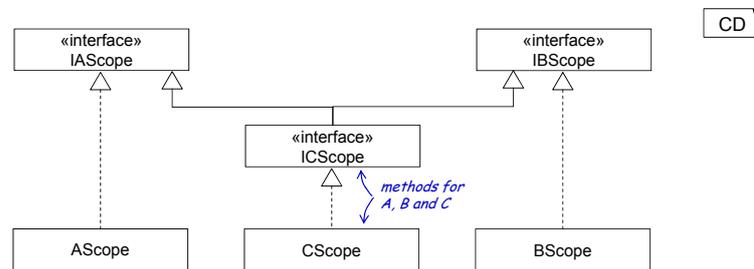


Figure 11.14: Interface Pattern for language composition

The *Multiple Interface Composition Pattern* addresses the absence of multiple inheritance for Java classes. While Java has good reasons to not provide multiple inheritance on classes, this sometimes enforces that a multiple inheritance mechanism has to be realized on interfaces to allow to reuse some functionality, while at the same time overriding it in subclasses (i.e., subinterfaces).

The MontiCore generator uses this design pattern on several occasions, which is systematically applicable, because usually all these classes and interfaces are generated. This design pattern is especially necessary to allow to inject handwritten code, for example, by the TOP mechanism as described in Section 14.3, because it allows to modify some functionality of a generated class and still inherit this modification to subclasses, i.e., into composed languages.

An example using this pattern was shown in Chapter 9. Scope classes are generated for each language. Since languages can be composed of many sublanguages, this means that the implementation of the scopes or the resolving has to be composed as well.

The idea of this pattern is that the implementation is carried out as much as possible within the interfaces by default methods. If attributes are needed, they are mapped to abstract getter and setter methods, which are then realized in concrete subclasses together with the actual attribute. As shown in Figure 11.14, for each component A, B, and C, an individual interface is created. The multiple inheritance is realized by the inheritance of the interfaces. In addition, there is a class for each language that implements the interface of the language.

This pattern ensures that the potentially overriding implementations of the compositions (subinterfaces) are reused. As said, this is compatible with the TOP mechanism, when adaptations via TOP mechanism are only made to the interfaces. If this is respected, these adaptations are also passed on to composed languages.

Chapter 12

FreeMarker for Code Generation

co-authored with Robert Eikermann

The *generator engine* in the backend of MontiCore produces textual files from internal representations of the abstract syntax. MontiCore uses the freely available generator engine FreeMarker [Fre21], which can easily be customized using templates. This feature allows to write pieces of code with holes in it, which will be filled upon generation. MontiCore currently uses FreeMarker's version 2.3.28¹. FreeMarker's version history is available online on the Freemarker homepage².

This chapter explains some essentials about the FreeMarker template language, which are especially helpful for using FreeMarker when integrated in MontiCore. FreeMarker has much more functions described in this tutorial [Fre21].

The various APIs that MontiCore provides for use within FreeMarker are described in the next Chapter 13 including AST access, template calls, or adaptation of templates.

12.1 The FreeMarker Template Languages

FreeMarker stores its templates as files with extension ".ftl". Templates describe the general structure of the target to be generated in combination with an *expression* and a *control* language that are evaluated when the templates are being processed.

FreeMarker was chosen, because it is comfortable to use, flexible, and easily integrable into larger projects. This section concentrates on a number of basic mechanisms that FreeMarker provides, independent of MontiCore. Listing 12.1 shows a first example producing the result shown in Listing 12.2, when applied with the `ast` variable pointing to information about a `Person` class.

```
1 <!-- we assume variable ast is pointing towards an AST object FTL  
2     containing class information (this is a FreeMarker comment)  
3 -->
```

¹Status Mar 2020

²http://freemarker.org/docs/app_versions.html

12. FreeMarker for Code Generation

```
4 package ${ast.packageName};  
5  
6 <#assign cname = ${ast.name}>  
7 /* Definition of a standard ${cname}Factory (a Java comment) */  
8 public class ${cname}Factory {  
9  
10     protected static ${cname}Factory f = null;  
11  
12     protected ${cname}Factory() {}  
13  
14     public static ${cname} create() {  
15         if (f == null) { ... }  
16     }  
17 }
```

Listing 12.1: Principle of FreeMarker: Copy the template content, execute FreeMarker commands, and inject their results into the output

```
1 package de.myOwnPackage; Java «gen» PersonFactory  
2  
3 /* Definition of a standard PersonFactory (a Java comment) */  
4 public class PersonFactory {  
5  
6     protected static PersonFactory f = null;  
7  
8     protected PersonFactory() {}  
9  
10    public static Person create() {  
11        if (f == null) { ... }  
12    }  
13 }
```

Listing 12.2: Result when applying the template

FreeMarker simply copies everything written in the target language, such as HTML-commands, Java, etc. to the output. However, it is sensitive to

- expressions enclosed in `${ast.name}`,
- control directives, such as `<#if ...>`, and
- variable assignments with `<#assign ...>`, and
- comments like `<#-- ... -->`

12.2 Expressions in FreeMarker

Expressions are almost ordinary *Java expressions* that can use accessible variables (see Section 13.4.4 on variable management) and can contain method calls to underlying Java


Tip 12.3: Real Java is not fully equal to FreeMarker's Java

When Java is also the target for a generation process, it is important to differentiate between the target and the expression Java language.

FreeMarker's Java expressions enclosed in `${...}` are executed when generating the target Java. Target language Java is not interpreted, but only copied to the result. This is compiled only later and, if not erroneous, finally executed in the product.

To avoid confusion, the user is advised to use disjoint sets of variable names.

objects. When an expression is evaluated, the result is transformed into a string by the method `toString()`. The resulting string is inserted in place of the expression.

A FreeMarker expression can contain Java method calls and also attribute access. For example `${ast.name}` at first tries to apply the get-method `${ast.getName()}` to the object `ast`. Only if this method does not exist, the FreeMarker engine attempts to directly read the attribute name.

Please note that FreeMarker was designed as part of a web server and thus tries to be robust by continuing to operate, even if the variable is not assigned to an object or the attribute does not exist. Sometimes, this conflicts with the idea of quick failure of a generator, which runs non-interactive and should not deliver a result in case a failure exists during the generation process. Thus, FreeMarker is operated in a mode that detects errors and assists the fail quick principle.

In addition to standard Java, FreeMarker provides its own data types:

- String: `"Hello World"`
- Number: `15` or `-3.7`
- Boolean: reserved words `true` and `false`
- Date: `"10/25/1995"?date("MM/dd/yyyy")`
- Hash: `{"name":"mouse", "price":50}`
- Sequence: `["foo", "bar", 12.3]`

For complex data structures a sequence or a hash container with key-value pairs can be used. Sequence elements as well as hash keys or values can be arbitrary objects and their types may differ. Sequence and Hash are thus untyped, but otherwise very similar to Java's List and Map.

To manipulate data, FreeMarker provides a number of built-in functions. They extend the Java syntax in form of `exp?functionname`. The function must exist for the data type that `exp` evaluates to. Some examples for built-in functions are:

- `"abcde"?substring(1,3)` evaluates to `"bc"`
- `"abcde"?cap_first` evaluates to `"Abcde"`

- `"abcde"?contains("de")` evaluates to `true`
- `2.6?round` evaluates to `3`
- `["c", "a", "b"?first` evaluates to `"c"`
- `["c", "a", "b"?seq_index_of("a")` evaluates to `1`
- `["c", "a", "b"?sort` evaluates to `["a", "b", "c"]`

The following convenient functions deal with the date of execution:

- `${.now}` evaluates to, e.g., `02.04.2027 23:46:06`
- `${.now?date}` evaluates to, e.g., `02.04.2027`
- `${.now?time}` evaluates to, e.g., `23:46:06`

Other functions allow to examine whether a variable is defined or provides a default if it is not. Some examples are:

- `varName??` is `true` when `varName` is defined,
- `varName!` evaluates to the empty string if `varName` is undefined,
- `varName!"John"` evaluates to default `"John"`, if `varName` is undefined,
- `(obj.varName)!"John"` evaluates to the given default `"John"` if `obj` is undefined, `varName` attribute does not exist or does not have a value (`null`).



Tip 12.4: Use Additional FreeMarker Functions with Care

FreeMarker provides a lot of functions [Fre21]. The FreeMarker manual can be found here:

<http://freemarker.org/docs/>

However, extensive data manipulation should be implemented in Java directly as Java is much better suited for programming. A drawback, on the other hand, is that recompilation of the entire tool is necessary once changes are applied to the Java code. Modifications of the FreeMarker templates do not require a recompilation.

12.3 Control Directives in FreeMarker

FreeMarker directives allow to control the execution of a template similar to an ordinary programming language. This includes the usual control constructs, such as case distinction, loop, switch-statement and variable assignment. FreeMarker also allows to call other templates and the definition of custom functions. MontiCore, on the contrary, replaces these options by customized versions (cf. Section 13.4.2) and suggests to define new functions only in Java.

Directives are generally enclosed in tags of the form `<#directive parameters>` and `</#directive>` or they just consist of a single tag. The following directives are predefined in FreeMarker:

- Variable declaration: `<#assign name=value>`
- Variable access: `#{name}`

An example for a conditional is given in Listing 12.5. The conditional shown in the code snippet will be evaluated to `4 is smaller than 8`. The conditionals are expressions evaluating to `true`, while cases are expressions that evaluate to strings.

```

1 <#assign i=4>
2 <#assign j=8>
3 <#if (i==j) >
4   #{i} and #{j} are equal
5 <#elseif (i<j) >
6   #{i} is smaller than #{j}
7 <#else>
8   #{i} is bigger than #{j}
9 </#if>
```

Listing 12.5: FreeMarker conditional

An example for a switch case is given in Listing 12.6. The template will evaluate to the String `Size is neither small nor big`.

```

1 <#assign size="medium">
2 <#switch size>
3   <#case "small">
4     Size is really small <#break>
5   <#case "big">
6     Size is really big <#break>
7   <#default>
8     Size is neither small nor big
9 </#switch>
```

Listing 12.6: FreeMarker switch statement

The loop directive can be used to iterate over a FreeMarker sequence or a Java list. The definition of a loop directive is shown in Listing 12.7 where `item` is the loop variable. Inside the `<#list>` directive, two special variables are automatically available. The variable `item_index` holds the current index number of `item` in the sequence. The variable `item_has_next` is true, if and only if `item` has a successor in the sequence.

```

1 <#list sequence as item>
2   text-body with variables
3     item:           value in current iteration of the loop
4     item_index:    index number of the current item
```

12. FreeMarker for Code Generation

```
5     item_has_next: true if not at the end of the sequence
6 </#list>
```

Listing 12.7: FreeMarker loop

In Listing 12.8 this is applied to an object `p` of class `Person` which has a `children` attribute of type `List<Person>`. Java lists are handled like FreeMarker sequences. Loops can also be nested. Listing 12.9 shows an extended form with various extras.

```
1 Children of ${p.firstName}:
2 <#list p.children as child>
3     ${child_index + 1}) ${child.name} born in ${child.age}
4 </#list>
```

Listing 12.8: Example for a FreeMarker loop

```
1 <#list sequence-expression>
2     text-header executed once if we have items
3     <#items as item>
4         text-body repeated for each item
5     </#items>
6     text-footer executed once if we have items
7 <#else>
8     alternate text executed when sequence is empty
9 </#list>
```

Listing 12.9: Extended form of a FreeMarker loop

Besides these, FreeMarker provides more directives. For readers of the generated source code, `<#compress> ... </#compress>` might be interesting to look up. This construct produces texts with condensed white spaces. Unfortunately, starting white spaces are completely omitted and thus the directive does not format produced code according to Java's indentation guidelines.

12.4 FreeMarker Add Ons

FreeMarker unfortunately is completely untyped and heavily relying on reflection, which can lead to bad or sporadic forms of errors during runtime that are difficult to identify and heal. The availability of the robust generator framework, however, and that it easily runs in batch tools were two arguments to integrate it into MontiCore. Freemarker's template errors occur at development time and thus need not be handled by end users.

But we also have added some precautions e.g. using a signature statement, discussed in Section 13.4.2 that adds some additional safety.

```
1  ${signature("stimuli",  
2      "className") }  
3  
4  public abstract class ${className} {  
5  
6  <!-- Add the list of stimuli as method calls -->  
7  <#list stimuli as stimulusName>  
8      void handle${stimulusName?cap_first} (${modelName} sc) {  
9          ...  
10     }  
11 </#list>
```

Listing 12.10: signature asserts variables to be defined

Listing 12.10 shows a *parameterized template* that can be used to map state machines to Java code. The template enforces in ll. 1 that it is called with two parameters (see Section 13.4.2) that are stored in the two local variables `stimuli` and `className`. Please note, that this only ensures that the variables do have a value, but not what kind of value it is. For example the usage of the two variables in l. 4 respectively l. 8 further demand that `className` is a `String` and `stimuli` is a list of `Strings`. More variables may exist, for example the global variable `modelName` is used in the template body as well, but not mentioned in the signature on purpose.

Chapter 13

Generator Engine using Flexible Templates

The *generator engine* is used to produce textual files from internal representations of the abstract syntax and thus is the last step in a typical generation process. MontiCore uses the freely available generator engine FreeMarker [Fre21] for its backend, because FreeMarker provides a rich and flexibly adaptable infrastructure for templates and assists extensions well. The FreeMarker language has already been described in Chapter 12. In addition to this, this chapter explains:

- How to use the FreeMarker template engine in a MontiCore based tool.
- The API MontiCore provides for usage within FreeMarker templates, including AST access, calls of other templates, or adaptation of the call structure between template.
- The hook point mechanism to flexibly adapt the generation process.
- How this adaptation flexibility can be used demonstrated by an illustrative example at the end of this chapter.

13.1 Methodical Considerations

Template engines are a powerful and flexible mechanism and thus are suited for generation of code or many other forms of possible output, such as documentation or webpages.

However, there are other possibilities to generate code. One could manually write the print statements that produce the target artifacts while traversing the AST, e.g., assisted by visitors. This is often sufficient when pretty printing the AST only. In this case, a collection of templates executing the pretty print is not necessary.

However, it is typically not sufficient to pretty print the AST many times, but additional outputs should be generated which are not available in the input. This is the case for example when adding access methods for attributes or when generating whole new classes such as builders or visitors.

In the following, we use the term *conceptual distance* in an informal way. The more concepts of the source language are present in the target, the smaller is the amount of work to map between the language. The more concepts of the source language are not available

in the target, the more complex the mapping will become. It would be nice to have a measurement that quantifies such a conceptual distance between languages. The conceptual distance between source AST and target language typically also increases necessities that the recursive descent along the source AST tree structure must be interspersed increasingly often. The *conceptual distance* between the available AST and the target language is to a large extent responsible for the complexity of the templates and in particular for the complexity of the template call structure. In most cases, the AST structure and the output order fit relatively well (and e.g., 100% in pretty printing situations). In case different files are generated but the AST structure is well-suited for the templates used, it is sufficient to execute these different template sets from the top level. However, in case the generation process needs information from different parts of the AST, additional infrastructure for calculations based on the AST is helpful. Especially the symbol table typically is used for navigation shortcuts from the usage position of a symbol to its definition or the symbol itself carries extra information, such as how to access or modify the item represented by the symbol. For example local variables are mapped to relative stack addresses in compilers.

If the conceptual distance, however, becomes too large, it is advisable to transform the AST of the input language to an AST that is better suited for the output language (if not directly the output language itself).

In case the output is Java, it is advisable to use a target AST that is directly Java or conceptually similar to it such as class diagrams. A good balance between conceptual similarity (i.e. only a small conceptual distance) and simplicity of the internal transformation is desirable.

As an example: The MontiCore generator processes grammars (cf. Chapter 4). To produce the output it maps the grammar to a class diagram of the language CD4Code [Rot17] internally. CD4Code provides classes, attributes, associations and method signatures. Missing method bodies are implemented in templates and attached to the respective method signatures using hook points (see Section 13.5). This allows to write method implementations in a rather compact and understandable form within templates, yet ensures that method signatures and attributes are only generated once, preventing potentially uncompileable code.

Templates are *untyped*. On the one hand this offers a lot of flexibility, but on the other hand it is a burden, because it raises the number of potential errors. Fortunately, when generating into a solid compilable language, such as Java, the language compiler detects quite a number of those errors. It would be much better, if the template engine itself would detect errors early, but something like *static typing* of templates is still a research direction in its infancy.

If the number of templates becomes larger and the interaction between templates becomes more complex, we advise to explicitly describe the template, e.g., in its header. We argue that each template has a *template signature*. This signature should contain:

- What is the result: The result can typically be described by a nonterminal of the target language, potentially equipped with a cardinality. Possible characterisations could be `Type`, `ImportStatement*` or `Attribute?`.

- What is the input (arguments): Which variables are defined and which elements do they carry. In MontiCore, the `ast` variable is mostly expected to point to a valid AST node. Each template typically expects a certain type (i.e. nonterminal of the input language) for the `ast` variable (which varies between the templates as the `ast` pointer describes the recursive descent). More variables may be expected, either in the local or in the global variable store (see Section 13.4.4).
- Templates may expect an array of additional template arguments that are passed to the template when being called (see e.g. Sections 13.2, 13.4).
- Which explicit hook points are provided (see Section 13.5).
- Does the template open new files and thus write its content to a new file or does it assume the target file is already open. This usually concurs with the resulting nonterminal of a template.
- Which templates are used by the template. This is important because (1) all used templates need to be shipped together with the template and (2) each template call creates further (implicitly defined) hook points that can be used for adaptation (see Section 13.5).

In addition, we suggest to separate the list of used templates into groups: (1) Templates that produce output and assume that the target file is already open and add content to it, and (2) templates that control the output files that should be created. The latter do not directly produce output, but concentrate on controlling the AST traversal and call other templates that create the output, using the `write` methods of the template controller as explained later in this chapter. Furthermore, special templates can be used for configuration, i.e., assigning values to variables, but that can also be managed through the Groovy interface (see Section 16.5) or of course directly within Java.

As an alternative, it is always possible to organize the control of what to be written within the Java part itself and use templates only to describe e.g. the structure of a target class or the body of specific methods. Using Java for the control is typically more robust, but less flexible, because changes of what is to be generated and which additional hooks are to be bound must then be organized within Java and a recompilation of the tool is needed. To reduce this challenge, we have also added Groovy as configuration language in Section 16.5.

13.2 Generator API

To start the generation process in MontiCore and process the templates, we use the `GeneratorEngine` class.

There are several parameters that can be configured to adapt the generation process. Because quite a number of these parameters are rather stable and reusable in the generation process, we use the class `GeneratorSetup` as a configuration class for the `GeneratorEngine`. The `setup` is passed to the generator engine as a parameter of the constructor. `GeneratorSetup` contains various configurable attributes, such as the form

13. Generator Engine using Flexible Templates

of comments, the file I/O object, the paths to templates or handcoded files, the output directory, and more. See Section 13.3 for a detailed description.

The `GeneratorEngine` offers several `generate` (and `generateNoA`) methods with varying signatures. All methods called `generateNoA` do not have an instance of an `ASTNode` as a parameter, where as `generate` methods have one. In total the following parameters are provided to run a generation process:

String `templateName` provides the qualified template name that shall be executed. It is assumed that the template produces a complete artifact as output. The template name is searched in the list of provided template paths.

Path `filePath` is the artifact name that is to be created upon executing the template. It is recommended that the file path is relative to the configured output directory specified in the `GeneratorSetup`, but `filePath` could also be an absolute path.

Writer `writer` is an alternative to the `filePath`. Here, the file or a string writer is already opened.

ASTNode `node` is the starting point, i.e., the AST that the templates act on.

Object... `templateArguments` allow to provide additional arguments to the initial template, which can be accessed after the `signature` method is executed in the template. The number and form of arguments highly depends on the individual template and should be explained in the template header itself (see Section 13.1).

The other parameters are usually provided when calling a `generate` method. Listing 13.1 shows the signature of six `generate` methods, which can be used as the starting point for the generation. The different names are necessary to distinguish if an `ASTNode` is passed as explicit value for the variable `ast`.

```
1 class GeneratorEngine { Java GeneratorEngine
2
3   public GeneratorEngine(GeneratorSetup gs)
4
5   void generate(String templateName,
6                 Path filePath,
7                 ASTNode node,
8                 Object... templateArguments);
9
10  void generateNoA(String templateName,
11                  Path filePath,
12                  Object... templateArguments);
13
14  void generate(String templateName,
15                Writer writer,
16                ASTNode node,
17                Object... templateArguments);
18
19  void generateNoA(String templateName,
20                  Writer writer,
```

```

21         Object... templateArguments);
22
23     StringBuilder generate(String templateName,
24                           ASTNode node,
25                           Object... templateArguments);
26
27     StringBuilder generateNoA(String templateName,
28                              Object... templateArguments);
29 }

```

Listing 13.1: Signature of a generate and generateNoA method

The first `generate` method (ll. 5f) opens a file and executes the given template on the AST node. It incorporates two flexibility mechanisms: (1) The template replacement discussed in Section 13.5 may interfere with the template execution and (2) the TOP mechanism explained in Section 5.10 may adapt the output filename as well as potential content (like the class name).

The additional arguments `templateArguments` can be empty. MontiCore passes those arguments to the template and with the special operator `signature` they become arguments available in the called template. See Section 13.4 for details. The second, fourth and sixth `generateNoA` methods (ll. 10f, 19f, 27f) omit the AST node argument, assuming that the template called does not rely on it. The `generate` methods (ll. 14-19) do not open a file, but use the `writer` argument to write to an already open file. The last two methods (ll. 23-27) also do not open a file, but return the produced text in form of a `StringBuilder`. These methods can also be used, when the called template is a control template (instead of an output template), which will open a file itself and the control template does not produce any useful output.

Please note that all `generate` methods have a template name as one of its arguments, which is subject for a possible replacement and extension as described in Section 13.5.

In Listing 13.2, we demonstrate how the `GeneratorEngine` can be created and used. The `GeneratorSetup` acts as the storage of the configuration of the generator engine. In the case shown, the output directory for the generated files is defined in l. 2. Lines 3-5 define the start and end marker to be used for generated comments and switch the tracing of templates on. If tracing is on, at the beginning of the code produced by a template a comment is added that states the templates name. For details see Section 13.3.

Afterwards, an instance of the `GeneratorEngine` is created (l. 7). With this instance the `generate` method is called for two templates (ll. 9) to generate two files.

```

1  GeneratorSetup s = new GeneratorSetup();
2  s.setOutputDirectory(new File("gen"));
3  s.setCommentStart("/*-- ");
4  s.setCommentEnd(" --*/");
5  s.setTracing(true);
6
7  GeneratorEngine ge = new GeneratorEngine(s);
8

```

Java

```

9   ge.generate("tpl/DemoStateMachine1.ftl", Paths.get("demo1"), ast);
10  ge.generate("tpl/StateMachine.ftl", Paths.get("pingPong.aut"), ast);

```

Listing 13.2: How the GeneratorEngine can be used

13.3 Configuring the Generation Process

In Listing 13.2 we have already seen that MontiCore provides some options to configure the generation process. This configuration is typically defined within Java or could be defined in a Groovy script (see Section 16.5). The special configuration class `GeneratorSetup` contains all relevant information and is best explained by showing its attributes.

Listing 13.3 shows the attributes that are stored in the generator setup to configure potential output.

```

1  class GeneratorSetup {
2      File                outputDirectory;           // def: "out"
3      String              defaultFileExtension;     // def: "java"
4      IterablePath       handcodedPath;           // IterablePath.empty()
5      List<File>          additionalTemplatePaths;  // def: empty
6      boolean            tracing;                  // def: true
7      String              commentStart;            // def: "/*"
8      String              commentEnd;              // def: "*/"
9      Optional<String>    modelName;               // def: Optional.empty()
10     GlobalExtensionManagement glex;              // defaults
11     FreeMarkerTemplateEngine freeMarkerTemplateEngine; // exist
12
13     // and a configuration method
14     TemplateController getNewTemplateController(String templateName);
15 }

```

Listing 13.3: Configuration options of the GeneratorSetup class

The `GeneratorSetup` allows to set each configuration attribute independently. If the attribute is not set, defaults apply.

Therefore, each configuration attribute has an accessor and mutator having the same name as the attribute but with a `get` and `set` prefix.

- `outputDirectory`: The output directory specifies where the generated files are placed. If none is specified, the directory `out` is used by default in the current path. (Also compare this to the `-o` argument when calling the CLI discussed in Section 16.1.2).
- `defaultFileExtension`: Files that shall have a default extension do get this string as suffix of their filename. In case no explicit file extension is given as an argument the default is used during execution of a template that creates a file using the `write` method. The default is `java`.

- `handcodedPath`: This is a list of directories, where handwritten classes can be found. As explained in Chapter 14, MontiCore allows handwritten replacements for the generated Java classes. It detects handwritten classes in any of the specified paths during code generation and adapts the generated sources. By default this path is empty and has to be manually configured (cf. CLI argument `-handcodedPath`; see also Chapter 14).
- `additionalTemplatePaths`: This is a list of paths where additional templates can be found that will be used for code generation. These templates can be injected into the generation process by either replacing or adding a template using the `glex` mechanism. These additional templates are considered during code generation if the path containing the templates is listed in the `additionalTemplatePaths` variable. These additional paths need to be added manually, because by default an empty path is configured (cf. CLI argument `-templatePath`).
- `tracing`: When tracing is enabled, the generated files will contain information on which template contributed which piece of the generated artifact. This information will be added in form of comments before the template content is executed. It is especially helpful when debugging a code generator, but generally leads to unreadable artifacts and contributes to performance as well as storage overhead. By default, tracing is enabled.
- `commentStart`: If, e.g., tracing is enabled, this string describes the begin of a comment in the target language. Because the MontiCore code generator may target different target languages, the comments to be generated have to be defined. Using the mutator for the `commentStart` the start symbol(s) for the comment can be defined. By default the comment start is `"/**`. Please note that the comment start can be for a single line comment, but then the comment end should contain a newline.
- `commentEnd`: Besides the start of the comment, the end of a comment has to be defined. This variable stores the symbol(s) for the comment end, which is by default `**/`.
- `modelName`: The source model name is printed in a comment in the generated code if tracing is turned on. By default the model name is absent and then the according tracing info is not printed at all. The model name is optional and has no default.
- `glex`: The `GlobalExtensionMangement` manages hook points, global variables, and template replacements. In particular, it allows to create and bind hook points as well as global variables. For templates it is allowed to replace existing ones or add a template before or after an existing template. See Section 13.4.4. The default for an unset `glex` is an instance of class `GlobalExtensionManagement`.

Reuse of the `glex` object allows even to reuse global variables that may be defined or changed in one `generate` call, and accessed or changed in a subsequent call. If not used with care this may unfortunately also be a source of multiple definition of global variables which leads to an error.

- `freeMarkerTemplateEngine` is the instance of the FreeMarker template engine that will be used. The default for an unset configuration attribute is an instance of class `FreeMarkerTemplateEngine`.

In addition, the generator setup provides the method `getNewTemplateController`, which iteratively creates new `TemplateController` objects – one for each template execution. This method is meant for overriding, when a different `TemplateController` class should be used.

13.4 MontiCore APIs for Templates

Adapting existing templates or writing new templates requires an understanding of how to access the AST, the symbol tables and other helpful functionalities within the templates. The available Java data structures are defined in Chapter 5, but custom extensions are possible. Additional Java APIs of the tooling are available allowing the template developers to access auxiliary functions of various forms within the templates.

MontiCore provides an elaborated API to support generator developers. There are standard objects and methods that are directly available within the templates through the variables `tc`, `glex`, and `ast`, respectively through the statically available methods in class `Log`:

- `TemplateController tc` provides typical template operations and template-specific information (see Section 13.4.2).
- `Log` provides static methods for logging, warning and error methods (see Section 13.4.3).
- `GlobalExtensionManagement glex` manages global variables and the handling of extensions and hook points (see Section 13.5).
- Variable `ast` allows access to the processed model. It points to the currently processed node of the abstract syntax (i.e., the internal representation of the model discussed in Chapter 5). The type of variable `ast` changes, and the available methods are therefore dependent on the currently processed node.

The objects (respectively class `Log`) provide various methods that will be discussed in the following sections. Throughout all templates `glex` always refers to the same `GlobalExtensionManagement` object. In contrast, `tc` holds information specific to each template and is thus instantiated on each template invocation, but always with an object of type `TemplateController`. Finally, the templates can process different parts of the model, and hence, `ast` contains the corresponding AST node that changes while processing different parts of the AST usually but not necessarily for each template.

13.4.1 Shortcuts: Aliases in Templates

For the sake of convenience, the MontiCore template engine provides aliases for methods that are often invoked within a template. Thus, the objects `tc` and `glex` can often be omitted when accessing their functions from templates.

Table 13.4 shows the signatures of these aliases. As a rule, we favor the alias version of a method instead of its long version since it improves the readability of the

templates. So, for example, we write `${include("my.Template")}` instead of `${tc.include("my.Template")}`.

Table 13.4: Aliases provided for templates

Alias	Expanded Command
Include and read signature commands, see Section 13.4.2	
<code>include(template)</code>	<code>tc.include(template)</code>
<code>include2(tpl, ast)</code>	<code>tc.include(tpl, ast)</code>
<code>includeArgs(tpl, par...)</code>	<code>tc.includeArgs(tpl, par...)</code>
<code>signature(par...)</code>	<code>tc.signature(par...)</code>
Warnings, errors, infos ..., see Section 13.4.3	
<code>error(message)</code>	<code>Log.error(message)</code>
<code>warn(message)</code>	<code>Log.warn(message)</code>
<code>info(message, logger)</code>	<code>Log.info(message, logger)</code>
<code>debug(message, logger)</code>	<code>Log.debug(message, logger)</code>
<code>trace(message, logger)</code>	<code>Log.trace(message, logger)</code>
Global variable management, see Section 13.4.4	
<code>defineGlobalVar(name, value)</code>	<code>glex.defineGlobalVar(name, value)</code>
<code>changeGlobalVar(name, value)</code>	<code>glex.changeGlobalVar(name, value)</code>
<code>addToGlobalVar(name, value)</code>	<code>glex.addToGlobalVar(name, value)</code>
<code>getGlobalVar(name)</code>	<code>glex.getGlobalVar(name)</code>
<code>requiredGlobalVar(name)</code>	<code>glex.requiredGlobalVar(name)</code>
<code>requiredGlobalVars(name...)</code>	<code>glex.requiredGlobalVars(name...)</code>
Hook point management, see Section 13.5	
<code>bindHookPoint(name, hp)</code>	<code>glex.bindHookPoint(name, hp)</code>
<code>defineHookPoint(name)</code>	<code>glex.defineHookPoint(tc, name)</code>
<code>defineHookPoint(name, ast)</code>	<code>glex.defineHookPoint(tc, name, ast)</code>
<code>defineHookPointWithDefault(name, default)</code>	<code>glex.defineHookPointWithDefault(tc, name, default)</code>
<code>defineHookPointWithDefault3(name, ast, default)</code>	<code>glex.defineHookPointWithDefault(tc, name, ast, default)</code>
<code>existsHookPoint(name)</code>	<code>glex.existsHookPoint(name)</code>

Please note that class `Log` needs to be qualified with the package if explicitly accessed, that means `de.se_rwth.commons.logging.Log` needs to be used. For this case, the shortcuts are helpful. Because `FreeMarker` is untyped, unfortunately overloading of methods is not possible. Therefore, sometimes numbers are added to methods, e.g., `include2` with two parameters.

The list of aliases is stored in the template

```
de.monticore.generating.templateengine.freemarker.Aliases
```

It is called by default at the beginning of each generation process and can be overridden

by using the `replaceTemplate` mechanism when desired, e.g., to add more aliases.

13.4.2 The `TemplateController`

The `TemplateController` provides methods for typical template operations such as the inclusion of sub-templates or the instantiation of further auxiliary classes as helpers.

The `TemplateController` additionally provides access to template-specific information, for example, the name and the package of the current template. Consequently, every template execution holds a new `TemplateController` object, which can be accessed through variable `tc` within the template.

It is possible to adapt the template controller by defining a subclass of `TemplateController`. For a repeated instantiation of this class the factory method `getNewTemplateController` needs to be adapted in a subclass of `GeneratorSetup`.

Including Templates without Arguments



Tip 13.5: Template Names Pointing to Files

In many methods, Strings are used as names for templates. Those template names are qualified names that point to the file containing the template. They could be fully qualified, but normally only define the *package* they can be found in. In the latter case, they are looked for in the *template path*.

The qualifier path can be defined like a Java style package name, like `"tp14.F.ftl"`, and also without default extension, like `"tp14.F"`.

It is also possible to use a pathname of the underlying operating system, such as `"tp1/F.ftl"` in Unix. However, `"tp1/F"` does not work.

To include sub-templates into a template, the `include` methods are used. Their signatures are shown in ll. 3-12 of Listing 13.6. By calling `tc.include(templateName, ast)`, the template `templateName` is processed on the AST node `ast`. The result is included into the current output, i.e., the corresponding position of the template, where the call was issued. If the included template works on the same `ast` object (or the `ast` is just not of interest), we can use the method in ll. 11-12 as a shortcut. If the methods are called with lists of templates or lists of AST nodes, then a method applies all templates of the first list on all nodes of the second list. This mainly acts as a shortcut for iterative application. If both arguments are lists, this is equivalent to having two nested loops where the outer loop iterates over the templates and the inner loop over the AST nodes.

```
1 class TemplateController {
2
3   StringBuilder include(String      templateName,
4                       ASTNode    ast);
5   StringBuilder include(List<String> templateNames,
```

Java TemplateController

```

6         ASTNode         ast);
7     StringBuilder include(String         templateName,
8                           List<ASTNode> astlist);
9     StringBuilder include(List<String>  templateNames,
10                          List<ASTNode> astlist);
11    StringBuilder include(List<String>  templateNames);
12    StringBuilder include(String         templateName);
13 }

```

Listing 13.6: Include methods provided by the `TemplateController tc`

Please note that the `include` commands are subject for substitution by the hook point mechanism as described in Section 13.5.

The example in 13.7 shows how to use the `include` methods within FreeMarker templates. The `include` methods are used from within templates by generator developers, but these methods could also be called from within Java source code.

```

1     `${include("my.Template")}`
2     `${include(["a.Template1", "a.Template2"])}`
3     `${include2("my.Template", ast.getOneChild())}`
4     `${include2(["a.Template1", "a.Template2"], ast.getSomeChildren())}`

```

Listing 13.7: Examples for including sub-templates within a template

In general, these methods replace and improve the template inclusion mechanism that FreeMarker provides by a better management of variables and template hook points. Thus, we ignore FreeMarker's own template inclusion and use that of MontiCore.

Including Templates with Explicit Arguments

A second group of `include` methods uses a slightly different approach for variable passing. The `includeArgs` methods allow us to call a new template and pass a list of arguments as additional parameters. Listing 13.8 shows their signatures.

```

1 class TemplateController {
2     StringBuilder includeArgs(String templateName,
3                               ASTNode node,
4                               List<Object> templateArguments)
5     StringBuilder includeArgs(String templateName,
6                               List<Object> templateArguments)
7
8     StringBuilder includeArgs(String templateName,
9                               String... templateArgument)
10
11    void signature(List<String> parameterNames)
12    void signature(String... parameterName)
13 }

```

Listing 13.8: The `includeArg` methods provided by the `TemplateController tc`

The `includeArgs` methods accept one template to be executed, optionally an explicit `ast` node and a list respectively array of (untyped) arguments. Within the called template only the variables `glex`, `tc` and `ast` are available. The `ast` variable contains the currently processed AST object of the calling template if it is not explicitly given as argument.

The list of further arguments is only implicitly passed to the called template. To make this implicit list explicit and accessible through variable names, there exists a method called `signature` provided by the template controller. This method can be used inside the called template and allows a template designer to describe what the additional parameters of a template are and initializes these parameters. This is a workaround to deal with the problem that FreeMarker itself does not allow to declare parameters and pass arguments. The `signature` method should be one of the first commands of a template: It defines a part of the *signature* of the template, because it lists the names of the variables (parameters) where the arguments shall be stored in. The parameter list must have as many entries as the arguments. This is checked at runtime and leads to an error if the number of arguments is wrong. Unfortunately, no type checking happens which may lead to errors when using these variables within the template.

The `signature` method can only be called once per template and stores variables locally only. The `signature` can be omitted, if no argument is passed. An empty list is also possible to clarify that no extra arguments are expected.

Please note that the `TemplateHookPoint` class discussed in Section 13.5 also allows to add parameters. These parameters are passed to the template with the same mechanism: The first list of parameters comes from the Java method `includeArgs` and the second part of the list comes from the `TemplateHookPoint` constructor.

The example in Listing 13.9 shows a template call and the `signature` command that would bind the variables `ast` to the caller's child, the string `prefix` to the variable `"text1"`, and the string `postfix` to the `"text2"`. In the second part, the content of the variable `ast` is bound to the value `32+10` and `i` becomes `42`. Unfortunately, the variable names need to be enclosed in quotation marks.

```
1 // Call of a template FTL  
2 `${tc.includeArgs("my.Template", ast.getAChild(), "text1", "text2")}`  
3  
4 // Line 1 of the called "my.Template"  
5 `${tc.signature("prefix", "postfix")}`  
6 // binds variable prefix to String "text1", ...  
7  
8 // Call of another template  
9 `${tc.includeArgs("my.Template2", 32+10)}`  
10  
11 // Line 1 of the called "my.Template2"  
12 `${tc.signature("i")}`  
13 // binds variable i to value 42
```

Listing 13.9: Examples for using `signature`

This form of template calls introduces more flexibility and also a better form of reuse, as it allows to avoid passing information along globally defined variables. It mimics the spirit of ordinary method calls between Java methods, although it does not provide the advantages of static typing.

Please note that if a template is replaced or decorated using the hook point mechanism, then the same form of argument passing occurs, which means that the replacing or decorating template has to have the same signature as the replaced template. The only exception may be that the `TemplateHookPoint` adds additional arguments, which enlarges the parameter list in the replacing template accordingly.



Tip 13.10: Template Signature: Parameters

The `signature` method can be used within templates to describe which parameters need to be set, when executing the template with the `includeArgs` or `writeArgs` methods.

It checks correctness of the number of arguments of the call and assigns the arguments to the listed parameters.

This is not a full type check, but at least provides some safety and comfort, because it mimics traditional parameterized method calls.

Writing Results of Template Executions to Files

The `write` methods, e.g., Listing 13.11 (ll. 2), are used to create complete artifacts. These methods process the template `templateName` and stores the result in the newly created file `fileName.extension`. `fileExtension` can be `null` or the empty string `""`. If the `fileExtension` does not start with `."`, but is not empty, a dot is inserted. So `".java"` and `"java"` have the same effect. Unless not absolute, the `fileName` (respectively `filePath`) is relative to the configured target directory.

The template filename may be qualified (using `."`). In case it is not qualified, the qualifier is taken from the current package (same as the calling template).

Other versions of `write` methods use the default extension (ll. 5) or an already defined `Path` object (ll. 8).

The file is opened, content is written, and the file is closed. For this purpose, the `write` methods use the class `FileReaderWriter`. This class is implemented as static delegator, which means it can be adapted, and handles a number of additional tasks, such as checking, which handcoded files are existing, and recording for the reporting described in Section 15.5. It is recommended to use the `write` methods, respectively directly the `FileReaderWriter` in all occasions to ensure proper recording of activities - and efficiency when using builds incrementally (like in `make` or `gradle`).

```
1 class TemplateController {
2   void write(String templateName,
```

Java TemplateController

13. Generator Engine using Flexible Templates

```
3         String qualifiedFilename, String fileExtension,
4         ASTNode ast);
5     void write(String templateName,
6         String qualifiedFileName,
7         ASTNode ast);
8     void write(String templateName,
9         Path filePath,
10        ASTNode ast);
11
12    void writeArgs(String templateName,
13        String qualifiedFileName, String fileExtension,
14        ASTNode ast,
15        List<Object> templateArguments);
16    void writeArgs(String templateName,
17        Path filePath,
18        ASTNode ast,
19        List<Object> templateArguments);
20 }
```

Listing 13.11: Write methods provided by the `TemplateController tc`

The `writeArgs` versions (ll. 12-19) also allow developers to pass additional arguments to the template. In this respect they behave like the `includeArgs` methods.

The main difference between the `include` methods described above and the `write` methods is that the latter open files and write contents to them. The `write` methods are therefore the entry points for code generation and usually called from Java. The `GeneratorEngine` uses these methods.

In principle, it is possible to write to an artifact, while one artifact is already being written to, i.e. `write` into a new file can be called within templates that contribute to other files. However, nested writing processes may be difficult to understand.



Tip 13.12: Controlling Templates

It is possible to use templates for controlling the output. Such a controlling template contains variable definitions, some control decisions and `write` commands, but does not itself produce text.

The advantage of a controlling template is that it can be adapted without touching Java files and thus without recompilation of the tool. Unfortunately, Java is better suited for complex control algorithms. Other possibilities would be a detailed Groovy script for output control or the use of hook points for extension or replacement.

For manageability, controlling templates and producing templates should be strictly separated and clearly marked.

More Methods in the `TemplateController`

The `TemplateController` object `tc` provides some more methods. An overview is given in Listing 13.13.

```

1 class TemplateController {
2
3   String getTemplatename();
4
5   Object instantiate(String className);
6
7   Object instantiate(String className, List<Object> params);
8
9   boolean existsHandwrittenFile(String fileName);
10
11  boolean existsHandwrittenFile(String fileName, String extension);
12 }
```

Listing 13.13: Further methods provided by the `TemplateController tc`

- `getTemplatename` (l. 3) allows to retrieve the name of the template.
- `instantiate` (ll. 5-7) allows to instantiate a Java class from its name. If the name is not qualified, the same package as the calling template is used. Qualification is dot-separated. The version in line 5 assumes a constructor without arguments, while the version in line 7 allows to set the arguments of a constructor.

The `instantiate` methods allow to create additional objects that can for example be used as helpers. If stored in a local or global variable, they extend the API by providing additional methods to access Java AST, symbols, or other structures to be used in templates. Please note that this is a reflective mechanism and should be handled with care, because it has no compile-time time check, but can fail at runtime.

- `existsHandwrittenFile` (l. 9-11) check whether a file exists in the handcoded path. This allows a template to react on whether a handcoded class exists (and generate something different). If the `extension` is omitted as second argument, the default extension is taken.

13.4.3 Logging within a Template

Error management and logging are always important components for a helpful tooling. The details of logging are defined in Section 15.3. This section especially introduces statically available methods in class `Log` that can also be used from templates.

As a shortcut a subset of the logging API (described in Section 15.3) is directly available through aliasing within templates. Hence, the example shown in Listing 13.14 is intended for template developers who use logging information in templates.

```

1 // error and warning go to stdout
2 ${error("0x12345 A critical error occured.")}
3 ${warn("AST value is empty, skipping template ...")}
4
5 // infos, debug and trace have additional component names
6 ${info("Starting template.", "component-name")}
7 ${debug("Value of node is " + ast.getValue(), "component-name")}
8 ${trace("Generating line 5.", "component-name")}

```

Listing 13.14: Logging examples from within templates

As shown in Listing 13.14, this API allows to issue log messages with five different levels of severity in descending order. While the higher severity levels `error` and `warn` are used to signal critical events or failures, the lower level severities are used for information. The distinction between these different levels allows to control the verbosity of the actual logging output. Log messages can be filtered according to their severity level and per component. This is what the second parameter of the log API for `info`, `debug`, and `trace` is for. Issuing an error leads to *immediate termination* of the generation process, as we strictly follow the *fail quick* policy when the generation cannot be completed successfully.

13.4.4 Variables in the Templates with GlobalExtensionManagement

Two kinds of variables are available in the templates: local and global variables. *Local variables* are only visible in the scope of the template that defines them, whereas *global variables* are stored globally, hence, can be defined and accessed in any template as well as from the underlying Java.

Local variables can be defined and assigned using the built-in `assign` directive that FreeMarker offers (see Section 12.1). Because FreeMarker does not offer a global variable management, MontiCore provides the `glex` (`GlobalExtensionManagement`) object that allows to define and manipulate variables that are visible in template executions.

Global variables should be used rarely, because they are shared and thus can have unexpected side effects. To reduce unwanted side effects the `GlobalExtensionManagement` class provides functionality to define and access global variables and handle them as if they were constants. Often these variables are used to access additional Java objects that help generating from the AST or symbol infrastructures or contain additional template paths.

To set, change or retrieve a global value one of the methods in Listing 13.15 can be used.

```

1 public class GlobalExtensionManagement {
2     void defineGlobalVar(String name, Object value);
3
4     void changeGlobalVar(String name, Object value);
5     void addToGlobalVar(String name, Object value);
6
7     boolean hasGlobalVar(String name);
8     Object getGlobalVar(String name);

```

```

9   Object getGlobalVar(String name, Object default);
10
11  void requiredGlobalVar(String name);
12  void requiredGlobalVars(String... names);

```

Listing 13.15: Methods to manage global variables with `glex`

`defineGlobalVar(name, value)` defines a new global variable called `name` and assigns it the value `value`. If the variable is already defined, an error is issued. Because FreeMarker is untyped, values generally are of type `Object`. `changeGlobalVar` replaces the value of an already existing global variable. If the value does not exist yet an error is logged, so check whether the global variable is present, before trying to change it is necessary.

`addToGlobalVar(name, value)` assumes that the argument for `name` refers to a global variable of type `List` and adds the argument for `value` to the list. This is convenient, e.g., when building up a list of templates that shall later be executed and thus allows some kind of configuration within the templates themselves. Variable name needs to be initialized with a list, like `defineGlobalVar("name", [])`.

`hasGlobalVar(name)` checks if a global variable exists (boolean) and `getGlobalVar` returns the value of the global variable. If the global variable does not exist then either a default is provided as a second argument, or it exits with exception to facilitate the fail quick policy.

A value of a global variable can be used in many ways. By calling `requiredGlobalVars`, we can require global variables to be defined. If a variable does not exist an error is thrown during execution. Using this early in templates defines a weak form of precondition for template execution. Together with the `signature` command, `requiredGlobalVars` defines a second form of input signature.

While all the above methods can be called from Java, they can also be called within a template. Listing 13.16 demonstrates this.

```

1  {glex.requiredGlobalVar("v3")}
2
3  {glex.defineGlobalVar("v1", 33+2)}
4  Var v1 is {glex.getGlobalVar("v1")}
5
6  <#if glex.hasGlobalVar("v1")>
7  Ok.
8  {glex.changeGlobalVar("v1", "Aha")}
9  </#if>
10
11 {glex.defineGlobalVar("v2", [])}
12 {glex.addToGlobalVar("v2", 16)}
13 {glex.addToGlobalVar("v2", [18, 19])}
14 {glex.addToGlobalVar("v2", 17)}
15 <#list glex.getGlobalVar("v2") as elem> {elem}, </#list>

```

Listing 13.16: Manipulating global variables from within a template

Please note that the global variables are the same within all template executions, and thus allow to transport data from Java to the templates and between templates calling each other. It even allows to share data between different generator calls, if the `GlobalExtensionManagement` object `glex` remains the same.

13.5 Hook Points for Adaptation

Sometimes a code generator does not deliver the optimal form of code, e.g. if additional generated functionality is desired, a generated modifier shall be adapted, or additional annotations shall be attached to generated attributes. Therefore, it is helpful if a generator provides mechanisms for adaptation of the generator and thus of the generated code.

MontiCore provides a flexible mechanism for generator adaptation that is based on *hook points* in templates.

13.5.1 The Concept of Hook Points

A *hook point* is a place within a template that is meant for adaptation [Rot17]. A hook point can either be defined explicitly or exists by default for decorating or replacing a called template. If a hook point is not explicitly bound to a value it defaults to an empty string.

A *hook point* consists of a *name*, which is a unique string that identifies the place in the template, where to hook in, and a *value* that is bound to the hook point name. The hook point is defined explicitly by giving it a *name* or implicitly, because every template itself acts as hook point name.

Explicit hook points in templates are therefore providing the same adaptation power as hook methods in a programming language [Pre95]. Furthermore, *decorator hook points* are used to add code before or after a template and act like decorating aspects [KLM⁺97]. Figure 13.17 shows a classic hierarchical calling structure of templates. Here, the caller knows and includes the called template via the `include` statement. Each template called serves as a hook point. Please note the direction of the arrows. The resulting text on the right side exhibits the execution order of the template parts.

Figure 13.18 demonstrates the effects of decoration with hook points. The `include("B")` in template A provides the option to hook in a template before or after the included template B. The effect is demonstrated by hooking in the template C before and the template D after the template B. The result is shown on the right, the start of A is printed first. The decorated `include` results in D being printed next, followed by B and then D. Finally, the rest of A is printed.

It is particularly important to notice that the decoration is defined outside the affected templates A and B. Both templates neither have explicit knowledge about the hooks they are decorated with, nor need to be changed. Only their execution is adapted. This gives developers the possibility to add decorations later in the development process and especially

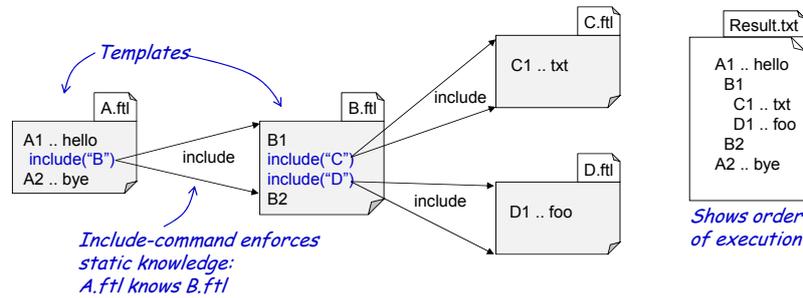


Figure 13.17: Hierarchical include structure induced by the include commands. The knowledge direction goes from A to B to C and D

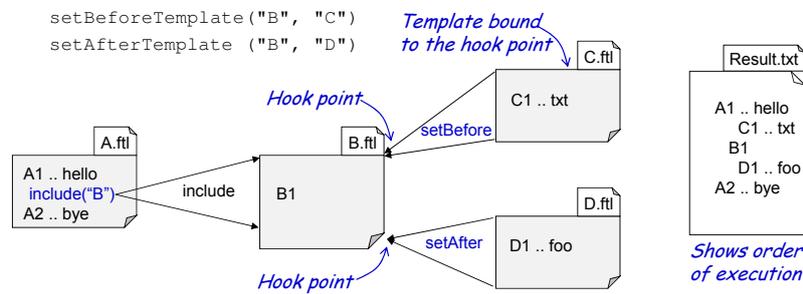


Figure 13.18: Decoration before and after a template. The knowledge direction is inverted: C and D know B

allows to adapt a generation process (defined in A, B) that is already fixed in a library without having to change A, B directly or make a copy/paste adaptation.

There are two kinds of hook points:

1. A hook point can be *explicitly defined* within a template (see Section 13.5.3).
2. Each *template* acts as an implicitly defined hook point, through its template name, which can be used to decorate or replace the template.

Figure 13.20 shows the effect of a *template replacement*, where the template name is used as hook point name.

 **Tip 13.19: Kinds of Hook Points**

Hook points can be *explicitly defined* within a template introducing an explicit *hook point name*, but also each *template itself* (by its qualified name) acts as an implicitly defined hook point.

Both kinds of hook points can be used for non-invasive adaptation of the generation process.

Again the template A includes the template B. Besides the hook points before and after an included template, the `include` itself can be used to hook in a different template and thus replacing the originally included template. In this example, the original template B is replaced by the template E. As a result, the content of B is not printed, only the content of E is.

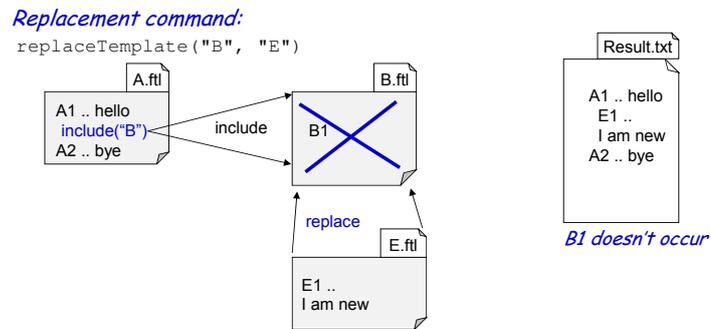


Figure 13.20: External replacement of a template

Figure 13.21 shows the effect of an explicit definition and binding. This time the template A defines an explicit hook point P. The command in the upper part of the figure is neither part of A nor of B, but from externally binds the template B to the explicit hook point P. This results in the content of B being printed at the position of the hook point P.

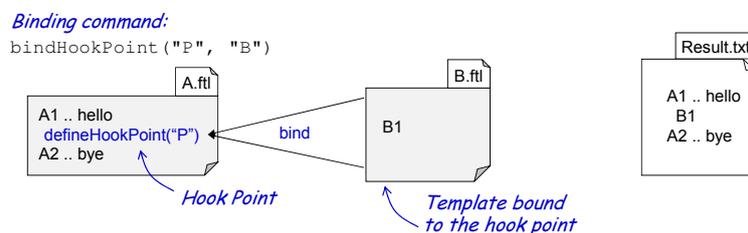


Figure 13.21: Defining an explicit hook point and binding it

An even more specific replacement is possible such that a replacement is only defined for some AST nodes the templates is called for: We use the paired combination of *template and an AST object* the template acts on as hook point name (i.e. more precisely "*identifier*") and allow an individual decoration or replacement.

Therefore, MontiCore provides the following mechanisms:

- A hook point can be used to *replace* an existing template (see Figure 13.20).
- Existing templates can be *decorated* with additional hook points that are executed before or after the original template (see Figure 13.18).
- For a fully fine grained output control, a template and the AST object it acts on, qualifies as hook point name and can individually be decorated or replaced.

- *Explicit hooks points* within the generation process can be filled with content (see Figure 13.21).

The paired, AST-dependent hook point replacement is rather helpful, e.g., when the output AST shall not be constructed completely, but some parts of it remain in templates. For example, method signatures can be constructed in the AST, but this results in a rather complicated AST construction. Thus, individual method bodies remain in templates and are attached individually to the corresponding AST method node.

If a hook point is bound several times (using `bind`, `setBeforeTemplate`, `setAfterTemplate`, or `replaceTemplate` multiple times), then only the last statement is effective. The last binding overrides the earlier ones.

Decoration and replacement is only effective for the explicitly included templates, such as A in Figures 13.20 and 13.21. In these figures, templates C, D, and E cannot be decorated or replaced. However, the decoration of template B in Figure 13.22 remains active, when B itself is replaced.

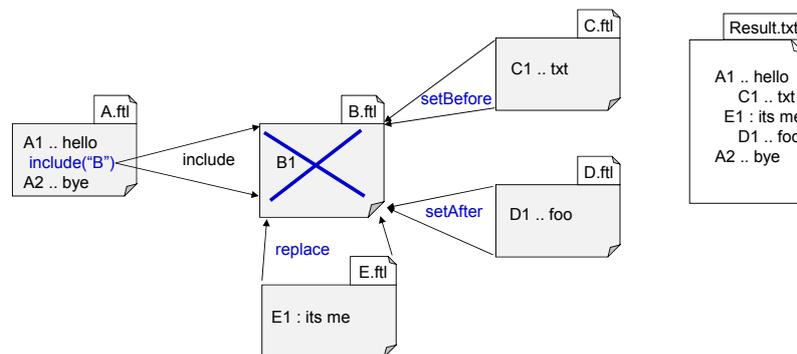


Figure 13.22: External template replacement keeps its decoration (execution order is 1..5)

It is noteworthy, that we could also replace an existing and used template by defining a new one with the same name and path, copying the new one into a local directory and including the new template in the *template path*. The new template just needs to have the same name (including package) and the storage place must be mentioned before the overridden old one in the template path. However, this process is brittle and somewhat difficult to understand, when you do not fully control the order of the template path.

It is also noteworthy, that direct handcoded adaptations of the resulting Java code are possible, but affect only single artifacts and are thus somewhat limited in their power. Such adaptations are described in Chapter 14.

Adding functionality through template adaptations can be challenging, because the developer requires knowledge about the template structure, the target and the generator language, the generator architecture, and the generated artifact architecture.

Configuring template replacements and hook point bindings can be done before the generation is started or during the generation. Using Java code for this enforces recompilation.

Alternatively, templates can themselves configure the use of other templates, as explained in Section 13.5. Or the MontiCore Groovy script is adapted accordingly (see Section 16.5).

13.5.2 Forms of Hook Points

So far we have discussed filling hook points with templates. However, MontiCore also provides the following forms of hook point values that allow to inject a normal string, an inline template or even Java code. A hook point can be filled by one (or if allowed several) *hooks*. We distinguish between four types of hooks that define hook point values:

1. *String hook*: The hook point is filled with a simple, uninterpreted string value.
2. *Template hook*: The hook point is filled by executing a template given by a qualified template name. During code generation this template is loaded, executed and the result is added to the template defining the hook point.
3. *Template-string hook*: Like the template hook, but the template content is already in the string, i.e., a string that contains FreeMarker expressions. During code generation this template string is evaluated and the returned string is embedded in the template defining the hook point. No file is loaded.
4. *Code hook*: The hook point is filled with a code hook, namely the value resulting from its execution. This is the most powerful type of hook point. A code hook is a Java class that can be used to implement additional functionality. During code generation this Java class is executed and the returning string is inserted in the template that defines the hook point.



Tip 13.23: Hooks in Object Oriented Programming

Hook points share properties with hook methods in OO programming frameworks, where the hook is defined as empty method and is meant to be overridden in a subclass. But, do not use hook points over excessively, because this complicates understanding the execution order of templates and hooks.

Each of these forms of hooks are defined by a subclass of `HookPoint`. Its signature is shown in Listing 13.24.

```
1 package de.monticore.generating.templateengine;
2 public abstract class HookPoint {
3
4     public abstract
5     String processValue(TemplateController controller,
6                       ASTNode ast);
7
8     public abstract
9     String processValue(TemplateController controller,
10                      List<Object> args);
11
12     public abstract
```

Java HookPoint

```

11 String processValue(TemplateController controller,
12                     ASTNode node,
13                     List<Object> args);
14 }

```

Listing 13.24: Signature that HookPoints provide

The subclasses that implement the forms of hooks are:

1. StringHookPoint implements a *string hook*, i.e., an uninterpreted string value.
2. TemplateHookPoint implements a *template hook*, i.e., a template given by qualified name is interpreted. All additional arguments are handed over to the called template using the signature command.
3. TemplateStringHookPoint accepts an inlined template text, i.e., no file is involved, but the text is executed as template content.
4. CodeHookPoint is itself an abstract class and is intended to be subclassed by all forms of *code hooks*.

The first three classes are meant for direct use. They are instantiated by one of the constructors shown in Listings 13.25- 13.27.

```

1 public class StringHookPoint extends HookPoint {
2     // constructor of StringHookPoint
3     public StringHookPoint(String value);
4 }

```

Listing 13.25: Constructor for StringHookPoints

```

1 public class TemplateHookPoint extends HookPoint {
2     // constructors of TemplateHookPoint
3     public TemplateHookPoint(String templateName);
4
5     public TemplateHookPoint(String templateName,
6                             Object... templateArguments);
7 }

```

Listing 13.26: Constructors for TemplateHookPoint

```

1 public class TemplateStringHookPoint extends HookPoint {
2     // constructors of TemplateStringHookPoint
3     public TemplateStringHookPoint(String statement)
4                                     throws IOException;
5 }

```

Listing 13.27: Constructor of TemplateStringHookPoint

The `GlobalExtensionManagement` provides access to global variables and also for hooks. In particular, template and code hooks can access and modify global variables. Parameters that are defined, e.g., in the `includeArgs` method calls, are also passed to the hook templates and can be extracted to local variables using the `signature` command.

The second constructor in class `TemplateHookPoint` allows to pass additional arguments to the template. These arguments are managed like the arguments passed to `includeArgs`. They are at first implicit in the called template, but they can be assigned to variables using the `signature` command. The list of parameter names in the `signature` consists of two parts: first the arguments of the `includeArgs` class and then the parameters from the constructor in class `TemplateHookPoint`.

This form of arguments is only necessary for `TemplateHookPoint`, because it allows to decouple the definition and the provisioning of arguments to three points: (a) definition of a reusable, parameterized template, (b) creation of the hook point, where some arguments are fixed and (c) execution of the hook point in the context of the replaced template, where the rest of the arguments is provided.

As said, hooks themselves are not subject to further replacement and thus names of template hooks do not act as hook names. In particular, the template defined in class `TemplateHookPoint` can not be replaced or decorated, but again hook points can be defined in this template and further include calls are subject for replacement and decoration.

The abstract class `CodeHookPoint` is not instantiated directly, but used by implementing subclasses. Such a subclass needs to implement three methods called `processValue` which receive all necessary information through the arguments, i.e. variables `tc` and `ast`. The methods differ in their parameters: while all have a template controller as a parameter, the first receives only the AST in addition, the second only a list of arbitrary arguments and the third the AST as well as further arguments. The `tc` contains e.g. access to the `glex` or the `GeneratorSetup` object. If a `CodeHookPoint` wants to access the `signature` arguments of a called template, it may ask the `tc` with `getArguments` to retrieve the list of objects that were passed as arguments.

Using a subclass of `CodeHookPoint` provides of course a general and mighty form of hook point definition. However, using this kind of hook points requires recompiling the tool (respectively `MontiCore`) before using it, while templates are interpreted.

13.5.3 Defining Explicit Hook Points in Templates

To simplify the adaptation and extension of templates, *hook points* can be explicitly defined in templates. A hook is a place in a template that is planned for extension. Every hook gets a *name* through its definition. If the name was bound to one of the above mentioned *hook points*, this hook point is executed and the result is inserted in the template.

To define a hook point in a template the `TemplateController` class provides `defineHookPoint` methods that are used only within templates as shown in Listing 13.28. In this listing the hook point with the name

"<JavaBlock>?TemplateName:member" is called. How hook point names like this one are constructed will be discussed later. If the hook point should work on the same ast node as the including template then the ast argument can also be omitted. If an alternative default is needed, several defineHookPointWithDefault methods allow to define what is included if the hook point is left empty.

To check if a hook point with a specific name is already existing, the existsHookPoint method is typically used within template control structures.

```

1 // defining a hook point (if it is bound,
2 // the hook point value is inserted here)
3 ${glex.defineHookPoint(tc, "<JavaBlock>?TemplateName:member", ast) }
4 ${glex.defineHookPoint(tc, "<JavaBlock>?TemplateName:member",
5                          ast.getAChild()) }
6 ${glex.defineHookPoint(tc, "<JavaBlock>?TemplateName:member") }
7
8 // either hook point value or a given default
9 ${glex.defineHookPointWithDefault(
10      tc, "<JavaBlock>?TemplateName:member", ast, "default text")}
11 ${glex.defineHookPointWithDefault(
12      tc, "<JavaBlock>?TemplateName:member", "default text")}
13
14 // to find out, whether a hook point is bound (if necessary)
15 ${glex.existsHookPoint("<JavaBlock>?TemplateName:member")}
16
17 // shortcuts with the same effect as above
18 ${defineHookPoint("<JavaBlock>?TemplateName:member", ast) }
19 ${defineHookPoint("<JavaBlock>?TemplateName:member") }
20 ${defineHookPointWithDefault("<JavaBlock>?TemplateName:member",
21                              ast, "default text")}
22 ${defineHookPointWithDefault3("<JavaBlock>?TemplateName:member",
23                               "default text")}

```

Listing 13.28: Methods to define a hook point in a template

The aliases described in Section 13.4.1 allow developers to use the shortcuts shown in ll. 18f.

Please note that it is not possible to pass additional arguments from explicit hook point definitions to the executing hook point except through global variables. Only the standard ast and tc are usually available.

Hook Point Naming Conventions

Hook points need memorable names such that developers have a unique identification. Due to a lacking *type system* for templates in general and thus also hook points, we suggest the following convention for hook point names. The following naming scheme helps developers to better recognize what to do and how to achieve the desired effect. A hook point name, like "<Block>?ClassImpl:ConstructorInit" consists of

13. Generator Engine using Flexible Templates

1. the type of the expected result (given as nonterminal `<Block>`). It is associated with a cardinality, such as `*` or `?` (default: `?`, i.e., maximum one), to describe that omission or repetition is allowed,
2. the template name in which the hook point is defined, if the hook point with that name is defined only once (here: `ClassImpl`),
3. the type of the ast where the hook point is applied to (here: `ConstructorInit`), and
4. optionally the purpose of the hook point.

This convention encodes some typing information in the hook point name: For example, a hook point is defined in the `ClassImpl` template that requires a `Class` ast, returns an arbitrary set of `JavaBlock` statements, and is defined to add class *members*. This hook point can be named with `<JavaBlock>*ClassImpl:member`.

But for convenience often used hook point names can be defined even simpler. For example, the hook point called `JavaCopyright` expects a comment containing some copyright information for each generated Java class. A call of this hook point with the simple name is shown below.

```
1  ${glex.defineHookPoint("JavaCopyright")} FTL
```

13.5.4 Binding Hook Points

Binding a hook point means to assign one or several hook point values to a previously defined hook point name. Binding is usually applied, when the hook point was defined explicitly by name and is not a template. To set a hook point the `GlobalExtensionManagement` provides the method `bindHookPoint`.

```
1  class GlobalExtensionManagement { Java GlobalExtensionManagement  
2      String defineHookPoint(TemplateController controller,  
3          String hookName, ASTNode ast);  
4      String defineHookPoint(TemplateController controller,  
5          String hookName);  
6      String defineHookPoint(TemplateController controller,  
7          String hookName, Object... args);  
8      String defineHookPoint(TemplateController controller,  
9          String hookName, ASTNode ast, Object... args);  
10  
11     String defineHookPointWithDefault(TemplateController controller,  
12         String hookName, String defStr);  
13     String defineHookPointWithDefault(TemplateController controller,  
14         String hookName, ASTNode ast, String defStr);  
15     String defineHookPointWithDefault(TemplateController controller,  
16         String hookName, String defStr, Object... args);  
17     String defineHookPointWithDefault(TemplateController controller,
```

```

18         String hookName, ASTNode ast, String defStr,
19         Object... args);
20
21     void bindHookPoint(String hookName, HookPoint hp);
22     void bindStringHookPoint(String hookName, String content);
23     void bindTemplateHookPoint(String hookName, String tpl);
24
25     boolean existsHookPoint(String hookName);
26 }

```

Listing 13.29: Methods of the `GlobalExtensionManagement` class for hook point management

The class `GlobalExtensionManagement` is not only responsible for managing global variables, but also organizes hook points in templates. Listing 13.29 describes the methods it provides for this purpose.

The effect of the methods which are called from templates, namely `defineHookPoint`, `defineHookPointWithDefault` and `existsHookPoint` are already described in Section 13.5.3.

The `bindHookPoint` method is used to bind a hook point name to one of the forms of hook points described in Section 13.5.2. This method can be used from Java as well as within templates. The convenience methods `bindStringHookPoint` and `bindTemplateHookPoint` allow to define hook points more easily from within templates. If the hook point already had a binding, a warning is issued and the hook point name gets bound to the new value, so the new value overrides the old.

Binding Hook Points in Templates

A hook point can also be bound or adapted within a template. The code examples in Listing 13.30 demonstrate this. We advise to use the shortcuts only.

```

1  ${glex.bindHookPoint("aComment1",
2     tc.instantiate(
3     "de.monticore.generating.templateengine.TemplateHookPoint",
4     ["tpl4/SE-Copyright.ftl"])}
5
6  <!-- or with these shortcuts: -->
7  ${glex.bindTemplateHookPoint("aComment2",
8     "tpl4/SE-Copyright.ftl")}
9
10 ${glex.bindStringHookPoint("aComment3",
11     "// Developed by SE RWTH\n")}

```

Listing 13.30: Example: setting a hook point

13.5.5 Replacing and Decorating Hook Points

When the hook point is a template, then replacement and decoration of the template is possible. The following methods can be applied for template names and for explicitly named hook points.

As already mentioned, this mechanism mimics aspect orientation for templates: an "*aspect*" template can decorate or replace the basic template, without the basic template knowing that its effect is modified.

The class `GlobalExtensionManagement` provides appropriate methods shown in Listing 13.31.

```
1 class GlobalExtensionManagement { Java GlobalExtensionManagement
2   replaceTemplate(String oldTemplate,
3                   HookPoint hp);
4   replaceTemplate(String oldTemplate,
5                   ASTNode node,
6                   HookPoint newHp);
7
8   setBeforeTemplate(String template,
9                    HookPoint beforeHp);
10  setBeforeTemplate(String template,
11                  List<? extends HookPoint> beforeHps);
12
13  setAfterTemplate(String template,
14                 HookPoint afterHp);
15  setAfterTemplate(String template,
16                  List<? extends HookPoint> afterHps);
17 }
```

Listing 13.31: `GlobalExtensionManagement` for hook point management

`replaceTemplate` replaces a template call (via `include`, `includeArgs`, `write`, or `writeArgs`) by the provided hook point, which is executed instead. The version of `replaceTemplate` with the three argument only replaces the template if called on a specific ast node.

The two methods `setBeforeTemplate` and `setAfterTemplate` allow to decorate templates (as well as explicitly named hook points).

This mechanism provides a powerful approach to extend a code generator. Likewise, it should be used with care, because it may provide unwanted side effects.

13.5.6 HookPoint Replacement and Decoration Strategy

Irrespective, whether the hook point name is explicitly defined or a template name, the hook point binding respectively replacement and decoration strategies are the same.

However, the strategy is complex and thus is detailed here:


Tip 13.32: Use Template Replacement And Decoration with Care

To avoid confusion with the aspect-like template decoration and replacement, developers should only either replace a template or decorate it with extensions, but not both. Template replacement is also error prone and should thus be handled with care.

If a *hook point name* `hpn` and a concrete *AST node* `ast` are given and the hook point shall be executed, then the following is calculated:

1. If `hpn` is decorated with one or more *before* hook points, then this *before* decoration is executed. This may be a list of hook points.
2. If there is a specific replacement (depending on `hpn` and `ast`), then this replacement hook point is executed.
3. Otherwise, if there is no specific replacement (depending on `hpn` and `ast`), then it is checked, whether a general replacement (dependent on `hpn`) exists. If so, then this replacement hook point is executed.
4. Otherwise, if there is no replacement at all and `hpn` is a template, then the template is executed (this typically happens within `include` and `write` methods).
5. Otherwise, if there is no replacement at all and the hook point was explicitly defined, it defaults to the empty string (this typically happens with the `defineHookPoint` method).
6. At the end: If `hpn` is decorated with one or more *after* hook points, then this *after* decoration is executed. This may be a list of hook points.

The various binding, replacement and set methods do have a variety of effects:

`bindHookPoint(hpn, hp)` binds `hp` to `hpn`. If `hpn` was already bound, the value is overwritten, but also a warning issued. `bindHookPoint` does not affect decoration.

`replaceTemplate(hpn, hp)` binds `hp` to name `hpn`. If name `hpn` was already replaced, the value is overwritten, but there is no warning issued. `replaceTemplate` does not affect decoration and does not affect specific replacements.

`replaceTemplate(hpn, ast, hp)` binds `hp` to identifier (`hpn,ast`). If name (`hpn,ast`) was already replaced, the value is overwritten, but there is no warning issued. `replaceTemplate` does not affect decoration, but setting a specific replacement has the effect that it overrides general replacement or binding.

`setBeforeTemplate(hpn, hp)` decorates the hook point named `hpn` such that hook point value `hp` is executed before. `setBeforeTemplate` does also take a list of hook points, but a second call of `setBeforeTemplate` overrides the first value. Only the last call counts. If the decorated template is also replaced, the decoration remains active.

setAfterTemplate(hpn, hp) : like `setBeforeTemplate`, but applied after the main hook point execution.

None of the binding, decoration and replacement methods is transitive. That means a template mentioned within a `TemplateHookpoint` is not a hook point name and will therefore not be further replaced or decorated.

All replacing and decorating hook points have the same signature as the basis template. In particular, `TemplateHookPoints` should have the same signature command, even though the names of the parameters might vary.

`bindHookPoint` and `replaceTemplate` have very similar effects, when applied on template names. The main difference is that `bindHookPoint` is usually applied on explicitly defined and thus otherwise empty hook point names, while `replaceTemplate` is usually applied to replace a non empty template.

If the hook point is defined explicitly with `bindHookPoint`, decoration or replacement have the same effect, as long as only one form is used, because replacement of an empty default looks like decoration.



Technical Info 13.33: How hook points are managed internally

Hook point names are just strings. This holds for template names as well as freely defined strings for explicit hook points.

The `bindHookPoint` and `replaceTemplate` (with two arguments) methods store their replacement in the same `Map<String, HookPoint>`.

`setBeforeTemplate` and `setAfterTemplate` have multimaps with type `Multimap<String, HookPoint>` to be able to store lists of hook points.

Finally, the three argument version of `replaceTemplate` stores its replacements in a map of maps: `Map<String, Map<ASTNode, HookPoint>>`. This map stores a hook point individually for each hook point name and AST node.

13.5.7 A HookPoint Replacement and Decoration Example

The mechanisms to replace hook points are powerful and can be used in various ways. We demonstrate this on a relatively simple example, where an existing generator translating automata into Java classes using the state pattern is adapted. We assume the state pattern is realized through a set of templates:

```
templates/src/main/resources/Statechart.ftl
templates/src/main/resources/StatechartStateAttributes.ftl
templates/src/main/resources/AbstractState.ftl
templates/src/main/resources/ConcreteState.ftl
```

We assume these templates together with the language defining the relevant context conditions etc. are part of a fixed library project `lib` that must be reused in the following without changing the files.

Replacing the Generation of State Attributes

Template `AbstractState` describes the implementation of the abstract superclass of all states and template `ConcreteState` is used for generation multiple times, mainly for each state of the automaton. Template `StatechartStateAttributes` describes, how the instantiated objects of these generated state classes are stored in an attribute in the generated `Statechart`. It uses a static attribute and instantiates objects for all state classes initially as shown in Listing 13.34¹.

```

1 <#assign name = ast.getName() >
2   public static _${name}State _${name?uncap_first}
3     = new _${name}State();

```

Listing 13.34: Producing attributes in the State Pattern

When applied for example to our well known `PingPong` automaton the resulting code contains among others (Listing 13.35):

```

1 public static NoGameState noGame
2   = new NoGameState();
3 public static PingState ping
4   = new PingState();

```

Listing 13.35: Resulting code for State attributes

Because the public access of internal attributes is to some extent problematic, we decide to adapt the generation. Unfortunately, the original template cannot be modified, because it is shipped in a library project. We therefore locally define a replacement template `MyStateAttributes` shown in Listing 13.36.

```

1 <#assign n = ast.getName() >
2   protected static _${n}State _${n?uncap_first}
3     = new _${n}State();

```

Listing 13.36: Modified generation template

We now can enforce the template replacement by a piece of code that can be added as Java code in the tool, in a Groovy configuration script or even in another template that is executed early in the generation process. Here, we just define a new tool class that adds the replacement (Listing 13.37):

```

1 glex.replaceTemplate("StatechartStateAttributes.ftl",
2   new TemplateHookPoint("MyStateAttributes.ftl"));

```

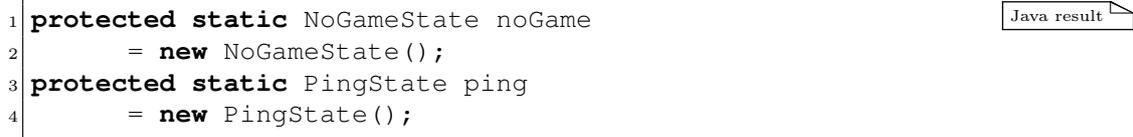
Listing 13.37: Replacing the default template

¹The example has illustrative character and is only a simplification of a really usable and well engineered design pattern.

13. Generator Engine using Flexible Templates

Now we get the desired result in which all the attributes are protected (Listing 13.38):

```
1 protected static NoGameState noGame
2     = new NoGameState();
3 protected static PingState ping
4     = new PingState();
```

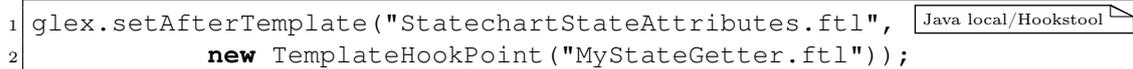


Listing 13.38: Resulting State attributes for adapted generation

Decorating the Generation of State Attributes

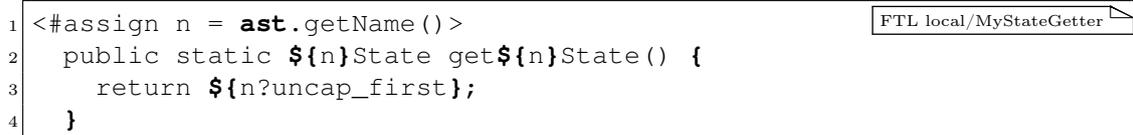
It was also decided to add an access function for each of the generated attributes. Unfortunately, the fixed Statechart template does not provide an explicit hook point for this. But because attributes and methods are defined on the same level in a class, we reuse the StatechartStateAttributes as hook point and *decorate* the template by adding a getter-function defined in Listing 13.40 by the call in Listing 13.39:

```
1 glex.setAfterTemplate("StatechartStateAttributes.ftl",
2     new TemplateHookPoint("MyStateGetter.ftl"));
```



Listing 13.39: Decorating the default template

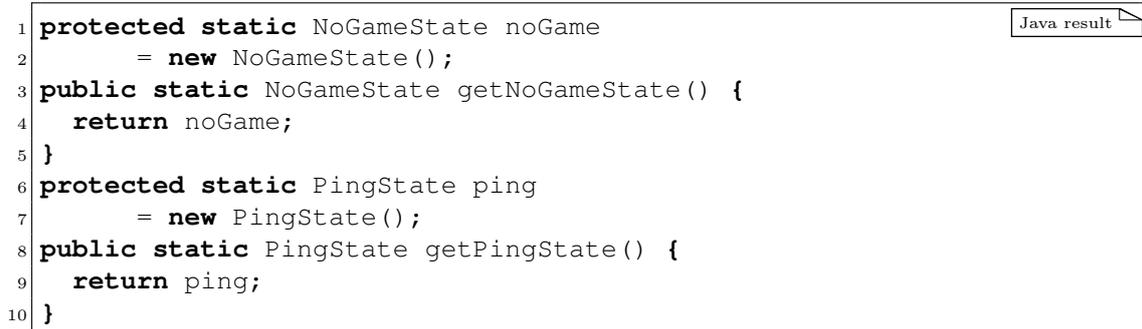
```
1 <#assign n = ast.getName()>
2     public static ${n}State get${n}State() {
3         return ${n?uncap_first};
4     }
```



Listing 13.40: Decorating template for StatechartStateAttributes

The desired result now contains protected attributes and public get-functions (Listing 13.41):

```
1 protected static NoGameState noGame
2     = new NoGameState();
3 public static NoGameState getNoGameState() {
4     return noGame;
5 }
6 protected static PingState ping
7     = new PingState();
8 public static PingState getPingState() {
9     return ping;
10 }
```



Listing 13.41: Resulting code including get functions

This example demonstrates two aspects. On the one hand, thanks to the template StatechartStateAttributes we were able to add the desired code to the generation, but on the other hand it also shows that we had to extend a template here contrary

to the original intention. When designing a template for a reusable library it is advisable to think about possible extensions and to integrate some hook points for additional elements.

Using Explicit Hook Points to Count Method Calls

It shall also be counted how often the methods of a state are called. Fortunately, the `ConcreteState` template shown in Listing 13.42 provides a number of explicit hook points, where we can add our additional pieces of code. To understand the example: Variable `outgoing` contains a map of pairs with stimulus name and respective transition.

```

1  ${defineHookPoint("<Import>*ConcreteState")}
2
3  public <#if existsHWCEExtension>abstract </#if>
4      class ${className} extends Abstract${modelName}State {
5
6  <#list outgoing as stimulusName, transitionAST>
7      @Override
8      public void handle${stimulusName?cap_first}(${modelName} sc) {
9          sc.setState(${modelName}.${transitionAST.getTo()?uncap_first});
10         ${defineHookPoint("<JavaBlock>?ConcreteState:handle")}
11     }
12 </#list>
13     ${defineHookPoint("<Field>*ConcreteState")}
14 }

```

Listing 13.42: Template generating a State class

Relevant for us are l. 10, where the body of the handle methods can be extended, and l. 13, where additional attributes and methods can be added to the body of the state class. If needed we could also extend the import list via the hook point in l. 1. Because the template content is small and simple, we use inlined Strings to bind the hook points (Listing 13.43):

```

1     "<JavaBlock>?ConcreteState:handle", "count++;");
2 glex.bindStringHookPoint("<Field>*ConcreteState", "int count;");

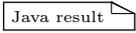
```

Listing 13.43: Binding strings to the hook points

The resulting code now contains, for example, the newly added statements in l. 4 and 9, which is equal for all handle methods in all state classes (Listing 13.44):

13. Generator Engine using Flexible Templates

```
1 @Override
2 public void handleReturnBall(PingPong sc) {
3     sc.setState(PingPong.ping);
4     count++;
5 }
6 @Override
7 public void handleStopGame(PingPong sc) {
8     sc.setState(PingPong.noGame);
9     count++;
10 }
11 int count;
```



Listing 13.44: Resulting method bodies and count attribute

Chapter 14

Integrating Handwritten Code

co-authored with Klaus Müller, Alexander Roth

Not every piece of code can and should be generated. Sometimes no appropriate modeling language exists, algorithms are already well implemented, or various external libraries shall be used. Thus, quite often specific functionality is implemented by hand. We call this *handcoding* and the result *handwritten code* often marked by «hw». Efficient techniques for a smooth integration of handwritten and generated code are essential in a practical development process. This integration is also necessary when using several generators producing different pieces of code.

Hence, in this chapter we discuss two solutions to add *handcoded* functionality into the code generated by a MontiCore tool.

As described in Chapter 13, it is possible to adapt the generation process itself by writing new *templates* that are aware of the handcoded functionality and are thus able to directly use these functions.

14.1 Integration of Handwritten Code

Reusability of a code generator is an essential property. This includes that it is possible to change the source model and to re-generate new code easily. This allows to reuse and evolve the source model, because in practice models will be adapted, enhanced, extended, and refactored plenty of times. Therefore, the generated code must not be modified by hand. We repeat this important principle:



Tip 14.1:

Generated code must not be modified by hand.

Thus, any handcoded parts need to be located in separate artifacts. Each artifact is either completely handcoded or generated, but not a mixture. Actually, there are approaches that try to mix hand coded and generated code within artifacts, but they seem to be not robust against manual errors, as well as refactorings and optimizations of smart IDEs.

Furthermore, they put completely unnecessary burden of heavy conflict resolution of independently generated code to the user.



Tip 14.2: Keep Generated and Handwritten Code Separate

- Keep generated and handwritten code in separate artifacts.
- Generated code must not be modified by hand.
- Ideally separate the artifacts in different directory hierarchies, such that cleaning of all generated artifacts is simple.
- Do not put any generated artifacts under version control.

A good generator aims for extensibility of the resulting code. This can be accomplished by integrating design patterns (see e.g. Chapter 11 and [GHJV94]) into the generated code. As a consequence, developers are able to add their own subclasses and to inject them appropriately. This kind of usage facilitates the generated code like a framework. In particular, the *template method pattern*¹ is a standard technique for handcoded extensibility.

MontiCore also uses a second technique: It generates the code while being aware of existing handcoded classes and directly integrates them into the generated code. For that purpose the Monticore generator examines the *handcoded path* (argument `-hcp`) and reacts on existing handwritten classes.

Because all generated classes are stored in the `out` directory (argument `-out`) the hand-coded and the generated classes can be kept separate. This allows to *clean up* generated files easily and also prevents developers from accidentally storing generated files in version control.

In Java, the package structure is reflected in the directory structure. Consequently, the output directory can have a larger substructure, representing the different packages that contain generated code while the directories above the package structure are used to separate handcoded and generated classes. Handcoded classes are thus part of the same package, but reside in a different directory structure.

14.2 Adaptation of Generated Code by Subclassing

It is always possible to define subclasses of generated classes or implement generated interfaces. Depending on the concrete form of usage, it is, however, simpler or more complicated to inject objects of subclasses into the generate code structures. For example, the parser creates lots of instances of the AST-classes. In case subclasses should be used instead, the parser needs to create objects of appropriate subclasses. This, however, needs to be done without changing the generated parser.

¹The template method pattern has nothing to do with our Freemarker templates. A template method contains an implemented, reusable part of code and uses (empty) hook methods for extensibility. Subclasses redefine these hooks and thus add functionality.

For that purpose, many classes (including all AST-classes and symboltable-classes) have corresponding *builder* classes. Builders and their providers, the *builder mills*, are generated in form of the static delegator pattern (see Section 11.1) and can be adapted through building a subclass, instantiating the subclass and injecting the single instance (see Figure 14.3). The AST mill for a language L is generated as a class `LMill` and the symboltable mill for a language is generated as a class `LSymTabMill`.

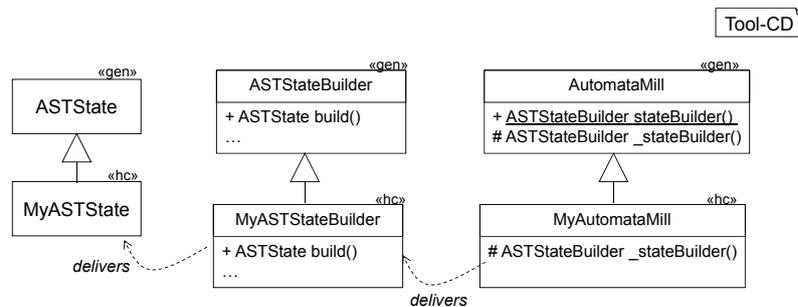


Figure 14.3: How a given class can be extended by building a subclass



Tip 14.4: Adaptation through Subclasses

When building a subclass of a generated class, you typically also need to adapt the builder for that class. This is usually done by subclassing the generated builder as well.

This approach is robust, because it allows re-generation. When the superclass is not re-generated (in the same form), the handcoded subclass becomes erroneous and does not compile anymore, which allows developers to detect necessary changes during compile time.

This approach is robust, but has the disadvantage that it is not possible to add new methods to the signature of the `ASTState` class directly, but to the subclass only. So either the subclass needs to be explicitly known in the rest of the system, and downcasts of `ASTState` objects are necessary, or no additional functionality is available.

14.3 Adaptation of Generated Code using the TOP Mechanism

A second and for developers less labor-intensive approach is to directly write the desired class by hand. This has the advantage that the implementation can be extended and method bodies overridden, but also the method signature can be extended and the newly defined methods are available for users of the generated class. There is no need for users, to explicitly know (and import) subclasses to be able to use these methods.

Figure 14.5 demonstrates this. Here a handwritten class `ASTState` is existent in the path for handcoded files (argument `-hcp`) that replaces the generated `ASTState` class. The

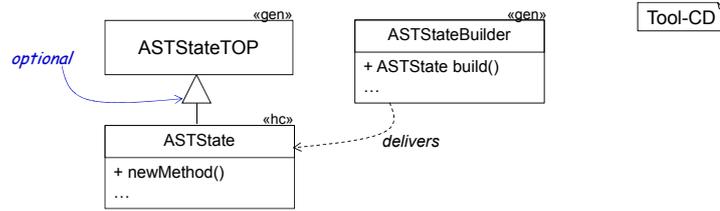


Figure 14.5: How a given class can be replaced by a handwritten class

handwritten class `ASTState` has the same name and package as we would expect for the generated class, however, it is located in the source path, which is passed to the generators via `-hcp` argument. In this case, a class `ASTStateTOP` is generated instead, which contains the identically generated code, but is abstract and has obviously a different name. This class is not explicitly used in the framework, but may mainly serves as superclass for `ASTState`. The developer has the following options:

- The handwritten class extends the generated class (as shown and recommended).
- The handwritten class is a copy/pasted version of the original class with specific modification, but does not extend the TOP class. Not recommended!
- The handwritten class is completely written afresh and ignores the generated class. In this case the generated class is not used at all, but the handwritten class needs to provide at least the same interface, i.e., the same methods, as the generated class but may provide additional methods and individual implementations.

If MontiCore detects a handwritten class (here `ASTState`), it creates all other classes in the usual form, except if these classes are replaced by handwritten classes as well. The TOP mechanism is applicable to every generated class.

Please note that the generation process is now *sensitive to the existing set of classes*. This means: when adding new handwritten classes or removing a handwritten class, a re-generation might be necessary. In the worst case, a cleanup and re-generation or an improvement of the generation script (gradle, maven or make) should help. Furthermore, when configuring the classpath yourself, it is useful to always include handcoded classes before generated classes in the Java classpath.

Please also note that the handwritten class may not be of a different kind than the generated TOP class. In particular, if the generated class is instantiatable, then the handwritten class must be instantiatable too. Either both are abstract respectively an interface, or none of them is.

When the constructor for the handcoded class remains the same, then the generated builder can be reused directly. This is typically the case when no new attributes are added or all added attributes have default values to start with. If the constructor of a handcoded class has, however, changed, the same TOP mechanism can be applied to the builder. Figure 14.7 demonstrates how a class and its builder are replaced using the TOP mechanism.



Tip 14.6: How to Add Handwritten Code

When handwritten classes shall be added, the following steps may help:

- Create (empty) class `ASTState` in a directory `dir`.
- Don't forget to put the new file under version control.
- If not yet done, add `dir` to the handcoded path via the `-hcp` argument.
- Execute the MontiCore generator.
- Now, let `ASTState` extend `ASTStateTOP`.
- Adapt `ASTState` at will to implement changed and additional functionality reusing signatures from `ASTStateTOP`.
- Do not forget to initialize additional attributes.
- Rerun the generator.

This introduces a handcoded version of the class and thus allows versioning and modification.

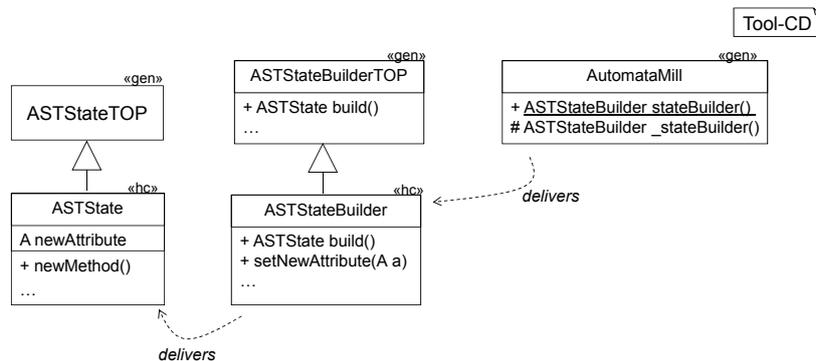


Figure 14.7: How a given class and its builder can be replaced by handwritten versions

This problem with the constructor is the reason, why many, and especially the AST-TOP-classes provide an empty constructor and the corresponding builders use this constructor together with the respective setters for the attributes to build the objects. It is up to the responsible developer to not misuse empty constructors without completing the object afterwards. When you want to provide a different version of a TOP mechanism to your own developers, you might use a single constructor that takes exactly the builder as argument.

As a big advantage, the TOP mechanism can also be used, when the generated classes are embedded in an inheritance hierarchy. Each class in an inheritance hierarchy can be extended individually, while the generator for other classes does not need to take notice at all. Figure 14.8 demonstrates possible resulting structures.

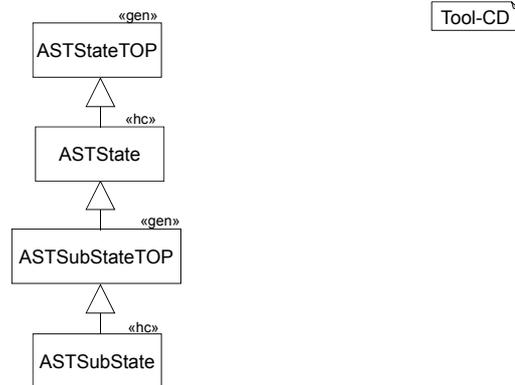


Figure 14.8: How a given class and its subclass can be replaced by a handwritten class

In the MontiCore language workbench itself and usually for the MontiCore tools the same principle is used for all generated classes.

Chapter 15

Error Handling, Logging and Reporting

co-authored with Andreas Horst

This chapter serves two purposes: First, it describes general considerations on error handling and second it describes how MontiCore implements these. The first part of this chapter is also useful for writing your own generator.

In addition, this chapter describes, which reports are generated in a MontiCore generation run and how to configure the reports.

Generators in general and language workbenches like MontiCore have to cope with errors on several levels: (1) the generator itself may issue error messages while analyzing its input models or generating code and (2) the generated code itself usually must include error messages, which are then issued when executing the generated code. MontiCore, however, applies the same rules of defining and raising errors both for the generation phase as well as for the code execution. This is possible because the MontiCore runtime environment is available for MontiCore itself, all generated tools, and the final products. This in particular includes logging, error messaging, etc. as discussed in this chapter.

15.1 Where to find Concrete Help for an Error, Warning, or other Message

The rest of this chapter are more general considerations on errors, warnings and logs as well as their configuration. When you have a concrete error message that you don't know what to do with, please have a look at:

```
1 // Explanations to errors and potential help
2 www.monticore.de
3 www.monticore.de/bestpractices
```

Here you can find a more current and up-to-date list of help and suggestions.



Tip 15.1: Internal Errors in MontiCore

MontiCore tries hard to avoid internal errors. In case you got one, please send it to `bugreport@monticore.de` ideally including the source model, configuration, MontiCore-version and other potentially interesting information. It will not be generally visible, but only used for improvement of MontiCore by its developers.

15.2 Errors, Warnings and Log Messages

In this section the different kinds of error messages are explained. Furthermore, the difference between internal errors and errors due to incorrect usage of a generator is elaborated.

15.2.1 Errors

Many forms of errors may occur during model processing and generation. These errors can have various causes and address different user groups. MontiCore distinguishes at least two kinds of errors, namely *internal errors* caused by MontiCore *developers* and *usage errors* caused by the *users* of the MontiCore tool, i.e., by erroneous input models or wrong configurations.

In addition to errors MontiCore may identify flaws that may be a problem when using the generated result and issue a warning. Regardless of the different severities it is vital to appropriately report these situations.

Apart from errors and non critical flaws as described above, sometimes it is also desirable to *report* general *information* about the complex model processing and generation process to the user.

We identify these two kinds of errors:

Internal error is an error which prevents the tool from performing its tasks and which is caused by erroneous implementations. As such, if experienced by a user, the user cannot avoid or fix the error but should report it to the developers.

Usage error is relevant to the user of a tool. It is caused by erroneous input, configuration or other behavior of the user and can be handled directly by the user.

Internal errors are caused by implementation faults (e.g., missing initialization of a class member before first access) and ideally discovered and fixed during extensive testing. For a code generator an internal error should lead to the immediate termination as it is better to have no result instead of an incomplete and incorrect result. Furthermore, stopping immediately avoids unnecessary waiting times for the user. MontiCore tries to avoid internal errors, but more importantly avoids erroneous output.

A modeler typically has no implementation insights into the used tool and therefore receives a friendly apology with the remark "*internal error*". Stack traces obviously do not help the users, but can be stored in a log file that can be sent to the developers. Nowadays

there typically exist several channels of feedback which allow users to provide relevant information (such as log files) about a program crash to the developers. They can then use these additional information for fixing and improving the tool.

Usage errors are, for example, caused by configuration faults or erroneous input models. User errors must be reported with concise and user readable error messages. Ideally they include what caused the problem and a hint on how to fix the error.

As for internal errors, the tool should terminate as quickly as possible allowing the user to adapt the erroneous input for the next iteration. However, it has proven useful to complete parsing and context condition checks, in order to allow the user to handle several user errors at once. In the backend, namely the generation, immediate termination is however useful. MontiCore has taken some effort to be helpful here.

The roles of users and developers blur, when the user of a tool adapts the tool with additional templates or Java classes. In this case, the user code replaces or adapts internal code and thus may produce internal errors resulting from erroneous user code.

15.2.2 Warnings and Information

MontiCore provides further levels of escalation, corresponding to the interests and roles of their users.

Warning does not prevent the successful execution of the tool or generator, but might lead to unwanted results the user should be aware of.

Information message contains neutral information about the executed process and help to comprehend what is currently or has already been done by the tool.

Debug message contains detailed internal information about specific states of the executed process.

Trace message allows to comprehend the execution of a program in very detailed form and is based on the implementation internals such as method names or even numbers of lines of codes.

Warnings are reported, because the user may want to react and fix it or explicitly decide to live with it. Warnings are always on the user level as "internal" warnings are useless, but it may happen that only the tool developer can interpret the warning.

Information messages typically signal what is currently done by the tool (e.g., the different execution steps) or what has already been done (e.g. successful parsing of a model, successful generation of an artifact). They signal to the user that in fact something is being done by the tool as opposed to a quiet execution where, when in doubt, it may be unclear if something is happening at all. Information messages should be selected carefully and not be too numerous.

Debug and trace messages are targeted at tool developers and potentially also tool adapters, i.e., users that not only use the tool, but add additional handwritten code. They are intended to help to debug a program by inspection of the internal states of a program.

Debug and trace messages provide detailed insight into the executed program (in this context DSL tools). They allow for instance to inspect concrete arguments passed to a method and hence can provide hints about what caused an error. The interpretation of such information of course requires detailed knowledge of the implementation. Hence debug messages are explicitly aimed at developers and not normal users.

MontiCore in its default mode provides some, but not too verbose information and the debug messages are switched off completely. See below how to configure both via script as well as using them from additional Java classes or Monticore templates.

15.2.3 Form of Errors, Warnings and Log Messages

Summarizing these definitions and considerations yields the following types of messages and their semantics:

error messages denote critical errors during the execution of the application. Typically this means that the program cannot or could not be executed successfully.

warn messages denote situations which do not hinder the execution of the program but should be fixed to avoid unwanted behavior.

info messages are used to communicate general information to the user such as the start or successful termination of a process.

debug messages as opposed to **info** messages may contain detailed information about the state of the program. Such detailed information are not required during normal execution but help to debug the program.

trace messages are typically used for even more detailed information such as the currently called method or even line of code or the number of iterations. Such detailed information is mainly used for optimization or complicated debugging.

This list of types of messages can be used in a concise and centralized logging component which will be presented in Section 15.3.

Good error messages help to solve the problem or at least to communicate it to a potential expert. Complex, highly adaptable software such as Monticore, has many kinds of errors with individual solution strategies. This includes e.g. the many context conditions of a language. It is useful to simplify communication (for example the chat over telephone) by attaching unique and simple to find identifiers to an error message. For quick finding of information, errors in Monticore and its DSL tools have an *error identification* code in the form of e.g. "0xD0016" or "0xAF21C". They use a familiar representation known from hex representations with (unfamiliar) five digits and upper letters and are thus of form "0x[0-9A-Z]^5". Advantages of such an identifier are:

- Easy to communicate and less likely to misunderstand an error communication (e.g. through the telephone).
- Easy to find in the code (usually only these have 5 digits).

- Easy to check their uniqueness.
- Easy to search for potential solutions in the web.

Too many or detailed warnings and especially *informative messages* often pollute the output and prevent the user - and potentially also the developer - from seeing the relevant information (such as errors). It is therefore important to configure where to announce information:

- *Console*: Console output is the most direct form of information. It is created and viewed directly during the execution of the tool/generator.
- *Log file*: Log files typically contain more detailed information to prevent pollution of the direct output in the console. MontiCore by default writes logs to a single log file in `monticore.date.log`.
- *Reports*: In addition, reports of various kinds contain special information about the executed process. See Section 15.5 for details.

In interactive systems such as web servers, log messages often contain timestamps for each message. Administrators and developers use this information to retrospectively analyze issues. Since generators typically run in batch mode, their logs do not require such information. It might at most be valuable to add starting and end time or the total duration of an execution.

Below we examine the MontiCore error management in order to see how error handling and logging are implemented, and can be reused for DSL tools.

Please note that the generated code or product also may experience errors. These error messages address users of the generated code. However, depending on the form of product, the MontiCore error handling may be completely inappropriate. For instance, code that runs in embedded systems may have no detailed log file, while multi-user systems usually have extensive logs including timestamps, error codes and messages, and so on.

15.3 The Error and Logging Component

MontiCore uses a centralized *logging component* for issuing error, warning, and other log messages. This logging component is made available using the *static delegator* pattern (see Section 11.1), and thus can be reached globally from everywhere in the code and templates. This was a deliberate decision, because we expect a unified logging mechanism to be acceptable for all components of a DSL tool.

Although the logging API is static, there are extensive configuration mechanisms, as the static methods basically delegate to a hidden object that can be configured as well as replaced by own implementations. As mentioned above, we use the static delegator design pattern for this purpose.

The class `Log` defines and constitutes the core API (as shown in Listing 15.2) for the logging component and acts with its methods as a central delegator for issuing errors, warnings, and log messages.

15. Error Handling, Logging and Reporting

```
1 Repository: MontiCore/se-commons github
2 Directory:  se-commons-logging/src/main/java/
3 Files:     de.se_rwth.commons.logging.Log.java
4           de.se_rwth.commons.logging.LogStub.java
5           de.se_rwth.commons.logging.Slf4jLog.java
```

While the `Log` class uses standard printing (`System.out`) and also exits the program in case of an error. The `LogStub` instead only internally stores the messages, but does not produce any side effects, i.e. prints. `LogStub` is thus useful for testing (see e.g. Section 10.3). A third subclass `Slf4jLog` is a highly configurable subclass that is used by default. See Section 15.4.4 for its configuration.

```
1 public class Log {
2     public static void error(String msg)
3     public static void error(String msg, Throwable t)
4
5     public static void warn(String msg)
6     public static void warn(String msg, Throwable t)
7
8     public static void info(String msg, String logName)
9     public static void info(String msg, Throwable t, String logName)
10    public static void debug(String msg, String logName)
11    public static void debug(String msg, Throwable t, String logName)
12    public static void trace(String msg, String logName)
13    public static void trace(String msg, Throwable t, String logName)
14
15    public static void enableFailQuick(boolean enable)
16    public static long getErrorCount()
17    public static boolean isFailQuickEnabled()
18 }
```

Listing 15.2: Logging API in class `Log`

Using this central logging component, developers can issue log messages at the respective severity. For instance, error messages are to be issued using the respective `error` log methods shown in Listing 15.2 in lines 2 and 3. Log messages intended for debugging have to be respectively issued using the `debug` log methods in ll. 10f. All log methods provide one parameter for the log message and a second overloaded variant with an additional parameter for exceptions (e.g., for printing out stack traces).

The different log levels `error`, `warn`, `info`, `debug` and `trace` form a hierarchy of severities where the level `error` is of highest severity and `trace` the lowest. A common approach in logging is to use the log level for controlling the verbosity of logs. The idea is that `trace` messages are far more numerous than `error` messages which only occur when something went wrong. With this in mind log levels can be seen as threshold filters for controlling which messages actually get printed in the output (e.g., console, log files, etc.). The desired log level is meant to also include all log messages of levels with higher severity. For example, most systems by default log with level `info` which will output messages

of the levels `info`, `warn`, and `error` in the logs. That is because the levels `warn` and `error` are considered more severe than `info`. One could hence choose to ignore all log messages except for `error` or activate all log messages by using `trace` as threshold. The configuration of which log level to use for controlling the output verbosity is provided by many different logging frameworks and described in Section 15.4.

As can be seen in Listing 15.2, the methods for issuing log messages of level `info`, `debug`, and `trace` provide an additional argument `logName`. This argument is used for another output control mechanism which allows to control the output for specific parts or components of a program. These parts or components are specified by the parameter `logName`. As an example in MontiCore consider a scenario where one wants to see only the log messages of the parser or the symbol table. Logging frameworks provide configuration options for achieving this using the values of the `logName` parameter. Using this feature efficiently of course requires detailed knowledge of the program code. For the log levels `error` and `warn` this mechanism is considered unimportant as their high severity is global, i.e., it is not relevant which part of the program issued the error or warning. Detailed information about what caused the error or warning is nonetheless to be given in the respective messages (or even stack traces).

The implementation of the centralized log component also realizes the fail quick paradigm described in Section 15.2.1. By default, issuing log messages of level `error` leads to the immediate termination of the application. This behavior may be controlled with the methods `enableFailQuick` and `isFailQuickEnabled`. The former allows to temporarily deactivate the fail quick mechanism such that `error` log messages do not automatically terminate the application. However, the log component keeps track of any issued `error` log messages such that if fail quick is re-enabled later on, it terminates the application as well. This might be useful for example when multiple model artifacts have to be processed where failure to process one model artifact should not immediately cancel the entire process but after all processing is completed. How to use the logging API for such a use case is shown in Listing 15.3.

```
1 public void processModels() {
2     // disable fail quick
3     Log.enableFailQuick(false);
4     // iterate and process some models, but completely
5     // ...
6
7     // re-enable fail quick
8     Log.enableFailQuick(true);
9 }
```

Listing 15.3: Example for controlling fail quick

In Listing 15.3 the fail quick mechanism is temporarily disabled in line 3 to process a set of models. In this example errors occurring while one model is processed do not terminate the whole process. The logging component keeps track of issued `error` log messages. As soon as the fail quick mechanism is re-enabled in line 8, it is checked whether any `error` messages was issued and if so, the fail quick is applied, which terminates the application.

15.4 Logging Configurations in MontiCore

As described in Section 15.3, logs may contain detailed information. For controlling this information, MontiCore uses and provides different means which are described in this section.

Log message can be printed to the *console*, in much more detailed form to *log files* and, depending on the specific purpose, in an aggregated form of so called *reports*. The log files can be different for each DSL tool or even single components of an application and vary in desired verbosity (see Section 15.3). The logging APIs SLF4J [QOS21b] provides mechanisms to configure the logging output for specific requirements. The default implementation of the centralized log component described in Section 15.3 is based on SLF4J and its implementation logback [QOS21a]. With this, MontiCore provides three configuration mechanisms for the logging component. These three mechanisms are:

- Selection of one of two built-in logback configurations (Section 15.4.1).
- Usage of a custom logback configuration (Section 15.4.2).
- Implementation of a custom log component (Section 15.4.4).

15.4.1 Selecting one of the given Configurations

MontiCore ships with two externally usable standard logging configurations and four Java based mechanisms to select a configuration internally.

The externally selectable configurations control output based on the severity. Both print `info`, `warn`, and `error` messages to the console and into the same log file. `debug` messages are only printed into the log file. Log files are created for each execution of MontiCore individually and stored in the configured output directory of MontiCore. An overview over the characteristics and differences of the two default configurations is given below.

MontiCore default logback configuration for *users*:

- Console
 - Contains `info`, `warn`, and `error` messages
 - No logger names (see parameter `logName` in Section 15.3)
 - No stack traces of any logged exceptions
 - No timestamps etc.
- Log file: `monticore.yyyy-MM-dd-HH:mm:ss.log`
 - Contains `debug`, `info`, `warn`, and `error` messages
 - Contains logger names (see parameter `logName` in Section 15.3)
 - Contains stack traces of any logged exceptions

- Contains timestamps of the log messages

MontiCore default logback configuration for *developers*:

- Console
 - Contains `info`, `warn`, and `error` messages
 - Contains logger names (see parameter `logName` in Section 15.3)
 - Contains stack traces of any logged exceptions
 - Contains timestamps of the log messages
- Log file: `monticore.detailed.yyyy-MM-dd-HHmms.log`
 - Contains `debug`, `info`, `warn`, and `error` messages
 - Contains logger names (see parameter `logName` in Section 15.3)
 - Contains stack traces of any logged exceptions
 - Contains timestamps of the log messages

By default MontiCore uses the configuration for users. The configuration for developers can be selected by using the `-d` (or `-dev`) option over command line.

15.4.2 Using a Custom logback Configuration

An entirely custom logback configuration can be passed to the execution of MontiCore using the command line option `-cl` (or `-customLog`). Detailed information about the configuration of logback is available online [QOS21b, QOS21a].

15.4.3 Initializing the Log within Java

If a Java developer chooses to initialize the log via one of the following `init` methods (from within Java), then the above discussed defaults or custom configurations with SLF4J are not effective anymore, but a direct implementation is provided:

- `LogStub.init()` leads to sideeffect free log: all output is internally stored. It also does not terminate upon errors. This is mainly usable for test automation.
- `Log.init()` leads to a logging component that does not write any file, suppresses trace and debug messages, but writes infos, warnings and errors to standard output (console). It also terminates on error, if fail quick is enabled and the returned error code is then 1 to distinguish erroneous termination from a correct program end¹.
- `Log.initDEBUG()` is like `Log.init()`, but writes all messages to the console.
- `Log.initWARN()` is like `Log.init()`, but even suppresses info messages and only writes warnings and errors.

Furthermore, some of the reporting functions are then disabled, because reporting also acts as subclass of `Log`.

¹This is usable in make, shell or CLI environments

15.4.4 Providing a Custom Log Implementation

This mechanism allows the highest flexibility and control. In contrast to the first two methods, this mechanism requires to write substantially more Java code.

Log implements the *static delegator* design pattern described in Section 11.1. As such, subclasses can be used to implement custom log components.

All public static methods of the `Log` class (as depicted in Section 15.3) delegate to an corresponding protected `do` method. A subclass has to register itself as the new log component through the protected static method `setLog(Log)`. `LogStub` is such a subclass and can be used as blueprint if desired.

15.5 Reports

Reports differ from logs in that they need not follow a time line, but usually aggregate their information and are produced at the end of a processing run.

15.5.1 Where to Find Reports

To understand the MontiCore generator and in particular the generated code, MontiCore offers a possibility to produce a number of reports. Reports provide detailed information about the execution process and states of the generator. This includes different statistics as well as detailed reports about occurrences of generator events such as template executions, file generations and AST transformations. All generated reports are available in the reporting output directory:

```
1 reports/           // directory with generated reports 
```

Information contained in reports is usually presented short and dense. Therefore, a short explanation of the content can be found at each report's end. The purpose of the report and an overview of the contained information can be found in section 15.5.4.

15.5.2 How to Configure Reporting

All reports are enabled when using the default configuration. When using a user specific configuration file, the report generation can be enabled and then is automatically switched on by adding the following lines:

```
1 Reporting.init(outputDirectory, reportDirectory,   
2   reportManagerFactory)  
3 // ...  
4 Reporting.flush(anAST); // finally writes the reports
```

Listing 15.4: How to enable reporting

where `reportManagerFactory` is a predefined variable in the `MontiCoreScript` environment that Groovy is interpreted in. This variable comes by default with a factory instantiating class `MontiCoreReports`, but we could override this by our own class.

If the reporting shall temporarily be switched off for some activities of the generator, the methods

```

1  Reporting.off();
2  // generator activity without reporting
3  // ...
4  Reporting.on(aName)

```

Listing 15.5: How to stop and start reporting

can be used between enabling and writing (`Reporting.flush()`) the reports.

Reports can be adapted e.g. by adding more content. For example, the report `08_Detailed.txt` introduced in Section 15.5.4 can be extended by using the following method of the `Reporting` class:

```

1  Reporting.reportToDetailed("Additional info");

```

Listing 15.6: Additional information reported in `08_Detailed.txt`

Each call of this method produces an additional line written to the detailed report.

15.5.3 Identifiers contained in the Reports

Some reports contain information in temporal order of appearance and thus can be understood as log files on certain aspects of the generation process. Other reports aggregate information during the generation process and are thus only produced at the end. Reports contain some general references to templates, AST nodes etc.

All AST nodes and other objects, e.g. from symbol tables, are uniquely identified in all reports. `MontiCore` uses a semi-readable object identifier (OID) that reflects some content of the AST node (at the time the identifier is created). The OID does definitely not change over time, once it is computed, even though object attributes may change. Examples for AST node identifiers of a CD4A model are:

```

1  @Person!CDInterface
2  @PersonImpl!CDClass(5,2)
3  @age!CDAttribute(7,4)
4  @_!Modifier
5  @_!Modifier(!2)
6  @_!Modifier(3,4!2)

```

Listing 15.7: Exemplaric object identifiers in reports

Line 1 shows an identifier for an AST node with name `Person` and type `CDInterface`. Line 2 contains an AST node of type `CDClass` and name `PersonImpl`. Moreover, this identifier carries two additional comma separated numbers denoting the line and column position of the corresponding model element in the input model. If these numbers are missing, the respective AST node is usually not a direct result of model parsing, but added to the AST afterwards.

Line 3 contains another example of an identifier with source position. The last lines 4-6 contain identifier for AST nodes of type `Modifier`. All except letters and numbers are escaped with `"_"`. In line 5, a number `!2` is added to indicate that the corresponding object is not the same as reported in line 4. Line 6 shows this number added when the source position is present too (which rarely happens).

In general, the representation of AST Nodes within the reports has the structure of one of the following lines:

```

1  @content!type(line,col)
2  @content!type
3  @content!type(!nr)
4  @content!type(line,col!nr)

```

Listing 15.8: Object identifiers in reports

`type` refers to the class of the AST node that it identifies, `content` is a small identifier that is extracted from the attributes of the object. As seen before, `content` is dependent on the kind of node, e.g. if the node has an attribute called `name`, usually this name is taken. If the AST node was created as result of parsing, it comes with a source position that is added in form of `(line,col)`. If there is no source position present, the AST node was probably created during a transformation process. Finally, if there are several objects that would have the same identifier, we distinguish them by an appendix of form `!nr` within the brackets. In such a situation the identifier for the first node does not have an appendix, the second one has the appendix `!2`, the third one `!3` and so on.

Note that AST nodes are always uniquely identified in the whole generation process. However, an identifier may not fit to the actual content of an AST node as it's content can change over time while the identifier remains stable.

The reports also reference templates, hookpoints, Java classes, Java files and variables in the following forms:

```

1  NameOfTemplate.ftl      // Templates occur with extension
2  NameOfClass            // Java classes occur without extension
3  NameOfSourceFile.java // Java source files occur with extension
4  Nonterminal           // Nonterminal from grammar and
5                        // also stands for Java class ASTNonterminal
6  HP:"NameOfHookPoint"  // Hookpoints are prefixed and quoted
7  NameOfVariable

```

Listing 15.9: Representation of various entities

Generally, qualifiers (or path names) are omitted for the sake of brevity. Therefore, it helps not only here to generally use unique names.

15.5.4 List of the Reports

In this section the different reports and their purposes are introduced. A detailed description of the content can be found in the explanation section, which is located at the end of each reported file.

01_Summary.txt contains some numbers summarizing the overall generation process. Examples for information reported here are the number of generated files, the number of used templates or the number of called hook points.

02_GeneratedFiles.txt contains the list of all created files. In addition, the source of the file creation is given by the responsible template and ast node.

03_HandwrittenCodeFiles.txt contains all used and unused handwritten source code files. Inspecting this report helps the generator user to ensure that necessary handwritten code files are taken into account by the generator.

04_Templates.txt contains the list of templates used in the generation process and how often they were called to open a new file for generation or how often they were executed for existing files. In addition, a list of available but unused templates of the corresponding project is provided. Both lists are generated for standard templates provided by the generator as well as for user specific templates which are extracted by examining the `template` path.

05_HookPoint.txt contains detailed information of all kinds of hook point related events. This includes both, usual hook points as well as AST specific hook points. Hook points are regarded as AST specific if they are registered via the `replaceTemplate` method of the `GlobalExtensionManagement` class.

The information given in this report helps generator users and generator developers to understand in which order hook points are registered and executed. Moreover the different possibilities of hook point registrations (before template, after template etc.) are reported. The execution of a hook point is reported together with additional information such as the type and content of the hook point.

06_Instantiations.txt contains the list of Java classes, which have been instantiated from templates during the generation process. In MontiCore Java classes can be instantiated directly from templates using the `instantiate` method of the template controller. The reported information about instantiations from templates can help the generator developer to ensure that the usage of Java classes from templates works properly.

07_Variables.txt contains the list of global template variables used during the execution of the generator. Global variables can be read and written from all templates and Java objects through the `GlobalExtensionManagement`. For each variable the number of value changes of the variable is reported. These information help

15. Error Handling, Logging and Reporting

generator developers to identify undesired overriding of variable values during the generation process.

08_Detailed.txt is a fine grained protocol of all events reported in temporal order of occurrence. This includes events, which are reported in other reports as well as instantiations from templates, write operations of global variables, file generations, template executions and hookpoint events. Moreover, warnings and errors are reported. The purpose of this report is to comprehend the overall generation process by analyzing the individual process steps. This facilitates localizing the source of failures.

09_TemplateTree.txt shows the call hierarchy of the templates as tree structure. In addition to the involved templates, variable assignments, instantiations of Java classes via the template controller and hook point executions are reported. Like the detailed report, this report overviews the overall execution process of the generator, but it focuses on the template execution. Thus, this report helps especially to reveal weak spots corresponding to the templates of a generator.

10_NodeTree.txt depicts the AST structure captured after finishing the generator's generation step. In addition to the AST nodes and the structure of the whole tree, it reports how often a node has been used as parameter `ast` for template execution. The provided information help identifying mistakes after transformation steps of the AST and moreover to identify AST nodes which might be unused and potentially unnecessary.

11_NodeTreeDecorated.txt contains a more detailed version of report `10_NodeTree.txt`. The additional information can help the generator developer to understand which files are generated based on specific AST nodes and which templates have been executed with specific AST nodes as input.

12_TypesOfNodes.txt provides information about the final AST in a summarized manner. The report focuses on the type of AST nodes, how many objects of each type exist in the AST and how often objects of each type were used as input for a template. The purpose of this report is to get a better intuition about the different elements and the magnitude of elements of the different types which are part of the final AST.

14_Transformations.txt contains some information about explicitly defined transformations used in the generator process. A transformation event is reported when a transformation creates or modifies a specific AST node. Generator developers can use this report to check if the transformations work as supposed.

18_InvolvedFiles.txt describes the models that have been parsed, as well as the files that have been used for the tool execution, e.g. templates, and the produced output files as well as files that are influencing the generation. All files are given with their complete paths, including the jar containers where they had been retrieved.

DataStructure_grammarname.cd contains the classes, associations and attributes that define AST structure in the form of a class diagram.

grammarname_AST.od contains the AST structure in the form of an object diagram.

Every AST node represents an object with an unique name and a type. Attributes of objects include their corresponding name and value.

A graphical representation of the participating templates can be derived from two additional reports, which are generated in special language formats (GML [Gro21] and Graphviz text language [Gra17]) for this purpose. Within the graphical representation, relations between templates used during the execution process of the generator are displayed. Moreover, it is shown which Java objects are instantiated from which template files and the directories which contain the Java source files of the instantiated Java objects and the template files. Behind the name of each template and Java source file a number is displayed which indicates the total number of executions for templates and the total number of instantiations of the Java type defined within the Java source file. Some more information are shown dependent on the chosen graphical representation as described below:

15_ArtifactGml.gml is generated in the format of the Graphical Modeling Language (GML) [Gro21] which can be graphically presented by yEd [yWo21] for example. An excerpt of such a diagram is shown in Figure 15.10

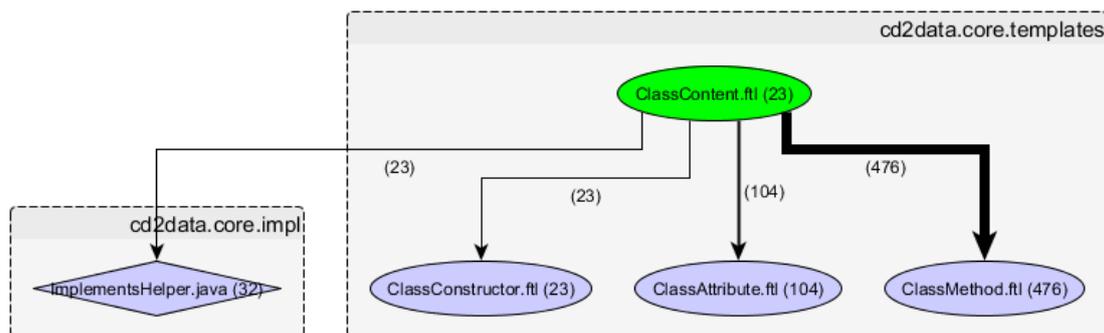


Figure 15.10: Relationship between artifacts (templates and Java, excerpt)

In this graphical representation, template files are displayed as ellipses, Java source files of instantiated Java objects are displayed as diamonds and directories containing the files are displayed as boxes (their names are displayed in a dot separated notation.) In addition to the information described above, the edges between two elements are labeled with a number. This number indicates how often a template is executed from the linked template or how often a Java type is instantiated from the linked template. Moreover, the line width of the link is adapted according to this number. A graphical template element colored in green represents a template file which has generated one or more files during the generation process.

16_ArtifactGv.gv is generated in the format of the Graphviz text language, which can be transformed into a graphical representation by Graphviz layout programs [Gra17]. An excerpt of a diagram created by a Graphviz layout program is shown in 15.11.

In this graphical representation, template files are displayed as ellipses, Java source files of instantiated Java objects are displayed as arrows and directories containing the

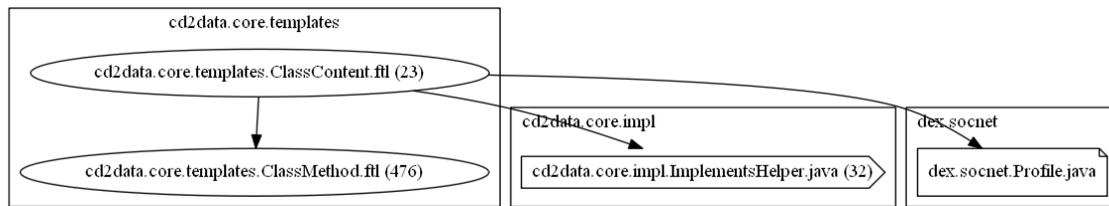


Figure 15.11: Relationship between template artifacts (excerpt)

files are displayed as boxes (their names are displayed in a dot separated notation.) In addition to the general graphical information, the created files are shown as squares each with a snapped corner. A link between a template and a generated file indicates that the template is responsible for the file creation.

In addition to the standard textual and graphical reports there may exist additional, only internally used reports such as the report `18_InvolvedFiles.txt`. For example, the latter is used in a subsequent generator execution to enable incremental generation.

15.6 For Developers: How to Deal with Errors and Warnings

Exceptions are basically handled with the mechanisms provided by Java. They carry important technical information such as the stack trace and typically occur when fatal internal errors happen. It is commonly recommended that exceptions shall not be used to implement validation failures/results to be regularly coming from user errors. Instead of eagerly printing out stack traces etc. MontiCore applies these techniques:

1. Relevant information about the occurred exception is stored in an appropriate log. If the exception signifies an internal error it is logged with level `error` using a brief but informative message together with the stack trace of the exception. The logging configuration (see Section 15.3) can ensure that the informative message is logged to the console while the detailed stack trace is only printed to a log file. This ensures that the immediate feedback on the console is not too verbose thus not polluting relevant messages by unnecessary information.
2. Additional information about the context in which the exception occurred is also logged, if appropriate. The log levels `debug` and `trace` (see Section 15.3) are suitable for this as they are intended for detailed analysis and not for general feedback to the user. As above the logging configuration ensures that these log messages are printed to a log file.
3. All further execution is terminated if the system cannot recover in a sensible way (except when parsing, which only stops at the end of parsing). The logging API (see Section 15.3) terminates the application if an `error` message is issued.

The idea is that if an exception cannot be handled in a meaningful way, then the exception should not be caught. For generators (which usually are embedded in sophisticated build

processes) it is better to terminate with an exception than to pretend being successful and to produce incomplete and incorrect results. The default logging configuration of MontiCore as described in Section 15.4 is designed to support the aforementioned steps of handling exceptions.

Chapter 16

MontiCore Use and Configuration from CLI or Gradle

co-authored with Nico Jansen

While the previous chapters mainly have been dealing with specific parts of MontiCore, this chapter concentrates on the embedding of a fully automated MontiCore generation into a larger automatic build process. There are mainly four ways to use MontiCore from outside and some of them have already been shown in the getting started introduction in Chapter 2:

Commandline interface (CLI) The CLI of MontiCore allows to call MontiCore from a shell or a makefile and is therefore well suited for scripting in batch mode.

Gradle Integration The Gradle plugin of MontiCore allows to integrate MontiCore calls into Gradle build scripts.

Maven integration is used for integration into Maven, but this is regarded as legacy. We recommend not to use the relatively dumb Maven, but a smart Gradle solution.

MontiCore Framework . MontiCore can also be understood as an adaptable Java framework that provides classes with certain reusable or adaptable methods. The API provided by the many MontiCore classes is described in the previous chapters.

The CLI is available in two variants: (1) As a jar file downloaded from a storage site, or (2) the project is locally available and has been built, thus calling main class `MontiCoreCLI` with the respective class path is relevant.

MontiCore itself does not manage incremental building, but when called it (re-)starts the generation process. It therefore behaves like many other generation tools and compilers, like e.g. the C++ compiler, but differently from Java. The smartness of Java is both a benefit for ordinary build processes and a huge problem for build processes that include generation (potentially in several layers), which is relatively common when using a MontiCore generated generator.

Maven fits to ordinary Java projects, but because of the reduced understanding of generation dependencies, Maven is not helpful for projects with efficient, smart, and incremental generation dependencies in the kind of settings that MontiCore needs. MontiCore can be

```
1 java -jar monticore-cli.jar \  
2     -g Automata.mc4 \  
3     -mp monticore-rt.jar \  
4     -hcp src/main/java \  
5     -out target
```

Listing 16.1: Executing MontiCore via CLI

integrated into Maven, but the repeatedly executed tool chain is slow compared to make or Gradle. We recommend to use Gradle or for simpler cases batch scripting with the CLI and therefore concentrate on those two forms in the following.

Furthermore, MontiCore embeds a configurable workflow that describes how the transformation from the processed grammars to the finally produced pieces of code and reports is executed. This workflow is realized using an exchangeable Groovy script which is detailed in Section 16.5 and also shows how the standard MontiCore workflow is realized. As a flexible configuration of this workflow is needed in certain configurations, we have added possibilities to configure MontiCore:

Configuration parameters allow to adapt the processing results of the MontiCore CLI and the Gradle plugin. They are described in Sections 16.1.2 and Section 16.3.

Exchangeable Groovy script for the main workflow A custom *Groovy* script provides flexibility for adapting the logging and reporting, adding additional hook points, restricting the grammar for certain purposes, or generating completely different or additional artifacts. It is discussed in Section 16.5.

Template script adaptation There is also a more specific form of adaptation for the generation process, which is largely based on *FreeMarker* templates. MontiCore's output can be configured by exchanging or adding templates, e.g., in additional hook points (cf. Section 13.3).

16.1 MontiCore from Commandline

When called from commandline, MontiCore can be used by executing compact jar-files. This is already explained in Section 2.2. The jar file contains a manifest that points to the class `MontiCoreCLI`, which contains the main method that is actually started when starting MontiCore.

16.1.1 How to Call the CLI

The call shown in Listing 16.1 demonstrates how a downloaded or precompiled jar file is included, but is actually equivalent to a direct call of the `MontiCoreCLI` class.

The jar in l. 1 contains the full MontiCore generator including all needed libraries. Line 2 specifies the processed grammar (using the `-g` option). For a search of importable grammars, the model path (`-mp`) argument is used (l. 3), which in this case consists of the

grammars provided by MontiCore. Lines 4ff describe where hand coded classes can be found for the TOP mechanism (`-hcp`) and the directory, where to generate all the output files (`-out`).



Technical Info 16.2: Obtaining MontiCore Jars via Command Line

The MontiCore jars are stored in a Maven repository to easily obtain them when using Gradle or make build scripts. There are two most relevant jars:

monticore-cli the generator CLI Tool used to generate code for a given grammar,

monticore-rt the jar containing the compiled runtime and grammar code as well as the grammar files of the MontiCore grammar library. These jars are additionally made available for download via the website <http://monticore.de>:

```
1 wget http://monticore.de/download/monticore-cli.jar
2
3 wget http://monticore.de/download/monticore-rt.jar
```

shell

16.1.2 Parameters of the CLI

The MontiCore CLI can be adapted in various ways. For that purpose, MontiCore accepts the following parameters:

- g filelist** the list of input grammars given as ordinary list of arguments. It is possible to name several grammars (space separated), which are then processed sequentially and independently (equivalent: `-grammar filelist`).
- o path** Optional output directory for all generated code; defaults to relative directory `out` (equivalent: `-out path`)
- mp pathlist** Optional list of directories or files to be included for import of other model, i.e., in this case grammars (equivalent: `-modelPath pathlist`).
- hcp pathlist** Optional list of directories to look for handwritten code to integrate (equivalent: `-handcodedPath pathlist`). See TOP mechanism in Section 14.3.
- sc file.groovy** Optional *Groovy* script to control the generation workflow (equivalent: `-script file`). For further explanation on custom Groovy scripts see Section 16.5. This option offers great flexibility, but also risks of failure.
- gh1 file.groovy** Optional Groovy script that is hooked into the workflow of the standard script (cf. Section 16.5) at hook point one, which is called after initialization, before the actual workflow begins (equivalent: `-groovyHook1 file`).
- gh2 file.groovy** Optional Groovy script that is hooked into the workflow of the standard script (cf. Section 16.5) at hook point two, which is called before the generation step (equivalent: `-groovyHook2 file`).

- fp pathlist** Optional list of directories to look for handwritten templates to integrate (equivalent: `-templatePath pathlist`). See Chapter 13 for an explanation for the use of templates. This option offers great flexibility for the generation process, but also risks of failure.
- ct file.ftl** Optional template to configure the integration of handwritten templates (equivalent: `-configTemplate file`). The filename may be qualified or relative to the `-fp` argument.
- d** specifies whether much more detailed MontiCore developer level logging should be used. Default is MontiCore user level, i.e., product developer level. (equivalent: `-dev`). This option selects another predefined Log configuration (cf. Section 15.4).
- cl file.xml** change the logback configuration to a customized file, e.g., log level, message format (equivalent: `-customLog file`). This option offers great flexibility for logging aspects, but is mainly dedicated for MontiCore developers.
- r path** specifies the directory for printing reports based on the given MontiCore grammars (equivalent: `-report path`).
- h help**: list or parameters (equivalent: `-help`)

A `pathlist` is a list of paths that are separated by spaces " ", i.e., in terms of the CLI these are separated arguments.

When composing grammars, dependent grammars need to be located in the model path using the `-mp` option. If the logging shall be adapted, option `-cl` can be used. For the configuration of the generation process (without changing the overall set of generated files), option `-fp` is useful. For a deep control of the generation workflow, including various configuration options, it is possible to use a Groovy script with option `-sc`. Thus, Groovy can be used to adapt the generation process.

16.2 Embedding the CLI in a Makefile Build Process

The CLI version is mainly dedicated for being called from a (traditional) shell and is therefore well-suited for being embedded in a larger *build script* or a *makefile*. Arguments are usually directly added at the commandline. We assume the reader of this section is familiar with the Unix `make` facility.

We recall that MontiCore does not internally check, whether an execution is needed, because the startup phase of such a tool takes too much time. Thus, the decision whether a regeneration of the code is needed has to be managed by a build tool, e.g., by a `makefile`. Make is relatively well suited for this, but needs to know two things to accurately understand dependencies between build tasks: (a) what are the actual source files that a MontiCore call is using, and (b) what are the (dependent) generated target files.

The output files (b) can relatively easily be targeted in a `makefile`, because MontiCore – like other tools – simply writes all its output into a dedicated target directory. The target directory, however, itself should not be used as a `makefile` dependency object, because

if `MontiCore` fails, it still produces the `targets` directory and a part of the files. Instead `make` offers a surrogate mechanism, the *dummy* goals.

On the input side, the situation is more complex, because `MontiCore` not only uses the explicitly mentioned input grammar, but on demand also loads a lot of additional files, such as imported grammars or symbol tables from other sources. If one of those input files changes, a rerun of the generation process is inevitable. A (traditional) `makefile` respectively its developer should not be forced to know – and even worse manually adapt – these dependencies, in particular as they may change over time.

Even more complicated: the `MontiCore` generator uses the TOP mechanism to include handwritten code into the generated code (see Section 14.3). This means that the generation process is sensitive to some handwritten Java classes and therefore needs regeneration, if these classes are added or removed. Changes of these files however, do not require a new generation. Again, a `makefile` respectively its developer shall not handle these dependencies manually.

On the other hand a general dependency of the generation process to all handwritten classes would enforce a regeneration virtually every time. To prevent this, `MontiCore` has added a *report* that tells the `makefile` (and equally also the Gradle infrastructure), which files are sensitive and therefore need to be observed.

There are essentially three factors that must lead to a regeneration: (a) a new Java file has been defined to which the TOP mechanism has to react, the opposite case (b) a Java file has been removed from the project which was previously used with the TOP mechanism, and (c) changes to all other input artifacts, such as used grammars or templates. For efficient detection `MontiCore` generates `IncGenCheck.sh` which contains a shell script that quickly detects these changes.

The following snippets are taken from a `makefile` that can be found in `MontiCore's 01.experiments/makeuse`. Its excerpts show how a straightforward integration of generation, compilation, testing and execution can be defined within one `makefile`.

This part shows the initial configuration needed to define what needs to be done:

```

1 # Individual Configuration:      <<-- adapt this part
2 # M2-model to be processed (the grammar)
3 GRAMMAR=src/main/grammars/Automata.mc4

```

From the name of the grammar, the whole project structure is being derived using typical defaults, which is very similar to Gradle. All other elements, e.g. where additional source files are located, where the target will be placed, etc. are then derived using defaults. One advantage of `makefiles` is that these defaults can easily be overridden. The defaults are:

```

1 # Configuration B: the generic part
2 # (which needs normally not be touched)
3 # a few things are derived from the grammar name
4 GramNameCap=$(patsubst src/main/grammars/%.mc4,%,$(GRAMMAR))
5 GramName=$(shell echo $(GramNameCap) | tr '[:upper:]' '[:lower:]')

```

16. MontiCore Use and Configuration from CLI or Gradle

```
6 |
7 | # path for the generated sources
8 | GenSrc=target/$(GramName)
9 |
10 | # name of resulting tool and its main class:
11 | Tool=target/$(GramNameCap)Tool.jar
12 | ToolClass=$(GramName).$(GramNameCap)Tool
13 |
14 | # M2+M1-input: All handwritten sources (from M1) for the tool
15 | SRC=$(wildcard src/main/**/$(GramName)/*.java)
```

After the variable configuration part, makefiles usually define a set of rules that describe the execution order and dependencies. These are the rules in an Automata project that depend on the tool being prepared such that it can be executed on different models:

```
1 | # <<-- adapt this part
2 | # specific goal: applying the generated tool on a model
3 | target/PingPong.result: src/test/resources/example/PingPong.aut \
4 |                         $(Tool) $(MCRTE)
5 |     java -cp "$(Tool);$(MCRTE)" \
6 |           $(ToolClass) $< target/symtab/ > $@
```

Here the tool is executed on `PingPong.aut` and the results stored in `target/PingPong.result`. A makefile can easily generalize this rule for systematic execution of the tool on all available models.

But before the tool can be applied to automata, it needs to be created. For this the following four activities are used, each of them fully generic in that sense that no tool specific name, directory, etc. needs to be adapted here at all:

```
1 |
2 | # Start of generic rule part B (which needs normally not be touched)
3 | # Activity 1 (M2): run MontiCore generator for $(GRAMMAR)
4 | $(GenSrc).f: $(GRAMMAR) $(MCJAR) $(MCRTE) \
5 |             $(GenSrc).incGenStamp.f
6 |     java -jar $(MCJAR) \
7 |           -g $(GRAMMAR) \
8 |           -mp $(MCRTE) \
9 |           -hcp src/main/java \
10 |          -out target
11 |     @touch $@
```

The above rule executes the MontiCore generator.

The MontiCore generator is only called incrementally, i.e. if one of the explicitly mentioned prerequisites changes. The rule includes `$(GenSrc).incGenStamp.f`, which acts as a dummy object. It summarizes all the implicit prerequisite files, such as grammars, templates, or sensitive Java files, that MontiCore needs. If `incGenStamp.f` is absent or the check of `IncGenCheck.sh` determines that one of the implicit input files has changed,

`incGenStamp.f` is updated and MontiCore generation is started. The rule for its inclusion in the makefile is as follows (creating it if absent and executing `IncGenCheck.sh` if present:

```

1 # Check dependencies for $(GRAMMAR)
2 $(GenSrc).incGenStamp.f: $(InitTarget) force
3     @[ -e $@ ] || touch $@
4     @[ -e $(GenSrc)/IncGenCheck.sh ] \
5         && sh $(GenSrc)/IncGenCheck.sh $@ || true

```

Again this rule normally needs no change or adaptation.

The included script `IncGenCheck.sh` contains entries like these:

```

1 [ -e A1.java ] || (touch $1; echo A1.java removed!; exit 0;)
2 [ -e A2.java ] && (touch $1; echo A2.java added!; exit 0;)
3
4 md5sum -c <<<"39...fe *G.mc4" \
5     || (touch $1; echo G.mc4 changed!; exit 0;)

```

Line 1 checks if `A1.java` is still existing, because it had been used in the TOP mechanism generation. If it has been removed then the argument file `$1`, which is `incGenStamp.f`, gets a new timestamp, such that make recognizes that it needs to act. Furthermore, a message describing the reason for the regeneration is issued and the script terminates. The other way round, line 2 checks if `A2.java` has newly been added and then reacts. Line 4 shows the dependency to another artifact, here a grammar, where the content is relevant.

The second activity is compiling the `AutomataTool` from generated classes and source files located in `src/main/java/automata`:

```

1 # Activity 2 (M2): compile the tool
2 $(GenSrc)-classes.f: $(GenSrc).f $(SRC) $(MCRTE)
3     @mkdir -p $(GenSrc)-classes/
4     javac -cp $(MCRTE) \
5         -d $(GenSrc)-classes/ \
6         -sourcepath "target;/src/main/java/" \
7         src/main/java/$(GramName)/$(GramNameCap)Tool.java
8     @touch $@

```

This rule also normally needs no change or adaptation.

Building a jar file, like `AutomataTool.jar`:

```

1 # Activity 3 (M2): build the tool jar
2 $(Tool): $(GenSrc)-classes.f
3     @echo "[MINFO]. 3: Create tool jar:" $<
4     jar cfe $(Tool) $(GramName).$(GramNameCap)Tool \
5         -C $(GenSrc)-classes .

```

Compiling the JUnit tests, which are all located in `src/test/java/automata`:

```

1 TESTSRC=$(wildcard src/test/**/$(GramName)/*.java)
2
3 # Activity 4 (M2): compile the tests
4 # (which was on purpose not integrated in tool compilation)
5 $(GenSrc)-testclasses.f: $(Tool) $(SRC) $(MCRTE) $(JUNIT)
6     @echo "[MINFO]. 4: Compile tests in" $(GenSrc)-testclasses
7     @mkdir -p $(GenSrc)-testclasses/
8     javac -cp "$(Tool);$(MCRTE);$(JUNIT)" \
9           -d $(GenSrc)-testclasses/ \
10          -sourcepath "target;/src/test/java/" \
11          $(TESTSRC)
12     @touch $@

```

And executing the tests (if desired):

```

1 # Activity 5 (M1): execute the tests
2 $(GenSrc)-tests.f: $(GenSrc)-testclasses.f $(JUNIT) $(HAMCREST)
3     @echo "[MINFO]. 5: Run tests in" $(GenSrc)-testclasses
4     @java -cp \
5          "$(GenSrc)-testclasses;$(Tool);$(MCRTE);$(JUNIT);$(HAMCREST)" \
6          org.junit.runner.JUnitCore \
7          $(TESTCLASSES)
8     @touch $@

```

The sensitivity of the MontiCore generator using the TOP mechanism has a number of advantages, but also enforces that the build management is smart enough to cope with it effectively. Thoughtlessly repeating each generation step every time is typical for Maven, but in the long run it's quite tedious. Make and Gradle do a much better job, because they listen to MontiCore, which tells them when to redo the generation.

In total, the above `makefile` construction covers each form of incrementality needed and is efficient in its execution. It is especially possible to add additional dependencies, further MontiCore tool generations or series of tool applications using advanced make rules. Furthermore, if the tool application also uses and generates multiple files, the tool itself can use the same mechanism allowing large *incremental tool chains* (actually, only partially ordered "tool nets").

16.3 MontiCore Used via Gradle Plugin

Gradle is a general purpose build tool that utilizes Groovy to define build scripts against an underlying API of methods and properties. Using Gradle a build is modeled as a directed acyclic graph of tasks. Each task is a unit of work. Thus, defining a build includes defining a number of tasks and wiring them together. Based on the task graph, Gradle determines the order in which the tasks need to be executed. Besides defining tasks and their relationships,

Gradle supports managing dependencies via Maven- and Ivy-compatible repositories and the file system.

Tasks are the unit of work in a Gradle build script. A task consists of actions, inputs, and outputs where actions are the work the task takes care of, inputs are values, files, or directories the actions operate on and outputs are directories and files that the actions create or modify. Inputs, actions, and outputs are optional. However, the inputs and outputs of a task need to be defined to utilize Gradle incremental execution feature because Gradle uses these to determine whether a task needs to be executed if triggered or whether it is already up to date. Wiring tasks can be either done by defining an explicit `dependsOn` relation or by using an output of one task as an input of another task. Tasks can be as simple as printing some text to the console or more complex including own properties and methods. In the latter case typically a custom task type is defined and instances of this task are used within the build process. Custom task types and instances of them can be integrated in a build script by using plugins. For Monticore we developed a custom task type called `MCTask`, which is provided by the Monticore Gradle plugin.

- Tasks can be defined **ad-hoc**:

```
task helloWorld {
  doLast {
    println 'Hello world!'
  }
}
```

task name → helloWorld

defines a new task → task

doLast adds the enclosed code to the task as its last step → doLast

task prints Hello world! → println

Gradle

- Or as instances of **provided task types**:

```
task copyFiles(type: Copy) {
  from "src"
  into "dest"
}
```

defined in Copy → Copy

from src → from

into dest → into

task of type Copy → Copy

Gradle

<https://docs.gradle.org/current/dsl/org.gradle.api.Task.html>

Figure 16.3: Different ways to define tasks

Figure 16.3 demonstrates both cases. In the upper part a new task is defined ad-hoc without a special type, while in the lower part an instance of the `Copy` task type is created and configured. For further background information regarding Gradle, the online documentation can be considered.

16.3.1 Defining a Monticore Task

The Monticore plugin can be integrated in a build script as shown in Listing 16.4. In case the project version does not match the desired Monticore version use the desired version instead of `$version`. The plugin offers the `MCTask` task type that executes Monticore, an `incCheck` closure, and creates a configuration named `grammar`, which

will be explained later in this section. The goal of the `MCTask` is to execute MontiCore, i.e. to process a given grammar and produce the corresponding source code for it.

```
1 plugins {  
2     id "monticore" version "$version" // MontiCore Plugin  
3 }
```

Listing 16.4: Integrating the MontiCore plugin in a build script

Similar to the CLI jar, the `MCTask` has a couple of configuration options. Two of the configuration options are mandatory while all others are optional as explained in the following. The mandatory parameters are the grammar parameter as well as the output directory (`outputDir`). The creation of a new `MCTask` instance is depicted in Listing 16.5. The instance is called `generate` and is of type `MCTask` (cf. l. 1). The task is configured to process the `HierAutomata` grammar (cf. l. 2). The variable `projectDir` is predefined in Gradle build scripts while `grammarDir` and `outDir` are custom variables. The grammar parameter expects a grammar file while the `outputDir` expects a directory. The grammar file serves as an input for the `MCTask` while the output directory and the files created by the task (and put into the output directory) are the tasks outputs. Based on this input and output Gradle determines whether the task needs to be executed or is up to date if it is triggered.

```
1 task generate (type: MCTask) {  
2     grammar = file "$projectDir/$grammarDir/HierAutomata.mc4"  
3     outputDir = file outDir  
4     def uptoDate = incCheck("HierAutomata.mc4")  
5     outputs.upToDateWhen { uptoDate }  
6 }
```

Listing 16.5: Creating a task to process the grammar `HierAutomata`

However, due to the TOP mechanism, the grammar file is not the only file that needs to be considered for the incremental execution. Hand-coded files added to extend generated implementations via the TOP mechanism as well as files removed that previously extended generated classes via the TOP mechanism require a task execution as well. However, only their existence/absence but not their changes must result in a task execution. Thus, it is not sufficient to declare this files as task inputs as changes would trigger a task execution. To solve this, MontiCore generates a file called `IncGenGradleCheck.txt` that contains information about the hand-coded files that were considered during the generation process and is used by the `incCheck` closure to determine whether a task execution is necessary. This file lists hand-coded files that were present as well as those that were checked for existence but were absent. This additional check can be used as shown in l. 4. Because of this, the outputs are marked as not being up to date in case a task execution is necessary due to the TOP mechanism.

Furthermore, the processed grammar typically is based on other grammars, e.g., the `MCBasics` base grammar. There are two cases to consider: inheriting from local grammars, i.e. grammars located in the same project, and inheriting from grammars provided

via dependencies. For local grammars, the `MCTask` provides the `modelPath` parameter (similar to the `-mp` parameter of the CLI jar), which can be used with a comma separated list of strings of paths. By default, the model path is set to `src/main/grammars` if the path is present in the current project. However, the model path can be configured explicitly as well. An example on how to set the model path explicitly is shown in Listing 16.6. For grammars provided via dependencies the `grammar` configuration is added. Dependencies added to this configuration as demonstrated in Listing 16.9 are considered by the `MCTask` as well. If these dependencies change a new generation is triggered as well.

```

1 task generate (type: MCTask) {
2     // grammar and outputDir configured as before
3     modelPath "$projectDir/grammars", "$projectDir/grammars2"
4 }

```

Listing 16.6: Configuring the model path

Further parameters are

handcodedPath "path1", "path2" Optional list of directories for detecting hand-written code that needs to be integrated (cf. TOP mechanism in Section 14.3). This default value is the directory corresponding to `src/main/java` if the directory is present in the current project.

script "file.groovy" Optional *Groovy* script to control the generation workflow. For further explanation on custom *Groovy* scripts see Section 16.5. This option offers great flexibility, but also risks of failure. By default the script `monticore_standard.groovy` that is shipped with MontiCore is used.

groovyHook1 "file.groovy" Optional *Groovy* script that is hooked into the workflow of the standard script (cf. Section 16.5) at hook point one, which is called after initialization, before the actual workflow begins.

groovyHook2 "file.groovy" Optional *Groovy* script that is hooked into the workflow of the standard script (cf. Section 16.5) at hook point two, which is called before the generation step.

templatePath "path1", "path2" Optional list of directories for detecting hand-written templates to integrate. See Chapter 13 for an explanation for the use of templates. This option offers great flexibility for the generation process, but also risks of failure. This default value is the directory corresponding to `src/main/resources` if the directory is present in the current project.

configTemplate "file.ftl" Optional template to configure the integration of hand-written templates. Thus, it can only be used in conjunction with a valid `templatePath`.

dev specifies whether much more detailed MontiCore developer level logging should be used. The default is the MontiCore user level, i.e. product developer level. This option selects another predefined `Log` configuration.

customLog changes the logback configuration to a customized file, e.g. log level and message format. This option offers great flexibility for logging aspects, but is mainly dedicated for MontiCore developers.

help help: list or parameters

For `handocedPath` and `templatePath` again String lists are used. Paths can be added as described above. The parameters `dev` and `help` are booleans and are false by default.

16.3.2 Compilation and Packaging

For compilation and packaging Gradle's Java Library Plugin that is depicted in Listing 16.7 can be used.

```
1 plugins {  
2     id 'java-library'  
3 }
```



Listing 16.7: Using the Java Library plugin

When using this plugin, typical tasks for Java projects are available such as `compileJava` or `jar`. The `build` task can be used to compile, test and package the project. However, by default the provided tasks are not aware of the generated code and the `MCTasks`. Thus, the generated sources need to be added to the main source set as shown in Listing 16.8 in l. 1ff and the `compile` task must be linked to the `MCTasks` (cf. l. 4ff). For creating an uber or fat jar, e.g. to create CLI tools, the Gradle's shadow plugin can be used.

```
1 sourceSets {  
2     main.java.srcDirs += [ outDir ]  
3 }  
4 compileJava {  
5     dependsOn project.collect { it.tasks.withType(MCTask) }  
6 }
```



Listing 16.8: Integrating the MontiCore plugin in a build script

16.3.3 Defining external Dependencies

After generating code from a grammar, the code needs to be compiled. To compile the generated code, the MontiCore runtime and typically the generated code of the included grammars are needed. Therefore, dependencies need to be added to the build script as shown in Listing 16.9.

In Gradle, configurations are used for dependency resolution. Configurations can be added manually or integrated using plugins. The Java plugin, for example, offers configurations for compilation (`implementation`) and testing (`testImplementation`). The MontiCore runtime as well as the base grammar library are used for compilation in Listing 16.9.

```

1 dependencies {
2     implementation "de.monticore:monticore-runtime:$mcversion"
3     implementation "de.monticore:monticore-grammar:$mcversion"
4     grammar "de.monticore:monticore-grammar:$mcversion:grammars"
5 }

```

Listing 16.9: Adding MontiCore dependencies

The grammar configuration used in Listing 16.9 is added by the MontiCore plugin. This configuration is used to add external grammars to the MCTask. In this example, the MontiCore base grammars are added. While the generated and compiled code is provided by the "normal" jar, grammars are provided by jars with the classifier grammars. Thus, this jar of the base grammars is used for the grammar configuration.

```

1 repositories {
2     maven {
3         url "https://nexus.se.rwth-aachen.de/content/groups/public"
4     }
5 }

```

Listing 16.10: Repository declaration in build.gradle

```

1 pluginManagement {
2     repositories {
3         maven {
4             url "https://nexus.se.rwth-aachen.de/content/groups/public"
5         }
6     }
7 }

```

Listing 16.11: Repository declaration in settings.gradle

Finally, the declared dependencies must be resolvable by Gradle. This holds for the dependencies as well as the MontiCore plugin. Therefore, a repository information must be added to the `build.gradle` file for project dependencies and to the `settings.gradle` file for the plugin. Listing 16.10 demonstrates the repository declaration for the `build.gradle` while Listing 16.11 demonstrates it for the `settings.gradle` file.

16.3.4 Example Build Script

Listing 16.12 serves as an example build file for defining the build process of MontiCore-based projects. It is also available in the git repository located here: `monticore-test/example`.

The script is a combination of the above discussed aspects and configures the following:

```
1 plugins {
2     id 'java-library'
3     id 'monticore' version '7.0.0' // MontiCore Plugin
4 }
5
6 group = "my.project"
7 version = '1.0.0-SNAPSHOT'
8
9 buildDir = file("${projectDir}/target")
10 def mcversion = '7.0.0'
11 // or alternatively use a snapshot, e.g. '7.0.0-SNAPSHOT'
12 def grammarName = 'Automata'
13 def outDir = "${buildDir}/generated-sources/"
14
15
16 dependencies {
17     implementation "de.monticore:monticore-runtime:$mcversion"
18     implementation "de.monticore:monticore-grammar:$mcversion"
19     grammar "de.monticore:monticore-grammar:$mcversion:grammars"
20     testImplementation "junit:junit:4.13.1"
21 }
22
23 repositories {
24     maven {
25         url "https://nexus.se.rwth-aachen.de/content/groups/public"
26     }
27 }
28
29 sourceSets {
30     main.java.srcDirs += [ outDir ]
31 }
32
33 task generate (type: MCTask) {
34     grammar = file "src/main/grammars/${grammarName}.mc4"
35     outputDir = file outDir
36     outputs.upToDateWhen { incCheck("${grammarName}.mc4") }
37 }
38
39 compileJava {
40     dependsOn project.collect { it.tasks.withType(MCTask) }
41 }
```

Listing 16.12: Example build.gradle

- In ll. 1ff the Java and MontiCore plugins are integrated as described in Section 16.3.1 and 16.3.2.

In ll. 6ff the group and version used to deploy the project are configured. The name of the project is derived from the folder name (or can be explicitly defined within the settings.gradle).

- In ll. 9ff several variables are initialized. The introduced local variables `mcversion`, `outDir` and `grammarName` are only for convenience and are used to store values that are used more than once.
 - `buildDir` is a variable predefined by Gradle. It specifies where all generated output shall be located. The default is directory `build`, but in this example it is configured to be `$projectDir/target` using the predefined `projectDir` variable.
 - As MontiCore’s version is used several times within the dependencies block the local variable `mcversion` is introduced here.
 - The local variable `outDir` is introduced as this information is used in the `MCTask` configuration for the output directory for MontiCore as well as to add this folder to the source set to include the generated sources in the compilation process.
 - The local variable `grammarName` holds the name of the grammar and is used for the input grammar and the `incCheck` file that contains additional dependencies of the last execution.
- In ll. 16ff dependencies to MontiCore and JUnit used for generation, compilation, and testing are defined as explained in Section 16.3.3.
- The repository for all relevant sources is declared in ll. 23ff as explained in Section 16.3.3.
- The output directory is added to the main source set in l. 29ff. This has the effect that it is included in the compilation as explained in Section 16.3.2.
- The `MCTask` instance is created in ll. 33ff. It processes the grammar `Automata`, produces the result in `outDir` and clarifies with `incCheck`, when an update is needed. The latter also includes the incremental re-generation, when a dependent artifact such as a template or a handwritten Java class for the TOP mechanism is modified. For a more detailed explanation please consider reading Section 16.3.1
- Finally, the compilation is linked to all `MCTasks` in ll. 39ff.

This build file is capable of executing MontiCore, compiling and testing the generated code as well as packaging the resulting class files in a jar file. It can be extended accordingly to several independent generation processes based on independent (or not so independent) grammar components and potentially other generation activities in the usual Gradle styles.

16.4 MontiCore in Maven

As explained at the beginning of this chapter, we recommend for your own effectiveness that you do not use Maven for a generation build chain, but only for downloading external sources from other projects. We assume, that Gradle or a similar tooling infrastructure will displace Maven in the long run. However, if you currently have to use it, MontiCore provides a Maven plugin that offers to configure the same parameters like in Gradle or the makefile CLI.

16.5 MontiCore Workflow Configuration with Groovy

Internally the MontiCore generator is controlled by a Groovy script which manages the high-level workflow. Groovy is an interpreted language [KLK⁺15] but looks very similar to Java.

The use of Groovy as configuration script has the big advantage that the behavior of MontiCore can be adapted to specific needs without any recompiling. The MontiCore tool can thus be used completely out of the box but still be adapted and extended rather flexibly. One major use of this flexibility is extending the generation, for example, by adding additional templates to the output process using the hooks discussed in Chapter 14.

The same approach cannot only be used for the language workbench MontiCore itself but also for derived tools that need flexible configuration as well. However, there is always a trade-off: Groovy-based configuration is probably individual to the dedicated tool, while in larger build scripts, it is useful to shovel certain kinds of configurations between several tools, such as verbosity or source and model paths. On the other hand, we might even use individual Groovy-configurations for each specific model if desired.

In MontiCore, the Groovy script is executed using the base class `MontiCoreScript`, which provides a set of available methods (cf. Listing 16.14). MontiCore provides a predefined Groovy script that can be used to execute the generator: `monticore_standard.groovy`. The script configures the generation process, as explained later in this chapter, and it is the default configuration for MontiCore if not customized individually. As described in Chapter 2, the user can choose between using the predefined Groovy script provided by MontiCore as well as develop a custom Groovy script. These scripts may use Java methods and variables as well as imported classes explained below. Groovy scripts are passed to the generator using the `-script` or `-s` parameter described in Chapter 2.

Groovy is used for the realization of a flexible top-level control workflow in order to provide a high degree of flexibility to cope with Javas customization and flexibility deficiencies: A Java-coded control workflow cannot be changed without recompilation. Even worse, recompilation requires not only the source code of the control workflow but also the entire original build environment (tools and dependencies). This is not optimal for efficient on-site customization, e.g., by a tool user (here, language developers using MontiCore). The Groovy integration chosen here is much more flexible and requires only little infrastructure.

MontiCore implements a generation process consisting of the following nine actions, where M2-M9 are repeated for each processed grammar in a loop:

- M1** Basic setup and initialization (logging, global scope, reporting)
- M2** Load and parse the input grammar
- M3** Derive the symbol table for the grammar
- M4** Check the grammar context conditions
- M5** Translate Grammar-AST to CD-AST (including symbol table)

M6 Generate parser using ANTLR

M7 Decorate CDs with classes and methods

M8 Generate AST classes, symbol table, visitor, and context condition infrastructure

M9 Write reports to files

Listing 16.13 depicts the actual implementation of the MontiCore control workflow in Groovy. This listing shows the Groovy script to generate the standard classes, as described in this book.

The first step, M1, initializes MontiCore. It initializes logging, creates the global scope, and enables reporting. Steps M2 to M4 belong to the frontend, which processes the input and checks the correctness of the processed grammar. Usually, the input grammar imports further grammars that have to be loaded via the symbol table (M3). When the frontend is finished, the backend starts to produce code and reports beginning with step M5. In M5, the grammar AST is translated to a class diagram AST, which will be used in subsequent generation steps and is written to a file as part of the reports created by MontiCore. Similarly, MontiCore creates corresponding class diagrams for the symbol management infrastructure. In M6, an input file for the ANTLR parser generator is created. Then the ANTLR parser generator is executed, which generates the desired language parser. Next, the class diagrams are enriched with additional classes and methods in M7 for the AST class generation as preparation for M8. Finally, in M8, the class diagrams are transformed into executable Java artifacts. This includes generating classes for the AST, symbol table, visitor, and context condition infrastructure. In the end, in M9, all reports are written into files.

The workflow is rather straight forward (see points M1 to M9 above). Simplicity in the controlling scripts is favored here. Thus, adaptation of the control workflow can be achieved by substituting the control script through a handwritten script. The handwritten script can reuse the various predefined functions provided by the MontiCore library because these functions are designed in particular for being used by control workflow scripts.

The individual steps performed in the control workflow, as depicted in Listing 16.13, are part of the MontiCore component library and are described in Section 16.5.2.

16.5.1 The Standard Groovy Generation Script

Listing 16.13 depicts the Groovy script `monticore_standard.groovy` that is used by the MontiCore generator to generate default classes. It generates the complete model processing infrastructure such as lexer, parser, AST classes, context conditions, visitors, and symbol table infrastructure. It is the default script used for MontiCore. This includes running the generator via CLI or a build script, such as Gradle, make, or Maven. Customized scripts can be passed as an argument using the parameters `-s` or `-script`. For minor adjustments to the workflow without changing the standard groovy script itself, MontiCore additionally offers predefined hook points at appropriate positions. Therefore, the script offers one hook point after initialization (l. 19), before the actual workflow begins. The

16. MontiCore Use and Configuration from CLI or Gradle

scripts can be flexibly injected using the designated parameters of the CLI (e.g., `-gh1`, `-gh2`) or the analogous variables of the Gradle script.

```
1 // M1: Basic setup and initialization
2 // M1.1: Logging
3 Log.info("-----", LOG_ID)
4 Log.info("MontiCore", LOG_ID)
5 Log.info(" - eating your models since 2005", LOG_ID)
6 Log.info("-----", LOG_ID)
7 Log.debug("Grammar argument      : "
8         + _configuration.getGrammarsAsStrings(), LOG_ID)
9 Log.debug("Grammar files         : " + grammars, LOG_ID)
10 Log.debug("Modelpath           : " + modelPath, LOG_ID)
11 Log.debug("Output dir          : " + out, LOG_ID)
12 Log.debug("Report dir          : " + report, LOG_ID)
13 Log.debug("Handcoded argument   : "
14         + _configuration.getHandcodedPathAsStrings(), LOG_ID)
15 Log.debug("Handcoded files      : " + handcodedPath, LOG_ID)
16
17 // groovy script hook point
18 hook(gh1, glex, grammars)
19
20 // M1.2: Build Global Scope
21 mcScope = createMCGlobalScope(modelPath)
22
23 // M1.3: Initialize reporting (output)
24 Reporting.init(out.getAbsolutePath(),
25              report.getAbsolutePath(), reportManagerFactory)
26
27 while (grammarIterator.hasNext()) {
28     input = grammarIterator.next()
29
30     // M2: Parse grammar
31     astGrammar = parseGrammar(input)
32
33     if (astGrammar.isPresent()) {
34         astGrammar = astGrammar.get()
35
36         // start reporting on that grammar
37         grammarName = Names.getQualifiedName(
38             astGrammar.getPackageList(), astGrammar.getName())
39         Reporting.on(grammarName)
40         Reporting.reportModelStart(astGrammar, grammarName, "")
41         Reporting.reportParseInputFile(input, grammarName)
42
43         // M3: Populate symbol table
44         astGrammar = createSymbolsFromAST(mcScope, astGrammar)
45
46         // M4: Execute context conditions
47         runGrammarCoCos(astGrammar, mcScope)
48     }
```

```

49
50 // M5: Transform grammar AST into a class diagram and report it
51 cd = deriveCD(astGrammar, glex, mcScope)
52
53 reportCD(cd, report)
54
55 // M6: Generate parser and wrapper
56 generateParser(glex, cd, astGrammar, mcScope, handcodedPath,
57               templatePath, out)
58
59 // M7: Decorate class diagrams and report it
60 decoratedCD = decorateCD(glex, mcScope, cd, handcodedPath)
61
62 // groovy script hook point
63 hook(gh2, glex, astGrammar, decoratedCD, cd)
64
65 // generator template configuration with -ct hook point
66 configureGenerator(glex, decoratedCD, templatePath)
67
68 // M8 Generate ast classes, symbol table, visitor,
69 // and context conditions
70 generateFromCD(glex, cd, decoratedCD, out, handcodedPath,
71               templatePath)
72
73 // M9: Write reports to files
74 // M9.1: Inform about successful completion for grammar
75 Log.info("Grammar " + astGrammar.getName() +
76         " processed successfully!", LOG_ID)
77
78 // M9.2: Flush reporting
79 Reporting.reportModelEnd(astGrammar.getName(), "")
80 Reporting.flush(astGrammar)
81 }

```

Listing 16.13: Groovy script used to generate the standard result

Please note that almost everything is configured and executed through this script, but the Log has already been initialized before, to allow earliest outputs with `Log.init()`. If an alternate Log mechanism should be used, it is possible to re-initialize it within this script (cf. Section 15.4).

16.5.2 MontiCore Base Class for Groovy Scripts

Listing 16.14 depicts the method signatures that are provided by the base class `MontiCoreScript` shipped with MontiCore and can be used within the Groovy script. The Groovy scripts provided by MontiCore rely on this base class. Furthermore, custom Groovy scripts can use this base class to define a custom generation process. But besides that, within a Groovy script, via the typical import mechanism, variable and method declarations known from Java can be used to implement a custom Groovy script. In the

16. MontiCore Use and Configuration from CLI or Gradle

following subsections, methods of the base class, predefined variable, and pre-imported classes are briefly described.

```
1                                     Java MontiCoreScript
2 IGrammarFamilyGlobalScope createMCGlobalScope(
3     ModelPath modelPath)
4 Optional<ASTMCGrammar> parseGrammar(Path grammar)
5 List<ASTMCGrammar> parseGrammars(IterablePath grammarPath)
6 generateParser(GlobalExtensionManagement glex,
7     ASTCDCompilationUnit astClassDiagram, ASTMCGrammar grammar,
8     GrammarFamilyGlobalScope symbolTable,
9     IterablePath handcodedPath, IterablePath templatePath,
10    File outputDirectory)
11 generateParser(GlobalExtensionManagement glex,
12     ASTMCGrammar grammar,
13     GrammarFamilyGlobalScope symbolTable,
14     IterablePath handcodedPath, IterablePath templatePath,
15     File outputDirectory, boolean embeddedJavaCode,
16     Languages lang)
17 ASTMCGrammar createSymbolsFromAST(
18     IGrammarFamilyGlobalScope globalScope,
19     ASTMCGrammar ast)
20 ASTCDCompilationUnit createSymbolsFromAST(
21     ICD4AnalysisGlobalScope globalScope,
22     ASTCDCompilationUnit ast)
23 runGrammarCoCos(ASTMCGrammar ast,
24     IGrammar_WithConceptsGlobalScope scope)
25 ASTCDCompilationUnit deriveCD(ASTMCGrammar astGrammar,
26     GlobalExtensionManagement glex,
27     ICD4AnalysisGlobalScope cdScope)
28 reportCD(ASTCDCompilationUnit astCd, File outputDirectory)
29 hook(Optional<String> file, Object... args)
30 configureGenerator(GlobalExtensionManagement glex,
31     ASTCDCompilationUnit cd,
32     IterablePath templatePath)
33 decorateCD(GlobalExtensionManagement glex,
34     ICD4AnalysisScope cdScope, ASTCDCompilationUnit cd,
35     IterablePath handCodedPath)
36 generateFromCD(GlobalExtensionManagement glex,
37     ASTCDCompilationUnit baseCD,
38     ASTCDCompilationUnit decoratedCD, File outputDirectory,
39     IterablePath handcodedPath, IterablePath templatePath)
40 decorateEmfCD(GlobalExtensionManagement glex,
41     ICD4AnalysisScope cdScope, ASTCDCompilationUnit cd,
42     IterablePath handCodedPath)
43 generateEmfFromCD(GlobalExtensionManagement glex,
44     ASTCDCompilationUnit baseCD, ASTCDCompilationUnit cd,
45     File outputDirectory, IterablePath handcodedPath,
46     IterablePath templatePath)
```

Listing 16.14: Methods available in the Groovy scripts

16.5.3 Methods Available within Groovy Scripts

The provided Groovy script uses several methods that are predefined by the `MontiCoreScript` base class. Moreover, to enable individual adaptations for custom scripts, MontiCore offers additional methods ready for configuration. For instance, MontiCore is able to generate AST classes that are compatible with the Eclipse Modeling Framework (EMF) [SBPM08]. In this section, these methods are briefly explained. Therefore, the expected arguments, as well as the realized functionality, and, if applicable, the returned value are described.

`createMCGlobalScope(...)` creates a unique global scope based on a model path received as input parameter. The global scope contains symbols and subscopes of the parsed grammar AST and the consecutively created class diagrams. It is used to effectively resolve corresponding symbols, e.g., imported grammars.

`parseGrammar(...)` parses the grammar received as input parameter and creates the corresponding AST. Returns the created AST.

`parseGrammars(...)` is similar to `parseGrammar(...)` but processes all grammars in the specified path.

`createSymbolsFromAST(...)` creates the symbols and scopes for the symbol table of the given grammar and attaches them to the AST elements. Returns the AST including its symbol table.

`runGrammarCoCos(...)` executes the context conditions for grammars to ensure well-formedness of the processed grammar. If the grammar violates any context condition, an appropriate message is displayed and the generation process aborted.

`deriveCD(...)` translates the grammar AST to a class diagram AST for further processing. Returns the created class diagram AST, which serves as input for the code generation.

`generateParser(...)` generates the parser and lexer for the processed grammar using ANTLR. Therefore, a `g4` file is created and passed to ANTLR to create the corresponding parser and lexer.

`reportCD(...)` writes the given class diagram to a file located in the specified output directory for reporting purposes. Can be used to report a base class diagram as well as a decorated one.

`hook(...)` executes an additional groovy script for the specified hook point. This hook points can be defined using e.g. `-gh1` or `-gh2` parameters. The executed sub-script receives an array of customizable arguments, enabling the sub-script to perform further operations on objects available during the standard workflow.

`configureGenerator(...)` calls a specified FreeMarker template, e.g. given as `-ct` parameter, before the actual generation starts and thus allows further configuring of the generation process.

`decorateCD(...)` enriches the class diagram AST with further methods for the generated AST and symbol table classes for the processed grammar. Furthermore, the

method derives additional classes for the visitor and context condition infrastructure. These classes are generated by the `generate(...)` method.

generateFromCD(...) generates the AST classes, symbol and scope classes, the visitor interfaces and classes, and classes to define and check context conditions for the language defined by the processed grammar.

decorateEmfCD(...) is similar to `decorateCD(...)` but creates methods for EMF compatibility as well.

generateEmfFromCD(...) is similar to `generateFromCD(...)`, but creates classes and methods for EMF compatibility as well.

16.5.4 Variables Available within Groovy Scripts

The provided Groovy script uses several variables that are predefined by the MontiCore base class. In this section, those variables are briefly explained. Therefore, their types and purposes are described.

IterablePath grammars corresponds to the `grammars` parameter used to create the `grammarIterator`.

Iterator<Path> grammarIterator used to process grammars sequentially.

ModelPath modelPath created from the `model path` parameter and used to load grammars such as imported grammars, if needed.

IterablePath handcodedPath created from the `path` parameter describing where handcoded classes are and used for the handwritten code integration mechanism.

File out is the output directory.

IterablePath templatePath created from the `path` parameter for FreeMarker templates and used to find handwritten templates.

Optional<String> gh1 optional path to a custom groovy script for the first predefined workflow hook point.

Optional<String> gh2 optional path to a custom groovy script for the second predefined workflow hook point.

String LOG_ID is the name of the logger which is "MAIN" by default. Can be overridden in custom Groovy scripts.

GlobalExtensionManagement glex is used for the template attachment and template hook point mechanism.

MontiCoreReports reportManagerFactory initializes and provides the set of reports desired for MontiCore to the reporting framework.

16.5.5 Available preimported Classes within Groovy Scripts

To further ease the implementation of (custom) Groovy scripts, several classes provided by MontiCore are imported by default. In this section, these classes are briefly explained.

Log offers methods to write to the log files. Includes methods for log warnings, informations and errors.

Reporting offers methods to write reports.

Names offers methods for name handling such as creating lists from qualified names and vice versa.

InputOutputFilesReporter offers methods to track which files are read, written, or considered (e.g., for the handwritten code integration mechanism) during a generation run.

Chapter 17

Example MontiCore Grammars

MontiCore provides a number of component grammars organized in library projects. These can be found in appropriate `github` projects under the MontiCore group structure. Their reuse supports language engineering because typically these languages have been tested and come with lots of reusable extra functionalities.

One major group of reusable and extensible component grammars are the literal, expression, type and statement grammars explained in the following Chapters 18 and 19. They and many other grammars build on `MCBasics` explained in Section 17.1.

Furthermore, this chapter shortly describes some additional grammars that the MontiCore core project provides. This and the following chapters therefore serve two purposes: First, they explain some of the available grammars and nonterminals for potential reuse and extension. Second, they also demonstrate how to write reusable, and hopefully also well engineered grammars. For that purpose, the chapters discuss the nonterminals and their arrangements in productions as well as some design decisions and best practices for grammar definition.



Tip 17.1: Grammar Components Explained in this Chapter

The grammars discussed below can be found in the MontiCore repository under:

```
1 Repository: MontiCore/monticore github
2 Directory: monticore-grammar/src/main/grammars/
3 Files:    de.monticore.MCBasics.mc4
4           de.monticore.Cardinality.mc4
5           de.monticore.Completeness.mc4
6           de.monticore.UMLStereotype.mc4
7           de.monticore.UMLModifier.mc4
8           de.monticore.MCCommon.mc4
9 Directory: monticore-grammar/src/main/examples/
10 Files:   de.monticore.MCNumbers.mc4
11         de.monticore.MCHexNumbers.mc4
12         de.monticore.StringLiterals.mc4
```

Files

17.1 Component Grammar MCBasics.mc4

Lexicals are basic elements of a language, such as names, numbers, math operators, whitespaces, and comments. Several of these tokens do not even show up in the AST.

The header defines the package and that the `MCBasics` grammar is a component grammar that depends on no other grammar.

```
1 package de.monticore;
2
3 component grammar MCBasics {
```

MCG MCBasics

Names have a special meaning. They are used as symbol references that can either point to a symbol defined elsewhere or introduce a new symbol. Nonterminal `Name` is therefore relevant in grammars. If a token does not explicitly define a different type, its Java type automatically is `String`.

```
1 token Name =
2   ( 'a'..'z' | 'A'..'Z' | '_' | '$' )
3   ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' ) *;
```

MCG MCBasics

`NEWLINE` accepts all three variants of line breaks. Whitespaces are captured with `WS` and the Java code after it leads to an ignoring of that token. So white spaces do not show up in the AST.

```
1 fragment token NEWLINE =
2   ( '\r' '\n' | '\r' | '\n' ) : ;
3
4 token WS =
5   ( ' ' | '\t' | '\r' | '\n' ) : ->skip;
```

MCG MCBasics

Comments are defined in Java style either as single line (`"/"`) or are enclosed in `"/*` and `*/`, but are not nested. Comments are also not stored as token but are attached to the currently processed token. This is ensured by the enclosed Java code (not completely shown here):

```
1 token SL_COMMENT =
2   "/" (~('\n' | '\r' )) * : ->skip
3   {storeComment();};
4
5 token ML_COMMENT =
6   "/*" .*? "*/" : -> skip
7   {storeComment();};
```

MCG MCBasics

The semantic predicate `storeComment()` is handcoded by the MontiCore developers. It manages to attach the comment to the nearby AST node, such that it can be retrieved if

needed. It is generally a good practice to factor out methods to check certain properties because larger pieces of Java are less efficient to develop and test within a grammar.

17.2 Component Grammar StringLiterals.mc4

There are two important literals to manage strings: the `CharLiteral` and the `StringLiteral`. Both are defined in such a way, that they embody the typical literals of a programming language like Java. In particular, both nonterminals embody the typical escape sequences and character encodings that Java uses. It may therefore be that a totally different kind of language cannot directly reuse these literals.

The `StringLiterals` grammar basically contains three blocks: (1) The two tokens, `CharToken` and `StringToken` are defined to identify the core literals. (2) Two atomic AST nonterminals, called `CharLiteral` and `StringLiteral`, are defined that contains the literal and are meant to be used in other grammars. (3) Each AST class resulting from these nonterminals is extended by a basic method `getValue` that allows to retrieve the decoded value. For that purpose, the code uses an additional class from the RTE, called `MCLiteralsDecoder` that provides numerous methods decodings strings to the respective values.

`StringLiterals` build on `MCBasics`:

```

1 MCG StringLiterals
2 component grammar StringLiterals extends de.monticore.MCBasics {

```

The following block of token precisely defines characters including all possible escape sequences. Both, characters and strings are parsed with their delimiters, but only the contents is stored. Note, however, that the escape sequences are not expanded, but remain as parsed. This form of storage is common, because repeated stripping of a string on each reuse is somewhat inefficient, but escapes potentially should remain for later processing.

```

1 CharLiteral = MCG StringLiterals
2   source:CharToken;
3
4 astrule CharLiteral =
5   method public char getValue() {
6     return
7     de.monticore.literals.MCLiteralsDecoder.decodeChar(getSource());
8   }
9 ;
10
11 token CharToken
12   = '\\' (SingleCharacter|EscapeSequence) '\\'
13   : {setText(getText().substring(1, getText().length() - 1));};
14
15 fragment token HexDigit
16   = '0'..'9' | 'a'..'f' | 'A'..'F' ;

```

17. Example MontiCore Grammars

```
17
18 fragment token OctalDigit
19     = '0'..'7' ;
20
21 fragment token SingleCharacter
22     = ~ ('\''');
23
24 fragment token EscapeSequence
25     = '\\\' ('b' | 't' | 'n' | 'f' | 'r' | '"' | '\'' | '\\\'')
26         | OctalEscape | UnicodeEscape;
27
28 fragment token OctalEscape
29     = '\\\' OctalDigit
30         | '\\\' OctalDigit OctalDigit
31         | '\\\' ZeroToThree OctalDigit OctalDigit;
32
33 fragment token UnicodeEscape
34     = '\\\' 'u' HexDigit HexDigit HexDigit HexDigit;
35
36 fragment token ZeroToThree
37     = '0'..'3' ;
```

While the token `CharToken` could be used directly, it is convenient to embed the pure string in an AST object that provides additional functionality and that can be targeted by the visitors. It is therefore a matter of taste, whether the token `CharToken` is used directly or the nonterminal `CharLiteral` that embeds the token.

This grammar also shows that smaller Java functions can relatively easily be added to the generated AST classes, but it is generally not recommended to write much functionality in this form. Instead the handcoded extension mechanism described in Chapter 14 should be used.

The definition of `CharToken` relies on a complex structure of token fragments, because the escapes need to be precisely defined to ensure that only correct characters are actually parsed. For this purpose the grammar uses a number of fragments that cannot be used as direct token, but merely as shortcuts within a concrete token definition. However, they can be reused in other, importing grammars for defining more tokens.

Strings are defined using a similar structure to characters:

```
1 StringLiteral =
2     source:StringToken;
3
4 astrule StringLiteral =
5     content:String
6     method public String getValue() {
7         if(content == null) {
8             content =
9                 MCLiteralsDecoder.decodeString(getSource());
10        }
11        return content;
```

MCG StringLiterals

```

12     };
13
14     token StringToken
15       = '"' (StringCharacters)? '"'
16       : {setText(getText().substring(1, getText().length() - 1));};
17
18     fragment token StringCharacters
19       = (StringCharacter)+;
20
21     fragment token StringCharacter
22       = ~ ('"' | '\\') | EscapeSequence;

```

One difference to a single character is that both, the decoded string and the original parsed string are both (redundantly) stored to prevent repeated calculation. On the other hand if the AST object is modified, then both attributes, `source` and `content` need to be kept in synchronization.

Please note that Strings do not allow arbitrary form of escapes, e.g. `"\a "` is forbidden. Furthermore, it is important that a string is parsed in detail, because escape sequences may contain a string delimiter `"`, without actually terminating the string.

17.3 Component Grammars for Numbers

Numbers can be positive decimals only, or also come with a negative sign. In many programming language they also can be provided as hexadecimal, octals or binary numbers and they could be integers or longs.

The following two grammars demonstrate, how to define numbers and extend the way how numbers are encoded in a subgrammar. The following grammar introduces an interface `Number` that is meant for a subsumption of different potential encodings. The interface comes with an extension of three methods, for either getting the direct source that has been parsed or the decoded integer with `getValueInt` respectively long with `getValue`. Please note that the methods are attached to the interface and need to be overwritten in all implementing nonterminals.

17.3.1 Component Grammar MCNumbers.mc4

Grammar `MCNumbers` provides two nonterminals for positive decimals and integers:

```

1
2 component grammar MCNumbers extends de.monticore.MCBasics {
3
4     interface Number;
5
6     astrule Number =
7         method public String getSource()

```

17. Example MontiCore Grammars

```
8     { throw new UnsupportedOperationException(  
9         "0xFF230 Method not implemented"); }  
10    method public int getValueInt()  
11    { throw new UnsupportedOperationException(  
12        "0xFF231 Method not implemented"); }  
13    method public long getValue()  
14    { throw new UnsupportedOperationException(  
15        "0xFF232 Method not implemented"); }  
16    ;
```

The interface methods `getSource()`, `getValueInt()`, and `getValue()` (cf. ll. 6) have been given default implementations, which in this case basically means throwing an exception, because the actual implementation can only be provided in the implementing classes. However, the signature is common to all known terminals implementing interface `Number` (cf. l. 4).

One nonterminal implementing this interface parses exactly the `Decimals` and thus produces only positive numbers (≥ 0):

```
1  Decimal implements Number = MCG MCNumbers  
2     source:DecimalToken;  
3  
4  astrule Decimal =  
5     method public int getValueInt() {  
6         return Integer.parseInt(getSource());  
7     }  
8     method public long getValue() {  
9         return Long.parseLong(getSource());  
10    }  
11    ;  
12  
13    token DecimalToken  
14        = '0' | (NonZeroDigit Digit*);  
15  
16    fragment token Digit = '0'..'9' ;  
17  
18    fragment token NonZeroDigit = '1'..'9' ;
```

Again, `Decimal` is a nonterminal that is built on the token `DecimalToken` which would only provide a string and could not be reached by visitors, neither implement the signature desired by the `Number` interface. Please note, that only two methods are implemented directly, while the third method `getSource` is generated because we called the token source in the right inside of the `Decimal` production.

Please note that we deliberately decided, that `00` is not parsed as a single decimal and that separating dots, spaces or underscores to group large numbers are also prohibited. However, there is no prevention against large number overflows.

The following part of the grammar introduces integers, which optionally have a minus sign:

```

1 Integer implements Number =
2     (negative:["-"])? decimalpart:DecimalToken;
3
4 astrule Integer =
5     method public int getValueInt() {
6         int a = Integer.parseInt(getDecimalpart());
7         return negative ? -a : a;
8     }
9     method public long getValue() {
10        long a = Long.parseLong(getDecimalpart());
11        return negative ? -a : a;
12    }
13    method public String getSource() {
14        String s = getDecimalpart();
15        return (negative ? "-" +s : s);
16    }
17 ;

```

Please note that the literals with a prefixed negation consist of two token and thus allow spaces in between. The reason for this separation is that otherwise the infix operator "-" would not be lexically recognized anymore when followed by a number. On the other hand this means that method `getSource` has to be implemented explicitly and the containing attribute gets a different name (`decimalpart`).

17.3.2 Component Grammar MCHexNumbers.mc4

While the `MCNumbers` grammar can be used directly, it also allows to extend the forms of how to describe numbers. This is shown in the following grammar by introducing hexadecimal numbers:

```

1 component grammar MCHexNumbers extends
2     MCNumbers, de.monticore.MCBasics {
3
4     Hexadecimal implements Number =
5         source:HexadecimalToken;
6
7     astrule Hexadecimal =
8         method public int getValueInt() {
9             return Integer.parseInt(getSource().substring(2),16);
10        }
11        method public long getValue() {
12            return Long.parseLong(getSource().substring(2),16);
13        }
14    ;
15
16    token HexadecimalToken
17        = '0' ('x' | 'X') HexDigit HexDigit*;
18

```

17. Example MontiCore Grammars

```
19 fragment token HexDigit
20 = '0'..'9' | 'a'..'f' | 'A'..'F' ;
```

The principle is again the same: a new nonterminal `Hexadecimal` implements the given interface `Number` and is based on an appropriate token definition, that describes encodings of hexadecimal numbers.

The language developer may decide, whether only decimal encodings or in addition hexadecimal encodings are allowed for numbers, simply by extending `MCNumbers` or `MCHexNumbers`. Depending on which grammar is extended, the nonterminal `Number` provides different language elements being parsed, but the further reuse is basically harmonized through providing a common signature with a method like `getValue`. Please note that if only hexadecimal should be allowed, then the nonterminal `Hexadecimal` can also be used directly.

The following part of the grammar also allows negative hexadecimal:

```
1 HexInteger implements Number = MCG MCHexNumbers
2 (negative:["-"])? hexadecimalpart:HexadecimalToken;
3
4 astrule HexInteger =
5 method public int getValueInt() {
6 int a = Integer.parseInt(getHexadecimalpart().substring(2),16);
7 return negative ? -a : a;
8 }
9 method public long getValue() {
10 long a = Long.parseLong(getHexadecimalpart().substring(2),16);
11 return negative ? -a : a;
12 }
13 method public String getSource() {
14 String s = getHexadecimalpart();
15 return (negative ? "-" +s : s);
16 }
17 ;
```

17.4 Component Grammars for UML Languages

In modeling languages, such as the UML, it is quite common that additional information needs to be attached, that is officially not part of the modeling language. This may include information about the representation on screen, about the mapping to computational or physical devices, the necessary level of security, as well as meta information about the developer, the time of development, test coverage, etc. For that purpose UML has introduced the concept of *stereotype*.

In the grammars discussed in this section, we define a textual variant of *stereotypes*, *modifiers* and also the possibility to define *cardinalities* in a form as they are used e.g. in associations. All of these nonterminals are extended with functions available on the AST. E.g. stereotypes allow to set and retrieve their values.

17.4.1 Component Grammar UMLStereotype.mc4

An example for a stereotype is `<<singleton>>`. On the lexical level, scanning a composed token like `>>` can be difficult, because as explained in Section 4.3, the scan does not use backtracking or other predicting mechanisms to understand, whether `>>` shall be parsed as a single token for the stereotype or as two separate tokens, for example occurring in `List<List<String>>`. Line 4 in the following listing does introduce two such combined tokens. This is why the grammar also adds the parser directive `splittoken` to split this token in Line 8. This reduces parser execution speed and therefore, we do not apply that for `<<` immediately, but recommend to add the appropriate directive, when it becomes necessary in a subgrammar.

The parser directive `splittoken` has the same effect as if we would parse two separate `<` and use the `noSpace` predicate to glue two tokens together.

```

1 package de.monticore;
2 component grammar UMLStereotype extends MCommonLiterals {
3   Stereotype =
4     "<<" values:(StereoValue || ",")+ ">>" ;
5
6   // Due to possible scanner clashes with "List<List<String>>"
7   // we split the token:
8   splittoken ">>";
9
10  StereoValue =
11    Name& ("=" text:StringLiteral)?;
12
13  astrule Stereotype =
14    method public boolean contains(String name) {...}
15    method public boolean contains(String name, String value) {...}
16    method public String getValue(String name) {...}
17
18  astrule StereoValue =
19    content:String
20    method public String getValue() {...}
21 }

```

Method `getValue` in class `ASTStereoValue` uses a cache to store the value, when decoded the first time. This redundancy needs caution, because if the value is changed in the AST, the cached content is outdated.

17.4.2 Component Grammar Cardinality.mc4

A cardinality is for example written in form `[3..17]`. Boolean flag `many` indicates an unconstrained cardinality (`*`). The various pieces of Java code directly calculate the upper and the lower bound storing them as integer values in the additional attributes defined using the `astrule` statement.

```

1 package de.monticore;
2 component grammar Cardinality
3     extends de.monticore.MCBasics,
4             MCCCommonLiterals
5 {
6     Cardinality =
7         "["
8         ( many:["*"] {_builder.setLowerBound(0);
9                     _builder.setUpperBound(0);}
10        | lowerBoundLit:NatLiteral
11          { _builder.setLowerBound(
12            _builder.getLowerBoundLit().getValue());
13            _builder.setUpperBound(_builder.getLowerBound()); }
14        ( ".." (
15            upperBoundLit:NatLiteral
16            ( {_builder.setUpperBound(
17              _builder.getUpperBoundLit().getValue());})
18            |
19            noUpperLimit:["*"] {_builder.setUpperBound(0);} ) )?
20        ) "]" ;
21
22     astrule Cardinality =
23         lowerBound:int
24         upperBound:int;
25 }

```

If the upper limit is absent, then attribute *upperBound* is 0. It is better to use the flags generated for *many* and *noUpperLimit* to determine absence of the upper bound.

Cardinality is an example, where some extra code is directly added to the parser in form of semantic predicates that actually adapt the AST. The code is executed directly when parsing and is used to calculate some additional attributes defined using the *astrule* statement. Alternatively, it would have been possible to embed that code in some extra methods, e.g. like the *getValue()* methods used in the *MCNumbers* grammar. Here the code is still manageable directly within the grammar. If the code becomes more complex, we recommend to outsource this code into external Java classes, e.g. external, reusable helpers or methods added through the TOP mechanism explained in Chapter 14.

17.4.3 Component Grammar UMLModifier.mc4

Programming as well as modeling languages usually provide a set of modifiers that can be applied to its entities, such as classes, methods or attributes. The following non-terminal *Modifier* presents a standard set of these modifiers and allows a purely keyword representation, but also a shorthand alternative used in modeling languages.

```

1 package de.monticore;
2 component grammar UMLModifier extends UMLStereotype {
3     Modifier =

```

```

4      Stereotype?
5      (    ["public"]          | [public:"+"]
6          | ["private"]       | [private:"-"]
7          | ["protected"]     | [protected:"#"]
8          | ["final"]
9          | ["abstract"]
10         | ["local"]
11         | ["derived"]       | [derived:"/"]
12         | ["readonly"]     | [readonly:"?"]
13         | ["static"]
14     ) *;
15 }

```

Please note, that it was deliberately decided through the design of the production that after parsing it cannot be distinguished anymore, whether the keyword `protected` or the iconic shorthand `#` led to the Boolean attribute in the AST to become true. Furthermore, it cannot be distinguished, how often a keyword was applied, and in which order the keywords were used. This leads to a more efficient storage and management.

If any of this information is still necessary, for example for a precise pretty printing, or the order is semantically relevant (which is neither in Java nor UML the case), then a different definition for nonterminal `Modifier` needs to be used.

It would also be possible to define an interface, e.g. `SingleModifier`, and then realize each modifier as a nonterminal implementing the interface. This would have the advantages that a visitor could act on a modifier directly and that the list of modifiers are extensible in subgrammars, but also the disadvantage that more classes would be generated.

17.4.4 Component Grammar Completeness.mc4

The UML and especially UML/P defined in [Rum16, Rum17] make explicit use of the possibility to define models or part of models as complete or incomplete. The following textual representation allows to differentiate between two compartments, for example the list of attributes and the list of methods in classes.

```

1 package de.monticore;
2 component grammar Completeness {
3     Completeness =
4         // separate brackets to avoid lexer-symbol clashes
5         {noSpace(2,3)}? "(" [complete:"c"] ")" // "(c)"
6         | {noSpace(2,3)}? "(" [incomplete:"..."] ")" // "(...)"
7         | [incomplete:"(..., ...)"]
8         | {noSpace(2,3,4,5)}? "(" [complete:"c"] ", " "c" ")" // "(c, c)"
9         | [rightComplete:"(..., c)"]
10        | [leftComplete:"(c, ...)"];
11
12    // to allow use of "c" e.g. as variable:
13    nokeyword "c";
14 }

```

17. Example MontiCore Grammars

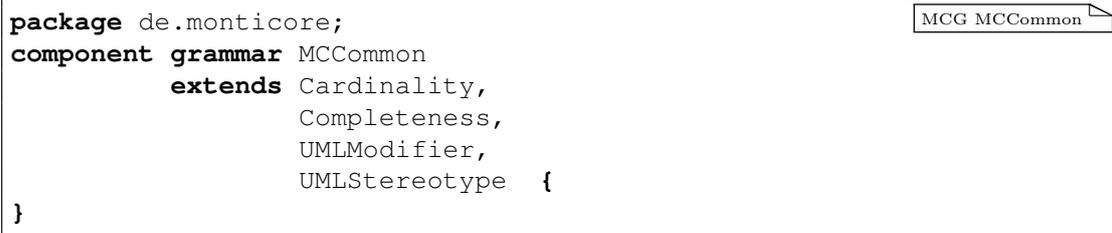
As white spaces, in general, should not be included, it is in this case necessary to glue the tokens together using the `noSpace` method to avoid a clash with similar symbols from other languages. Line 8 completely deconstructs the token, while in Line 6 only a partial split (leaving the "... " together) is used. In both cases the directive `splitToken` was not used because it always completely splits the tokens and it does not apply to tokens with characters included. As an advantage, it allows to apply the same name e.g. `incomplete` to several alternatives: the AST has only four alternatives to be dealt with.

The grammar directive `nokeyword` (l. 13) is needed to avoid the use of "c" as variable, e.g. in `foo(c)` or `bar(c,c)`, with definitions in lines 5 and 8. The grammar abstains from deconstructing in lines 10 and 13, because it is unlikely that these definitions will clash with other grammars.

17.4.5 Component Grammar `MCCCommon.mc4`

The above grammars are defined individually, but are often used together. It therefore makes sense to compose the grammars in a new grammar. `MCCCommon.mc4` realizes this.

```
1 package de.monticore;
2 component grammar MCCCommon
3     extends Cardinality,
4             Completeness,
5             UMLModifier,
6             UMLStereotype {
7 }
```



The grammar only consists of a header that includes the four subgrammars. The body can deliberately be left empty because it neither has to extend the imported productions nor has to connect external or interface nonterminals with implementations. This phenomenon of an antibody is a relatively common when composing grammars.

Chapter 18

Expression and Type Language Components

Expressions, literals, and their types, and procedural statements are essential and quite common in many kinds of languages. Historical developments of languages have shown that DSLs dedicated for a single use, e.g. as report preparation language¹ are often extended over time by various additional language constructs to be capable of handling further tasks. Therefore, to simplify the development of DSLs, the availability of a variety of language components for these typical language constructs, such as expressions, literals, types, and statements, is helpful.

The grammars provided in this chapter and the following Chapter 19 build a zoo of selectable and composable languages, which are based on the grammar `MCBasics.mc4` defined in Section 17.1.



Tip 18.1: Expressions, Literals, Symbols and Types Grammar Components

The grammars discussed in this chapter can be found in the MontiCore repository under:

```
1 Repository: Monticore/monticore github
2 Directory: monticore-grammar/src/main/grammars/
3 Packages:  de.monticore.expressions
4             de.monticore.literals
5             de.monticore.symbols
6             de.monticore.types
```

Files

The hierarchy is visualized in Figure 18.2 in the form of a language component feature diagram (LCD) exhibiting the extension dependencies and in particular the resulting variability. In total 1140 configurations are possible². Subsequent figures will highlight parts in more detail. Each of the grammar hierarchies has a root, such as

- `ExpressionsBasis` for expressions,

¹in German "Allgemeiner Berichts Aufbereitungs Prozessor", short: "ABAP" (trademark applies)

²1140 include useless configurations, such as the empty configuration or where `JavaClassExpressions` cannot fill their external nonterminals adequately.

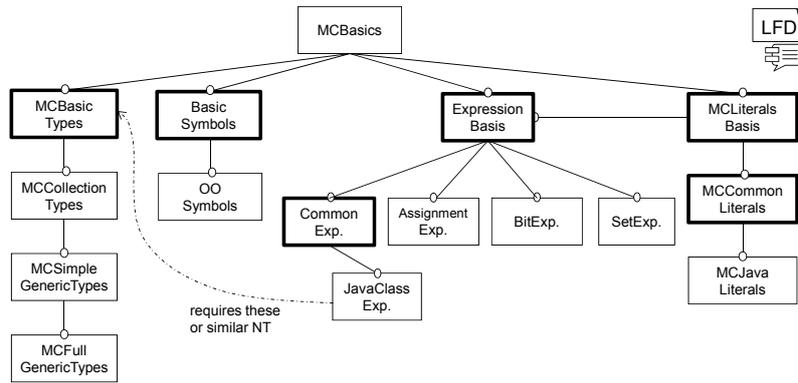


Figure 18.2: Component grammar hierarchy of Chapter 18

- `MCCCommonLiterals` for literals,
- `BasicSymbols` for symbol structures, and
- `MCBasicTypes` for types.

Each of these core grammars introduces a central nonterminal that acts as an extensible interface in all subsequent grammars. As described in Chapter 7 the selection of the appropriate variant of this nonterminal is simply defined by extending the selected grammars.

18.1 Literals as Basis for Expressions

The literals grammar collection introduces all useful forms of basic literals like strings, characters, numbers, and Booleans. They offer a wide range of different number definitions like `int`, `double`, `float` constants in signed and unsigned variants. While `MCLiteralsBasis` only introduces the respective interface nonterminal `Literal`, the grammar `MCCCommonLiterals` should be sufficient for most of the applications since these already cover the typical literals. The grammar `MCJavaLiterals` provides additional comfortable versions for numbers, for example, `"1_000"` and octal, binary and hexadecimal representations. Figure 18.3 gives an overview of the three predefined literal grammars.

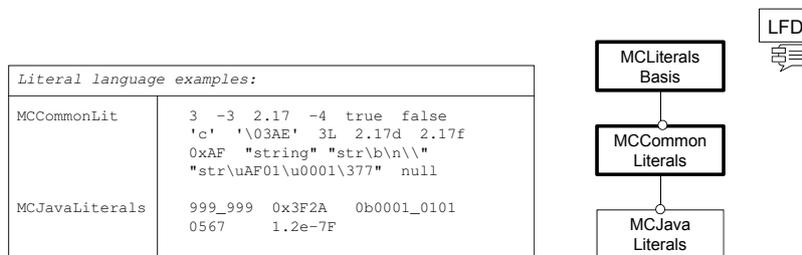


Figure 18.3: Grammars defining `Literal`

When using the predefined literals in a language it is advised to use the nonterminals that wrap the literals instead of the raw tokens as the AST classes of these nonterminals offer helpful methods like `getValue`. The nonterminals, such as `Literal` itself provide an AST implementation and its subclasses provide content extractors, in form of `getValue()` functions.

18.1.1 MCLiteralsBasis

As mentioned before, `MCLiteralsBasis` is the basic grammar for the literals group. It only contains one interface called `Literal`, which is implemented by all other literals.

```

1 component grammar MCLiteralsBasis {
2   interface Literal;
3 }

```

The principle used here is very similar to the definition of interfaces and object-oriented programming: This grammar and especially its only contained interface nonterminal `Literal` act as generally known abstract definitions. This allows each kind of grammar to

1. *extend* the available realizations of the nonterminal `Literal` by interface implementation, or
2. *reuse* available realizations, by including the nonterminal `Literal` in the productions.

Most importantly, the including grammars do not need to know anything about the extensions. A *complete decoupling* of both sides is achieved, which even allows to extend a language through additional forms of literals after the importing grammars (i.e. language components) have been defined. For example SI-units, date literals, and other specific forms can easily be added.



Tip 18.4: Decoupling through Grammars with a single Interface Nonterminal

`Literals` as well as `Expressions` show how to decouple grammars for independent development and extension.

This is achieved by defining an abstract grammar component with only a single interface nonterminal that can both be (1) extended and (2) used in production bodies of otherwise independent grammars.

18.1.2 MCCCommonLiterals

The grammar `MCCCommonLiterals` contains all typically used literals, but admittedly uses a Java-style, e.g. for characters, strings, Booleans, or numbers.

Many of the defined tokens and their wrapping nonterminals are straightforward, however, it is worth mentioning that the scanner exhibits difficulties to distinguish the binary subtraction "-" from the unary minus before number literals. Therefore, negative numbers are parsed as two tokens and combined in the wrapping nonterminal.

MCCCommonLiterals provides a number of grouping for literals, such as

SignedLiteral, which is another interface for all literals, independent of the more general **Literal**,

NumericLiteral for positive numbers only, and

SignedNumericLiteral for all numbers.

```

1 component grammar MCCCommonLiterals
2     extends de.monticore.MCBasics,
3         MCLiteralsBasis {
4     interface SignedLiteral;
5     interface NumericLiteral extends Literal <100>;
6     interface SignedNumericLiteral extends SignedLiteral <100>;

```

The number <100> provides a parsing priority, which is especially important for the order of the alternatives in the generated parsing method. The numbers are chosen freely and with gaps in order to have expansion possibilities also in between. The number <1> guarantees that this is the last alternative.

String and character (Char) use the standard encoding mechanism, which is typical for many languages, including Java. They allow encoding of typical form (e.g. "\n "), octal and unicode escapes (e.g. "\u 07AF").

```

1 NullLiteral implements Literal, SignedLiteral =
2     "null";
3 BooleanLiteral implements Literal, SignedLiteral =
4     source:["true" | "false"];
5
6 CharLiteral implements Literal, SignedLiteral =
7     source:Char;
8 StringLiteral implements Literal, SignedLiteral =
9     source:String;
10
11 // Character literals with an extraction of the Character
12 token Char
13     = '\'' (SingleCharacter|EscapeSequence) '\''
14     : {setText(getText().substring(1, getText().length() - 1));};
15 fragment token SingleCharacter
16     = ~ ('\''');
17
18 // String literals with an extraction of the String content
19 token String
20     = '"' (StringCharacters)? '"'
21     : {setText(getText().substring(1, getText().length() - 1));};

```

```

22  fragment token StringCharacters
23      = (StringCharacter)+;
24  fragment token StringCharacter
25      = ~ ('"' | '\\') | EscapeSequence;
26
27      // Escape sequences for Character and String Literals
28  fragment token EscapeSequence
29      = '\\\' ('b' | 't' | 'n' | 'f' | 'r' | '"' | '\\') | '\\\'
30          | OctalEscape | UnicodeEscape;
31  fragment token OctalEscape
32      = '\\\' OctalDigit | '\\\' OctalDigit OctalDigit
33          | '\\\' ZeroToThree OctalDigit OctalDigit;
34  fragment token UnicodeEscape
35      = '\\\' 'u' HexDigit HexDigit HexDigit HexDigit;

```

All four literals above provide a source that contains their content as string. For convenience, the grammar adds `getValue()` functions for all forms of literals with content. Here are two of them.

```

1  astrule CharLiteral =
2      method public char getValue() {
3          return de.monticore.literals.MCLiteralsDecoder.decodeChar(
4              getSource());
5      }
6  ;
7  astrule StringLiteral =
8      method public String getValue() {
9          return de.monticore.literals.MCLiteralsDecoder.decodeString(
10             getSource());
11     }
12 ;

```



Tip 18.5: Core Convenience Functions like `getValue()`

A certain form of functionality can be added to the AST nodes using the `astrule` construct.

`getValue()` is added for all literals (that contain a real value), but independently, because the resulting type differs on the literal kinds. This is the reason, why the `getValue()` signature cannot be added to the top level nonterminal `Literal` directly.

Again, we do not use Java within the method definitions excessively, but delegate to an ordinary Java function.

Various forms of numbers exist. Signed forms need to be parsed as two tokens. The semantic predicate `{noSpace(2)}` ensures that between these two nonterminals, no whitespace occurs. The same approach is applied to a following marker of the kind of numbers, such as "L".

18. Expression and Type Language Components

```
1 NatLiteral implements NumericLiteral<1> = MCG MCCCommonLiterals
2   Digits;
3
4 SignedNatLiteral implements SignedNumericLiteral<1> =
5   { noSpace(2) }? (negative:["-"]) Digits |
6     Digits;
7
8 BasicLongLiteral implements NumericLiteral<1> =
9   { cmpToken(2,"l","L") && noSpace(2) }? Digits key("l" | "L");
10
11 SignedBasicLongLiteral implements SignedNumericLiteral<1> =
12   { cmpToken(3,"l","L") && noSpace(2,3) }?
13     negative:["-"] Digits key("l" | "L")
14   |
15   { cmpToken(2,"l","L") && noSpace(2) }?
16     Digits key("l" | "L");
```



Tip 18.6: Literals composed of Several Tokens

Because of the limited capability of the scanner, sometimes it is better to parse a literal as several tokens, as for `SignedNatLiteral`.

`noSpace(n)` ensures the absence of whitespace between two tokens and `cmpToken(n, ...)` allows to initially refute an alternative (enforcing the parser to take a different one), which would not happen if only the `key(.)` statement was refuted. Section 4.2 describes these semantic predicates.

Because composed literals do not contain a `source` attribute anymore, the access to the contents is mimicked by providing an appropriate method, that acts like a get-function:

```
1 astrule SignedNatLiteral = MCG MCCCommonLiterals
2   method public String getSource() {
3     return (negative?"-":"" ) + getDigits();
4   }
5   method public int getValue() {
6     return de.monticore.literals.MCLiteralsDecoder.decodeNat (
7       getSource());
8   }
9   ;
```

Similarly floats and doubles are defined:

```
1 MCG MCCCommonLiterals
2 BasicFloatLiteral implements NumericLiteral<1> = ...
3 SignedBasicFloatLiteral implements SignedNumericLiteral<1> = ...
4 BasicDoubleLiteral implements NumericLiteral<1> = ...
5 SignedBasicDoubleLiteral implements SignedNumericLiteral<1> = ...
```

The above literals need a variety of tokens, which themselves are defined using fragments:

```

1 token Digits
2   = Digit+;
3 fragment token Digit
4   = '0'..'9';
5 fragment token ZeroToThree
6   = '0'..'3' ;
7 fragment token HexDigit
8   = '0'..'9' | 'a'..'f' | 'A'..'F' ;
9 fragment token OctalDigit
10  = '0'..'7' ;

```

18.1.3 MCJavaLiterals

MCJavaLiterals extends MCommonLiterals and defines additional Java-specific numbers with four new literal nonterminals:

```

1
2 component grammar MCJavaLiterals extends MCommonLiterals {
3   IntLiteral implements NumericLiteral <100> =
4     source:Num_Int ;
5   LongLiteral implements NumericLiteral <99> =
6     source:Num_Long ;
7   FloatLiteral implements NumericLiteral <100> =
8     source:Num_Float ;
9   DoubleLiteral implements NumericLiteral <100> =
10  source:Num_Double ;

```

IntLiteral implements only NumericLiteral and not SignedNumericLiteral because MCJavaLiterals treats `-1_000` as two individual tokens and also allows spaces in between for example `- 1_000`.

The appropriate tokens consist of a larger number of token fragments (not shown in detail):

```

1 token Num_Int
2   = DecimalIntegerLiteral | HexIntegerLiteral
3     | OctalIntegerLiteral | BinaryIntegerLiteral;
4 token Num_Long
5   = DecimalIntegerLiteral IntegerTypeSuffix
6     | HexIntegerLiteral IntegerTypeSuffix
7     | OctalIntegerLiteral IntegerTypeSuffix
8     | BinaryIntegerLiteral IntegerTypeSuffix;
9 token Num_Float
10  = DecimalFloatingPointLiteral | HexadecimalFloatingPointLiteral;
11 token Num_Double
12  = DecimalDoublePointLiteral | HexadecimalDoublePointLiteral;

```

18. Expression and Type Language Components

However, it can be observed that the extension of literals even affects some of the already defined tokens from the inherited grammar. For example `Digits` now also includes underscores in the middle, allowing numbers like `1_000`:

```

1  @Override
2  fragment token Digits
3    = Digit (DigitOrUnderscore* Digit)?;
4  fragment token DigitOrUnderscore
5    = Digit | '_';

```

The AST nodes for the new literals again provide `getValue()` and `getSource()` methods. The following list summarizes additional features in the Java numbers in comparison to `MCCCommonLiterals`:

- underscores within the numbers for better readability,
- a leading 0 before a number identifies it as an octal number, e.g. `016` in octal corresponds to 14 in decimal,
- a leading `0x` or `0X` before a number identifies it as a hexadecimal number, e.g. `0X9F`,
- a leading `0b` or `0B` identifies it as a binary number, e.g. `0b10110` equals 22.

18.2 Expressions in various Variants

The expression definition grammars are split in a top level `ExpressionsBasis` and a number of implementing component grammars, each focusing on a certain aspect. Figure 18.7 shows their extension structure as well as some operators provided by the respective grammars.

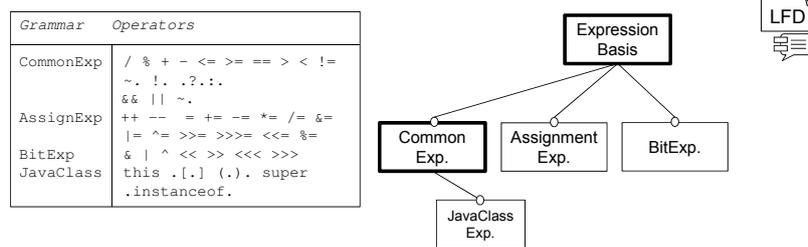


Figure 18.7: Grammars defining Expression

ExpressionsBasis provides the core nonterminal `Expression`, but also offers the core elements that are typically helpful.

CommonExpressions provides the standard form of expressions, including numeric operators, logical operators, etc. as common in programming languages. Again these are inspired by Java and may differ from certain other kinds of languages.

AssignmentExpressions groups all operations with side effects so that they can easily be omitted for pure functional or logic languages.

BitExpressions provides operators for manipulating numbers, which are then interpreted as bit fields.

JavaClassExpressions defines Java specific class expressions like `super`, `this`, type casts, etc. This grammar should only be included, when a mapping to Java is intended and the full power of Java should be available in the modeling language.

We discuss certain design decisions for the `Expression` grammar hierarchy, but omit a complete description here.

It is a good design principle to define component grammars as independent as possible. Ideally, the extension hierarchy as shown in Figure 18.7 should be flat. However, in the case of `JavaClassExpressions` some of the elements of grammar `CommonExpressions` are used, which stacks up the dependencies slightly. Furthermore, expressions are dependent on the literals, i.e. the `ExpressionsBasis` grammar includes the `MCLiteralsBasis` grammar, but no deeper dependencies exist.

The `Expression` and `Literal` hierarchies have also split their duties with respects to defining tokens: With the exception of implicit tokens, such as infix operators, all literal tokens have already been defined and can all be just imported.

18.2.1 ExpressionsBasis

Like `MCLiteralsBasis`, a core interface nonterminal, here `Expression`, is defined in the following listing. Furthermore, the embedding of literals is defined and the very typical argument syntax is provided:

```

1 component grammar ExpressionsBasis
2     extends MCBasics, MCLiteralsBasis {
3     interface Expression;
4
5     NameExpression implements Expression <350>
6         = Name;
7
8     LiteralExpression implements Expression <340>
9         = Literal;
10
11     Arguments
12     = "(" (Expression || ",") * ")";
13 }
```

In production `NameExpression` nonterminal `Name` does not refer to a specific kind of symbol, because in general many different kinds of symbols are possible. Actually, when in a concrete language only one kind of symbol is possible, this specific production can be overridden and a reference, like `Name@Attribute` added.

Expression productions all have a priority to ensure correct parsing of prefix, infix and postfix operations. A more detailed explanation of priorities can be found in Section 4.2.8.

18.2.2 CommonExpressions

CommonExpressions contains the typical expressions, such as various arithmetic expressions, method respectively function calls, brackets and the functional if-then-else (.?.:.).

The excerpt below also shows the nonterminal FieldAccessExpression (l. 10). It allows to select fields (attributes).

An extra nonterminal InfixExpression (l. 3) serves as a common interface for all infix operations, that always implement a left and a right hand side as well as the operator in form of a string. This common interface nonterminal can for example be used in visitors to uniquely handle all infix operations within one method. On the contrary, if individual handling is needed, then the visitor methods may directly apply on the implementing AST nodes, such as MultExpression. This allows us to prevent a lengthy switch statement that handles all the operators. Even more important extensibility with additional infix operations is only guaranteed by use of the generated visitors.

The priority <180> on MultExpression vs. <170> on PlusExpression (ll. 14) ensures that $a+b*c$ is parsed as $a+(b*c)$. If priorities are equal, then a left associative parsing is used: $d-e+f$ is equal to $(d-e)+f$. We choose the numbers 180 and 170 with a gap in between because if a new operator shall be between both, it can be added with e.g. 175 without renumbering and thus touching the given ones. The priority <180> needs to be added directly after Expression because the priority is defined with respects to the Expression hierarchy.

```
1 component grammar CommonExpressions MCG CommonExpressions
2     extends ExpressionsBasis {
3     interface InfixExpression =
4         left:Expression operator:"" right:Expression;
5
6     CallExpression implements Expression <240> =
7         Expression Arguments;
8     astrule CallExpression = Name;
9
10    FieldAccessExpression implements Expression <290> =
11        Expression "." Name;
12
13    // some of many infix expressions:
14    MultExpression implements Expression <180>, InfixExpression =
15        left:Expression operator:"*" right:Expression;
16    PlusExpression implements Expression <170>, InfixExpression =
17        left:Expression operator:"+" right:Expression;
18    MinusExpression implements Expression <170>, InfixExpression =
19        left:Expression operator:"-" right:Expression;
20 }
```

Most interesting is the `CallExpression` (ll. 6). This expression is meant for method calls, like `foo(a,b)`, but method names may be qualified, such as `x.y.foo(a,b)`, field access, like `x.y.foo` looks similar, and even mixed forms, where `x.y` is a field access are possible. These expressions are intrinsically ambiguous. Therefore, call expression is at first parsed like a normal expression with arguments and after parsing the AST is rearranged to form a call expression. A corresponding transformation is implemented in form of the visitor `NameToCallExpressionVisitor`. After a model is parsed the visitor traverses the AST once and adapts it accordingly. An error is issued, if the expression was ill-formed.

18.2.3 BitExpressions

`BitExpressions` defines the expressions for bit manipulations on numbers. This part is again identical to Java.

However, the grammar contains a peculiarity that has already been discussed in Section 4.1.1: Token `>` can be combined in the following piece `List<List<String>>`. Parsing will fail, if `>>` is defined as token too. The grammar directive `splittoken` helps here:

```

1 component grammar BitExpressions MCG BitExpressions
2     extends ExpressionsBasis {
3
4     splittoken ">>", ">>>";
5
6     RightShiftExpression implements Expression <160>, ShiftExpression =
7     left:Expression
8     shiftOp:">>"
9     right:Expression;
10    LogicalRightShiftExpression implements Expression <160>,
11                                ShiftExpression =
12    left:Expression
13    shiftOp:">>>"
14    right:Expression;
15 }
```



Tip 18.8: Decomposing Tokens with `splittoken`

`splittoken` can be used to ensure correct parsing of accidentally subsequent individual tokens, e.g. `>>` in `List<List<String>>`.

`splittoken` can be added in extending grammars and should not be added without need, because it slows down parsing (and complicates error messages).

18.2.4 AssignmentExpressions

The grammar `AssignmentExpressions` contains all known assignments, such as `=`, `+=`, as well as the prefix and postfix increment and decrement operations `++` and `--`.

AssignmentExpressions encapsulates all expressions that deal with side effects and are not useful in pure logic or description languages, such as the OCL.

18.2.5 JavaClassExpressions

The grammar JavaClassExpressions adds typical Java-specialties to the expression language. Among them are `this` and `super`, type casts, object creation with `new`, `instanceof` and array expressions. These constructs define Java-specific expressions and should only be used when a mapping to Java is intended.

The following excerpt shows some typical definitions. Although it is an expression grammar, in several places it needs to make explicit use of types, which are defined in the type hierarchy. To decouple the grammars and leave open which types are actually available, this grammar uses another mechanism of decoupling, namely *external nonterminals*. Three external nonterminals are introduced:

ExtType may be a type expression, e.g. `List<Person>`, `Set<A>` or only a simple type like `int`,

ExtReturnType is similar to `ExtType`, but in Java also includes the `void` pseudotype.

ExtTypeArgument is used to specify the type in a generic invocation, such as the content of the brackets in `<? extends Person>`.

These external nonterminals are to be filled by appropriate nonterminal definitions for example from the type hierarchy.

```

1 component grammar JavaClassExpressions
2     extends CommonExpressions {
3     // Types
4     external ExtType;
5     // Types including a void return type
6     external ExtReturnType;
7     // Type arguments
8     external ExtTypeArgument;
9
10    ThisExpression implements Expression <280> =
11        Expression "." "this";
12
13    // casting expression uses a type
14    TypeCastExpression implements Expression <230> =
15        "(" ExtType ")" Expression;
16
17    // access of class object uses a type
18    ClassExpression implements Expression <360> =
19        ExtReturnType "." "class";
20
21    // generic invocation may specify type arguments
22    PrimaryGenericInvocationExpression implements Expression <370> =
23        "<" (ExtTypeArgument || ",") + ">" GenericInvocationSuffix;

```

```

24 |
25 | // instance of needs a type as argument
26 | InstanceofExpression implements Expression <140> =
27 |     Expression "instanceof" ExtType;
28 | }

```



Tip 18.9: Decoupling of Grammars with External Nonterminals

External nonterminals, like `ExtType`, can be used to decouple a grammar, by creating a hole that needs to be filled when composing the grammars.

External nonterminals give more flexibility because each hole can have its individual name and be filled independently, even if some external nonterminals are filled with the same content.

The MontiCore grammar itself relies on `JavaClassExpressions` and fills the external nonterminals in a straightforward form using the type infrastructure together with the `MCBasicTypes` via import of `JavaLight`:

```

1 | grammar Grammar_WithConcepts extends MCG Grammar_WithConcepts
2 |     de.monticore.expressions.JavaClassExpressions,
3 |     de.monticore.JavaLight,           ...
4 | {
5 |     ExtType = MCType;
6 |     ExtReturnType = MCReturnType;
7 |     ExtTypeArgument = "<" (MCTypeArgument || ",")+ ">*";
8 |     // Empty TypeParameters
9 |     ExtTypeParameters = ;
10 | }

```

When using `JavaClassExpressions` together with generic types, an additional context conditions disallowing the creation of a generic expression on a `ClassExpression` should be added. An expression like `A<String>.class` is also not allowed in Java. The class `NoClassExpressionForGenerics` provides the context condition implementation.



Tip 18.10: Handle different Forms of Expressions via Context Conditions

It is possible that a language uses expressions in several places. If different forms of expressions are to be permitted at these places, then it is advisable to regulate this via specifically applicable context conditions.

Context conditions allow specific messages to be issued that are comprehensible to the user, especially if expression elements are used at positions where they are not permitted.

18.3 Symbols

Symbols have names. Whenever a new name is introduced, a symbol is created and can be used wherever it is visible. Expressions use for example variable names and function calls of different forms because variables may also be parameters or attributes. MontiCore therefore decouples the introduction of symbols from their use.

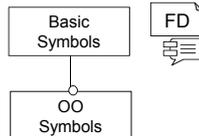


Figure 18.11: Component grammar defining symbols

The following two grammars (as shown in Figure 18.11) introduce typical symbols for expression languages. We found it helpful to provide two layers:

BasicSymbols introduces the core version of several symbols, usable in many forms of languages, and

OOSymbols refines these symbols towards object-oriented languages, such as Java. This refinement is basically an extension with additional properties (symbol attributes) that OO languages typically have.

These two grammars do not provide any concrete syntax and influence the abstract syntax only on the symbol and scope level. For this they introduce only interface nonterminals that are also symbols and sometimes contain their own scopes. These grammars are meant for reuse and sharing of common symbols. Extending grammars may reuse these symbols directly or build additional extending kinds of symbols with additional attributes.



Tip 18.12: Sharing Common Symbols

`BasicSymbols` shows a possibility to share simple infrastructure among different languages, that otherwise do not know from each other.

This typical mechanism of decoupling can be applied to AST nodes as well as for symbol nodes.

`BasicSymbols` or its a refined version `OOSymbols` may serve as well reusable core sets of symbols, but typically additional kinds of symbols, for example `StateSymbols` in automata, `ActionSymbols` in connectivity diagrams, etc. are needed and should not be coupled tightly to the provided symbols (i.e. as fixed subclasses) because there are many forms of mappings possible.

18.3.1 BasicSymbols

A diagram usually has a name. If the diagram name is used as a type as well, such as in Java or MontiArc, then a more specialized version might be useful. For the standard case

it seems however sufficient to simply use the resulting `DiagramSymbol` for references to other diagrams.

We found it convenient to manage diagram symbols as simple symbols without any scope containing capabilities because the artifact scope already manages the set of symbols in an artifact (again: that changes when the diagram name becomes a type as e.g. in Java).

Formally we need to distinguish between the name of the diagram and the name of the containing artifact. Please note that `import` statements formally do not target the diagram name, but the artifact name and location. However, the convention that the diagram and the artifact have the same name is useful and simplifies symbol lookup.

```

1
2 component grammar BasicSymbols extends de.monticore.MCBasics {
3
4   interface symbol Diagram = Name;
5
6   interface scope symbol Type = Name ;
7   symbolrule Type =
8     superTypes: de.monticore.types.check.SymTypeExpression* ;
9
10  interface symbol TypeVar extends Type = Name;
11
12  interface symbol Variable = Name ;
13  symbolrule Variable =
14    type: de.monticore.types.check.SymTypeExpression
15    isReadOnly: boolean ;
16
17  interface scope symbol Function = Name ;
18  symbolrule Function =
19    returnType: de.monticore.types.check.SymTypeExpression ;
20 }

```

The `TypeSymbol` is meant to represent the essence of a type. It may be extended to refer to possible supertypes, which are added through a `symbolrule` directive. The `TypeSymbol` is meant to potentially contain variables (record elements, fields, attributes, etc.) or associated functions (respectively OO methods) and is therefore not only a symbol, but also has a scope associated to manage those future extensions.

The MontiCore RTE provides the class `SymTypeExpression`, which is essentially a composite structure describing a complete *type expression*, such as `Map<int, Set<T>>` (cf. Section 18.6). `SymTypeExpression` differs from `TypeSymbol` because the latter only contains a type symbol, i.e. `Map`, but not concrete arguments.

A `TypeVarSymbol` is used as an unbounded argument in generic types, for example type variable `T` in `Map<Integer, T>`. Type variables have a similar duty as normal variables, however, they carry types that need to be instantiated upon use. Type variables only occur when defining new generic type classes or additional functionality (methods) that is generic in some of the type arguments.

`VariableSymbol` describes relevant information about variables, which includes the type and potential read-only access to the variable. This is a common abstraction for local variables in methods and functions, parameters, attributes, but also the read-only ports in architectural descriptions, such as in SysML or MontiArc.

A function is defined by its signature. `FunctionSymbol` stores such signatures in form of an explicitly added return type using again `symbolrule`, and a list of parameters. The parameters are stored as variable symbols in the associated scope.

18.3.2 OOSymbols

The grammar `OOSymbols` extends the grammar `BasicSymbols` because it mainly adds refinements of the previously defined symbols. These refinements are dedicated to store additional information, that is especially typical for object-oriented languages and especially Java-based languages, such as visibility, static accessibility, or the distinction between class and interface.

Subclassing has an interesting advantage: Assume model B only knows `BasicSymbol` kinds, but model A only defines symbols of `OOSymbol` kinds. B can simply ignore the extra attributes when importing symbols of A, which is directly implementable in B's symbol loading facilities without the languages of A and B introducing dependencies.

```
1 component grammar OOSymbols extends BasicSymbols { MCG OOSymbols
2
3 interface scope symbol OOType extends Type = Name ;
4 symbolrule OOType =
5     isClass: boolean
6     isInterface: boolean
7     isEnum: boolean
8     isAbstract: boolean
9     isPrivate: boolean
10    isProtected: boolean
11    isPublic: boolean
12    isStatic: boolean
13    isFinal: boolean ;
14
15 interface symbol Field extends Variable = Name ;
16 symbolrule Field =
17     isPrivate: boolean
18     isProtected: boolean
19     isPublic: boolean
20     isStatic: boolean
21     isFinal: boolean ;
22
23 interface symbol Method extends Function = Name;
24 symbolrule Method =
25     isConstructor: boolean
26     isMethod: boolean
27     isPrivate: boolean
```

```

28     isProtected: boolean
29     isPublic: boolean
30     isStatic: boolean
31     isFinal: boolean
32     isElliptic: boolean ;
33 }

```

Using so many booleans is simple and efficient from the developers point of view, but of course wastes some memory. More compact solutions are possible. Most boolean attributes are self explained, so we mention only that `isElliptic` is true when the method allows to repeat the last argument.

18.4 Types: From Simple To Generic

Types are an important concept for many languages. Types introduce a controlled form of redundancy that is very helpful for efficient quality assurance because certain kinds of errors are detected by the compiler instead of extensive run-time tests to detect those. Furthermore, in modern languages types are also used as a constructive mechanism to select the appropriate functions to call. This includes overloading of functions, but in particular dynamic lookup in object-oriented languages needs a notion of types and extensibility on these types.

This is the reason why MontiCore provides a set of grammars representing commonly used types like primitive, generic, and array types. These type grammars extend each other mainly in four layers as shown in Figure 18.13. However, array types were extracted and only extend the basis types to flexibly choose whether to include arrays. Figure 18.13 also contains a number of examples of types that can be expressed.

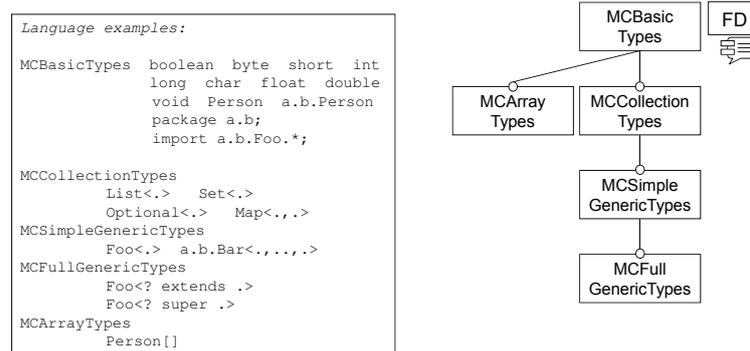


Figure 18.13: Overview over the types grammar hierarchy

- As usual, there is a top level `MCBasicTypes` grammar that introduces the core nonterminal `MCType`. `MCBasicTypes` also introduces primitive types and related commonly used elements.

- The grammar `MCCollectionTypes` provides exactly the four named generic types. The reason for these four types is that in the early stages of development, generic typing is not useful since generic types are usually added during implementation and therefore not yet relevant in the analysis phase. For example in class diagrams that capture requirements generic types should be avoided. On the other hand associations provide exactly these four generic types as realization techniques.
- The grammar `MCSimpleGenericTypes` provides the possibility to use arbitrary generics and introduce your own additional generics, however, does not allow to constrain the argument types in any form.
- The grammar `MCFullGenericTypes` provides the full type infrastructure that for example Java realizes. The typing system will be complicated and we honestly advise to avoid this kind of typing infrastructure in your own language.
- The grammar `MCArrayTypes` provides the syntax to express array type expressions. Arrays are orthogonal to the generic extensions and thus can be combined with any of the above variants.

The grammars shown in Figure 18.13 are explained in the following subsections in detail.

18.4.1 MCBasicTypes

As already mentioned, `MCBasicTypes` contains the basic interface `MCType`. It is the top level interface for all kinds of types except `void`, which is added in `MCType`. `MCType` is the extension point for all other forms of types.

`MCType` represents all primitive types supported by Java. `MCType` is introduced to contain names of freely defined types, like `Person`. It is also an extension point for generic types and, therefore, introduced as interface, even though only one implementation, namely `MCType`, exists here.

```

1
2 component grammar MCBasicTypes extends de.monticore.MCBasics {
3
4   interface MCType;
5
6   MCType implements MCType =
7     primitive: [ "boolean" | "byte" | "short" | "int"
8                 | "long" | "char" | "float" | "double" ];
9
10  interface MCType extends MCType;
11
12  MCType implements MCType = MCType;
13
14  MCType = MCType | MCType;
15  MCType = "void";
16 }
```

The grammar also defines the additional reusable nonterminals `MCQualifiedName`, `MCPackageDeclaration`, and `MCImportStatement`, commonly used in many artifacts as shown in this excerpt:

```

1 MCG MCBasicTypes
2 component grammar MCBasicTypes extends de.monticore.MCBasics {
3
4   MCQualifiedName =
5     parts:(Name || ".")+;
6
7   MCPackageDeclaration = "package" MCQualifiedName & ";";
8
9   MCImportStatement =
10    "import" MCQualifiedName ( "." Star:["*"] )? ";";
11 }
```

18.4.2 MCCollectionTypes

Grammar `MCCollectionTypes` introduces the interface nonterminals `MCGenericType`, which extends `MObjectType`, and `MTypeArgument`, which is later extended for the general generic types. This grammar allows exactly the four predefined generic types, e.g. usable as `List<Integer>`, `Set<Integer>`, `Map<String, socnet.Person>`, and `Optional<ASTAutomaton>`.

This grammar hardcodes the names of generic types using the `key(.)` statement for temporary keywords that can be used as normal names in other contexts.

In this grammar `MTypeArgument` allows only primitive and qualified types, which disables nesting of generic types and thus disallows e.g. `List<List<int>>`. Only the next grammar `MCSimpleGenericTypes` will allow this.

```

1 MCG MCCollectionTypes
2 component grammar MCCollectionTypes
3   extends MCBasicTypes {
4   interface MCGenericType extends MObjectType;
5
6   MListType implements MCGenericType <200> =
7     key("List") "<" MTypeArgument ">";
8   MOptionalType implements MCGenericType <200> =
9     key("Optional") "<" MTypeArgument ">";
10  MMapType implements MCGenericType <200> =
11    key("Map") "<" key:MTypeArgument "," value:MTypeArgument ">";
12  MSetType implements MCGenericType <200> =
13    key("Set") "<" MTypeArgument ">";
14
15  interface MTypeArgument;
16  MBasicTypeArgument implements MTypeArgument <200> =
17    MCQualifiedType;
18  MPrimitiveTypeArgument implements MTypeArgument <190> =
```

```

18 |         MCPPrimitiveType;
19 |     }

```

The next grammar `MCSimpleGenericTypes` will also overload the nonterminal `MCGenericType` with a more generalized processing of generic types. However, by clever choice of the priorities it is ensured that types, which are parsable by `MCCollectionTypes`, are still parsed as such. For example, `List<Integer>` is still parsed as a `ListType` when using the `MCSimpleGenericTypes` even though parsing it as the nonterminal `MCBasicGenericType` would be possible.

18.4.3 MCSimpleGenericTypes

Simple generics are nestable type expressions, such as `List<Foo<int>>` with arbitrary generic types, but without possibilities to constrain the argument types (i.e. no `<? super Foo>`).

For this generalization only two adaptations have to be made: (1) `MCBasicGenericType` extends `MCGenericType` and overrides the already existing four special forms of generic types (`List`, etc.) into the more general form. (2) `MCCustomTypeArgument` extends the possible type arguments two arbitrary types, therefore allowing *nested composition* of type expressions.

```

1 component grammar MCSimpleGenericTypes MCG MCSimpleGenericTypes
2     extends MCCollectionTypes {
3
4     MCBasicGenericType implements MCGenericType <20> =
5         (Name || ".")+ "<" (MCTypeArgument || ",")* ">";
6
7     MCCustomTypeArgument implements MCTypeArgument <20> = MCType;
8 }

```

18.4.4 MCFullGenericTypes

The grammar `MCFullGenericTypes` covers Java types completely. The nonterminal `MCWildcardTypeArgument` represents a wildcard type as a type argument that is constrained by an upper- or a lower bound. `MCMultipleGenericType` extends the typing also for inner generic types, which are allowed in Java as well.

```

1 component grammar MCFullGenericTypes MCG MCFullGenericTypes
2     extends MCSimpleGenericTypes {
3
4     MCWildcardTypeArgument implements MCTypeArgument =
5         "?" ( ("extends" upperBound:MCType)
6             | ("super" lowerBound:MCType) )?;
7

```

```

8  MCMultipleGenericType implements MCGenericType, MCType =
9      MCBasicGenericType // complex Outer Type qualification
10     "." (MCInnerType || ".")+ ;
11
12  MCInnerType = Name ("<" (MCTypeArgument || ",")+ ">")?;
13 }

```

18.4.5 MCArrayTypes

The grammar `MCArrayTypes` introduces arrays. Since arrays are rarely used in modeling languages, they have been placed separately in their own grammar. This allows language designers to integrate arrays if required, but they are not forced to do so.

The nonterminal production `MCArrayType` adds the possibility to model array types, which are generally defined using `[]`, possibly repeatedly. Therefore, the dimension of the array type is incremented upon each parse step, instead of storing the parse tree. For this, a dimension attribute is added to the AST node of nonterminal `MCArrayType` using the `astrule` directive.

```

1  component grammar MCArrayTypes MCG MCArrayTypes
2      extends MCBasicTypes {
3
4      MCArrayType implements MCType =
5          MCType
6          ("[" "]" {_builder.setDimensions(_builder.getDimensions()+1);}+);
7
8          // counter dimensions counts the array depth
9          astrule MCArrayType =
10             dimensions:int;
11 }

```

18.5 Using Base Grammars

This section demonstrates how to use these previously explained base grammars in a language. In general, the new language just extends the needed grammars.

Listing 18.15 shows an example grammar called `MyBasicLanguage` that provides some syntax and integrates expressions, types, and statements in its basic forms. The grammar is marked as a component and is thus meant for reuse. Therefore it only uses the most basic grammars for expressions, types, and statements: `ExpressionsBasis`, `MCBasicTypes`, and `MCStatementsBasis`. This allows the grammar to determine where expressions, statements and types are used without having to specify which form is used.

After defining the grammar `MyBasicLanguage`, now specific extensions of this basic grammar can be created with different type, expression, literal, and statement alternatives.



Tip 18.14: Choosing Grammars to Extend

When choosing the grammars to extend it is advisable to consider which grammars are extended transitively and which need to be extended directly. It is not necessary to extend transitively reachable grammars directly, however, in case non-terminals of a grammar are locally used in the grammar under development it is advised to extend the grammar explicitly.

For example, if the grammars `MCommonStatements` and `MCBasicTypes` are extended, it is sufficient to extend `MCommonStatements` because then the `MCBasicTypes` is already extended transitively.

However, in case nonterminals of `MCBasicTypes` are used in the grammar directly, `MCBasicTypes` can be extended explicitly in addition. Extending a grammar multiple times by extending it directly and transitively does not create errors but can make the dependency graph more complex than needed.

Another advice when choosing the grammars to extend is to choose only those grammars that are currently needed. For example, if a language uses types, there are five grammars to choose from. However, if the language does not need a certain form of types yet, it is advised to choose the most basic grammar as more complex grammars can easily be integrated via language extension. This is especially useful if the language is meant for extension and the choice of the used types should be delayed. The other way around, i.e. restricting the language, is more complex but also possible. If a larger grammar was chosen than actually needed then not allowed nonterminals can be forbidden by context condition.

```

1 component grammar MyBasicLanguage extends ExpressionsBasis,
2                                     MCBasicTypes,
3                                     MCStatementsBasis {
4   MyLanguage = "myLang" "{" (MyDefinition | Expression)* "}";
5   MyDefinition = MCType Name "=" MCStatement;
6 }

```

Listing 18.15: Grammar `MyBasicLanguage`, showing the usage of the base grammars

Listing 18.16 shows an example of a grammar extending `MyBasicLanguage` which only uses simple variants of types, expression, literals, and statements.

If the provided forms of e.g. types and literals are not sufficient, then of course further forms can be added locally or by separate grammars. The newly defined nonterminals just need to implement the predefined interface of the grammar. For example, if a new type is needed, the `MCType` interface can be implemented. In some cases e.g. when adding expression forms a priority also should be defined for the new production. For more information about priorities in MontiCore grammars check out Subsection 4.2.8.

Since the base grammars are mostly orientated on Java, the chosen example extend these base grammars with additional definitions similar to Kotlin. Kotlin is a programming language based on Java, but containing some additional features. For this purpose a new grammar was created that extends the `MCCollectionTypes` with a special type for

```

1 package mc.basegrammars;
2
3 grammar SimpleLanguage extends MyBasicLanguage,
4                               CommonExpressions,
5                               MCCollectionTypes,
6                               MCCCommonStatements,
7                               MCCCommonLiterals {
8 }

```

Listing 18.16: Grammar `MySimpleBaseLanguage.mc4`, containing simple base grammar alternatives

arrays as shown in Listing 18.17. The priority `<200>` is important and is based on the priorities of the other generic types in `MCCollectionTypes`.

```

1 grammar KotlinCommonGenericTypes extends MCCollectionTypes {
2
3     MCArryGenericType implements MCGenericType <200> =
4         key("Array") "<" MCTypeArgument ">";
5
6 }

```

Listing 18.17: The grammar `KotlinCommonGenericTypes.mc4`

18.6 Type Checking in MontiCore Languages

As explained before, types are useful to detect typing errors at compile time. Type checking is thus only a special, but sophisticated form of context condition checking.

Types, such as `boolean`, `Person`, `List<int>`, are well known for *values* calculated in *expressions*, stored in *variables* and transported in *parameters*. They can also be extended to other kinds of modelling elements, such as *pins*, *ports*, or *channels*, where values (or objects) play a role. Type systems may be extended to functions, encoding the function signature and allowing higher-order-functions using other functions as parameters, etc.

It is also possible that entirely different kinds of type systems are developed for a language. For example, an action language may encode types of actions, including a notion of subtyping that realizes a refinement for the actions. It is also an option to realize a sophisticated structure of stereotypes for the UML that act as a kind of meta-type system for classes in a class diagram.

In the following, we only concentrate on the relatively familiar type system for expressions, which builds on the previously discussed expression grammars.

MontiCore expressions and therefore also the type system described below offer an extensible and modular infrastructure. To achieve this, MontiCore offers an individual, composable type check for each of the expression grammars and also allows several variants of

type systems. These appropriate type check parts can then be selected analogously to the grammars and combined to a composed type check.

A type checking infrastructure consists of the following elements:

- A representation of the *type system* is discussed in Section 18.6.1. It is used for storage of type information in the symbol table.
- A *type check algorithm*. Given a model element, e.g. an expression, and a type, it decides whether the runtime evaluation of the element will deliver a value fitting to the desired type (i.e. whether the evaluation is *type safe*, see Section 18.6.2).
- A *type inference algorithm*. Given a model element, the algorithm calculates the minimal type that an evaluation we will always deliver (see Section 18.6.2).

In the following, we are going to roughly describe how to use the existing type checking infrastructure (Section 18.6.2) and also gives some hints how to extend it (Section 18.6.3).

18.6.1 Types in a Symbol Table: SymTypes

MontiCore uses the RTE class `SymTypeExpression` and its subclasses for its internal representation of types. This is a compact, composite structure as shown in Figure 18.18.

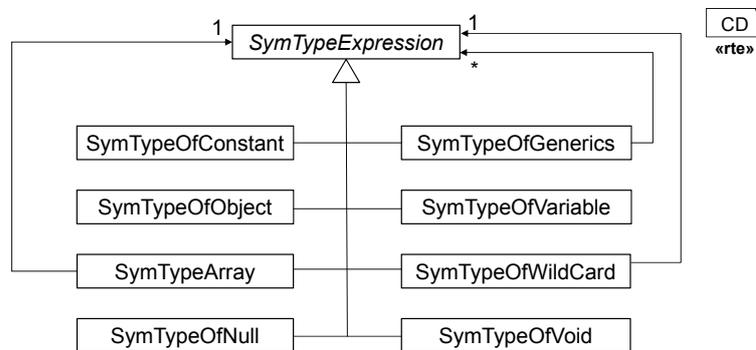


Figure 18.18: The `SymTypeExpression` class hierarchy

The central class is the abstract class `SymTypeExpression` that acts as a composite. The composite is the optimal structure to efficiently represent complex types, such as generics, arrays, etc. We generally speak of *type expressions* because the structure of these types is built in a similar way as ordinary expressions. However, type expressions and normal expressions are not to be confused. The same holds for type variables and normal variables.

SymTypeExpression is the top composite class representing all forms of type expressions.

SymTypeConstant represents *primitive types* like `int` and `long`.

SymTypeOfObject models standard classes like `Person` or `Student`, which do not have generics in their signature.

SymTypeArray represents *arrays* over any other type, i.e. e.g. `int[][]` or `Person[]`.

SymTypeOfGenerics is to describe *generic types*, such as `List<...>`, `Map<...,...>`. The generic type may have several arguments, which are themselves types. In its most general form, the typed arguments may themselves be constrained (i.e. there must be of a specific subtype or supertype).

SymTypeVariable models a *type variable*, which is not to be confused with an ordinary variable. In a type expression, like `Person<T>`, the `T` acts as type variable, which can be instantiated upon use of the generic type `Person`.

SymTypeOfWildcard allows to model type expressions of forms like `List<? extends Person>`.

SymTypeVoid represents the special "type" `void` that is, for example, used in Java to represent the absence of a return type.

SymTypeOfNull models the type of the special value `null` (or `nil`).

The above classes allow to represent a variety of types, known from Java or also from the UML. If needed, they can be extended, e.g., to cover additional typing information, such as SI-Units, like `km/h`.

The `SymTypeExpression` class hierarchy is independent from any AST and symbol infrastructures. We have deliberately chosen this form of realization to allow different concrete syntactic shapes for the definition of these types, but share the same algorithmic realization of the type check. Therefore, this type check cannot only be applied to Java, but also to C++ like or other kinds of languages. Furthermore, the stored symbol tables are better decoupled because they do not enforce sharing of foreign AST classes. It may even be that there is no concrete syntax to represent certain types, although these are highly relevant for type checking, e.g. there is often no `Null` type.

In practical languages, we recommend to not fully explore the possible type system infrastructure, but to restrict to a practically useful subset only. For example full generics are only rarely necessary in modeling languages. A relatively small reduction of typing expressibility greatly produces the complexity of type checks, both from the view of development, but also from runtime execution times.

The representation of type expressions above is primarily used with the symbol table infrastructure where, for example, `VariableSymbols` are typed using these types. Because symbols of various kinds are dedicated to be usable in foreign models, symbols and their types are also stored in the symbol table artifacts. For this purpose, the RTE provides the `SymTypeExpressionDeSer` and a set of additional classes to store and load the above type structures. MontiCore handles this relatively automatic.

18.6.2 Using Type Checks: the Type Check API

The provided type checks for the expression languages mentioned above concentrate on literals and expressions, but also incorporate the possibility to define concrete types in the

18. Expression and Type Language Components

language. This is also reflected in the central `TypeCheck` class of MontiCore as shown in Listing 18.19. This class shows the core type checking methods.

```
1 Java  
2 public class TypeCheck {  
3   // derive a symtype from the AST node representing it  
4   public SymTypeExpression symTypeFromAST(ASTMCType ast)  
5   public SymTypeExpression symTypeFromAST(ASTMCVoidType ast)  
6   public SymTypeExpression symTypeFromAST(ASTMCReturnType ast)  
7   public SymTypeExpression symTypeFromAST(ASTMQualified Name ast)  
8  
9   // derive the most precise possible symtype of an expression  
10  public SymTypeExpression typeOf(ASTExpression expr)  
11  
12  // derive the symtype of a literal  
13  public SymTypeExpression typeOf(ASTLiteral lit)  
14  
15  // is right type assignment compatible to left type  
16  // e.g. right is a subtype or a specialized numeric type  
17  public static boolean compatible(SymTypeExpression left,  
18                                   SymTypeExpression right)  
19  
20  // is the result of an expression compatible to a needed type  
21  public boolean isOfTypeForAssign(SymTypeExpression type,  
22                                   ASTExpression      exp)  
23  
24  // some convenience  
25  public static boolean isBoolean(SymTypeExpression type)  
26  public static boolean isInt(SymTypeExpression type)  
27  public static boolean isDouble(SymTypeExpression type)  
28  public static boolean isFloat(SymTypeExpression type)  
29  public static boolean isLong(SymTypeExpression type)  
30  public static boolean isChar(SymTypeExpression type)  
31  public static boolean isShort(SymTypeExpression type)  
32  public static boolean isByte(SymTypeExpression type)  
33  public static boolean isVoid(SymTypeExpression type)  
34  public static boolean isString(SymTypeExpression type)  
35 }
```

Listing 18.19: Methods of the `TypeCheck` class

symTypeFromAST: The first group of methods `symTypeFromAST` converts a type, which is explicitly defined in the AST of a model into a `SymTypeExpression`, which is then used for the type check and is therefore a preparatory function.

The `symTypeFromAST` methods are able to convert the concrete types defined in the typing grammars shown in Section 18.4 to `SymTypeExpressions`. If other concrete representations of types are desired, these functions can be easily replaced or extended.

typeOf methods derive the `SymTypeExpression` from an Expression AST. These

methods embody the actual *type inference algorithm* that is needed to infer the type of an Expression.

This method is recursive over the examined Expression and assumes that the symbol table is already in place so that the types of all symbols in the expressions (i.e. variables, functions, enums, etc.) can be obtained.

compatible takes two type expressions and checks if the right type is a subtype of the left type, so that in all places where the left type is expected the right type is usable. For example, `int` and `int`, `int` and `long` as well as `Person` and `Student` are compatible assuming the class `Student` extends the class `Person`.

isOfTypeForAssign: The method call `isOfTypeForAssign(type, exp)` checks whether the `ASTExpression exp` will always result in a value that is of a type that is compatible to the type given as the `SymTypeExpression type`.

This is the *actual type check* for expressions.

The function `isOfTypeForAssign` mainly combines the type derivation with `typeOf` for the expression and the compatibility check using `compatible`.

18.6.3 How the Type Check is Configured

The `TypeCheck` class is configured by two special classes to which it delegates the required computations (cf. Figure 18.20).

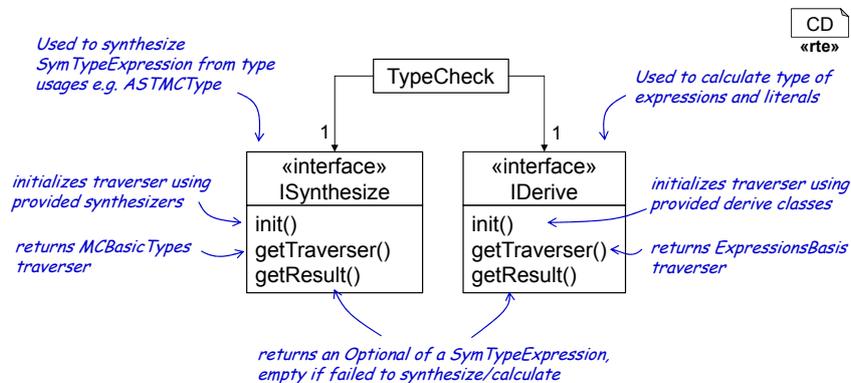


Figure 18.20: Overview over the `TypeCheck` class and Interfaces

`ISynthesize` is the common interface that is used to synthesize a `SymTypeExpression` from a `MCType`.

`IDerive` contains the *type inference algorithm* to calculate a `SymTypeExpression` from an Expression AST. Both interfaces have an `init`, a `getTraverser` and a `getResult` method.

The general approach of the *type inference algorithm* in `IDerive` is to use recursive descent with the visitor pattern to calculate the actual type of an Expression AST. The traverser

in that visitor pattern is stateful and therefore the `init` method resets the traverser each time, before it is used. When initialized, the `Traverser` queried via the `getTraverser` method is executed. Finally, the result can then be retrieved via the `getResult` method.

The class `ISynthesize` acts similar, but maps an `MCType` AST to a `SymTypeExpression` and thus uses a `MCBasicTypesTraverser`.

As described before, the `TypeCheck` has the same modular structure as the `Expression` and as the `MCType` grammars (cf. Section 18.2 and 18.4). In detail, both mechanism are realized by a composed visitor pattern, where individual visitors deal with the respective nonterminals of one individual grammar component.

As a consequence, the MontiCore RTE provides a set of implementations for each of the two interfaces allowing to select and compose the appropriate algorithm.

The standard form to create a `TypeCheck` for a language is to reuse the available implementations of the lists below. If the language contains new forms of syntactic constructs for expressions, then additionally a class that handles the new language constructs need to be written and added.

ISynthesize: Mapping AST Types to SymTypeExpression

The class `ISynthesize` consists of a visitor for calculating types. While traversing the AST of a type, it creates the relevant `SymTypeExpression`.

The following classes are available for the mapping, where the name of each class describes for which kind of types it is applicable:

- `SynthesizeSymTypeFrom`

Each synthesizer handles exactly all nonterminals from the respective grammar. This is why a composition of the `ISynthesize` objects into a combined visitor as shown below is needed.

When a different concrete syntax for types is used or the type expressions are extended by new syntactic constructs, then a new subclass must be defined.

IDerive: Type Inference for Expressions and Literals

The `IDerive` implementations also use a recursive visitor pattern to calculate types for expressions and literals. Of course, developers can create and integrate own classes for extensions or implementations for the synthesis or calculation of types.

Again, many classes are available, each containing a part of the *type inference algorithm*, where the name of the class describes, for which forms of expressions it is applicable:

- `DeriveSymTypeOf`

Example Application of a Type Inference

Listing 18.21 shows an example language MyLang, which allows simple variable definitions. For this purpose, the language combines the provided languages MCBasicTypes, MCArrayTypes, MCCCommonLiterals, and CommonExpressions.

```

1 grammar MyLang extends MCBasicTypes, MCArrayTypes,
2                               MCCCommonLiterals,
3                               CommonExpressions {
4   MyVar = type:MCType var:Name "=" exp:Expression;
5 }

```

Listing 18.21: Example language MyLang

For the type check of the language, the two classes SynthesizeFromMyLang and DeriveFromMyLang are created as explained below (cf. Figure 18.22).

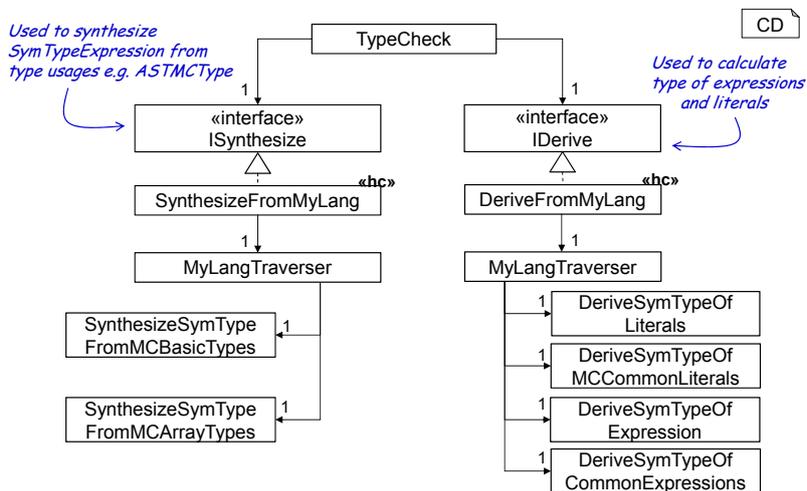


Figure 18.22: TypeCheck configuration for MyLang

As shown, the class SynthesizeFromMyLang combines the classes

- SynthesizeSymTypeFromMCBasicTypes and
- SynthesizeSymTypeFromMCArrayTypes

to synthesize SymTypeExpression from types of types grammars.

DeriveFromMyLang, on the other hand, combines the provided classes

- DeriveSymTypeOfExpression,
- DeriveSymTypeOfCommonExpressions,
- DeriveSymTypeOfLiterals and
- DeriveSymTypeOfMCCCommonLiterals

to calculate the types of expressions and literals. Even though only two imported grammars are explicitly mentioned in the MyLang grammar above, it is necessary to include explicitly all transitively imported grammars as well.

```

1                                     Java SynthesizeFromMyLang
2 public class SynthesizeFromMyLang implements ISynthesize {
3   protected MyLangTraverser traverser;
4   protected TypeCheckResult typeCheckResult;
5
6   @Override
7   public void init() {
8     // use new result wrapper and traverser
9     traverser = MyLangMill.traverser();
10    typeCheckResult = new TypeCheckResult();
11
12    // add Synthesize for MCBasicTypes
13    SynthesizeSymTypeFromMCBasicTypes bt =
14        new SynthesizeSymTypeFromMCBasicTypes();
15    bt.setTypeCheckResult(typeCheckResult);
16    traverser.add4MCBasicTypes(bt);
17    traverser.setMCBasicTypesHandler(bt);
18
19    // add Synthesize for MCArrayTypes
20    SynthesizeSymTypeFromMCArrayTypes at =
21        new SynthesizeSymTypeFromMCArrayTypes();
22    at.setTypeCheckResult(typeCheckResult);
23    traverser.add4MCArrayTypes(at);
24    traverser.setMCArrayTypesHandler(at);
25  }
26 }

```

Listing 18.23: `init()` method of `SynthesizeFromMyLang`

The implementation of the `init` method of the `SynthesizeFromMyLang` class can be seen in Listing 18.23. As described before, it fills a traverser with visitors. To do this, it retrieves the traverser from the mill. Then, for the two types grammars, instances of the provided synthesize classes are added to the traverser. The class `TypeCheckResult` is also provided by MontiCore and serves as a container for the calculated result of the synthesize classes. The implementation of the `init` method of the class `DeriveFromMyLang` is analogous, whereby the provided classes for the involved expression and literals grammars are added to the traverser.

The implementations of the getters `getResult` and `getTraverser` can be seen in Listing 18.24.

The two classes are then used as a `TypeCheck` configuration. Listing 18.25 shows various uses of the `TypeCheck` to check and calculate types using the example language in Listing 18.21.

In line 2, the model is defined as a string and the AST is built through parsing. Here, the boolean variable `x` is assigned to the result of `Expression 3 > 4`.

```

1                                     Java SynthesizeFromMyLang
2  @Override
3  public Optional<SymTypeExpression> getResult() {
4      if (typeCheckResult.isPresentCurrentResult()) {
5          return Optional.of(typeCheckResult.getCurrentResult());
6      }
7      return Optional.empty();
8  }
9
10 @Override
11 public MCBasicTypesTraverser getTraverser() {
12     return traverser;
13 }

```

Listing 18.24: Methods of the TypeCheck class

```

1                                     Java MyLangTest
2  Optional<ASTMyVar> varOpt = parser.parse_String("boolean x = 3 > 4");
3
4  TypeCheck tc = new TypeCheck(new SynthesizeFromMyLang(),
5                               new DeriveFromMyLang());
6
7  ASTMyVar var = varOpt.get();
8  ASTMCType type = var.getType();
9  ASTExpression exp = var.getExp();
10
11 // synthesize SymTypeExpression from type
12 SymTypeExpression symType1 = tc.symTypeFromAST(type);
13
14 // calculate SymTypeExpression for exp
15 SymTypeExpression symType2 = tc.typeOf(exp);
16
17 // check whether the type is boolean
18 assertTrue(TypeCheck.isBoolean(symType1));
19 assertTrue(TypeCheck.isBoolean(symType2));
20
21 // check whether both types are compatible
22 assertTrue(TypeCheck.compatible(symType1, symType2));
23
24 // check whether the expression is of assignable type
25 assertTrue(tc.isOfTypeForAssign(symType1, exp));

```

Listing 18.25: Usage of TypeCheck methods demonstrated in a JUnit test

The TypeCheck is created in l. 4 and during the creation configured with the two hand-crafted classes.

In l. 12ff, the type of the variable (x) and then the type of the expression are determined.

Then in l. 18f, it is checked whether the two SymTypeExpressions are indeed of type

`boolean`. This is a demonstration that the calls of l. 12ff work correctly.

In l. 22, the compatibility of the `SymTypeExpression` is checked with the help of the method `compatible` of the class `TypeCheck`. This test demonstrates the use of the compatibility check.

Finally, in l. 25, the assignability of the expression to the variable is checked. This test finally demonstrates the use of the assignability check, which is essentially only a combination of `typeOf` and `compatible`.

The short version would actually be (1) derive the `symType1` of the variable `x` and the `exp` and call `isOfTypeForAssign` as shown in l. 25.

Chapter 19

Statement Language Components

Procedural statements are often needed in domain specific languages. Statements are executable, which means that their inclusion normally goes more towards an executable DSL that shall be mapped to a normal programming language. This chapter demonstrates on in total ten grammars how to define statements, and especially how to allow language developers to select an appropriate subset.

While all statement grammars are in the spirit of Java and their composition in grammar `MCFullJavaStatements` provides the complete set of Java statements, several of these grammars provide only rather general statements that can easily be translated into other general purpose programming languages.



Tip 19.1: Statement Grammar Components

The grammars discussed in this chapter can be found in the MontiCore repository under:

```
1 Repository: Monticore/monticore github
2 Directory:  monticore-grammar/src/main/grammars/
3 Packages:  de.monticore.statements
```

Files

Rather generic and well suited even for mappings to other target languages are:

```
1 Grammars:   MCCCommonStatements.mc4
2             MCVarDeclarationStatements.mc4
3             MCReturnStatements.mc4
```

Statements build on `ExpressionsBasis` and thus on `MCBasics`, because we allow any expression to be understood as statement as well, even though that is only useful for expressions with side effects and method calls. Component grammar `MCVarDeclarationStatements` and therefore also its descendants, like `MCCCommonStatements`, furthermore includes `MCBasicTypes` and `OOSymbols`. The import structure is visualized in Figure 19.2, also exhibiting the extension dependencies to foreign grammars. Again, root interface nonterminals for statements are introduced in `MCStatementsBasis`. While the extension structure is mainly a tree, MontiCore also offers `MCFullJavaStatements`, a composing grammar that unites all others. Figure 19.3 shows some examples.

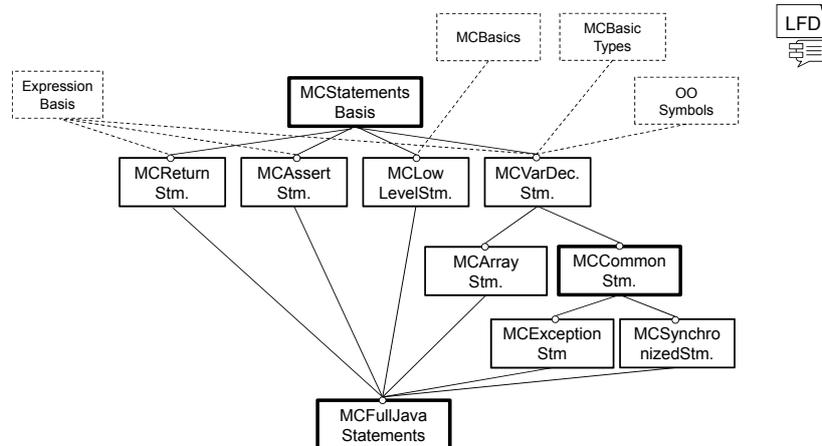


Figure 19.2: Component grammar structure hierarchy of Chapter 19

```

Statement examples:
Person p[] = { foo(3+7), p2, ...}
if (.) then . else .
for ( i = .; .; .) { . }
while (.) .
do . while (.)
switch (.) { case .: .; default: . }
return .
assert . : .
try { . } catch (.) { . } finally { . }
throw .
break .
continue .
label: .
private static final native ...
    
```

Figure 19.3: Some statement examples

19.1 MCStatementsBasis

The basic grammar `MCStatementsBasis` already introduces the main interface `MCStatement` accompanied by a more general `MCBlockStatement` that is expected to be used only within blocks with scopes, such as `{ ... }` and allows additionally to define local variables, like `int a = 7`.

Interface nonterminal `MCModifier` allows adding various modifier variants, e.g., `public`, `final` when needed.

```

1 component grammar MCStatementsBasis {
2   interface MCBlockStatement;
3
4   interface MCStatement extends MCBlockStatement;
5
    
```

```

6  interface MCModifier;
7  }

```



Tip 19.4: Extensible list of Enum Values

Interface nonterminal `MCModifier` is the starter for an extensible list of modifiers.

If they were defined as an enumeration, the number of possible elements would have been fixed. Advantage of the interface approach are extensible modifiers, but these generally need to be handled by (extensible) visitors instead of a fixed `switch` statement.

19.2 MCVarDeclarationStatements

Grammar `MCVarDeclarationStatements` concentrates on the definition of local variables as specific statements. Local variables are typed using `MCType` and can be initialized with concrete values. Arrays are not part of this grammar, but an extension of it called `MCArrayStatements`, which will be explained later in this chapter (cf. Section 19.3). In case arrays are needed, this grammar needs to be used, which only makes sense if arrays are available in the type hierarchy.

```

1  component grammar MCVarDeclarationStatements MCG MCVarDeclarationStatements
2      extends MCStatementsBasis,
3          MCBasicTypes,
4          ExpressionsBasis,
5          OOSymbols {
6      LocalVariableDeclarationStatement implements MCBlockStatement
7          = LocalVariableDeclaration ";" ;
8
9      LocalVariableDeclaration
10         = MCModifier* MCType (VariableDeclarator || ",")+ ;
11
12     VariableDeclarator
13         = Declarator ("=" VariableInit)? ;
14
15     interface Declarator extends Field
16         = Name;
17
18     DeclaratorId implements Declarator
19         = Name;
20
21     interface VariableInit;
22
23     SimpleInit implements VariableInit
24         = Expression;
25 }

```

LocalVariableDeclaration was factored out as an independent nonterminal to enable variable declarations without trailing ";" in other parts of a model.

19.3 MCArrayStatements

MCArrayStatements introduces the possibility to declare arrays with ArrayDeclaratorId and to initialize them with ArrayInit. The latter extends the variable initialization options provided by MCVarDeclarationStatements. Like in Java, ArrayDeclaratorId allows binding the array information not only to the type but also to the variable.

```
1 component grammar MCArrayStatements MCG MCArrayStatements
2     extends MCVarDeclarationStatements {
3 ArrayDeclaratorId implements Declarator
4     = Name (dim:"[" "]"")+ ;
5
6 ArrayInit implements VariableInit
7     = "{" (VariableInit || ",")* (",")? "}" ;
8 }
```

19.4 MCCCommonStatements

As explained, the grammar MCCCommonStatements introduces Java's common control statements and imports variable declarations. It thus builds a complete procedural language that can, for example, be used as an action language, e.g., as the body of transitions in executable statemachines.

MCJavaBlock realizes block statements and can directly be included if statements shall be enclosed in {...}. MCJavaBlock is designed to open up a new scope, allowing to define local variables that are visible in that scope only (non_exporting). Furthermore, a local variable is only accessible after it has been introduced (ordered). This is standard in programming language method bodies, including Java, and therefore should be well known to programmers.

The ExpressionStatement allows using any form of available expression as a statement, including for example method calls.

```
1 component grammar MCCCommonStatements MCG MCCCommonStatements
2     extends MCVarDeclarationStatements {
3
4     scope(non_exporting ordered) MCJavaBlock implements MCStatement
5     = "{" MCBlockStatement* "}" ;
6
7     JavaModifier implements MCM Modifier =
8     Modifier:["private" | "public" | "protected" | "static"]
```

```

9         | "transient" | "final" | "abstract" | "native"
10        | "threadsafe" | "synchronized" | "const" | "volatile"
11        | "strictfp"] ;
12
13 IfStatement implements MCStatement
14     = "if" "(" condition:Expression ")"
15         thenStatement:MCStatement
16         ("else" elseStatement:MCStatement)? ;
17
18 EmptyStatement implements MCStatement
19     = ";" ;
20 ExpressionStatement implements MCStatement
21     = Expression ";" ;
22 }

```

ForStatement embodies a relatively complex structure, including the two possibilities to define its own local variables, which also enforces that a non_exporting ordered scope is opened. Besides traditional loop variables, for can also iterate over lists.

Compared to the ForStatement structure, the WhileStatement and DoWhileStatement are relatively harmless.

```

1 component grammar MCCCommonStatements MCG MCCCommonStatements
2     extends MCVarDeclarationStatements {
3     scope (non_exporting ordered) ForStatement implements MCStatement
4         = "for" "(" ForControl ")" MCStatement ;
5     interface ForControl ;
6     CommonForControl implements ForControl
7         = ForInit? ";" condition:Expression? ";" (Expression || ",")* ;
8     ForInit
9         = ForInitByExpressions | LocalVariableDeclaration ;
10    ForInitByExpressions
11        = (Expression || ",")+ ;
12    EnhancedForControl implements ForControl
13        = FormalParameter ":" Expression;
14    FormalParameter
15        = JavaModifier* MCType DeclaratorId;
16
17    WhileStatement implements MCStatement
18        = "while" "(" condition:Expression ")" MCStatement ;
19    DoWhileStatement implements MCStatement
20        = "do" MCStatement "while" "(" condition:Expression ")" ";" ;
21 }

```

The next part of the grammar deals with the also relatively complex switch statement. It contains a SwitchBlockStatementGroup where both, labels and statements, need to be present and catches incomplete empty labels with an explicit SwitchLabel* at the end. With the BreakStatement, the loop and switch statements can be interrupted.

19. Statement Language Components

```
1 component grammar MCommonStatements MCG MCommonStatements
2     extends MVarDeclarationStatements {
3     SwitchStatement implements MCStatement
4         = "switch" "(" Expression ")"
5           "{" SwitchBlockStatementGroup* SwitchLabel* "}";
6
7     SwitchBlockStatementGroup
8         = SwitchLabel+ MBlockStatement+ ;
9
10    interface SwitchLabel ;
11    ConstantExpressionSwitchLabel implements SwitchLabel
12        = "case" constant:Expression ":" ;
13    EnumConstantSwitchLabel implements SwitchLabel
14        = "case" enumConstant:Name ":" ;
15    DefaultSwitchLabel implements SwitchLabel
16        = "default" ":" ;
17    BreakStatement implements MCStatement
18        = "break" ";" ;
19 }
```



Tip 19.5: Use MCommonStatements in Action-oriented languages

MCommonStatements contains all relevant typical statements that a general-purpose programming language needs to have, but avoids internal return statements, exceptions, low-level statements, asserts, or synchronizing statements.

19.5 MCReturnStatements

Grammar MCReturnStatements adds the return with or without return expression and therefore is also based on ExpressionsBasis.

```
1 component grammar MCReturnStatements MCG MCReturnStatements
2     extends MCStatementsBasis,
3           ExpressionsBasis {
4     ReturnStatement implements MCStatement
5         = "return" Expression? ";" ;
6 }
```

19.6 MCAssertStatements

MCAssertStatements realizes Java's assert statement.

```

1 component grammar MCAssertStatements MCG MCAssertStatements
2     extends MCStatementsBasis,
3         ExpressionsBasis {
4     AssertStatement implements MCStatement
5     = "assert" assertion:Expression (":" message:Expression)? ";" ;
6 }

```



Tip 19.6: Assert Statements with extended Forms of Expressions

In a specification oriented modeling language, assert statements are not meant for efficient execution but for effective writing of invariants and constraints by the developers.

Therefore, it might be useful to extend the assert mechanism massively, for example, by OCL and other kinds of logic or data structure querying expressions.

Please note that importing other forms of expressions has the "side effect" that these additional expressions are then available everywhere. This can be prohibited by issuing context conditions like *"This expression element is not allowed in XY expressions"*, which explain errors much better than if the context-free grammar would include different forms of expressions. Users tend to mix kinds of expressions up and will experience the resulting errors relatively often.

19.7 MCSynchronizedStatements

MCSynchronizedStatements realizes Java's synchronization statement.

```

1 component grammar MCSynchronizedStatements MCG MCSynchronizedStatements
2     extends MCommonStatements {
3     SynchronizedStatement implements MCStatement
4     = "synchronized" "(" Expression ")" MCJavaBlock ;
5 }

```

19.8 MCExceptionStatements

MCExceptionStatements realizes Java's try-catch and the exception throw statement. The three different variants of the TryStatement differ in the optionality of their elements, which are, in this case, encoded in the context-free grammar but might also have been encoded in extra context conditions.

```

1 component grammar MCExceptionStatements MCG MCExceptionStatements
2     extends MCommonStatements {
3     TryStatement1 implements MCStatement

```

19. Statement Language Components

```
4     = "try"
5       core:MCJavaBlock
6       CatchClause+
7       ("finally" finally:MCJavaBlock)? ;
8 TryStatement2 implements MCStatement
9     = "try"
10      core:MCJavaBlock
11      CatchClause*
12      ("finally" finally:MCJavaBlock) ;
13 TryStatement3 implements MCStatement
14     = "try" "(" (TryLocalVariableDeclaration || ";" )+ ";"? )"
15      core:MCJavaBlock
16      CatchClause*
17      ("finally" finally:MCJavaBlock)? ;
18
19 TryLocalVariableDeclaration
20     = JavaModifier* MCTYPE DeclaratorId "=" Expression ;
21
22 CatchClause
23     = "catch" "(" JavaModifier* CatchTypeList Name ")" MCJavaBlock ;
24 CatchTypeList
25     = (MCQualified Name || "|")+ ;
26
27 ThrowStatement implements MCStatement
28     = "throw" Expression ";" ;
29 }
```

19.9 MCLowLevelStatements

MCLowLevelStatements introduces labeled break and continue. In a modern programming style, these statements should rarely be used. Therefore, MontiCore separates them into an extra grammar.

Labels are handled as LabelSymbols, that only contain their name.

```
1 component grammar MCLowLevelStatements MCG MCLowLevelStatements
2     extends MCStatementsBasis,
3         de.monticore.MCBasics {
4     LabelledBreakStatement implements MCStatement
5     = "break" label:Name@Label? ";" ;
6
7     ContinueStatement implements MCStatement
8     = "continue" label:Name@Label? ";" ;
9
10    symbol Label implements MCStatement
11    = Name ":" MCStatement ;
12 }
```

19.10 MCFullJavaStatements

Grammar `MCFullJavaStatements` combines all previously described language components providing the full range of Java statements. It still does not fix types and expressions, which need to be added separately.

```
1 component grammar MCFullJavaStatements extends MCG MCFullJavaStatements
2   MCAssertStatements,
3   MCExceptionStatements,
4   MCLowLevelStatements,
5   MCReturnStatements,
6   MCSynchronizedStatements,
7   MCArraryStatements {
8
9 }
```



Tip 19.7: Composition only Grammars

For reusable library components, it is good to apply the separation of concerns principle, where each grammar focuses on one specific aspect and if desired, a fully integrating language, like `MCFullJavaStatements`, allows to directly import the full set of language components.

Such a compositional grammar typically only consists of a set of `extends`, sometimes external nonterminals are bound and the axiom needs to be clarified in non-component grammars.

Chapter 20

The JavaLight Language

co-authored with Marita Breuer

The `JavaLight` language defines a larger subset of the Java programming language. The grammar, the hand-written symbol table extensions, and a pretty printer are part of the `MontiCore` project. The language introduces Java method declarations, constructor declarations, interface method declarations, attributes, and annotations. The `JavaLight` language neither defines classes nor interfaces. Its purpose is to be easily reusable and extensible for the creation of more complex languages such as the complete Java programming language or software modeling languages. The `JavaLight` language is also used in the `MontiCore` grammar language for specifying ast rules and symbol rules.

In the following, Section 20.1 overviews the extension hierarchy of the `JavaLight` grammar. Afterwards, Section 20.2 describes the nonterminals introduced by the grammar.

20.1 Sublanguage Hierarchy of JavaLight

The feature diagram in Figure 20.1 depicts the hierarchy of the `JavaLight` sublanguages. The `JavaLight` grammar extends the grammars `JavaClassExpressions`, `AssignmentExpressions`, and `MCCCommonStatements`. The grammar `JavaClassExpressions` defines Java-specific class expressions containing `this`, `super`, `new`, and `instanceof`. The grammar `MCCCommonStatements` introduces statements for specifying the control flow via loops (`while`, `for`, `do-while`) and switches (`switch-case`). The grammar `AssignmentExpressions` defines expressions for assignments (e.g., `a = b + 4` and `i++`).

The grammar `JavaClassExpressions` extends the grammar `CommonExpressions`. With this, it embeds common expressions such as arithmetic expressions (e.g., `(a + 5) / 6`), comparisons (e.g., `a > b + 7`), and boolean expressions (e.g., `a && !c`). The grammar `MCCCommonStatements` extends the grammar `MCCVarDeclarationStatements`, which introduces statements for variable declarations (e.g., `private int i = 4;`).

The grammars `CommonExpressions`, `AssignmentExpressions`, and `MCCVarDeclarationStatements` extend the grammar `ExpressionsBasis`. This enables using expressions containing names and literals as well as to specify argument lists.

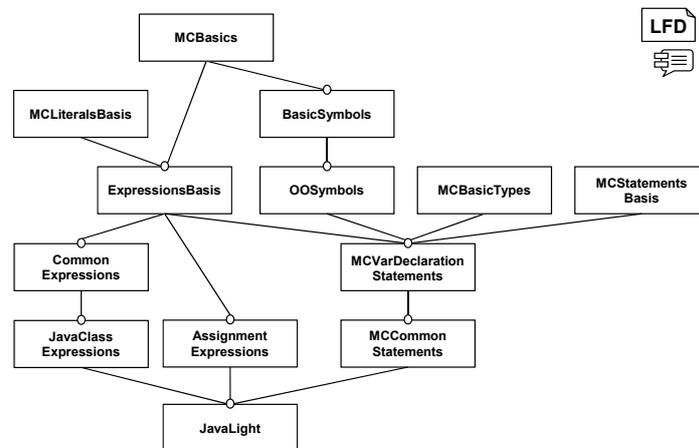


Figure 20.1: A language feature diagram depicting the languages included in the JavaLight language

The grammar `ExpressionsBasis` extends the grammar `MCLiteralsBasis`. The grammar `MCLiteralsBasis` solely introduces the `Literal` interface for introducing various forms of literals.

The grammar `MCVarDeclarationStatements` additionally extends the grammars `MCBasicTypes`, `MCStatementsBasis`, and `OOSymbols`. The grammar `MCBasicTypes` defines qualified names (e.g., `a.b.c`), package declarations (e.g., `package a.b;`), import statements (e.g., `import a.b;`), the extensible `MCType` and `MObjectType` interfaces for types, primitive types (e.g., `int`, `double`), qualified types for objects (e.g., `a.b.C`), and return types (`void` and other `MCTypes`). The grammar `MCStatementsBasis` introduces the reusable common interfaces `MCStatement`, `MCBlockStatement`, and `MModifier` for statements and modifiers. The `OOSymbols` grammar defines the symbol classes `OObjectType`, `Field`, and `Method` for object-oriented types. It extends the grammar `BasicSymbols`. The `BasicSymbols` grammar introduces the symbol classes `Diagram`, `Type`, `TypeVar`, `Variable`, and `Function`. With this, the grammars `OOSymbols` and `BasicSymbols` nearly define all required symbol kinds for the JavaLight language (JavaLight solely introduces one more symbol kind extending the kind `Method` as described below).

The grammars `ExpressionsBasis` and `BasicSymbols` extend the grammar `MCBasics`. The `MCBasics` grammar introduces tokens for white space and comment handling as well as the `Name` nonterminal. The nonterminal `Name` is required for expressions containing names and all symbols.

20.2 Nonterminals of JavaLight

The following first describes the nonterminals for methods, constructors, and attributes. Afterwards, the nonterminals for Java annotations are presented. Finally, this section

introduces the nonterminals introduced for Java array initialization.

20.2.1 Methods, Constructors, and Attributes

```

1 external ExtTypeParameters;
2 interface ClassBodyDeclaration;
3 interface InterfaceBodyDeclaration;

```

MCG JavaLight

Listing 20.2: External and interfaces introduced by the JavaLight grammar for comfortable reuse of type parameters and class and interface elements

The JavaLight grammar introduces the external `ExtTypeParameters` and the interfaces `ClassBodyDeclaration` and `InterfaceBodyDeclaration` as extension points (cf. Listing 20.2). The external nonterminal `ExtTypeParameters` is an extension point for the specific kinds of type parameters (e.g., generic types) to be embedded. The interface `ClassBodyDeclaration` is an extension point for elements that can be defined in the bodies of classes. Analogously, the interface `InterfaceBodyDeclaration` is an extension point for elements that can be defined in the bodies of interface nonterminals. Although the JavaLight grammar neither defines Java classes nor Java interfaces, it introduces the interface nonterminals as extension points for comfortable reuse of the nonterminals implementing these interface nonterminals. Providing interfaces for these elements supports developers in embedding specialized language elements at the specific positions where the interfaces are used. The required language elements that should be embedded may vary between different use cases.

```

1 interface scope (shadowing non_exporting ordered)
2 symbol JavaMethod extends Method = Name;
3
4 symbolrule JavaMethod =
5   exceptions: de.monticore.types.check.SymTypeExpression*
6   annotations: de.monticore.types.check.SymTypeExpression*
7   isAbstract: boolean
8   isSynchronized: boolean
9   isNative: boolean
10  isStrictfp: boolean;

```

MCG JavaLight

Listing 20.3: The JavaMethod symbol

The grammar defines the symbol kind `JavaMethod` (cf. Listing 20.3). The symbol class `JavaMethod` (l. 1f) extends the symbol class `Method`. The `Method` symbol class is defined in the grammar `OOSymbols` and represents methods of objects. Thereby, it inherits all its attributes. The symbol rule in ll. 4-10 adds all Java-specific attributes needed to describe modifiers for methods and also all exceptions the method can raise as well as annotations the method can have. Java methods open a shadowing scope and do not export the symbols defined in their scopes. The symbols are ordered.

```

1 Throws
2   = (MCQualifiedName || ",")+;
3
4 LastFormalParameter
5   = JavaModifier* MCType "... " DeclaratorId;
6
7 FormalParameterListing
8   = (FormalParameter || ",")+ ("," LastFormalParameter)?
9     | LastFormalParameter;
10
11 FormalParameters
12  = "(" FormalParameterListing? ")";

```

Listing 20.4: Nonterminals defined in the JavaLight grammar that are used by the nonterminals for methods and annotations

The definitions of the nonterminals `Throws`, `LastFormalParameter`, `FormalParameterListing`, and `FormalParameters` are depicted in Listing 20.4. These nonterminals are used by the nonterminals for methods and annotations. The `Throws` nonterminal can be reused where declarations for possible throws of exceptions are needed. The parameter-productions can be reused where the specification of parameters is necessary. The nonterminal `FormalParameter` is already introduced in the grammar `MCommonStatements`. The right-hand side of the `Throws` production consists of a comma-separated list of qualified names (ll. 1-2). We recall that a `FormalParameter` consists of an arbitrary number of `JavaModifiers`, followed by an `MCType` and an `DeclaratorId`. Analogously, a `LastFormalParameter` consists of an arbitrary number of `JavaModifiers`, followed by an `MCType`, three dots (`...`), and a `DeclaratorId` (ll. 4-4). A `FormalParameterListing` either consists of a comma-separated list of at least one `FormalParameter`, optionally followed by a `LastFormalParameter` or solely consists of a `LastFormalParameter` (ll. 7-9). The right-hand side of the `FormalParameters` production consists of an opening round bracket, optionally followed by a `FormalParameterListing`, which is required to be followed by a closing round bracket (ll. 11-12). For example, a valid word that can be produced by the `FormalParameters` production using the `FormalParameterListing` and `LastFormalParameter` productions is `(final int x, Object... objects)`.

```

1 MethodDeclaration implements JavaMethod,
2                   ClassBodyDeclaration
3   = MCMODIFIER* ExtTypeParameters?
4     MCReturnType Name FormalParameters (dim:[" "]*)
5     ("throws" Throws)? (MCJavaBlock | ";");

```

Listing 20.5: The `MethodDeclaration` nonterminal

The JavaLight grammar introduces the nonterminal `MethodDeclaration` for implementing methods (cf. Listing 20.5). The nonterminal implements the `JavaMethod` symbol interface (l. 1). Thus, for every `MethodDeclaration`, the corresponding model defines a `JavaMethod` symbol. It implements the interface `ClassBodyDeclaration`

(l. 2) to be reusable at every position where `ClassBodyDeclaration` is used. Every `MethodDeclaration` starts with an arbitrary number of `MCMModifiers`, optionally followed by a type parameter (l. 3). Afterwards, every `MethodDeclaration` consists of a `MCReturnType`, the method's Name, and `FormalParameters` (a list of parameters enclosed by round brackets) (l. 4). The parameters are followed by an arbitrary number of opening and closing brackets, defining the dimension of the return value (l. 4). Then, the `MethodDeclaration` optionally defines exceptions that can be thrown (l. 5). Finally, the `MethodDeclaration` either consists of an `MCJavaBlock` defining the method's body or a semicolon¹ (l. 5). For instance, when embedding appropriate syntax for type parameters into the external `ExtTypeParameters`, the word `public <E> int calc(E arg) [] throws ex.MyException;` can be produced by the `MethodDeclaration` production.

```

1 InterfaceMethodDeclaration implements JavaMethod,
2                               InterfaceBodyDeclaration
3   = MCMModifier* ExtTypeParameters?
4     MCReturnType Name FormalParameters (dim:[" "]*)
5     ("throws" Throws)? ";"
6
7 ConstructorDeclaration implements JavaMethod, ClassBodyDeclaration
8   = MCMModifier* ExtTypeParameters? Name FormalParameters
9     ("throws" Throws)? MCJavaBlock;

```

Listing 20.6: The nonterminals `InterfaceMethodDeclaration` and `ConstructorDeclaration`

The nonterminals `InterfaceMethodDeclaration` and `ConstructorDeclaration` are defined similar to the nonterminal `MethodDeclaration` (cf. Listing 20.6). An `InterfaceMethodDeclaration` (ll. 1-5) must not end with a `MCJavaBlock` (l. 5). Thus, it must end with a semicolon. It implements the interfaces `JavaMethod` and `InterfaceBodyDeclaration`. Thus, using the nonterminal produces a `JavaMethod` symbol and the nonterminal can be employed where `InterfaceBodyDeclarations` are expected. The nonterminal `ConstructorDeclaration` (ll. 7-9) does not define a return value and cannot end with a semicolon (l. 9). Thus, `ConstructorDeclarations` must end with a `MCJavaBlock`. The nonterminal implements the interfaces `JavaMethod` and `ClassBodyDeclaration` (l. 7).

```

1 ConstDeclaration extends LocalVariableDeclarationStatement
2                 implements ClassBodyDeclaration,
3                               InterfaceBodyDeclaration
4   = LocalVariableDeclaration ";"

```

Listing 20.7: The nonterminal `ConstDeclaration`

The `ConstDeclaration` nonterminal (cf. Listing 20.7) extends the

¹Java allows abstract classes to contain methods with empty bodies.

`LocalVariableDeclarationStatement` nonterminal² (l. 1). It implements the `ClassBodyDeclaration` and `InterfaceBodyDeclaration` interfaces (ll. 2-3). Thus, `ConstDeclarations` can be used in all places where the `LocalVariableDeclarationStatements` nonterminal, the `ClassBodyDeclarations` interface, or the `InterfaceBodyDeclaration` interface is expected. The right-hand side of the `ConstDeclaration` production contains a `LocalVariableDeclaration`, followed by a semicolon (l. 4). For instance, `public final int x,y; is a valid ConstDeclaration.`

20.2.2 Java Annotations

```
1 interface AnnotationArguments;
2 interface ElementValue;
```

MCG JavaLight

Listing 20.8: The interfaces `AnnotationArguments` and `ElementValue`

The JavaLight grammar also introduces Java annotations. To this effect, the grammar introduces the interface nonterminals `AnnotationArguments` and `ElementValue` (cf. Listing 20.8). The `AnnotationArguments` nonterminal represents a block of possible arguments for annotations. The `ElementValue` interface represents possible expressions for the right-hand sides of element/value pairs used in annotations. JavaLight does not support to introduce new annotation declarations. Only the call of existing annotations is described here.

```
1 AnnotationPairArguments implements AnnotationArguments
2   = (ElementValuePair || ",")+;
3
4 ElementValueOrExpr implements AnnotationArguments
5   = ElementValue | Expression;
6
7 ElementValuePair
8   = Name "=" ElementValueOrExpr;
9
10 ElementValueArrayInitializer implements ElementValue
11  = "{" (ElementValueOrExpr || ",")* (",")? "}";
```

MCG JavaLight

Listing 20.9: JavaLight nonterminals for annotations

Listing 20.9 presents the four nonterminals used for Java annotations. The `AnnotationPairArguments` nonterminal implements the interface `AnnotationArguments` (l. 1). The right-hand side consists of a comma-separated list of `ElementValuePairs` (l. 2). The `ElementValueOrExpr` nonterminal implements the `AnnotationArguments` interface (l. 4). Each `ElementValueOrExpr` is either an `ElementValue` or an `Expression` (l. 5). Each `ElementValuePair` consists

²The `LocalVariableDeclarationStatement` nonterminal is defined in the grammar `MCGVarDeclarationStatements`.

of a Name, followed by the keyword = and an ElementValueOrExpr (ll. 7-8). The ElementValueArrayInitializer nonterminal (ll. 10-11) can be used to specify an array of ElementValueOrExpr. As it implements the ElementValue interface (l. 10), it can be used in all places where element values for annotations are expected. An ElementValueArrayInitializer starts with an opening curly bracket, followed by a comma-separated list of ElementValueOrExpr, optionally followed by a comma, and ends with a closing curly bracket (l. 11). For instance, when embedding appropriate further syntax into the ElementValue interface, then `x = 5` is a valid ElementValuePair, `x = 5, y = 7` can be produced by the production AnnotationPairArguments, and `{4, 7}` is a valid ElementValueArrayInitializer.

```

1 Annotation implements MCModifier, ElementValue
2   = "@" annotationName:MCQualifiedName
3     ( "(" AnnotationArguments? ")" )?;
```

Listing 20.10: The nonterminal Annotation

Listing 20.10 depicts the definition of the Annotation nonterminal. It implements the interfaces MCModifier and ElementValue (l. 1). Thus, Annotations can be used in all places where MCModifiers and ElementValues are expected. Each Annotation starts with the lexical @, followed by the annotation's qualified name (l. 2), an opening round bracket, optionally followed by AnnotationArguments, and ends with a closing round bracket (l. 3). For example, when embedding appropriate further syntax into the ElementValue interface, the Annotation production can produce `@MyAnnotation(x = 5, y = {4, 7})`.

20.2.3 Java-Specific Array Initialization

```

1
2 ArrayDimensionByInitializer implements ArrayDimensionSpecifier
3   = (dim:"[" "]" )+ ArrayInit;
```

Listing 20.11: The nonterminal ArrayDimensionByInitializer

The ArrayDimensionByInitializer nonterminal (cf. Listing 20.11, ll. 2-3) implements the interface ArrayDimensionSpecifier³. Each ArrayDimensionByInitializer starts with arbitrarily many opening and closing square brackets specifying the array's dimension, followed by ArrayInit for initializing the array (l. 3).

³The interface ArrayDimensionSpecifier is defined in the grammar JavaClassExpressions

Chapter 21

Some Demonstrating Example Languages

co-authored with Robert Heim, Arvid Butting

Quite a number of languages have been developed using MontiCore, some of them are publicly available in the repositories¹. In this chapter we describe a small selection of these languages, because they are used as examples in parts of the handbook.

The main purpose is to demonstrate some features of the MontiCore language workbench. For a detailed and more complete overview of existing languages and tools behind that languages please see the various MontiCore projects.

21.1 A Simple Automata Language

The `SAutomata` language (S is short for Simple) is an example for a language that is delivered with standard MontiCore and can easily be used to build on.

When designing a language, the first step is to discuss its properties on the basis of concrete examples. Listings 21.1 shows a simple automaton having two states and two transitions and 21.2 shows an automaton describing how a ping-pong game is played. Graphical representations are displayed in Figure 21.3.

```
1 automaton Simple { Automaton Simple  
2   state A    <<initial>>;  
3   A - x > B;  
4  
5   state B    <<final>>;  
6   B - y > A;  
7 }
```

Listing 21.1: Simple automaton in text format

¹<https://github.com/MontiCore>

```

1 // The ping pong game
2 automaton PingPong {
3   state NoGame <<initial>> <<final>>;
4   state Ping;
5   state Pong <<final>>;
6
7   NoGame - startGame > Ping;
8
9   Ping - stopGame > NoGame;
10  Pong - stopGame > NoGame;
11
12  Ping - returnBall > Pong;
13  Pong - returnBall > Ping;
14 }

```

Listing 21.2: Example model for the Automaton language

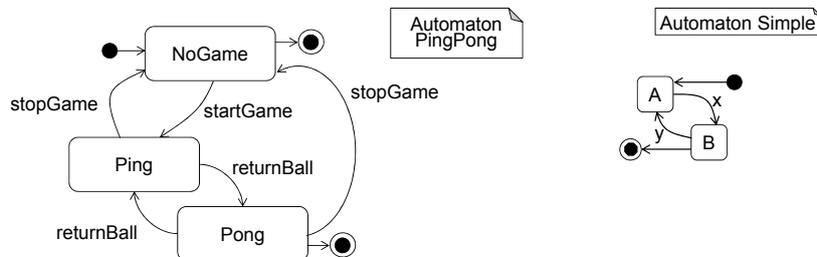


Figure 21.3: Two automata example models

Concrete Syntax

As can be seen in these examples an automaton consists of a basic block (l. 2 and 14 in Listing 21.2), states (l. 3ff. in Listing 21.2) and transitions (l. 7ff. in Listing 21.2). States start with the keyword `state` and have a name. They can be initial, final or none of it. Transitions connect two states where one state is the source and the other the target. Furthermore, transitions are triggered by events such as `stopGame` in the examples or `x` and `y`. After identifying the structure with fixed (e.g. keyword and multiplicities) and variable parts (e.g. names) that all models of the desired language have in common, a grammar can be defined. One such grammar that allows specifying automata as presented before is shown in Listing 21.4.

```

1 import de.monticore.*;
2 grammar SAutomata extends MCBasics {
3
4   symbol scope Automaton =
5     "automaton" Name "{" (State | Transition)* "}" ;
6
7   symbol State =

```

```

8   "state" Name
9   ((" <<" ["initial"] ">>" ) | (" <<" ["final"] ">>" )) * ";" ;
10
11  Transition =
12    from:Name@State "-" input:Name ">" to:Name@State ";" ;
13 }

```

Listing 21.4: MontiCore grammar for the SAutomata language

The grammar does not have an explicit package declaration and thus belongs to the default package. It uses the most basic grammar, namely `MCBasics`, and therefore starts with an `import` statement. `MCBasics` provides white space handling and common lexical nonterminals such as `Name`. The grammar itself defines three nonterminals `Automaton`, `State` and `Transition` corresponding to the different modeling elements of the automaton language. The MontiCore grammar describes the concrete and the abstract syntax, as well as a core symbol table infrastructure. If we are only interested in the concrete syntax, the reduced EBNF is easier to read:

```

1 Automaton = "automaton" Name "{" (State | Transition)* "}" MCG EBNF
2
3 State = "state" Name ( "<<initial>>" | "<<final>>" ) * ";"
4
5 Transition = Name "-" Name ">" Name ";"

```

Listing 21.5: EBNF of the SAutomata language

Abstract Syntax

To understand the AST, it suffices to use the grammar shown in Listing 21.6 that contains all essential information that MontiCore needs to derive a set of Java classes for the AST. The resulting AST classes are shown in Figure 21.7. As described in Chapter 5, MontiCore translates the keyword `final` into the attribute `boolean r__final` to avoid naming conflicts with the `final` keyword of Java.

```

1 grammar SAutomata extends MCBasics { MCG SAutomata (AST only)
2   Automaton = Name (State | Transition)* ;
3   State      = Name (["initial"] | ["final"])* ;
4   Transition = from:Name input:Name to:Name ;
5 }

```

Listing 21.6: MontiCore grammar for the SAutomata language

For a deeper look into the class structure of the AST, the connection with the runtime classes provided by MontiCore are shown in Figure 21.8.

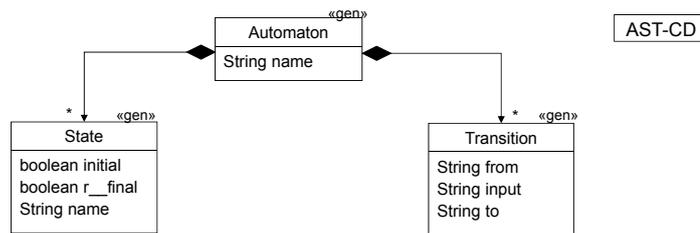


Figure 21.7: AST of the Simple Automaton language

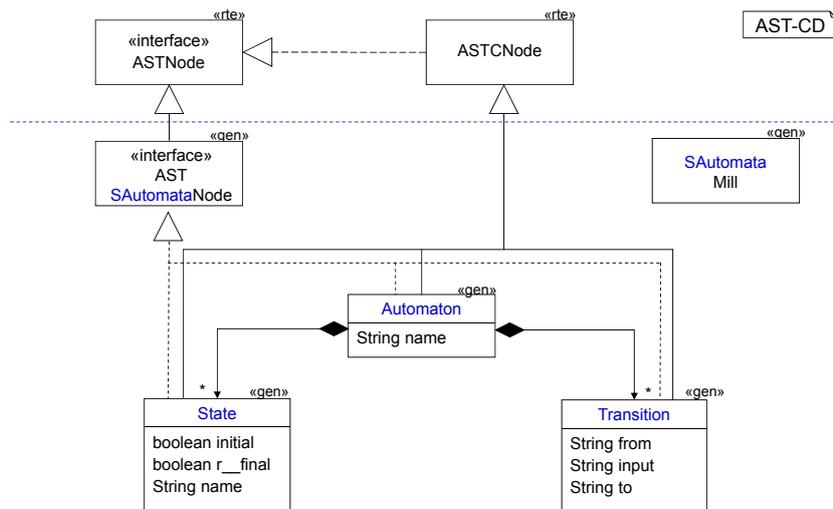


Figure 21.8: AST of the SAutomata language extended by runtime classes

Visitors

For the SAutomata language the usual classes and interfaces for visitors are generated as depicted in Figure 21.9. As described in Chapter 8 the generated interfaces and classes already provide default implementations for traversing the AST and visiting AST nodes. The default traversal strategy is depth first while the default visiting implementation is empty. Thus, when implementing a visitor, e.g., for realizing a pretty printer, it is sufficient to override visit methods relevant for the desired functionality. An example of an implemented visitor interface is the the symbol table genitor, which is explained in Chapter 9. Instances of the traverser SAutomataTraverser are obtainable via the mill, because this allows easy creation as well as some checks on the composite.

Symbols

Two of the nonterminals defined in the SAutomata grammar in Listing 21.4 (p. 390) are marked as symbols: Automaton and State. Transition sources and targets are references to states. Therefore, a symbol table infrastructure is needed and because of this relatively simple case, the generated classes completely cover what is needed. This includes symbols,

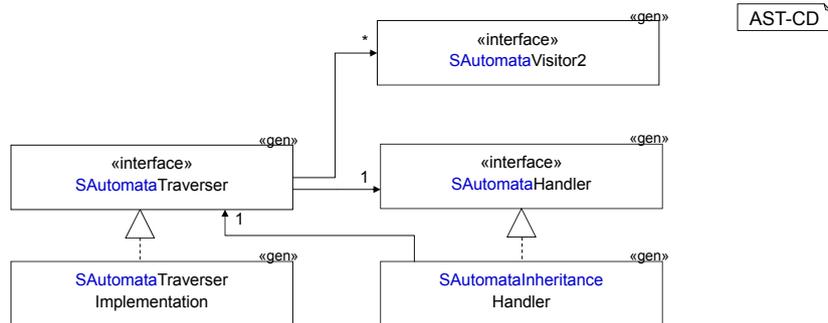


Figure 21.9: Visitors for the Automata language

scope classes as well as symbol table genitors and an infrastructure for storing and loading symbols on the file system.

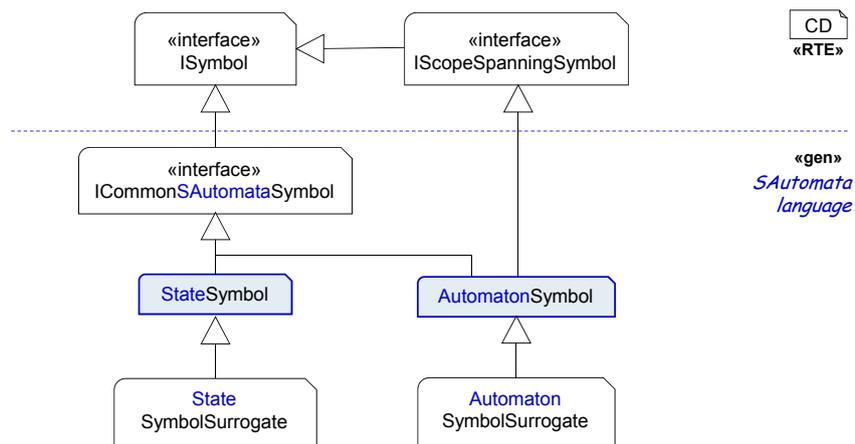


Figure 21.10: Symbols of the SAutomaton language

The generated symbols are shown in Figure 21.10. As described in Chapter 9, for every nonterminal X marked as a symbol, the classes X Symbol and X SymbolSurrogate are generated. The main classes are AutomatonSymbol and StateSymbol. Their surrogates should normally not be necessary. They are only used, when the symbol is not defined in the currently processed model, but in a foreign model and the according symbol table has not been loaded yet. This happens for example when only the check for the existence of the symbol is needed, but no further information about the symbol is required and there is a high potential that loading the entire symbol table of the model is not required in the whole process. This is typically useful when lazy loading the signatures of classes, where other classes are mentioned e.g. as parameter types.

Scopes

For each language, MontiCore always generates three interfaces and three implementation classes that manage scope infrastructures. Figure 21.11 shows the structure of these classes.

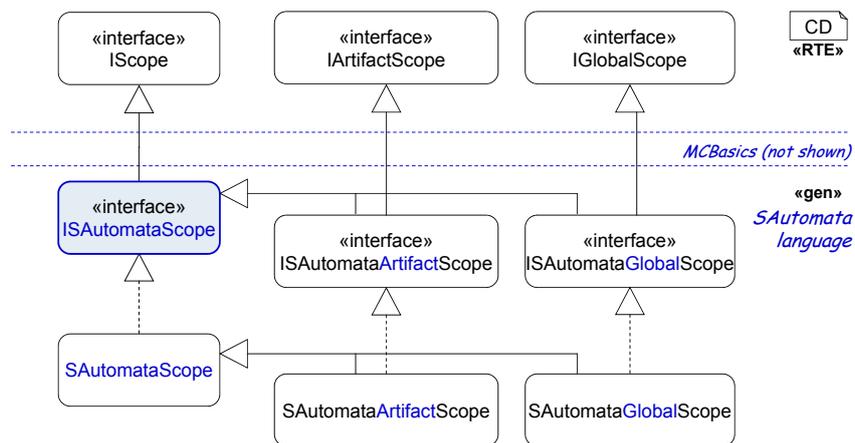


Figure 21.11: Generated scope classes and interfaces of the SAutomaton language

In this simple case, where only the nonterminal Automaton is marked as scope spanning, in a concrete model only the SAutomataScope scope is explicitly needed. For composition purposes, however, it is useful to use the also provided SAutomataScopeBuilder. The interfaces, like ISAutomataScope, are mainly generated, to resolve the inheritance diamonds (see Figure 21.11). The class ISAutomataScope also is the main interface to be programmed against.

The class SAutomataGlobalScope acts like an ordinary scope from the model side, but redefines the behavior of its implemented interface IGlobalScope to provide a global look up and load of needed symbol tables (not needed in the simple automaton case).

Figure 21.12 connects the symbol and the scope classes by showing the relationship between the StateSymbol, the ASTState node and the SAutomataScope. For the AutomatonSymbol, this relationship is slightly more complicated because it also spans a new scope.

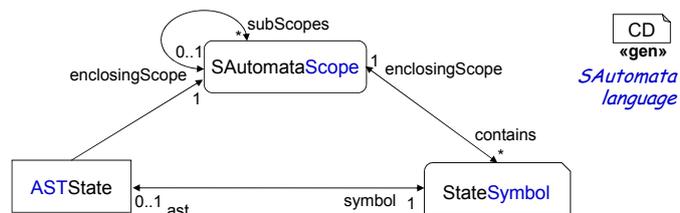


Figure 21.12: Relation between AST node, symbol and scope in the Automaton language

Creation, Storage and Loading of Symbols

For loading and storing symbol tables of the `SAutomata` language, MontiCore generates four classes that realize serialization and deserialization strategies for the language's symbols and scopes. Loading and storing symbol tables of a language is realized by storing each artifact scope as an individual artifact. Thus, the central class of the serialization infrastructure is `SAutomataDeSer`, which offers methods for loading and storing artifact scopes of the language. Further, the scope `DeSer` realizes methods for serializing and deserializing scope objects. As the language defines two symbol kinds, two `DeSer` classes – one for each symbol kind – are generated for serializing and deserializing the symbols.

The class `SAutomataSymbols2Json` is a visitor that traverses the artifact scopes for the purpose of their serialization. The `Symbols2Json` class and the `DeSer` classes collaborate to load and store symbols. During the traversal of a scope, the local symbols of the scope are visited. If a symbol spans a scope, this scope is traversed when traversing the symbol.

The `SAutomataDeSer` translates objects of the scope into JSON and JSON representations of the scope back into scope objects. To traverse the scope, it uses the `SAutomataSymbols2Json` class. For serializing and deserializing local symbols, it delegates to symbol `DeSer` classes. As this language defines two symbol kinds and no symbol kinds of super languages are reused, the `DeSer` uses the `AutomatonSymbolDeSer` and the `StateSymbolDeSer`.

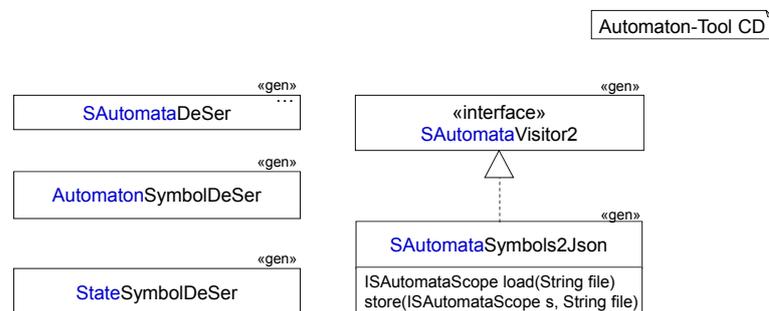


Figure 21.13: Generated classes for loading and storing symbol tables of the `SAutomaton` language

Context Conditions

Another important part of a language definition is well-formedness. Besides the restriction expressed by the grammar there are typically further constraints that need to be satisfied by the models of a language in order to be well-formed. For the `SAutomata` language these are for example:

- There must be at least one initial state.
- State names start with capital letter.
- Transition source and target must exist.

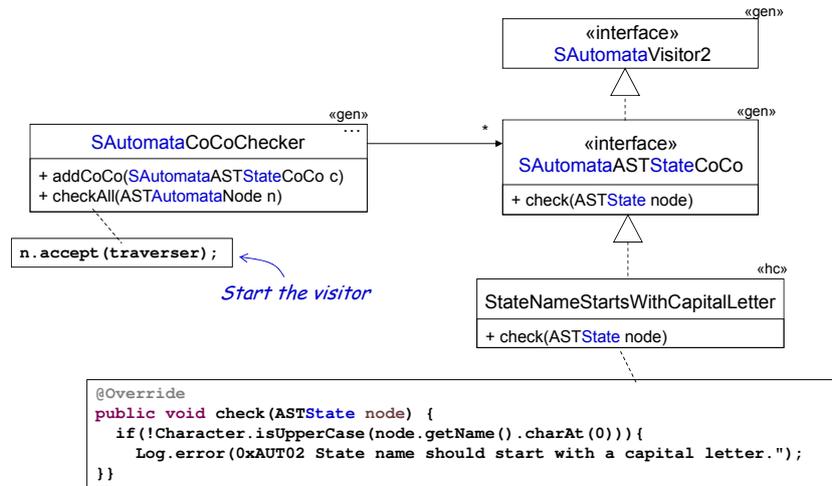


Figure 21.14: Defining context conditions for the SAutomata language

For defining well-formedness rules MontiCore generates infrastructure to implement context conditions. For every nonterminal an interface is generated that should be implemented by a context condition concerning this nonterminal. Furthermore, a context condition checker and a context condition interface for the base node of the language are generated. Figure 21.14 shows an example context condition implementation for *state names start with capital letters*. Implemented context conditions must be added to the context condition checker that afterwards is able to check the well-formedness of models (cf. Figure 21.14). The checker uses a traverser to traverse the AST and checks all context conditions suitable for the visited nodes. For the SAutomata language the files listed in Listing 21.15 are generated for defining context conditions.

```

1 sautomata/_cocos/SAutomataASTAutomatonCoCo.java
2 sautomata/_cocos/SAutomataASTSAutomataNodeCoCo.java
3 sautomata/_cocos/SAutomataASTStateCoCo.java
4 sautomata/_cocos/SAutomataASTTransitionCoCo.java
5 sautomata/_cocos/SAutomataCoCoChecker.java
    
```

Listing 21.15: Files for context conditions for the SAutomaton language

21.2 A Language Extension for Hierarchical Automata

In the previous section the language SAutomata for simple automata was explained and two example automata were considered. The next automaton in Listing 21.16 is an extended version of the automaton in Listing 21.2. Here, the automaton has an additional state InGame that contains the two sub-states Ping and Pong and their transitions. SAutomata does not allow containment and thus an extension is necessary for this kind of automata.

```

1 automaton PingPong {
2   state NoGame <<initial>>;
3   state InGame <<final>> {
4     state Ping ;
5     state Pong ;
6
7     Ping - returnBall > Pong;
8     Pong - returnBall > Ping;
9   }
10
11   NoGame - startGame > Ping;
12
13   InGame - stopGame > NoGame;
14 }

```

Listing 21.16: Example model for the hierarchical Automaton language

As this language is an extension of the SAutomata language, the grammar of HAutomata (H for Hierarchical) extends the grammar of SAutomata (l. 1 of Listing 21.17). Thereby all nonterminals of SAutomata are inherited. HAutomata uses the same starting non-terminal as SAutomata (cf. in l. 7). As states can be hierarchical in HAutomata, the nonterminal State is redefined such that the former state syntax is still valid but in addition states may have a body consisting of states and transitions (cf. l. 9).

```

1 grammar HAutomata extends SAutomata {
2   // keep the old start rule
3   start Automaton;
4
5   // redefine a nonterminal
6   @Override
7   State = "state" Name
8     ( "<<" ["initial"] ">>" | "<<" ["final"] ">>" ) *
9     ( ";" | "{" (State | Transition)* "}" );
10 }

```

Listing 21.17: MontiCore grammar for the HAutomata language

The resulting AST structure is depicted in Figure 21.18. The AST classes of SAutomata are reused for HAutomata. For the redefined State nonterminal a new AST class State is generated that extends the State class of SAutomata.

As for the SAutomata language, the typical four types of visitors are generated. They are all implementing their respective visitors from the supergrammar SAutomaton.

The grammar defines no new symbols or scopes itself, thus no symbol classes are generated. Instead, the symbols of the language SAutomata are reused for the language HAutomata. However, as states are hierarchical in HAutomata, the symbol table creation needs to be adapted to also consider sub-states. Furthermore, by default new scope classes are generated for the language, even though they do not really implement anything in addition to their superclasses from SAutomata.

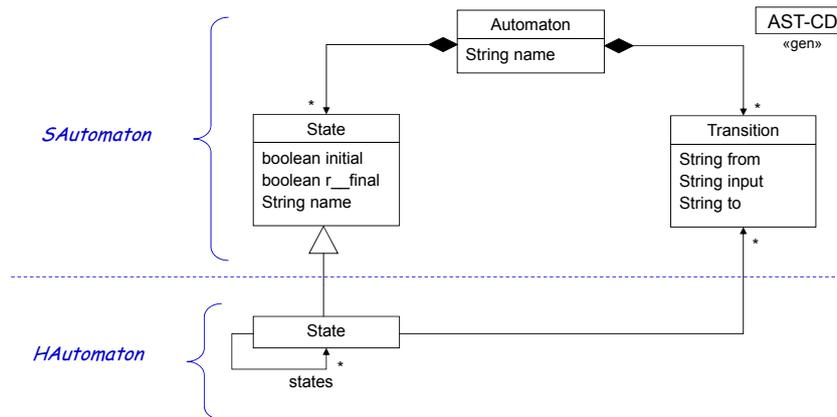


Figure 21.18: AST of the hierarchical Automaton language

A new context condition checker is generated and for the new state nonterminal of HAutomata, a context condition interface allows to define adapted CoCos.

21.3 A Language for Automata with Invariants

Another example of an automaton is shown in Listing 21.19. In comparison to the simple automata of SAutomata states have invariants, which are modelled as *external nonterminal*. A possible and flexible grammar for this kind of automata is shown in Listing 21.20.

```

1 automaton PingPong {
2   state NoGame - <<initial>>;
3
4   NoGame - startGame > Ping;
5
6   Ping - stopGame > NoGame;
7   Pong - stopGame > NoGame;
8
9   state Ping var1;
10  state Pong !var2 <<final>>;
11
12  Ping - returnBall > Pong;
13  Pong - returnBall > Ping;
14 }

```

Listing 21.19: Example model for the Automaton language with invariants

Grammar IAutomataComp (I for Invariant) defines states that have invariants. However, it does not yet define any syntax for invariants but uses an external nonterminal (cf. l. 6). Thus, this grammar is meant for extension and must be marked as a component (cf. l. 1). An extension of this grammar called IAutomata that defines its own syntax for invariants

is shown in Listing 21.21. `IAutomata` overrides the external nonterminal `Invariant` and thereby defines a syntax for it. The resulting AST structure is shown in Figure 21.22.

```

1 component grammar IAutomataComp extends MCBasics { MCG IAutomataComp
2
3   Automaton =
4     "automaton" Name "{" (State | Transition)* "}" ;
5
6   external Invariant;
7
8   State = "state" Name
9     Invariant ( "<<" ["initial"] ">>" | "<<" ["final"] ">>" )* ";" ;
10
11  Transition =
12    from:Name "-" input:Name ">" to:Name ";" ;
13 }

```

Listing 21.20: Grammar component for `IAutomataComp` that defines state with invariants

For `IAutomataComp` MontiCore generates the AST classes for `Automaton`, `State` and `Transition` and an interface `Invariant` for the external nonterminal. We now could embed the expressions provided by MontiCore as invariant language using `Invariant = Expression` and importing some of MontiCores expression grammars. But we deliberately decide against this (normally recommended, simple and powerful option) and define `IAutomata` instead.

For `IAutomata`, MontiCore generates an Interface for the interface nonterminal `LogicExpr`, AST classes for the nonterminals `Truth`, `Not` and `Var`, and an AST class for the overriding nonterminal `Invariant`. The AST class for `Invariant` implements the interface `Invariant` generated for `IAutomataComp`. `["-"]` in l. 6 is used to denote the absence of an invariant with a "-" sign and the brackets force the generator to store a boolean attribute `mINUS` in the class `ASTInvariant` indicating whether "-" was used.

```

1 grammar IAutomata extends IAutomataComp { MCG IAutomata
2   start Automaton;
3
4   // use this production as Invariant in Automata
5   @Override
6   Invariant = LogicExpr | ["-"] ;
7
8   interface LogicExpr;
9   Truth implements LogicExpr = tt:["true"] | "false" ;
10  Not   implements LogicExpr = "!" LogicExpr ;
11  Var   implements LogicExpr = Name ;
12 }

```

Listing 21.21: Grammar for automata with state invariants

As for `SAutomata` the typical four types of visitors are generated for `IAutomataComp` as well as for `IAutomata`. Both grammar define no symbols or scopes, thus no symbol

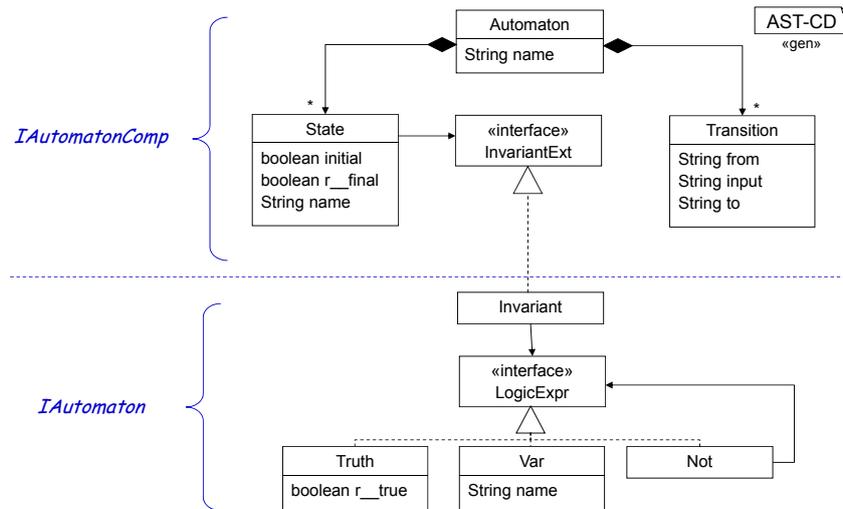


Figure 21.22: AST of the Automaton language with Invariants

or scope classes are generated. For all nonterminals regardless of whether their marked as interfaces or external or not marked at all context condition interfaces are generated.

21.4 Scannerless Parsing to Handle Complex Tokens

Section 4.1 discusses how the *lexer* groups individual characters into *tokens*, which are then used by the *context-free parser* to construct the abstract syntax. Because tokens are defined using regular expressions and are scanned without any context information, overlapping tokens may lead to problems. It depends on the order of definition, which tokens are recognized and given to the parser. Furthermore, if one token is a prefix of another, then only the longer token is recognized. For example the infix operator ">>" has a prefix ">", the negative decimals like "-42" are prefixed by "-".

To prevent these problems, we discuss two explicit mechanisms among several possible solutions in the following two sections. MontiCore provides the `noSpace(.)` semantic predicate and the `splittoken` directive that solve already most of the discussed cases.

21.4.1 Parsing with Whitespaces

When the simple comparison ">" needs to be recognizable as token, the infix operator ">>" cannot be directly defined, but needs to be defined as nonterminal with two token. Unfortunately, on the nonterminal level, whitespaces are not visible anymore and a non-terminal with right-hand side ">" ">" can not distinguish the correct input ">>" from an erroneous "> >".

One way to handle this is to switch off the parsing and ignoring of whitespaces as defined in the token `WS` by not including the `MCBasics` grammar. Token definition `WhiteSpace` achieves this:

```

1  S = WhiteSpace* ;
2  S1 = WhiteSpace+ ;
3
4  token WhiteSpace = ( ' ' | '\t' | '\r' | '\n' ) ;

```

Listing 21.23: Whitespaces made explicit in the productions

Nonterminal *S* groups an arbitrary sequence of whitespaces including the empty sequence, while *S1* demands at least one whitespace.

The advantage is that whitespaces can now be explicitly handled in the productions. The disadvantage is that they need to be handled everywhere explicitly as the following excerpt of a grammar shows:

```

1  interface Expression;
2
3  ShiftExpression implements Expression <160> =
4      leftExpression:Expression
5      (   shiftOp2:"<" "<"
6        |   shiftOp4:">" ">"
7        )
8      rightExpression:Expression;
9
10 BracketExpression implements Expression <310>
11   = S "(" Expression ")" S;
12
13 NameExpression implements Expression <350>
14   = S Name S;
15
16 Type = S Name S TypeArguments? S;
17
18 TypeArguments = S "<" (Type || ",")* ">" S ;

```

Listing 21.24: Whitespaces explicitly used in productions

The productions are decorated with the explicit whitespace nonterminal *S* in many places, to explicitly allow spaces at the beginning, between and after a language element. The paradigm is that each nonterminal in itself embeds whitespaces (*S*) before it really starts and ends. Thus, between two terminals an explicit *S* must be included. Furthermore, at the beginning and end of each production, *S* is added if the production starts respectively ends with a terminal.

For example, it is not necessary to repeat the initial *S* in left recursive definitions (it is not even allowed to repeat it). Explicit omission of *S* between two character terminals enforces both to stand in the input directly next to each other and thus allows us to achieve that "<" "<" in the production only parses "<<" in the input. Please note that the production still needs to contain two individual characters and not their combination, because that implicitly defines a new token.

This form of grammar definition almost does not use the lexer and thus is called almost *scannerless*. It puts the burden of token aggregation to the parser and thus slows down the parsing process by a not neglectable factor. However, it exploits the parsing capabilities to full context-free parsing. Not only whitespaces are managed by the parser, but many other tokens are separated into their character sequences and managed by the parser directly.

21.4.2 Temporarily Parsing with Whitespaces

As an alternative, it would also be possible, to switch off the omission of whitespaces temporarily. This means that we would generally add a status concept, stored in form of attributes in the lexer and change the behavior of certain tokens according to that status.

The following grammar excerpt demonstrates such a definition, again using an alternate definition of whitespaces. It relies on the possibility to add extra functionality to a token (discussed in Section 4.1.2) and to the lexer in general (introduced in Section 4.2.11). In addition, this example demonstrates, how to use a state to dynamically adapt behavior of the lexer and similarly of the parser if necessary.

```

1  token WhiteSpace = ( ' ' | '\t' | '\r' | '\n' )
2      :{ if(isSpaceOn()) {_channel = HIDDEN; } };
3
4  concept antlr {
5      lexerjava {
6          protected boolean spaceOnFlag = true;
7
8          public boolean isSpaceOn() {
9              return spaceOnFlag;
10         }
11     }
12 }

```

Listing 21.25: Whitespaces temporarily explicit in the production using a state switch in the lexer

In case the boolean variable `spaceOnFlag` is `true` the token `WhiteSpace` has the usual behavior. That means whitespaces are filtered from the input stream and not delivered to the parser. Setting `spaceOnFlag` temporarily to `false` changes this behavior. Whitespaces are now submitted to the parser and must explicitly be parsed if they occur in the input stream or cannot occur in the input stream.

Switching the `spaceOnFlag` can only be controlled from the scanner. Because of the variable lookahead of the scanner, multiple preprocessed tokens may already include (or filter) whitespaces, before the parser can switch the behavior. To be able to switch from the lexer to the parser, special tokens need to be defined for performing the switching. Unfortunately, these tokens need to be present in the input stream, which makes the process only usable for certain special cases. The following excerpt shows an impractical example for the `">>"` operator:

```

1  ShiftExpression implements Expression <160> = MCG SpaceOnOff
2      leftExpression:Expression
3      WSOff ( shiftOp2:"<" "<"
4          |  shiftOp4:">" ">"
5          ) WSON
6      rightExpression:Expression ;
7
8  token WSOff = "!"
9      :{ spaceOnFlag = false; };
10
11 token WSON = "!"
12     :{ spaceOnFlag = true; };

```

Listing 21.26: Switching whitespaces on and off temporarily in productions

Token WSON and WSOff adapt the behavior of token WhiteSpace. Both nonterminals are selected such that they do not interfere with other nonterminals. Unfortunately they cannot be empty and they need to be token (and not other nonterminals). The result is unsatisfactory for our use case.

21.4.3 Preventing Whitespaces between Tokens

If the tokens need to be defined individually, but should follow consecutively without spaces in between, we can also use the line and column position of tokens after being parsed. The following grammar shows how the extra function is used in the productions in form of a semantic predicate.

```

1  ShiftExpression implements Expression <160> = MCG SpaceFreeChecks
2      leftExpression:Expression
3      ( {noSpace(2)}? shiftOp2:"<" "<"
4        | {noSpace(2,3)}? shiftOp3:">" ">" ">"
5        )
6      rightExpression:Expression;
7
8  C = Name "." NoWSLast2 Name NoWSLast2 ;
9  D = NoWSNext3 Name "." Name ;
10
11 NoWSLast2 = {noSpace()}? ;
12 NoWSNext3 = {noSpace(2,3)}? ;

```

Listing 21.27: Disallowing spaces between last two token

The semantic predicate `{noSpace()}?` can be used directly or can be encapsulated in a nonterminal like e.g. `NoWSLast2`. It must be applied after the two tokens that shall be directly connected and can be used repeatedly, e.g. to define `">>>"` but also with tokens that consist of several characters as shown with nonterminals `C` and `D`.

21.5 Tip: Testing Grammars and their Models

Defining a new grammar is a task as complex and error prone as writing arbitrary software. Much can be said about how to assure the quality of a grammar. To ensure the quality of a grammar, a comprehensive set of parsable model, as well as negative examples are useful. It is necessary to

- review the grammar thoroughly,
- use a comprehensive set of input models to be parsed, and
- Identify a set of negative (not parseable) models to prevent false positives.



Tip 21.28: Testing a Grammar

A kind of "*unit tests*" can relatively easy be achieved by embedding the grammar to test into a testing grammar that includes the nonterminals to be tested e.g. into repeating lists, etc.

This simplifies the tests that need to be set up especially for grammars that deal with small literals.

The test framework JUnit [Bec15] allows to define small tests to check whether a model or a part of a model has been transformed into the correct AST.

Unit tests are among the most helpful and efficient techniques to check a grammar respectively its outcome for the desired behavior. Therefore, we use a JUnit infrastructure to check the desired behavior of a grammar's result.

This piece of Java code can be used in a similar manner for many unit tests, like in example in Listing 21.29.

```

1 package test;
2 import de.monticore.scannerless._ast.*;
3
4 public class CheckScannerlessTest {
5
6     // setup the language infrastructure
7     ScannerlessParser parser = new ScannerlessParser() ;
8
9     @BeforeClass
10    public static void init() {
11        // replace log by a side effect free variant
12        LogStub.init();
13        // LogStub.initPlusLog(); // for manual testing purposes
14        Log.enableFailQuick(false);
15    }
16
17    @Before
18    public void setUp() {
19        Log.getFindings().clear();

```

```

20 | }
21 | }

```

Listing 21.29: Setting up a JUnit test infrastructure

A typical testing class, such as `CheckScannerlessTest` defines a reusable parser for the grammar under test (here `Scannerless`). It also replaces the normal logging by a stub that collects and ignores all errors, which is necessary to handle negative models as well. The log is cleared for each test.

Figure 21.30 contains a positive and a negative test. Both use the parsing from a `String`. In the first case a `Type` is parsed and in the second case an `Expression` (here types are also expressions like in OCL, but the comparison `">>"` contains an erroneous space).

```

1 | @Test
2 | public void testType2() throws IOException {
3 |     ASTType ast = parser.parse_StringType( " List < Theo > " ).get();
4 |     assertEquals("List", ast.getName());
5 |     ASTTypeArguments ta = ast.getTypeArguments();
6 |     assertEquals("Theo", ta.getType(0).getName());
7 | }
8 |
9 | @Test
10 | public void testType8() throws IOException {
11 |     // This cannot be parsed because of the illegal space in ">>"
12 |     Optional<ASTExpression> ast0 = parser.parse_StringExpression(
13 |         "List<Set<Theo>>> >wert" );
14 |     assertFalse(ast0.isPresent());
15 | }

```

Listing 21.30: Some JUnit tests

Other possibilities would be to parse a string, pretty print it and compare the original and the pretty printed version, while completely ignoring all whitespaces.

For smaller examples parsing and pretty printing etc. can be done within the process and no files are needed, which speeds up the execution.

21.6 ColoredGraph Language

The `ColoredGraph` language enables modeling graphs that comprise vertices with a color attribute and directed edges. The colors in this language can be either symbolic color names or tuples of RGB values. The language showcases how the generated serialization and deserialization can be adjusted with handwritten extensions.

```

1 | graph Blinker {
2 |     initial vertex Off (black);
3 |     vertex BlinkBlue (0, 0, 204);

```

21. Some Demonstrating Example Languages

```
4 |
5 |   Off -> BlinkBlue;
6 |   BlinkBlue -> Off;
7 | }
```

Listing 21.31: Example model of the ColoredGraph language

Listing 21.31 depicts an example model of the language. The model realizes a graph with the name `Blinker` (l. 1). The graph has a state with the name `Off` that is assigned with a color that has the predefined symbolic name `black` (l. 2). It further has a vertex with the name `BlinkBlue` that has a color defined via a tuple of RGB values (l. 3). Both vertices are connected via two edges (ll. 5-6).

```
1 | grammar ColoredGraph MCG ColoredGraph
2 |     extends de.monticore.literals.MCCommonLiterals {
3 |
4 |     symbol scope Graph = "graph" Name "{" ( Vertex | Edge )* "}" ;
5 |
6 |     symbol Vertex = ["initial"]? "vertex" Name "(" Color ")" ";" ;
7 |
8 |     interface Color;
9 |
10 |     RGBColor implements Color = NatLiteral "," NatLiteral
11 |         "," NatLiteral ;
12 |
13 |     NameColor implements Color = Name;
14 |
15 |     Edge = s:Name@Vertex "->" t:Name@Vertex ";" ;
16 |
17 |     symbolrule Vertex = color:java.awt.Color initial:boolean;
18 |
19 |     scoperule = numberOfColors:int;
20 |
21 | }
```

Listing 21.32: Grammar of the ColoredGraph language

The grammar of the colored graph language is depicted in Listing 21.32. Each graph begins with the keyword `graph` and has a name as well as a body embraced by curly brackets, which contains vertices and edges (l. 4). A vertex (l. 6) can be marked `initial` and has a color, which is realized as interface nonterminal (l. 8). For the two options of defining colors in this language, the grammar uses two individual nonterminals implementing the color interface nonterminal. Edges (l. 15) connect a source vertex with a target vertex via their names.

Graph nonterminals span a scope and define a symbol (l. 4), vertices define vertex symbols (l. 6), and edges use the names of vertex symbols (l. 15). These properties are defined in the grammar with the respective annotations as explained in Chapter 9. The grammar further defines a `symbolrule` (l. 17) and a `scoperule` (l. 19) to indicate attributes of the `VertexSymbol` and of the language's scope. The `symbolrule` for vertex symbols defines

two attributes, one for the `color` of the vertex and another one for indicating whether a vertex is `initial`. The scoperule defines an integer attribute `numberOfColors` that indicates the number of distinct colors that vertices in the scope define. We added the scoperule for demonstration purposes only. In practice, this information could also be derived from the symbols.

For scoperule and symbolrule attributes, MontiCore generates Java attributes as well as access and manipulation methods for these in the symbol and scope classes. Beyond this, MontiCore generates the serialization and deserialization infrastructure for these symbol and scope attributes. The serialization and deserialization for attributes of built-in data types, such as `integer`, `Boolean`, `double`, and `String` as well as for iterations thereof or optionals is generated completely. In the example of the colored graph language, this applies to the attributes `initial` and `numberOfColors`. However, the serialization and deserialization for other data types is not fully generated. To this end, language engineers have to plug in their own functionality in the `DeSer` class as well as in the `Symbols2Json` class with the TOP mechanism (see Section 14.3) and override methods accordingly. Only the method skeletons are generated for these attributes.

We demonstrate how to serialize custom data on the example of colors, which can be serialized in various different forms. For instance, it is possible to serialize colors as:

- String with symbolic name (e.g., "blue")
- String with hexadecimal RGB values (e.g., "#0000cc")
- JSON array of integers for RGB values (e.g., [0, 0, 204])
- JSON object (e.g., {"r":0, "g":0, "b":204})

In this example, language engineers decided to serialize colors as Json arrays with RGB values. Listing 21.33 depicts the handwritten serialization of the color attribute that is located in the TOP mechanism extension of the `VertexSymbolDeSer`. The `serializeColor` method has parameter `s2j`, which is of the type `ColoredGraphSymbols2Json` and provides a printer that is used to print JSON syntax. For a passed color value, the method `serializeVertexColor` uses this `s2j` to obtain the printer to print colors as JSON arrays. The first value of the array is always the red value, followed by the green and the blue values.

```

1 /**
2  * Serializes Color as RGB values in form [0,0,0]
3  */
4 public class VertexSymbolDeSer extends VertexSymbolDeSerTOP {
5
6     @Override
7     public void serializeColor(Color color,
8                               ColoredGraphSymbols2Json s2j) {
9         JsonPrinter p = s2j.getJsonPrinter();
10        p.beginArray("color"); // Serialize color as arrays,
11        p.value(color.getRed()); // add red value first
12        p.value(color.getGreen()); // ... followed by green
13        p.value(color.getBlue()); // ... and blue.

```

21. Some Demonstrating Example Languages

```
14     p.endArray();           // Print the array end.
15 }
16 }
```

Listing 21.33: Handwritten serialization of the color attribute of VertexSymbols in the class VertexSymbolDeSer

The deserialization, on the other hand, uses a `JsonObject` of the vertex symbol passed as a method argument for deserializing the serialized JSON array to an instance of the class `java.awt.Color`. It is realized in the method `deserializeColor` located in the handwritten extension of the generated `VertexSymbolDeSer`.

```
1  @Override
2  public Color deserializeColor(JsonObject symbolJson) {
3      // get color attribute from the symbol represented in json
4      List<JsonElement> rgb = symbolJson.getArrayMember("color");
5
6      // cache each color value as integer number in a variable
7      int r = rgb.get(0).getAsJsonNumber().getNumberAsInteger();
8      int g = rgb.get(1).getAsJsonNumber().getNumberAsInteger();
9      int b = rgb.get(2).getAsJsonNumber().getNumberAsInteger();
10
11     // create new color using deserialized color values
12     return new Color(r, g, b);
13 }
```

Listing 21.34: Handwritten deserialization of the color attribute of VertexSymbols in the class VertexSymbolDeSer

Other forms of serialization and deserialization for colors can be realized similarly, by overriding the respective methods of the `DeSers`.

21.7 Questionnaire Language

The language `Questionnaire` provides modeling elements for questionnaires. The key idea is to easily define and develop questionnaires and allow a backend to setup a website with a database to let people fill the questionnaire. The user has specific demands on the language, which are partially reflected below.

As usual, we start with an example for a model, which in this case is a questionnaire definition for personal skills. A questionnaire defines items (i.e. questions) that ask for information as strings, alternatives, numbers or values from a specific scale. The listing below consists of four items asking for the name, age and programming skills in Java and C++ (ll. 2-5).

Often items share a scale (e.g., rating an item on a range from "Beginner" 1 to "Expert" 5). Consequently, scales can be defined once (as scale types) and items reuse them (see l. 6). The user has provided the following example:

```

1 questionnaire Personal {
2   item name "What is your name?" text 140
3   item age  "What is your age?"  number
4   item java "Rate your Java skill" skill
5   item cpp  "Rate your C++ skill"  skill
6   scale skill range ["Beginner" 1 .. 5 "Expert"]
7 }

```

Questionnaire model

The example allows us to identify a number of keywords, such as `questionnaire` or `item`. While the user did ask for a specific form of indentation, we as usual decided to provide a robust implementation, which means that white spaces are ignored. The language is only loosely inspired by programming languages, like Java, because it uses curly brackets to enclose the questionnaires body. We also identify a number of statements, namely the `item` and the `scale` definitions, which start with an appropriate keyword, but not have a regular terminator, such as `;` or `,`. We might either use the newline (which would make it sensitive to white spaces) or the implicit termination, when the next keyboard occurs. After clarification with the user, the second approach was chosen. The user denied our suggestion to introduce a regular terminator.

As the users have to experience the language, we usually try to accommodate their wishes. This differs, when the representation of the language is defined in a grammar, which is for tool developers only. In the following, the grammar details are described.

```

1 grammar Questionnaire
2   extends de.monticore.literals.MCCommonLiterals {
3
4   symbol scope QDefinition = "questionnaire" Name "{"
5     ( Item | Scale ) *
6   "}" ;
7
8   symbol Item = "item" Name question:String (scale:Name | ScaleType);
9
10  symbol Scale = "scale" Name ScaleType;
11
12  interface ScaleType;
13
14  Range implements ScaleType = "range" "["
15    minTitle:String? min:NatLiteral
16    ".."
17    max:NatLiteral maxTitle:String?
18  "]" ;
19
20  Number implements ScaleType = "number";
21  Text implements ScaleType = "text" maxCharacters:NatLiteral?;
22
23  Select implements ScaleType = "select" "{"
24    options:SelectOption+
25  "}" ;
26

```

MCG Questionnaire

21. Some Demonstrating Example Languages

```
27 |   SelectOption = id:NatLiteral ":" title:String;  
28 | }
```

The grammar omits an explicit package declaration and starts with the grammar defined as `Questionnaire` (l. 1). It makes use of MontiCore's common grammar `MCCCommonLiterals` and therefore extends it.

As defined by production rule `QDefinition`, each questionnaire model starts with the keyword `questionnaire` (l. 4) followed by its name and an arbitrary count of `Items` and `Scales` enclosed by curly brackets (ll. 4-6).

An `Item` (l. 8) is introduced by the keyword `item`. It has a name and specifies the question to ask. It either refers to a scale by its name or defines a new type of scale for that specific item.

A `Scale` (l. 10) starts with the keyword `scale` followed by a name and a definition of a `ScaleType`. There are several kinds of scale types and they all implement the interface production `ScaleType` (l. 12). Since `ScaleType` is an interface production it is easy to extend the language by other kinds of scales.

The language supports the following scale types. A `Range` (ll. 14-18) starts with the keyword `range` and an interval definition in square brackets. Each end of the interval has an optional title. The scale type `Number` (l. 20) simply states the keyword `number` meaning that any number is expected for the item (e.g., age of a person). Similar, the `Text` scale type (l. 21) starting with keyword `text` expects any free text (which can be restricted to a maximum character count). Choosing an element of predefined options is modeled by scale type `Select` (ll. 23-25). The keyword `select` introduces the modeling element and encloses the available options in curly brackets. Each `SelectOption` (l. 27) consists of an `id` and a `title` separated by a colon.

The presentation of such a grammar is relatively straightforward, we begin with the starting nonterminal that describes the overall language and introduces the next nonterminals. To keep readers in the flow, it is generally a best practice, to define the next nonterminals in order of their occurrence in the previous definition.

There is a second best practice, to start with nonterminals for larger parts of the language and define the small nonterminals, which are usually not further decomposed, at the end. And a third best practice is to try to define nonterminals that implement an interface directly below the interface. Finally for larger grammars, we try to group nonterminals that belong together. Unfortunately, these best practices are regularly in conflict.

We decided to introduce `ScaleType` as an interface, because this reflects that several potential implementations are available and furthermore, we assume that more variants will be coming. It would also have been a natural possibility to introduce an interface, e.g. `QStatement`, as a common super-nonterminal for `Item` and `Scale`. This is especially interesting, if it matters whether a scale has been defined before it is used in an item, because the current AST item and scale definitions are kept in separate lists.

We can also see from the productions for `QDefinition` and `Range`, that the representation (layout, indents, brackets) of the production mimics the expected representation of the text to be parsed.

The language obviously needs context conditions, a completed symbol infrastructure and a generative backend. From those aspects the visitors are discussed in Chapter 8 in detail.

Chapter 22

Developer's View on MontiCore

This chapter overviews the locations of MontiCore's source files. MontiCore is an open source framework primarily developed by the Software Engineering Group of RWTH Aachen University. MontiCore imports other sources, partly as open source. Some re-usable languages that are anticipated to be often reused in other languages (e.g., `Expressions`, `Literals`, `Types`, etc.) are part of the source files of the MontiCore project. Other languages are developed in external repositories.

MontiCore itself and languages based on MontiCore are generally developed with Java. Gradle¹ is the standard build automaton and dependency management tool for MontiCore itself and languages developed with MontiCore.

MontiCore's source code is structured in Gradle's default project layout. Each Gradle project should consist of a build script, a settings file, and a properties file. The build script defines and configures the tasks that are executed for the project (e.g., generation of source code, compilation, packaging, publishing archives to a repository, etc.). For this task, it can include external plugins that may manipulate the build (e.g., by adding new tasks). The settings file mainly defines the root project, all included subprojects, and the configuration for retrieving external plugins. The properties file configures startup options (e.g., JVM memory size) and properties (e.g., String constants) that can be used in the build script.

The build script file `build.gradle`, the settings file `settings.gradle`, and the properties file `gradle.properties` are located in the root folder of the Gradle project. The contents of the build script `build.gradle` and the settings file `settings.gradle` are based on an internal Groovy DSL. The properties file contains simple `key = value` pairs. It is also possible to specify Gradle build scripts and settings using an internal Kotlin DSL. The MontiCore project itself uses the Groovy DSL.

The following overviews the most important folders of MontiCore Gradle projects. The top level folder `src` separates handwritten source code from generated code. The generated code is located in the build folder, which is typically `target` or the Gradle default `build`. The `src` folder contains a `main` folder for the productive source code and a `test` folder containing source code for tests. Both folders then are structured by separating the actual java source code (`src/main/java`, `src/test/java`) from additional resources (`src/main/resources`, `src/test/resources`). Also,

¹<https://gradle.org/>

grammars (`src/main/grammars`) and models (`src/main/models`) are separated. By default, only the compiled source code (`target` folder) and resources of the `src/main/resources` folder are included when Gradle packages the final jar. Also, the `target` folder must never be part of a version control system since it is generated.



Tip 22.1: Different Kinds of Projects

The term *project* is used in multiple ways. Some common usages are specified in the following.

A *software engineering project* commonly consists of a project definition (often in form of a research or industry application), has a budget, a lifetime, potential project partners, responsible project managers, a project team, project management tools, a development infrastructure, etc. This is the business oriented interpretation of the term project.

To conduct a SE project, project management tools are used. Here, the term *management project* is a set of services provided by any of the project management tools (e.g., SSELab, GitLab, GitHub). These tools support the project development, documentation and communication between team members. Their services may include source code repositories and their version control (SVN, git), a ticket system, wikis, mailing lists, etc. A SE project may use several management projects (e.g., some may be accessible for customers or students, others may be internal or open source such as MontiCore).

A *Gradle project* is the technical bundling of specific source code that can be deployed as an artifact (this is often a library jar or an executable jar). A Gradle project is reusable in other Gradle projects by defining a dependency on the deployed artifact. Most management projects consist of multiple Gradle projects to structure modules of complex source code. Gradle projects can be nested. E.g., MontiCore as well as its runtime are available as single jars that bundle the different specific projects and subprojects.

MontiCore uses the two project management tools GitHub and an RWTH Aachen internal GitLab instance. The RWTH GitLab projects are not public. Access can be granted upon request. All source code specific to MontiCore that is not yet open source is located in its git repository. Besides that a main purpose of the GitLab project is its ticket system for planning development cycles. Additionally, some languages that are still in early development stages are located in the RWTH GitLab instance.

22.1 MontiCore's GitHub Repository

MontiCore's source code itself is open source and located in the MontiCore GitHub repository². There are several main branches:

²<https://github.com/MontiCore/monticore>



Tip 22.2: Repositories

MontiCore and its components can be found under:

```

1 // MontiCore core parts
2 Repository: https://github.com/MontiCore/monticore
3
4 // Basics, such as logging
5 Repository: https://github.com/MontiCore/se-commons
6
7 // MontiCore open source languages
8 Repository: https://github.com/MontiCore/
9
10 // Release notes about important changes, etc.
11 Release Notes: http://monticore.github.io/monticore/00.org/
12                 Explanations/CHANGELOG/
13
14 // Overview of language-projects in development
15 Repository: https://github.com/MontiCore/monticore/blob/dev/
16                 docs/Languages.md
17
18 // Packaged components, sources, snapshots etc.
19 Nexus: https://nexus.se.rwth-aachen.de/
20
21 // Management of open issues, enhancements, etc.
22 Tickets: https://git.rwth-aachen.de/monticore/monticore/-/
23         issues

```

The MontiCore *GitHub* organization is public. It contains the MontiCore project and projects of languages developed with MontiCore (e.g. an automaton language, json, sequence diagrams), and statecharts. The GitHub page containing the MontiCore release notes is public. The *Nexus* website can be used to download source files for integrating them in other projects and can be used as a repository location for using the projects as dependencies in Gradle projects. The MontiCore GitLab Project is private. It contains an overview of languages under development that will be release as open-source projects and an internal ticket system.

master contains the source code of the latest stable MontiCore release. On each release of MontiCore, the dev branch is integrated into the master branch (usually resulting in the master branch being replaced by the dev branch). As usual, the master branch mainly receives the consolidated changes from the dev branch.

dev is the main development branch. Changes such as enhancements as well as bug fixes are integrated through the dev branch.

other branches are created from the dev branch. Changes in these branches are either for bug fixing or for implementing new features. Once the bug is fixed or the feature is implemented, the branch is merged back into the dev branch.

22.2 For External Developers: How to Contribute

The projects located in the MontiCore organization³ of GitHub are public. The projects located in the organization are internally developed by MontiCore's core developers in the RWTH GitLab instance. The protected branches are mirrored into the corresponding GitHub project. The automated mirroring process ensures that the source code of mirrored branches on GitHub and GitLab is always the same.

External developers who plan to contribute to a GitHub project located in the MontiCore organization should work on the dev branch of the project. The best practice for contributing to the project is to fork the repository and continue working in the fork. The changes should usually be applied to the dev branch. When finished, the external developer must merge all new changes from dev branch of the project into the fork. Afterwards, the developer creates a pull request from the fork in the project. The core developers will review the pull request, respond with questions if necessary, and accept the pull request when integrable.

While many of the language projects in the MontiCore GitHub organization are actively developed, a number of language projects are located in the organization for documentation and reuse purposes only. This is the case, for example, if a corresponding research paper referenced specific parts of the source code. These projects sometimes use outdated MontiCore versions and can be identified by their version histories.

22.3 MontiCore's Gradle Projects

The MontiCore Repository consists of several Gradle projects, from which the following provide MontiCore's core features:

monticore-generator contains the grammar language for MontiCore-based grammars (cf. Chapter 4), a transformation to transform a given grammar to its internal CD representation and the generator that derives a DSL infrastructure from the CD. The Groovy-based MontiCore-script and all MontiCore configuration options are defined in this project as well.

monticore-runtime provides the source code required to execute MontiCore or a MontiCore-based DSL. This includes the Generator Engine (cf. Chapter 13), common infrastructure of Symbol Tables and ASTs (e.g., the `ASTNode` interface as described in Section 5.7), and helpers, e.g., for file operations.

monticore-grammar contains the library of the common grammars (cf. Chapter 17) providing lexical tokens, literal constants, types, expressions, and statements as well as some tools to work with these grammars (e.g., pretty printers).

The other Gradle projects complement the core functionalities of MontiCore:

³<https://github.com/MontiCore>

monticore-cli provides MontiCore's Command Line Interface by basically wrapping the MontiCore generator script with a CLI.

monticore-emf-runtime contains the runtime environment for EMF compatible MontiCore applications.

monticore-maven contains the Maven plugin that enables executing MontiCore as part of the Maven build lifecycle.

monticore-gradle contains the Gradle plugin that enables executing MontiCore as part of Gradle builds.

monticore-templateclassgenerator contains a MontiCore module to generate Java classes from templates to enable a Java-based type-safe workflow when calling templates (disabled by default).

22.4 Further Source Code Locations

The MontiCore git project includes source code specific to MontiCore and a core library of languages, e.g. for expressions and statements. The following list overviews further git projects belonging to the extensible MontiCore language zoo. These are easily reusable for the development of further domain-specific or generic languages. The projects are actively under curation of the MontiCore team.

cd4analysis mainly focuses on class diagrams for early activities of development projects. It contains full assistance for associations, composition, qualifiers, etc., but does not define method bodies. There is a code generator available, that ensures consistency of the defined model, storage, etc.

Feature Diagram provides grammars for a feature diagram and a feature configuration language. The project further contains advanced tools for performing semantic feature diagram analysis.

javaDSL is fully compatible with the Java 8 standard. It contains a complete grammar, symbol table infrastructure, context conditions, and related tooling. The language can be completely integrated into a larger languages or individual parts of the language can be selected for their integration.

JSON provides a grammar conforming to the JSON standard and advanced tooling such as a pretty printer and a model differencing tool.

MontiArc provides the language definition for a component and connector architecture description language. The project further contains a code generator that can generate simulations as executable Java source code from MontiArc component models.

Object Diagram provides grammars for defining several textual object diagram variants. The project also contains context conditions, a symbol table infrastructure and tooling such as a pretty printer.

OCL provides the OCL/P [Rum16] language definition, which is an enhanced variant of the UML Object Constraint Language with a syntax that is similar to Java.

se-commons The se-commons project provides the following parts that are used by MontiCore:

se-commons-logging contains the `Log` classes as described in Section 15.3.

se-commons-groovy contains auxiliary functions such as the `GroovyRunner` and `GroovyInterpreter` to interpret Groovy scripts (cf. Section 16.5). This is used in MontiCore itself, but also applicable for other tools.

se-commons-utilities contains utility classes such as `SourceCodePosition`, `CLIArguments`, and the `Names` helper.

sequence-diagram focuses on UML/P sequence diagrams [Rum16]. The project contains grammars, a symbol table infrastructure, context conditions, and advanced tooling such as a pretty printer and a semantic differencing tool.

si-units provides grammars and type checks for integrating data types based on physical units into larger languages.

Statechart provides the language definitions for building various Statechart variants, a symbol table infrastructure, and advanced tooling such as model-to-model transformations for normalizing Statecharts (e.g., expanding hierarchical states).

Chapter 23

Further Reading and Related Work

The following section gives an overview on related work done at the SE Group, RWTH Aachen. More details can be found on the website

<https://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 [Rum04]: “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 how digital and organizational techniques help to cope with 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 [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], 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, HR17] 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]. 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.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, size, and number of the 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] was introduced. [BGRW18] explains its applicability in systems engineering based on MDSE projects.

An artifact model basically 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 [KRRvW17] 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. 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. Smart semantic differencing operators have been defined for Activity Diagrams [MRR11a], Class Diagrams [MRR11d], 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 [KR18].

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. And we have applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, HRR12] and based it on the core ideas of Broy's Focus theory [RR11, BR07]. 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 and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP⁺11a, KLPR12] and city quarters [GLPR15] to optimize the operation efficiency and prevent unneeded CO2 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 based on the MontiCore language workbench [KRV10, GKR⁺06, GKR⁺08, HR17]. In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKRS06, GHK⁺15b] discuss mechanisms to keep generated and handwritten code separated. In [Wei12], we demonstrate how to systematically derive a transformation language in concrete syntax. [HMSNRW16] presents how to generate extensible and statically type-safe visitors. In [MSNRR16], 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 [HLMSN⁺15b, HLMSN⁺15a]. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], 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)

Starting with an 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] respectively [Rum12, Rum13] 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” [BCGR09a], [BCGR09b], [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 [MRR11c] and objects diagrams [MRR11e] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is 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⁺98], and how to use modeling in agile development projects [Rum04], [Rum02]. The question 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 [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR⁺06, KRV10, Kra10, GKR⁺08, HR17] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HR17] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR⁺07, Völ11, HLMSN⁺15b, HLMSN⁺15a, HRW18, BEK⁺18a, BEK⁺18b, BEK⁺19] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages in [KRV14]. [Wei12] presents a tool that allows to create 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]. DSL quality [FHR08], instructions for defining views [GHK⁺07], guidelines to define DSLs [KKP⁺09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using 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⁺15] and the concern-oriented language development approach [CKM⁺18]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HRW18, BEK⁺19]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08, HLMSN⁺15b, HLMSN⁺15a, HMSNRW16, HR17, BEK⁺18a, BEK⁺18b, BEK⁺19] and the backend [RRRW15, MSNRR16, GMR⁺16, HR17, BEK⁺18b]. In [GHK⁺15a, GHK⁺15b], we discuss the integration of handwritten and generated object-oriented code. [KRV14] describes the roles in software development using domain specific languages. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages [HRW18]. How to automatically derive such a transformation language using 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⁺15a, HHK⁺13], where a delta language is derived from a base language 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 [HRR12] for architecture design and extensions for states [RRW13b]. In [RRW13a], we introduce a code generation framework for MontiArc. 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 [NPR13] and with the robotics domain [AHRW17a, AHRW17b]. [GHK⁺07] and [GHK⁺08a] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14b] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. Co-evolution of architecture is discussed in [MMR10] and modeling techniques to describe dynamic architectures are shown in [HRR98, BHK⁺17, KKR19].

Compositionality & Modularity of Models

[HKR⁺09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically underpinned in [HKR⁺07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HR17] that can even be used to develop modeling tools in a compositional form [HR17, HLMSN⁺15b, HLMSN⁺15a, HMSNRW16, MSNRR16, HRW18, BEK⁺18a, BEK⁺18b, BEK⁺19]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP⁺09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF⁺15]) 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]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type 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]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a]

its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH⁺97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH⁺98] 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 on 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 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 for the development of CPS.

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. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12], and systematic model transformation language development [Wei12]. [Rum04] 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], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

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⁺08a] 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 [HRR⁺11, HRR⁺11] and to Delta-Simulink [HKM⁺13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK⁺13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability modeling languages in order 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 compositional reuse [BEK⁺18b], and applied it as a semantic language refinement on Statecharts in [GR11].

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied for 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], autonomous driving [BR12a, 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 for the European airspace [ZPK⁺11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. In [RRW13a], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition on contributing to systems engineering in automotive [GHK⁺08b], which culminated in a new comprehensive 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 modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for machining based on these concepts [BKL⁺18]. 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, BCGR09b, BCGR09a], (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 [RRW13a, RRW14, RRW13a, RW18] as well as 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 behaviour (2) based on information from previously stored and real-time monitored structural context and behaviour 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] or UML/P based languages [MNRV19]. [MM15] describes a process how languages for assistive systems can be created.

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 [SM18] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR⁺17] and in IoT manufacturing [MNRV19]. The user-centered view on 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.

Modelling 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 composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15, HR17] that perfectly fit robotic architectural modeling. The LightRocks [THR⁺13] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17a, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible to different robotics middleware platforms.

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 requirements management that connects requirements with features in all phases of the development 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⁺08a]. [HKM⁺13] describes a tool for delta modeling for Simulink [HKM⁺13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW⁺15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we introduce a framework for modeling the dynamic reconfiguration of component and connector architectures and apply it to the domain of cooperating vehicles. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current 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. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO₂ emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods 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 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 to 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 & Enterprise Information Systems

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, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK⁺14, HHK⁺15b], Big Data, App, and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. 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 own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

Model-Driven Engineering of Information Systems

Information Systems provide information to different user groups as main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HR17], we developed several generators for such data-centric information systems. *MontiGem* [AMN⁺20] 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⁺15b], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH⁺18].

List of Figures

1.2	Some languages MontiCore provides	3
1.4	Notational conventions	4
1.5	Agile use of models for coding and testing	5
2.7	Parts of the AST data structure generated for the Automata grammar . .	14
2.10	Parts of the class <code>AutomataParser</code> , which is generated from the Automata grammar	16
2.12	The scope classes generated from the Automata grammar	17
2.14	Parts of the symbol classes generated from the Automata grammar	17
2.17	Parts of the visitor infrastructure generated from the Automata grammar .	19
2.28	Eclipse after importing the example project and executing MontiCore	31
2.29	IntelliJ IDEA after importing the example project and executing MontiCore	32
3.1	Structure of a generator - external view	34
3.2	Internal architecture of a generator	35
3.3	Chapter structure of the handbook	36
4.32	Constraining the cardinality of a nonterminal, when parsing	56
5.2	Sequences of nonterminals in the AST	78
5.3	Optional nonterminals	78
5.4	How interfaces in a grammar map to the abstract syntax	79
5.5	How abstract nonterminals in a grammar map to abstract classes	79
5.7	Productions extending other productions	80
5.8	Implements in abstract and concrete syntax	81
5.9	Inheritance in abstract and concrete syntax	81
5.10	Extending the AST structure	82

5.11	Adding attributes in the AST with the <code>astrule</code> statement	82
5.12	Adding methods in the AST with the <code>astrule</code> statement	82
5.14	Terminals included in the AST	84
5.15	Choice of one of several values stored as <code>int</code>	84
5.16	Explicit definition of an enumeration	85
5.18	Common interfaces of AST classes	86
5.29	Example: Handwritten AST class <code>ASTState</code> injected into the parsing process	97
7.11	Language inheritance	122
7.17	Composition of a language is executed as late as possible: late binding . . .	128
8.6	Generated visitor infrastructure for a language <code>L</code>	138
8.7	Traverser for the <code>Questionnaire</code> language and an handcoded usage . . .	139
8.8	Control flow of a traverser with an attached visitor implementation	140
8.14	The traverser instantiation via <code>mill</code> always guarantees the most specific tra- verser type in composed languages	144
8.17	Overview of visitor classes for <code>Automata6</code>	146
8.23	Overview of visitor infrastructure for <code>Automata3</code>	150
9.4	Overview of the main concepts of SMI	159
9.5	An example structure of a hierarchy of scopes	160
9.7	Symbol interfaces provided by SMI RTE and a language specific class	162
9.9	Symbols may carry access modifiers defining their visibility	163
9.16	Scope classes generated for the <code>Automata</code> language	172
9.20	Relationship between AST, symbol, and scope by example	176
9.26	Infrastructure for parsing and printing JSON	185
9.36	Principle of resolution in a hierarchy of scopes	197
9.44	Effect of grammar inheritance on inheritance between scope interfaces	205
9.45	Effect of grammar inheritance on artifact and global scope interfaces	206
10.1	CoCo infrastructure for nonterminals	213
10.2	The generated CoCo checker	213
11.3	Components use <code>realThis</code> instead of <code>this</code> to enable close collaboration . .	224

11.4	The runtime program flow in a <code>realThis</code> composition	225
11.5	Splitting Classes in Interface and Implementation	226
11.6	Composing objects using the <code>realThis</code> approach	227
11.11	Template hook pattern variants that can be used for integrating handwritten and generated code	232
11.12	Publicly available interfaces of the mill pattern for object creation	234
11.13	Internal Structure of the Mill Pattern	235
11.14	Interface Pattern for language composition	236
13.17	Hierarchical include structure induced by the <code>include</code> commands. The knowledge direction goes from A to B to C and D	263
13.18	Decoration before and after a template. The knowledge direction is inverted: C and D know B	263
13.20	External replacement of a template	264
13.21	Defining an explicit hook point and binding it	264
13.22	External template replacement keeps its decoration (execution order is 1..5)	265
14.3	How a given class can be extended by building a subclass	281
14.5	How a given class can be replaced by a handwritten class	282
14.7	How a given class and its builder can be replaced by handwritten versions .	283
14.8	How a given class and its subclass can be replaced by a handwritten class .	284
15.10	Relationship between artifacts (templates and Java, excerpt)	299
15.11	Relationship between template artifacts (excerpt)	300
16.3	Different ways to define tasks	311
18.2	Component grammar hierarchy of Chapter 18	340
18.3	Grammars defining <code>Literal</code>	340
18.7	Grammars defining <code>Expression</code>	346
18.11	Component grammar defining symbols	352
18.13	Overview over the types grammar hierarchy	355
18.18	The <code>SymTypeEypression</code> class hierarchy	362
18.20	Overview over the <code>TypeCheck</code> class and Interfaces	365

18.22	TypeCheck configuration for MyLang	367
19.2	Component grammar structure hierarchy of Chapter 19	372
19.3	Some statement examples	372
20.1	A language feature diagram depicting the languages included in the JavaLight language	382
21.3	Two automata example models	390
21.7	AST of the Simple Automaton language	392
21.8	AST of the SAutomata language extended by runtime classes	392
21.9	Visitors for the Automata language	393
21.10	Symbols of the SAutomaton language	393
21.11	Generated scope classes and interfaces of the SAutomaton language	394
21.12	Relation between AST node, symbol and scope in the Automaton language	394
21.13	Generated classes for loading and storing symbol tables of the SAutomaton language	395
21.14	Defining context conditions for the SAutomata language	396
21.18	AST of the hierarchical Automaton language	398
21.22	AST of the Automaton language with Invariants	400

Listings

1.3	Example in Java	4
2.3	The Automata grammar	10
2.5	A model conforming to the Automata grammar	11
2.16	Different <code>resolve</code> methods	18
2.19	Attributes and constructor of the <code>PrettyPrinter</code> for the Automata language	20
2.21	Some methods of the <code>AutomataMill</code> API	22
2.23	The <code>CountStates</code> visitor implementation	23
2.24	Context condition implementation for checking that there exist at least one initial and at least one final state	24
2.25	Context condition implementation for checking that states used in transitions exist	25
2.26	Methods for parsing and creating symbol tables	25
2.27	Main method of the <code>AutomataTool</code> class	26
3.4	Example tool for the Automata DSL	36
4.1	Minimal grammar example	40
4.2	Lexical productions for <code>SimpleName</code> and <code>SimpleString</code>	41
4.3	Lexical productions for <code>Numbers</code> using token fragments	42
4.5	Lexical productions for white spaces	42
4.6	Lexical production for strings without quotation marks, which are removed in a Java action	43
4.7	Changing the result type of lexicals	43
4.8	Adding a conversion method for lexical types	44
4.12	Some production examples	46
4.13	<code>next</code> , <code>cmpToken</code> and <code>cmpTokenRegEx</code>	48
4.14	Augmentation of terminals for storage in the AST	49
4.15	A choice of alternate terminals is stored as integers	49
4.16	Explicit definition of an enumeration	50
4.17	Automatic naming of unnamed nonterminals	50
4.19	An interface nonterminal and several nonterminals implementing it	51
4.20	Alternative to interface nonterminal in Listing 4.19 accepting the same concrete syntax, but <code>I</code> knows <code>A</code> and <code>B</code>	51
4.21	Interface nonterminal defining its signature	52
4.22	Extending the production of a nonterminal	52
4.23	Equivalent alternative to extension in Listing 4.22 accepting the same concrete syntax for <code>A</code> , but <code>A</code> knows and thus is coupled to <code>B</code>	52

4.25	Abstract production in a grammar	53
4.26	Alternative to abstract nonterminal in Listing 4.25 accepting the same concrete syntax	53
4.27	Explicitly setting a top-level nonterminal that is inherited with <code>start</code> . . .	54
4.28	Grammar <code>ExpressionsBasis</code> which provides an interface <code>Expression</code> and basic expressions for the other expression grammars to use	54
4.29	Excerpt of grammar <code>AssignmentExpressions</code> for several forms of assignments	55
4.30	Excerpt of grammar <code>CommonExpressions</code> for common expressions like <code>a+b</code> including infix operation priorities	55
4.33	Constraining the cardinality of a nonterminal	56
4.34	Add Java code to the parser	57
4.35	Add Java code to the lexer	58
4.37	Add a conversion method for lexical types	61
4.39	Using <code>nokeyword</code> to define "automaton" as a local keyword	61
4.40	EBNF of the MontiCore grammar <code>MCG</code>	70
5.17	Signature of the <code>ASTNode</code> superclass of all AST nodes	85
5.19	Signature of the generated AST class to represent states: part 1	87
5.20	Attribute management signature of a generated AST class: part 2	88
5.21	Signature for a <code>List</code> attribute in a generated AST class: part 3	89
5.22	Comparison and cloning in a generated AST class: part 4	90
5.24	EMF version of the <code>ASTState</code> class signature	91
5.25	Signature of the builder mill for all <code>Automaton</code> AST classes	92
5.26	Signature of the <code>Builder</code> for <code>State</code> objects: part 1	92
5.27	Retrieving methods for a <code>Builder</code> class: part 2	94
5.31	Internal structure of the <code>AutomataMill</code>	98
5.32	Handcoded extension of the <code>AutomataMill</code>	98
6.1	Location of the MontiCore parser generator	102
6.2	Method signature used to generate a parser	102
6.3	Java code creates a parser for automata (using its grammar)	103
6.4	List of files produced during the generation of a parser	104
6.5	Methods that can be used for parsing	105
6.6	Various forms of parsing	106
6.8	Where to find the MontiCore grammar <code>grammar</code>	107
7.3	Example of a grammar component with an external nonterminal	117
7.4	External nonterminals are mapped to interfaces in the AST	118
7.5	Language embedding with binding the external nonterminal	118
7.6	Implementation of the <code>Invariant</code> nonterminal	119
7.9	Language inheritance: One grammar extending another and redefining an inherited nonterminal	120
7.10	The new <code>ASTState</code> class extends the old <code>ASTState</code> class and serves as a substitute	121
7.12	Language inheritance: One grammar extending another and redefining an inherited nonterminals inheritance structure without modifying the body . .	123

7.13	Language inheritance: One grammar extending another and redefining an inherited nonterminals by eliminating the body	124
7.14	Language inheritance: One grammar extending another and redefining an inherited nonterminals inheritance structure as well as the body	124
7.15	Language embedding: Filling extension points	125
8.2	Signature of a <code>Traverser</code> for language <code>L</code>	136
8.9	Simplified presentation of <code>visit</code> and <code>endVisit</code> operations in the visitor interface of language <code>L</code>	141
8.10	Simplified presentation of <code>handle</code> and <code>traverse</code> operations in the handler interface of language <code>L</code>	142
8.13	Implementation of an inheritance handler's <code>handle</code> method	143
8.15	Automaton language with interface nonterminal <code>AutElement</code> used for extension	146
8.16	Adding transitions with output to the <code>Automata5</code> language of Listing 8.15	146
8.18	A visitor implementation for the new language <code>Automata6</code>	147
8.19	Automaton language without explicit extension point	148
8.20	Conservative extension of transitions from <code>Automata15</code> of Listing 8.19 . .	148
8.21	A visitor of the new language for an overridden nonterminal	148
8.22	Prepare a handler implementation for compositional use	149
8.25	The implementation of the pretty printer for the <code>Automata3</code> sublanguage (with only one nonterminal)	151
8.26	Composing the three visitors through delegation and giving them the same shared state	152
8.27	The composed visitors can be used as if it is only one monolithic component	153
9.2	Example use of symbol defining grammar constructs	156
9.6	<code>Automata</code> with counters and transition statements	161
9.8	Interface of all <code>Symbol</code> classes	162
9.10	Excerpt of the generated implementation of class <code>StateSymbol</code>	165
9.11	<code>symbolrule</code> that defines three symbol attributes and a symbol method .	166
9.12	Configuring global scope attributes	169
9.13	Signature of all scope classes that implement <code>IScope</code>	169
9.14	<code>IArtifactScope</code> dealing with scopes for full artifacts and thus manages package information	170
9.15	<code>IGlobalScope</code> describes the interface of the global scope	171
9.17	<code>IAutomataScope</code> core functions	174
9.18	Method signatures of the <code>AutomataGlobalScope</code> class	174
9.19	A <code>scoperule</code> that defines two attributes and a method of the scope . . .	176
9.22	Repeated excerpt of <code>Automata</code> grammar	178
9.23	Extended signature of <code>ASTTransitions</code>	178
9.24	Methods of the <code>AutomataScopesGenitor</code>	180
9.25	Content of an Example Symbol Table <code>PingPong.autsym</code>	183
9.27	Signature of the <code>IDeSer</code> interface	186
9.28	Signature of the <code>ISymbolDeSer</code> interface	186
9.29	<code>AutomataDeSer</code> for scopes and artifact scope of the <code>Automata</code> language .	188

9.30	StateSymbolDeSer for State symbols with addon attribute adjacentStates of type List<String>	189
9.31	Optimized symbol table with state list only	193
9.32	Customization of the AutomatonSymbolDeSer	193
9.33	Customization of the AutomataSymbols2Json	194
9.35	Example for resolving a state symbol	197
9.37	Example class for bottom-up intra model resolution	198
9.38	Example class for inter model resolution	198
9.39	Example class for top-down inter model resolution	199
9.40	Standard resolving method signatures for StateSymbols in the IAutomataScope interface	200
9.41	More generated resolving method signatures for StateSymbols in the IAutomataScope interface	201
9.42	Handwritten adjustment of the method calculateModelNamesForState	202
9.43	Symbol table method signatures of a Visitor of a language L	204
9.46	Symbol adapter for CDCClassSymbols to StimulusSymbols	208
9.47	Resolving symbols with added adapters if the scope knows both symbol kinds	208
9.48	Example resolver for CDCClass2StimulusAdapters	210
10.3	Implementation of the AutomataCoCoChecker class	214
10.4	Configure the AutomataCoCoChecker and check the context conditions .	214
10.6	Implementation of a context condition for State objects	216
10.8	Using the symbol table in a context condition	216
10.9	Initial setup to test a context condition	217
10.10	Testing a context condition on a valid model	217
10.11	Testing a context condition on an invalid model	218
11.1	A static delegator method	222
11.2	Customized static delegator method	222
11.7	Classic get and set method signature for an attribute	228
11.8	Methods for an optional attribute	228
11.9	Methods for a List attribute	229
11.10	Manipulation methods provided by builders	231
12.1	Principle of FreeMarker: Copy the template content, execute FreeMarker commands, and inject their results into the output	237
12.2	Result when applying the template	238
12.5	FreeMarker conditional	241
12.6	FreeMarker switch statement	241
12.7	FreeMarker loop	241
12.8	Example for a FreeMarker loop	242
12.9	Extended form of a FreeMarker loop	242
12.10	signature asserts variables to be defined	243
13.1	Signature of a generate and generateNoA method	248
13.2	How the GeneratorEngine can be used	249
13.3	Configuration options of the GeneratorSetup class	250

13.6	Include methods provided by the <code>TemplateController tc</code>	254
13.7	Examples for including sub-templates within a template	255
13.8	The <code>includeArg</code> methods provided by the <code>TemplateController tc</code>	255
13.9	Examples for using signature	256
13.11	Write methods provided by the <code>TemplateController tc</code>	257
13.13	Further methods provided by the <code>TemplateController tc</code>	259
13.14	Logging examples from within templates	260
13.15	Methods to manage global variables with <code>glex</code>	260
13.16	Manipulating global variables from within a template	261
13.24	Signature that <code>HookPoints</code> provide	266
13.25	Constructor for <code>StringHookPoints</code>	267
13.26	Constructors for <code>TemplateHookPoint</code>	267
13.27	Constructor of <code>TemplateStringHookPoint</code>	267
13.28	Methods to define a hook point in a template	269
13.29	Methods of the <code>GlobalExtensionManagement</code> class for hook point management	270
13.30	Example: setting a hook point	271
13.31	<code>GlobalExtensionManagement</code> for hook point management	272
13.34	Producing attributes in the State Pattern	275
13.35	Resulting code for <code>State</code> attributes	275
13.36	Modified generation template	275
13.37	Replacing the default template	275
13.38	Resulting <code>State</code> attributes for adapted generation	276
13.39	Decorating the default template	276
13.40	Decorating template for <code>StatechartStateAttributes</code>	276
13.41	Resulting code including <code>get</code> functions	276
13.42	Template generating a <code>State</code> class	277
13.43	Binding strings to the hook points	277
13.44	Resulting method bodies and <code>count</code> attribute	278
15.2	Logging API in class <code>Log</code>	290
15.3	Example for controlling fail quick	291
15.4	How to enable reporting	294
15.5	How to stop and start reporting	295
15.6	Additional information reported in <code>08_Detailed.txt</code>	295
15.7	Exemplaric object identifiers in reports	295
15.8	Object identifiers in reports	296
15.9	Representation of various entities	296
16.1	Executing <code>MontiCore</code> via CLI	304
16.4	Integrating the <code>MontiCore</code> plugin in a build script	312
16.5	Creating a task to process the grammar <code>HierAutomata</code>	312
16.6	Configuring the model path	313
16.7	Using the Java Library plugin	314
16.8	Integrating the <code>MontiCore</code> plugin in a build script	314
16.9	Adding <code>MontiCore</code> dependencies	315

16.10	Repository declaration in <code>build.gradle</code>	315
16.11	Repository declaration in <code>settings.gradle</code>	315
16.12	Example <code>build.gradle</code>	316
16.13	Groovy script used to generate the standard result	320
16.14	Methods available in the Groovy scripts	322
18.15	Grammar <code>MyBasicLanguage</code> , showing the usage of the base grammars . .	360
18.16	Grammar <code>MySimpleBaseLanguage.mc4</code> , containing simple base gram- mar alternatives	361
18.17	The grammar <code>KotlinCommonGenericTypes.mc4</code>	361
18.19	Methods of the <code>TypeCheck</code> class	364
18.21	Example language <code>MyLang</code>	367
18.23	<code>init()</code> method of <code>SynthesizeFromMyLang</code>	368
18.24	Methods of the <code>TypeCheck</code> class	369
18.25	Usage of <code>TypeCheck</code> methods demonstrated in a JUnit test	369
20.2	External and interfaces introduced by the <code>JavaLight</code> grammar for com- fortable reuse of type parameters and class and interface elements	383
20.3	The <code>JavaMethod</code> symbol	383
20.4	Nonterminals defined in the <code>JavaLight</code> grammar that are used by the nonterminals for methods and annotations	384
20.5	The <code>MethodDeclaration</code> nonterminal	384
20.6	The nonterminals <code>InterfaceMethodDeclaration</code> and <code>ConstructorDeclaration</code>	385
20.7	The nonterminal <code>ConstDeclaration</code>	385
20.8	The interfaces <code>AnnotationArguments</code> and <code>ElementValue</code>	386
20.9	<code>JavaLight</code> nonterminals for annotations	386
20.10	The nonterminal <code>Annotation</code>	387
20.11	The nonterminal <code>ArrayDimensionByInitializer</code>	387
21.1	Simple automaton in text format	389
21.2	Example model for the Automaton language	390
21.4	MontiCore grammar for the <code>SAutomata</code> language	390
21.5	EBNF of the <code>SAutomata</code> language	391
21.6	MontiCore grammar for the <code>SAutomata</code> language	391
21.15	Files for context conditions for the <code>SAutomaton</code> language	396
21.16	Example model for the hierarchical Automaton language	397
21.17	MontiCore grammar for the <code>HAutomata</code> language	397
21.19	Example model for the Automaton language with invariants	398
21.20	Grammar component for <code>IAutomataComp</code> that defines state with invariants	399
21.21	Grammar for automata with state invariants	399
21.23	Whitespaces made explicit in the productions	401
21.24	Whitespaces explicitly used in productions	401
21.25	Whitespaces temporarily explicit in the production using a state switch in the lexer	402
21.26	Switching whitespaces on and off temporarily in productions	403
21.27	Disallowing spaces between last two token	403

21.29	Setting up a JUnit test infrastructure	404
21.30	Some JUnit tests	405
21.31	Example model of the ColoredGraph language	405
21.32	Grammar of the ColoredGraph language	406
21.33	Handwritten serialization of the color attribute of VertexSymbols in the class VertexSymbolDeSer	407
21.34	Handwritten deserialization of the color attribute of VertexSymbols in the class VertexSymbolDeSer	408

References

- [AHRW17a] 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.
- [AMN⁺20] 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)*, volume P-304 of *LNI*, pages 59–66. Gesellschaft für Informatik e.V., May 2020.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.
- [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.
- [BCGR09a] 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.
- [BCGR09b] 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.
- [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.

- [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.
- [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)*, volume 12127 of *Lecture Notes in Computer Science*, pages 85–100. Springer International Publishing, June 2020.
- [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.
- [Bec15] Kent Beck. *JUnit Pocket Guide*. O'Reilly, 2015.
- [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.
- [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⁺98] 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. 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.
- [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.
- [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.
- [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.
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.
- [BR12a] 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.

- [BR12b] 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.
- [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⁺15] 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 Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 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.
- [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.
- [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 Soft-*

-
- ware 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.
- [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.
- [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 Tërnavá, 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.
- [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. Kluwer Academic Publisher, 1999.
- [EHRR19] Robert Eikermann, Katrin Hölldobler, Alexander Roth, and Bernhard Rumpe. Reuse and Customization for Code Generators: Synergy by Transformations and Templates. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Model-Driven Engineering and Software Development*, pages 34–55. Springer, February 2019.
- [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 Tërnavá, 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.
- [FELR98] 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.
- [FLP⁺11a] 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*, pages 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.
- [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.
- [FPR01] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML/F Profile for Framework Architecture*. Addison-Wesley, 2001.
- [Fre21] FreeMarker website. <http://freemarker.org/>, 2021.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994.
- [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⁺08a] 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⁺08b] 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⁺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. 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⁺15b] 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*, volume 580 of *Communications in Computer and Information Science*, pages 112–132. Springer, 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.
- [GJS05] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, 3rd edition edition, 2005.
- [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 domänenspezifischer 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.
- [GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference, LNI 82*, pages 67–81, 2006.

- [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, chapter 56, pages 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*, volume 254 of *LNI*, 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.
- [Gra17] Graphviz. Graphviz - Graph Visualization Software, 2017. <http://www.graphviz.org/> [Online; accessed Jan-21].
- [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.
- [Gro21] GraphML Working Group. The graphml file format, 2021. <http://graphml.graphdrawing.org/> [Online; accessed Jan-2021].
- [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.

-
- [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⁺15a] 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⁺15b] 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.
- [HHRW15] 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 (Mod-Comp'15)*, volume 1463 of *CEUR Workshop Proceedings*, pages 18–23, 2015.
- [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 Engineering 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.

- [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.
- [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.
- [HMSNR15] Katrin Hölldobler, Pedram Mir Seyed Nazari, and Bernhard Rumpe. Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 23–30. ACM, 2015.
- [HMSNRW16] 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.
- [HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [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.
- [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.
- [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 eingebetteter Systeme 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 automotiv Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, pages 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.
- [JvdAB⁺21] Matthias Jarke, Wil van der Aalst, Christian Brecher, Matthias Brockmann, István Koren, Gerhard Lakemeyer, Bernhard Rumpe, Günther Schuh, Klaus Wehrle, and Martina Ziefle. *Mit "Digitalen Schatten" Daten verdichten und darstellen*, pages 18–23. image Druck + MEDIEN GmbH, Aachen, February 2021.
- [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.

- [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.
- [KLK⁺15] Dierk König, Guillaume Laforge, Paul King, Jon Skeet, and Hamlet D’Arcy. *Groovy in Action, 2nd Edition*. Manning Publications, 2015.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming, ECOOP’97*. Springer Verlag, 1997.
- [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.
- [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.
- [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, editors, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aach-

- ener 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, *ASE’19. Software Engineering Intelligence Workshop (SEI’19)*, pages 126–133. IEEE, November 2019.
- [KR18] 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. Chapman & Hall, 1996.
- [KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.
- [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)*, volume 2308 of *CEUR Workshop Proceedings*, pages 87–94. CEUR Workshop Proceedings, February 2019.
- [KRRvW17] 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*, pages 113–116. VDI Verlag, 2012.
- [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 Steven 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.
- [KRV14] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modeling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM' 06)*, volume abs/1409.6618. CoRR arXiv, 2014.
- [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.
- [LMK⁺11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Zieffle. 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*, volume 8217 of *LNCS*, 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.
- [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.
- [MRR11a] 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.
- [MRR11b] 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.
- [MRR11c] 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.
- [MRR11d] 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.

- [MRR11e] 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*, volume 227 of *LNI*, 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)*, volume 1723 of *CEUR Workshop Proceedings*, 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)*, volume 2628, pages 11–18. CEUR Workshop Proceedings, June 2020.
- [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.
- [MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

-
- [MSNRR15] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 37–42. ACM, 2015.
- [MSNRR16] 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*, volume 254 of *LNI*, pages 133–140. Bonner Köllen Verlag, March 2016.
- [NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, volume 1118 of *CEUR Workshop Proceedings*, pages 15–24, 2013.
- [Oli07] Bruno C. d. S Oliveira. *Genericity, extensibility and type-safety in the Visitor pattern*. Oxford University, 2007.
- [Par13] Terence Parr. *The definitive ANTLR 4 reference*. The pragmatic programmers. O'Reilly Vlg. Gmbh & Co., 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*, volume 254 of *LNI*, 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.
- [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.

- [Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.
- [QOS21a] QOS.ch. Logback project, 2021. <http://logback.qos.ch/> [Online; accessed Jan-2021].
- [QOS21b] QOS.ch. Simple logging facade for java, slf4j, 2021. <http://www.slf4j.org/> [Online; accessed Jan-2021].
- [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, 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.
- [RRRW15] 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.
- [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.
- [RRW13a] 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.
- [RRW13b] 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.
- [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.
- [RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Transforming Platform-Independent to Platform-Specific Component and Connector Software Architecture Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, volume 1463 of *CEUR Workshop Proceedings*, pages 30–35, 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.
- [Rum04] 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.
- [Rum13] Bernhard Rumpe. Towards Model and Language Composition. In Benoit Combemale, Walter Cazzola, and Robert Bertrand France, editors, *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013.
- [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.
- [SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.
- [SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.
- [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.
- [SM18] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0 (Extended Abstract): Integration of non-smart resources into cognitive assistance systems. *EMISA Forum*, 38(1):35–36, Nov 2018.
- [SM20] Claudia Steinberger and Judith Michael. *Using Semantic Markup to Boost Context Awareness for Assistive Systems*, pages 227–246. Computer Communications and Networks. 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.
- [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.
- [Vli98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Professional, 1998.
- [Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.
- [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.
- [yWo21] yWorks. yEd Graph Editor, 2021. <http://www.yworks.com/en/products/yfiles/yed/> [Online; accessed Jan-21].
- [ZPK⁺11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.

Index

`${...}`, 237
_ast, 13, 14
_auxiliary, 13
_channel, 42
_cocos, 13, 20
_od, 13
_parser, 13, 16
_symboltable, 13, 16
_visitor, 13, 19
actual type check, 365
-modelPath, 190
-cl, 293
-customLog, 293
-dev, 293
-d, 293
-handcodedPath, 251
-hcp, 280, 281
-mp, 120, 190
-out, 280
-script, 318, 319
-s, 318, 319
-templatePath, 251
.now, 240
01_Summary.txt, 297
02_GeneratedFiles.txt, 297
03_HandwrittenCodeFiles.txt, 297
04_Templates.txt, 297
05_HookPoint.txt, 297
06_Instantiations.txt, 297
07_Variables.txt, 297
08_Detailed.txt, 298
09_TemplateTree.txt, 298
10_NodeTree.txt, 298
11_NodeTreeDecorated.txt, 298
12_TypesOfNodes.txt, 298
14_Transformations.txt, 298
15_ArtifactGml.gml, 299
16_ArtifactGv.gv, 299
18_InvolvedFiles.txt, 298, 300
</#directive>, 240
<#-- ... --#>, 238
<#assign name=value>, 241
<#case ...>, 241
<#compress>, 242
<#default>, 241
<#directive parameters>, 240
<#else>, 241, 242
<#elsif ...>, 241
<#if ...>, 238, 241
<#items ...>, 242
<#list ...>, 241, 242
<#switch ...>, 241
<rightassoc>, 54
>>>, 349
>>, 349
?date, 239
ASTAutomata3Node, 87
ASTAutomatonBuilder, 92
ASTCNode, 80, 86, 87, 94
ASTConstantsGr, 84
ASTNodeBuilder, 92
 PostComments, 92
 PreComments, 92
 SourcePositionEnd, 92
 SourcePositionStart, 92
 build, 92
 deepClone, 92
 deepEqualsWithComments, 92
 deepEquals, 92
 equalAttributes, 92
 equalsWithComments, 92
ASTNode, 85, 86, 163
 PostComments, 85
 PreComments, 85
 SourcePositionEnd, 85
 SourcePositionStart, 85

- deepClone, 85
- deepEqualsWithComments, 85
- deepEquals, 85
- equalAttributes, 85
- equalsWithComments, 85
- ASTStateBuilder, 92–94
- ASTStateTOP, 96, 282
- ASTState, 86–88, 90, 96, 176
- ASTTransitionBuilder, 92
- Aliases, 253
- AnnotationArguments, 386
- AnnotationPairArguments, 386
- Annotation, 387
- ArrayDimensionByInitializer, 387
- ArrayDimensionSpecifier, 387
- ArrayInit, 387
- ArtifactScope, 170
- AssertStatement, 377
- AssignmentExpressions, 349, 381
- AttributeSymbol, 160
- AutElement, 145
- Automata15, 147
- Automata16, 147
- Automata1, 120
- Automata3PrettyPrinter, 152
- Automata3Tool, 153
- Automata3, 149, 150
- Automata5, 145
- Automata6PrettyPrinter, 146
- Automata6, 145
- AutomataASTAutomatonCoCo, 212
- AutomataASTStateCoCo, 212
- AutomataASTTransitionCoCo, 212
- AutomataArtifactScope, 167, 172, 173
- AutomataCoCoChecker, 213
- AutomataDeSer, 188
- AutomataGlobalScope, 168, 172–174
- AutomataMillTOP, 98
- AutomataMill, 92, 98
- AutomataParser, 36, 106
- AutomataScopesGenitor, 179, 180
- AutomataScope, 172, 173, 176
- AutomataSymbols2Json, 187, 188
- AutomataTool.jar, 309
- AutomataTraverser, 214
- Automata, 87, 161, 167, 213
- AutomatonElement, 79
- AutomatonSymbolDeSerTOP, 193
- AutomatonSymbolDeSer, 193
- Automaton, 391
- BasicAccessModifier, 163
- BasicLongLiteral, 343
- BasicSymbols, 352–354, 382
- BitExpressions, 349
- BlockScope, 160
- BooleanLiteral, 342
- BreakStatement, 378
- CARDINALITY, 43
- CD4A, 3
- CD4Code, 246
- CD, 3
- CallExpression, 348
- Cardinality, 336
- CharLiteral, 329, 342
- CharToken, 329
- Char, 342
- CheckScannerlessTest, 404
- ClassBodyDeclaration, 383, 384
- ClassExpression, 350
- ClassScope, 160
- ClassSymbol, 160
- Cloneable, 86
- CodeHookPoint, 267, 268
- ColoredGraph, 405
- CommonExpressions, 348
- Completeness, 337
- ConstDeclaration, 385
- ConstructorDeclaration, 385
- ContinueStatement, 378
- Counter, 161
- DeSer, 183–185, 206
- DecimalToken, 332
- Decimal, 332
- DeclaratorId, 384
- Deprecated, 58
- DeriveFromMyLang, 367
- DeriveSymTypeOf, 366
- DiagramSymbol, 352, 353
- Diagram, 353, 382
- Digits, 345
- Digit, 345
- DoWhileStatement, 375

-
- DoubleLiteral, 345
 - EObjectContainmentEList, 90
 - ElementValueArrayInitializer, 386
 - ElementValueOrExpr, 386
 - ElementValuePair, 386
 - ElementValue, 386
 - EmptyStatement, 374
 - EscapeSequence, 342
 - ExpressionStatement, 374
 - ExpressionSublangPP, 149
 - ExpressionsBasis, 347, 381
 - Expression, 123, 125, 365
 - ExtReturnType, 350
 - ExtTypeArgument, 350
 - ExtTypeParameters, 383
 - ExtType, 350
 - Ext, 118
 - Feature Diagram, 417
 - FieldAccessExpression, 348
 - FieldSymbol, 354
 - Field, 354, 382
 - FileReaderWriter, 257
 - FloatLiteral, 345
 - ForStatement, 375
 - FormalParameterListing, 384
 - FreeMarkerTemplateEngine, 251
 - FunctionSymbol, 353, 354
 - Function, 353, 382
 - GenerateAutomataParser, 103
 - GeneratorEngine, 247–249, 258
 - GeneratorSetup, 247, 249, 250, 254
 - GlobalExtensionManagement, 252, 260, 270, 272, 324
 - GlobalExtensionMangement, 251
 - Grammar_WithConcepts.mc4, 106
 - Grammar_WithConceptsParser, 106
 - HAutomata, 397
 - HIDDEN, 402
 - Hash, 239
 - HexDigit, 345
 - HexInteger, 334
 - HexadecimalToken, 333
 - Hexadecimal, 333
 - HierarchicalAutomata, 120
 - HookClass, 232
 - HookPoint, 266
 - IArtifactScope, 172
 - IAutomataArtifactScope, 172–174
 - IAutomataComp, 398
 - IAutomataGlobalScope, 168, 172, 173, 202
 - IAutomataScope, 172–174, 195, 200, 201
 - IAutomata, 398
 - ICommonAutomataSymbol, 164, 165
 - IDeSer, 184, 186
 - IDerive, 365
 - IGlobalScope, 171, 172, 394
 - IScope, 169, 172
 - IStateSymbolResolver, 164
 - ISymbolDeSer, 185–187
 - ISymbol, 162
 - ISynthesize, 365, 366
 - ITSymbolResolver, 209
 - IfStatement, 374
 - IncGenCheck.sh, 307–309
 - IncGenGradleCheck.txt, 312
 - Increment, 161
 - InfixExpression, 348
 - InputOutputFilesReporter, 325
 - InstanceofExpression, 350
 - IntLiteral, 345
 - Integer, 333
 - InterfaceBodyDeclaration, 383
 - InterfaceMethodDeclaration, 385
 - InvAutomata, 125, 150
 - Invariant, 118, 399
 - IterablePath, 324
 - JAVA_HOME, 8
 - JSON, 417
 - JavaArtifactScope, 160
 - JavaClassExpressions, 350, 381, 387
 - JavaCopyright, 270
 - JavaGlobalScope, 160
 - JavaLight, 381
 - JavaMethod, 383, 384
 - JavaModifier, 374, 384
 - Java, 381
 - JSONArray, 184, 185
 - JsonBoolean, 184, 186
 - JsonDeSers, 185
 - JsonElement, 184, 185
 - JsonNumber, 184, 186

- JsonObject, 184, 185
- JsonParser, 184, 185
- JsonPrinter, 184, 185
- JsonString, 184, 185
- LCoCoChecker, 213
- LDelegatorVisitor, 151
- LHandler, 137
- LInheritanceHandler, 137, 142, 143
- LTraverserImplementation, 137
- LTraverser, 137
- LVisitor2, 137
- LabelSymbol, 378
- Label, 378
- LastFormalParameter, 384
- List, 88
- Literal, 341, 381
- Log.init(), 293, 321
- Log.initDEBUG(), 293
- Log.initWARN(), 293
- LogStub.init(), 293
- LogStub, 217, 219, 223, 290, 294
- LogicExpr, 118, 125, 399
- LogicalRightShiftExpression, 349
- Log, 221, 223, 252, 253, 259, 289, 290, 294, 325
- LongLiteral, 345
- MCArraYStatements, 374
- MCArraYTypes, 46, 359
- MCArraYType, 359
- MCAssertStatements, 376, 377
- MCBasicGenericType, 358
- MCBasicTypeArgument, 357
- MCBasicTypes, 46, 181, 356, 357, 382
- MCBasics, 43, 126, 328
- MCBlockStatement, 372, 382
- MCCollectionTypes, 46, 357
- MCCommonLiterals, 44, 341
- MCCommonStatements, 374, 376, 381
- MCCustomTypeArgument, 358
- MCExceptionStatements, 377
- MCFullGenericTypes, 46, 358
- MCFullJavaStatements, 379
- MCHexNumbers, 333
- MCImportStatement, 46, 181, 357
- MCInnerType, 358
- MCJavaBlock, 374
- MCJavaLiterals, 44, 345
- MCListType, 46, 357
- MCLiteralsBasis, 43, 341, 381
- MCLowLevelStatements, 378
- MCMaPType, 357
- MCMoDifier, 372, 382, 384, 387
- MCMultipleGenericType, 358
- MCNumbers, 331
- MCOBJECTType, 356, 382
- MCOPTIONALType, 357
- MCPackageDeclaration, 181, 357
- MCPriMitiveTypeArgument, 357
- MCPriMitiveType, 356
- MCQualifiedName, 46, 357
- MCQualifiedType, 356
- MCReturnStatements, 376
- MCReturnType, 356, 384
- MCSetType, 46, 357
- MCSimpleGenericTypes, 46, 358
- MCStatementsBasis, 371, 372, 382
- MCStatement, 372, 382
- MCSynchronizedStatements, 377
- MCTask, 311
 - Gradle, 314, 315
- MCType, 356, 366, 382
- MCVarDeclarationStatements, 373, 381
- MCVoidType, 356
- MCWildcardTypeArgument, 358
- ML_COMMENT, 328
- MethodDeclaration, 384
- MethodScope, 160
- MethodSymbol, 160, 354
- Method, 354, 382
- MinusExpression, 348
- ModelPath, 168, 324
- Modifier, 336
- MontiArc, 417
- MontiCoreCLI, 303, 304
- MontiCoreReports, 295, 324
- MontiCoreScript, 295, 318, 321–324
- MultExpression, 348
- MyLang, 367
 - DeriveFromMyLang, 367
 - SynthesizeFromMyLang, 367
- MyTransitionBuilder, 98

-
- NEWLINE, 328
 - Names, 325
 - Name, 78, 88, 177, 328
 - NatLiteral, 343
 - Not, 399
 - NullLiteral, 342
 - Number, 331
 - NumericLiteral, 342
 - OCL, 417
 - OOSymbols, 352, 354, 382
 - OOSymbol, 354
 - OType, 354, 382
 - Object Diagram, 417
 - Observer, 80
 - OctalDigit, 345
 - Optional, 88
 - Override, 58
 - PATH, 8, 29
 - ParserGenerator, 102
 - PingPong.autsym, 183, 193
 - PingPong.aut, 308
 - PingPong, 183, 390
 - PlusExpression, 348
 - PostComments, 85, 92
 - PreComments, 85, 92
 - QuestionnaireTool, 138
 - Questionnaire, 408
 - Reporting.flush(), 295
 - Reporting.off(), 295
 - Reporting.on(.), 295
 - Reporting.reportToDetailed(), 295
 - Reporting, 295, 325
 - ReturnStatement, 376
 - RightShiftExpression, 349
 - SAutomata, 389, 390
 - SI Units, 418
 - SL_COMMENT, 328
 - Scannerless, 400
 - ScopeGenitors, 205
 - ScopesGenitorDelegator, 205
 - ScopesGenitor, 180
 - Scope, 394
 - Sequence, 239
 - SignedBasicLongLiteral, 343
 - SignedLiteral, 342
 - SignedNatLiteral, 343
 - SignedNumericLiteral, 342
 - SimpleName, 41
 - SimpleString, 41
 - Slf4jLog, 290
 - SopeBuilder, 394
 - SourcePositionEnd, 85, 92
 - SourcePositionStart, 85, 92
 - SpaceOnOff, 402
 - StateNameStartsWithCapitalLetter, 215
 - StateSymbolBuilder, 164
 - StateSymbolDeSer, 164, 188, 189
 - StateSymbolSurrogate, 164
 - StateSymbol, 164, 165, 176
 - Statechart, 418
 - Statement, 161
 - State, 118, 145, 147, 391
 - StereoValue, 335
 - Stereotype, 335
 - StringHookPoint, 267
 - StringLiterals, 329
 - StringLiteral, 330, 342
 - StringToken, 330
 - String, 78
 - SwitchStatement, 376
 - SymTypeArray, 363
 - SymTypeConstant, 362
 - SymTypeExpression, 353, 362, 364
 - SymTypeEypression, 362
 - SymTypeOfGenerics, 363
 - SymTypeOfNull, 363
 - SymTypeOfObject, 363
 - SymTypeOfWildcard, 363
 - SymTypeVariable, 363
 - SymTypeVoid, 363
 - SymbolResolver, 209
 - SymbolSurrogate, 393
 - Symbols2Json, 187, 190, 191, 205
 - Symbol, 393
 - Synthesize
 - init, 368
 - SynthesizeFromMyLang, 367, 368
 - SynthesizeSymTypeFrom, 366
 - Synthesize
 - getResult, 368
 - getTraverser, 368

- TOP mechanism, 233
- TemplateClass, 232
- TemplateController, 252, 254, 259
- TemplateHookClass, 232
- TemplateHookPoint, 256, 267
- TemplateStringHookPoint, 267
- ThisExpression, 350
- ThrowStatement, 377
- Throws, 384
- TransitionSourceExists, 215
- TransitionWithOutput, 145
- Transition, 145, 147, 391
- TraverserConfiguration, 142
- Truth, 399
- TryStatement*, 377
- TypeCastExpression, 350
- TypeCheck
 - compatible, 364, 365
 - isBoolean, 364
 - isByte, 364
 - isChar, 364
 - isDouble, 364
 - isFloat, 364
 - isInt, 364
 - isLong, 364
 - isOfTypeForAssign, 364, 365
 - isShort, 364
 - isString, 364
 - isVoid, 364
 - symTypeFromAST, 364
 - typeOf, 364
 - usage, 368
- TypeSymbol, 353
- TypeVarSymbol, 353
- TypeVar, 353, 382
- Type, 353, 382
- UMLModifier, 336
- UMLStereoType, 335
- UML, 1
- UnicodeEscape, 342
- Variable declarations, 373
- VariableSymbol, 353, 363
- Variable, 353, 382
- VisitorInfrastructure, 137
- WSoff, 403
- WSOn, 403
- WS, 328
- WhileStatement, 375
- WhiteSpace, 400, 402
- XYAntlr.g4, 104
- XYAntlrLexer, 104
- XYAntlrParser, 104
- XYParser, 104
- `\${...}`, 238
- _ast, 77
- _automatonBuilder, 98
- _channel, 402
- _cocos, 77
- _parser, 77
- _reachable, 96
- _report, 77
- _stateBuilder, 98
- _symboltable, 77, 164
- _transitionBuilder, 98
- _visitor, 77
- abstract, 88
- accept, 87, 137, 165
 - symbol, 165
- addAdaptedStateSymbolResolver,
 - 168
- addCoCo, 213
- addLoadedFile, 171
- addToGlobalVar, 253, 260
- additionalTemplatePaths, 251
- add
 - scope, 174
- adjacentStates, 190
- assign, 260
- astextends, 80
- astextend, 80
- astimplements, 80
- astrule, 80, 81
- ast, 248, 252, 256
- automatonBuilder, 92
- beginArray, 194
- bindHookPoint, 253, 270, 273, 274, 277
- bindStringHookPoint, 270
- bindTemplateHookPoint, 270
- bind, 265
- boolean, 46
- build(), 94
- build.gradle

- Gradle, 315
- buildDir
 - Gradle, 315
- build, 92, 413
 - Gradle, 314
- calculateModelNamesForC, 191
- calculateModelNamesForState, 202
- cd4analysis, 417
- changeGlobalVar, 253, 260
- checkAll, 213, 214
- check, 212
- clearLoadedFiles, 171
- clear, 171
- cmpToken(n, ...), 344
- commentEnd, 251
- commentStart, 251
- compatible, 364, 365
- compileJava
 - Gradle, 314, 315
- component, 116, 117
- configTemplate
 - Gradle, 313
- configureGenerator, 322, 323
- continueAsStateSubScope, 201
- continueStateWithEnclosingScope, 201
- createFromAST, 180
- createMCGlobalScope, 322, 323
- createScope, 180, 181
- createSymbolsFromAST, 322, 323
- customLog
 - Gradle, 313
- debug, 253, 260, 288, 290
- decorateCD, 322, 323
- decorateEmfCD, 322, 323
- deepClone, 85, 90, 92
- deepEqualsWithComments, 85, 90, 92
- deepEquals, 85, 86, 90, 92
- defaultFileExtension, 250
- defineGlobalVar, 253, 260
- defineHookPointWithDefault, 253, 270
- defineHookPoint, 253, 268, 270
- delegate, 207
- dependencies
 - Gradle, 314, 315
- dependsOn, 311
- deriveCD, 322, 323
- deserializeAddons, 189, 193, 194
- deserializeAdjacentStates, 190
- deserialize, 186, 187, 189
- dev, 414
 - Gradle, 313
- doInfo, 221
- double, 382
- eGet, 90
- eIsSet, 90
- eNotify, 90
- eSet, 90
- eUnset, 90
- enableFailQuick, 290, 291
- endArray, 194
- endVisit, 136
 - ScopesGenitor, 180
- enum, 49, 84
- equalAttributes, 85, 90, 92
- equalsWithComments, 85, 90, 92
- error, 253, 260, 288, 290
- existsHandwrittenFile, 259
- existsHookPoint, 253, 269, 270
- extends, 53, 120
- external, 116, 117
- flush, 295
- for, 381
- ftl, 237
- generateEmfFromCD, 322, 323
- generateFromCD, 322, 323
- generateNoA, 248
- generateParser, 102, 322, 323
- generate, 248
 - Gradle, 315
- getAccessModifier, 162
- getAdaptedStateSymbolResolverList, 174
- getAstNode, 165
- getDeSer, 171
- getEnclosingScope, 162, 169, 170
- getErrorCount, 290
- getFileExt, 171
- getFullName, 162, 170, 171
- getGlobalVar, 253, 260
- getLocalStateSymbols, 174

getModelPath, 171
getName, 162
getPackageName, 162, 170, 171
getResult, 366
getSource(), 344
getSourcePosition, 162
getSource, 331
getStateSymbols, 174
getSubScopes, 170
getSymbolDeSers, 171
getSymbolDeSer, 171
getSymbols2Json, 174
getSymbolsSize(), 170
getSymbolsSize, 169
getTemplatename, 259
getToSymbol, 195
getTraverser, 366
getValue(), 45, 343, 344
getValue, 331
glex, 251, 252, 256, 260, 324
grammarDir, 312
grammarIterator, 324
grammar
 Gradle, 314, 315
groovyHook1, 324
 Gradle, 313
groovyHook2, 324
 Gradle, 313
group
 Gradle, 315
handCoded, 96
handcodedPath, 96, 250, 324
 Gradle, 313
handle, 136
hasGlobalVar, 260
help
 Gradle, 313
hook, 322, 323
hwc, 23
implementation
 Gradle, 314
import, 116, 181
incCheck, 311, 312
 Gradle, 315
incGenStamp.f, 308
include2, 253
 includeArgs, 253, 255, 256
 include, 253, 254
 info, 221, 253, 260, 288, 290
 init(), 223
 initArtifactScopeHP1, 180, 181
 initArtifactScopeHP2, 180, 181
 initScopeHP1, 180, 181
 initScopeHP2, 180, 181
 initStateHP1, 180
 initStateHP2, 180
 init, 127, 171
 instanceof, 381
 instantiate, 259
 interface, 88, 116
 int, 46, 382
 isBoolean, 364
 isByte, 364
 isChar, 364
 isDouble, 364
 isExportingSymbols, 169, 170
 isFailQuickEnabled, 290, 291
 isFileLoaded, 171
 isFloat, 364
 isInt, 364
 isLong, 364
 isOfTypeForAssign, 364, 365
 isOrdered, 169, 170
 isPresentAstNode, 162, 165
 isReachable, 96
 isShadowingScope, 170
 isShadowing, 169
 isShort, 364
 isStateSymbolAlreadyResolved, 201
 isString, 364
 isVoid, 364
jar
 Gradle, 314
javaDSL, 417
key, 61
loadC, 191
loadFileForModelName, 174
loadState, 174
load, 190
makefile, 306–308
master, 414
mc4, 103

- method, 83, 166
- modelPath, 313
- monticore-cli, 416
- monticore-editor, 416
- monticore-emf-runtime, 416
- monticore-generator, 416
- monticore-grammar, 416
- monticore-maven, 416
- monticore-runtime, 416
- monticore-templateclassgen., 416
- monticore.YYYY-MM-DD-HHmms.log, 14
- monticore_standard.groovy, 318, 319
- new, 381
- noSpace(), 403
- noSpace(n), 343, 344
- noSpace, 335
- nokeyword, 42, 61, 338
- nospace(), 60
- null, 88, 95
- outDir, 312
- outputDirectory, 250
- outputDir, 312
 - Gradle, 315
- out, 13, 23
- package, 181
- parseGrammars, 322, 323
- parseGrammar, 322, 323
- parseNT, 105
- parse_StringNT, 105
- parse_String, 105
- parse, 105
- pluginManagement
 - Gradle, 315
- plugins
 - Gradle, 315
- processValue, 266
- projectDir, 312
- putOnStack, 181
- putSymbolDeSer, 168, 171
- reachableStates, 83
- remove
 - scope, 174
- replaceTemplate, 265, 272–275
- reportCD, 322, 323
- reportManagerFactory, 295, 324
- reportToDetailed, 295
- repositories
 - Gradle, 315
- requiredGlobalVars, 253, 260
- requiredGlobalVar, 253, 260
- resolveAdaptedStateLocallyMany, 201
- resolveAdaptedTLocallyMany, 208
- resolveAdaptedTSymbol, 209
- resolveDownMany, 196
- resolveDown, 196, 200
- resolveLocallyMany, 196
- resolveLocally, 196, 200
- resolveMany, 195, 200
- resolveSLocallyMany, 209
- resolveStateDownMany, 199, 201
- resolveStateDown, 199, 201
- resolveStateLocallyMany, 201
- resolveStateLocally, 201
- resolveStateMany, 200
- resolveState, 195, 200
- resolve, 195, 199, 200
- runGrammarCoCos, 322, 323
- scoperule, 176, 187
- scope, 167, 172
- script
 - Gradle, 313
- se-commons-groovy, 418
- se-commons-logging, 418
- se-commons-utilities, 418
- se-commons, 417
- sequence diagram, 418
- serializeAddons, 189, 193
- serializeAdjacentStates, 190
- serialize, 186, 187, 189
- setAccessModifier, 162
- setAdaptedStateSymbolResolverList, 174
- setAfterTemplate, 265, 272–274, 276
- setAstNode, 169
- setBeforeTemplate, 265, 272–274
- setDeSer, 171
- setFileExt, 168, 171, 203
- setModelPath, 168, 171
- setName, 169
- setSpanningSymbol, 169

- setStateSymbolAlreadyResolved, 201
- setSymbolDeSers, 171
- setSymbols2Json, 174
- setTraverser
 - ScopesGenitor, 180
- settings.gradle
 - Gradle, 315
- signature, 243, 249, 253, 267
- sourceSets
 - Gradle, 314, 315
- spaceOnFlag, 402
- splittoken, 42, 60, 335, 338, 349
- src/main/grammars, 414
- src/main/java, 413
- src/main/models, 414
- src/main/resources, 413
- src/test/java, 413
- src/test/resources, 413
- src, 23
- start, 54, 121
- stateBuilder, 92
- store, 190, 191
- super, 381
- switch, 381
- symTypeFromAST, 364
- symbolrule, 166, 187
 - attribute, 166
 - method, 166
- symbol, 160, 161
 - Name?, 161
 - getName(), 161
 - inherited, 161
 - interface, 161
 - on extended nonterminal, 161
- sym, 168, 192, 203
- target, 413
- tc, 252, 254, 256, 259
- templatePath, 324
 - Gradle, 313
- testImplementation
 - Gradle, 314
- this, 381
- trace, 253, 260, 288, 290
- tracing, 251
- transitionBuilder, 92
- traverse, 136
- typeof, 364
- varName??. 240
- visit, 136
 - ScopesGenitor, 180
- void, 382
- warn, 253, 260, 288, 290
- while, 381
- writeArgs, 257
- write, 257

- abstract, 52
- abstract nonterminal, 52, 79
- abstract syntax, 14, 39, 77, 114
- abstract syntax (AST), 34
- abstract syntax tree, 34
- AbstractState.ftl, 274, 275
- access modifier, 158
- action, 43
- adaptation
 - subclass, 281
 - TOP mechanism, 281
- adapter
 - symbol, 206
- Agile Model-Based Development, 6
- agile modeling, 5
- ambiguity
 - resolution, 195
- annotation, 381
- annotations, 58
- anonymous scope, 167
- Ansi-C++, 2
- ANTLR, 39, 57, 69, 103, 319
 - concept, 57
 - lexerjava, 57
 - parserjava, 57
- arrays, 46, 363
- artifact, 109, 279
 - generated, 279
 - handcoded, 279
- artifact scope, 167, 205
- AST, 14, 34, 77
 - extension, 81
 - handwritten extension, 96
 - spanning tree, 155
- AST classes, 77

-
- AST-access-conservative, 131
 - AST-conservation, 122
 - AST-conservative, 131, 145
 - AST-modification-conservative, 131, 132
 - attribute, 381
 - symbolrule, 166
 - Automata DSL, 36
 - autonomous driving simulation, 2
 - AutoSar, 2
 - axiom, 54

 - backend, 116
 - Base Grammars, 359
 - bidirectional association pattern, 230
 - black-box reuse, 109
 - body, 46
 - bottom-up intra model resolution, 197, 201
 - build script, 29, 413
 - builder, 91, 114, 126, 281, 282
 - AST-conservative extension, 127
 - handwritten extension, 97
 - initialization, 127
 - mill, 233
 - builder mill, 91
 - handwritten extension, 97
 - initialization, 127
 - builder mills, 281
 - building facilities, 2

 - cardinality, 55
 - Char, 44
 - checker, 213
 - CI, 414
 - class body, 384
 - Class Diagrams for Analysis, 3
 - CLI, 7, 8, 12, 303, 306
 - cloud service, 2
 - CoCo, 211
 - code hook, 266
 - collection types, 46
 - command line, 7
 - Command Line Interface, 417
 - command line interface, 7
 - commandline, 304
 - commandline interface, 303
 - comparison, 86
 - compiling, 22
 - component
 - grammar, 39
 - language, 109
 - component grammar, 112
 - composition, 109
 - abstract syntax, 114
 - backend, 116
 - builder, 114, 126
 - concrete syntax, 113
 - conservative, 112
 - context conditions, 115
 - language mill, 126
 - late binding, 111
 - parser, 114, 127
 - parser CS-conservative, 128
 - scopes, 115
 - semantic, 111
 - symbol tables, 204
 - symbols, 115
 - visitors, 115
 - concept, 57
 - conceptual distance, 245
 - concrete syntax, 113
 - ConcreteState.ftl, 274, 275, 277
 - configuration parameters, 304
 - conservative, 112
 - conservative extension, 129
 - console output, 289
 - constructor declaration, 381, 385
 - context condition, 20, 211
 - check, 213
 - implemenation, 212
 - implementation, 214
 - symbol table use, 215
 - tests, 216
 - context condition infrastructure, 212
 - context condition interface, 212
 - context conditions, 115
 - context-free grammar, 155, 211
 - context-free parser, 400
 - context-free syntax, 39
 - context-sensitive restrictions, 211
 - continuous integration, 414
 - CS-AST-compliance, 122
 - CS-conservation, 122

- CS-conservative, 129
- data structure, 39
- debug message, 287
- DecimalDoublePointLiteral, 44
- DecimalIntegerLiteral, 44
- decorator hook point, 262
- delegation pattern, 223
- DeSer
 - global scope, 184
- Deserialization, 183
- deserialization, 19
- design pattern, 221
 - bidirectional association, 230
 - list attribute, 228
 - mill, 233
 - multiple interface composition, 236
 - optional attribute, 228
 - ordinary attribute, 227
 - RealThis object composition pattern, 223
 - RealThis pattern, 223
 - RealThis pattern without common superclass, 226
 - static delegator, 221
 - template hook, 232
 - unidirectional association, 230
 - visitor, 135
- directory structure, 280
- dispatching in visitors, 137
- double dispatching, 137
- DSL, 109
- DSLs, 1
- dynamic dispatching, 137
- dynamic type, 137

- EBNF, 70, 391
- Eclipse, 7, 29
- Eclipse Modeling Framework, 90, 323
- Eclipse plugin, 7
- EMF, 90, 323, 417
- encapsulation, 109
- enclosing scope, 163, 164
- energy management, 2
- enumeration, 49, 84
 - mapped to AST, 84
- error, 300
 - by incorrect usage, 286
 - internal, 285, 286
- error handling, 285
- error management, 259
- error message, 215
- errors, 285
- exception, 300
- exp?functionname, 239
- explicit hook point, 262
- expression
 - type check, 363
 - type checking, 361
- expression problem, 137
- Expressions, 346
- expressions, 381
- extension, 52
 - AST, 81
 - AST-access-conservative, 131
 - AST-conservative, 122, 131
 - AST-modification-conservative, 131, 132
 - conservative, 122, 129
 - CS-AST-compliance, 122
 - CS-conservative, 122, 129
 - Java type errors, 133
 - multiple grammars, 124
- external nonterminal, 119
- external type parameter, 383
- Extreme Programming, 5

- fail quick, 260, 291
- feature diagram DSL, 2
- file extension
 - sym, 192
- flight control, 2
- fragment token, 42
- free modification, 122
- FreeMarker, 237, 245
 - Boolean, 239
 - Date, 239
 - Hash, 239
 - Number, 239
 - Sequence, 239
 - String, 239
 - control directives, 240

- control language, 237
- data types, 239
- drawbacks, 242
- expressions, 237, 238
- FreeMarker template language, 237
- generation gap, 233
- generator, 33
 - API, 247
 - architecture, 34
 - backend, 35
 - central part, 34
 - engine, 249
 - frontend, 34
 - setup, 250
 - workflow, 35, 36
- generator engine, 237, 245
- generator scripts, 33
- Generic invocation, 350
- generic type, 383
- generic types, 363
- generics, 46
- git, 414
- GitHub, 414
- github, 414
- GitLab, 414
- global scope, 168, 190, 199, 205, 209
 - DeSer, 184, 188
- Globalscope, 394
- Gradle, 29, 31, 303, 310, 413, 417
- gradle, 282
- Gradle build script, 311
- Gradle plugin, 311
- grammar, 10, 39, 77
 - Cardinality, 336
 - Completeness, 337
 - MCBasics, 328
 - MCCCommon, 338
 - MCHexNumbers, 333
 - StringLiterals, 329
 - UMLModifier, 336
 - UMLStereotype, 335
 - import, 116
 - 0xA4*** messages, 62
 - component, 39
 - context conditions, 62
 - deprecated, 58
 - derivates, 39
 - error messages, 62
 - nokeyword, 59
 - splittoken, 59
 - syntax, 39
 - testing, 404
- grammar component, 117
- Grammar concepts, 40
- Grammar directives, 40
- Grammar extension, 359
- grammar format, 39
- Grammar.mc4, 75
- Grammar_WithConcepts.mc4, 75
- grammars, 1
- Graphviz, 299
- Groovy, 247, 295, 304, 310, 318, 319, 321
- Groovy script, 35
- handcoded files, 281
- handcoded path, 280
- handcoding, 279
- handwritten class, 281
 - detection, 282
- handwritten code, 279
- handwritten extension, 96
- HexIntegerLiteral, 44
- hook
 - code, 266
 - string, 266
 - template, 266
 - template-string, 266
- hook method
 - in design pattern, 232
- hook point, 35, 262
 - binding, 262
 - decoration, 272
 - decorator, 262
 - explicit, 262, 268, 277
 - implicit, 262
 - name, 262
 - naming convention, 269
 - parameters, 256
 - replacing, 265, 272
 - value, 262
- how to deal with

- errors, 300
- exceptions, 300
- human brain, 2
- identifiers, 157
- implements, 51, 123
- implicit hook point, 262
- import, 112
- import statement, 382
- incremental compilation, 111
- infix, 54
 - associativity, 54
 - priority, 54
 - priority <180>, 54
- information, 286
- information message, 287
- inheritance, 120
- inheritance visitor, 147
- integration
 - handwritten and generated code, 232
- IntelliJ, 7, 31
- IntelliJ plugin, 7
- inter model resolution, 198, 201
- interface, 51
- interface method declaration, 381, 385
- interface nonterminal, 51, 79
 - with symbol, 161
- internal error, 286

- Java, 2, 7, 155
- Java classpath
 - configuring, 282
- Java Development Kit, 7
- JDK, 7
- Jenkins, 414
- JSON, 183, 184
- JUnit, 5, 217, 404

- keyword
 - temporary, 61
- kind, 157, 158
 - model entity, 157
- language
 - component, 109
 - extension, 52
- language aggregation, 110, 112, 119, 209
 - symbol tables, 204, 206
 - visitor, 144
- language component, 109
- language component feature diagram, 339
- language components, 110
- language composition, 109–111, 113
- language embedding, 110, 112
 - symbol tables, 204
 - visitor, 144, 145, 149
- language extension, 110, 112
 - symbol tables, 204
- language inheritance, 110, 112, 120, 147, 208
 - symbol tables, 204, 205
 - visitor, 144, 145
- language library, 111
- language mill, 126
- language parser, 39
- language workbench, 1
- LCD, 339
- left recursion, 47
- left-hand side, 46
- lexer, 40, 400
- Lexer productions, 40
- lexerjava, 57
- lexical production
 - action, 43
 - conversion, 43
 - result type, 43
- lexical rules, 41
- library, 111
- list attribute pattern, 228
- literal
 - type check, 363
- Literals, 340
- log file, 289
- logback, 292, 293
- logback configuration, 306, 314
- logging, 259, 306, 313
 - debug, 260, 290
 - enableFailQuick, 290
 - error, 260, 290
 - getErrorCount, 290
 - info, 260, 290
 - isFailQuickEnabled, 290
 - trace, 260, 290
 - warn, 260, 290

- severity level, 260
- logging API, 289
- logging APIs, 292
- logging component, 289
- logs, 285
- main control, 35
- make, 282
- Management Project, 414
- Maven, 303, 317, 417
- maven, 282
- Maven Project, 414
- MCBasicGenericType, 46
- MCG, 70
- MCGenericType, 46
- MCListType, 46
- MCPriimitiveType, 46
- MCType, 46
- MCTypeArgument, 46
- MCWildcardTypeArgument, 46
- message form
 - debug, 288
 - error, 288
 - info, 288
 - trace, 288
 - warn, 288
- meta-tool, 1
- method, 384
 - symbolrule, 166
- method body
 - template, 246
- method declaration, 381
- mill, 21, 91
 - initMe, 234
 - init, 235
 - parser, 234
 - composition, 91
 - handwritten extension, 97
- mill design pattern, 233
- ML_COMMENT, 44
- model, 33
- model loader, 34
- model parser, 34
- model path, 168, 190
- model processor, 33
- model transformation, 33
- Modeling in the large, 1
- modeling language, 33
- modularity, 109
- MontiCore, 1
 - adaptation, 102
 - configuration, 318
 - Continuous Integration, 414
 - design pattern, 221
 - Download, 9
 - error configuration, 285
 - features, 1
 - generator engine, 237
 - Gradle, 29, 31
 - grammar, 106
 - grammars, 327
 - libraries, 327
 - Nexus, 414
 - parameters, 305
 - Release Notes, 414
 - reports, 285
 - Repositories, 414
 - RTE, 106
 - run, 12
 - runtime environment, 106
 - Sonar, 414
 - Ticket system, 414
- multiple interface composition pattern, 236
- MyStateAttributes.ftl, 275
- MyStateGetter.ftl, 276
- Name, 44
- name, 157, 163
- name definition, 157
- name usage, 157
- named scope, 167
- NEWLINE, 44
- Nexus, 414
- node builder mill, 91
- nonterminal, 39, 50, 78
 - * mapped to AST, 78
 - + mapped to AST, 78
 - ? mapped to AST, 78
 - external, 116, 117
 - interface, 116
 - abstract, 52, 79, 87
 - cardinality, 55

- deprecated, 58
 - enumeration, 49
 - extension, 52, 117, 121
 - extension mapped to AST, 79
 - external, 119, 350
 - implements, 123
 - interface, 51, 79, 87
 - list, 88
 - mapped to AST, 78
 - optional, 88
 - override, 58
 - overriding, 117, 121
 - redefining, 121
 - token mapped to AST, 78
- nonterminals, 41
- Notational Conventions, 3
- Num_Long, 44
- OID, 295
- open source, 414
- optional attribute pattern, 228
- ordinary attribute pattern, 227
- package declaration, 382
- package name, 163
- package structure, 280
- parameter
 - groovyHook1 file.groovy, 305
 - groovyHook2 file.groovy, 305
 - cl file.xml, 305
 - configTemplate file.ftl, 305
 - ct file.ftl, 305
 - customLog file.xml, 305
 - d, 305
 - dev, 305
 - fp path-list, 305
 - g file-list, 305
 - gh1 file.groovy, 305
 - gh2 file.groovy, 305
 - grammar file-list, 305
 - grammars path-list, 305
 - h, 305
 - handcodedPath path-list, 305
 - hcp path-list, 305
 - help, 305
 - modelPath path-list, 305
 - mp path-list, 305
 - o path, 305
 - r path, 305
 - report path, 305
 - sc file.groovy, 305
 - script file.groovy, 305
 - templatePath path-list, 305
- parameterized generator, 33
- parser, 16, 39, 114
- parser generator, 102
- parserjava, 57
- Pretty printing, 38
- pretty printing, 245
- primitive types, 382
- production, 39, 46
 - cmpToken(n, st), 47
 - cmpTokenRegEx(n, a*b), 47
 - key(.), 47
 - next(0, *, none), 47
 - noSpace(n), 47
 - enumeration, 49
 - grouping (...), 47
 - lowerChar..upperChar, 47
 - name reference, 47
 - naming n:NT, 47
 - no keywords Name&, 47
 - optional ?, 47
 - recursion NT, 47
 - redefining, 121
 - repetition *, 47
 - repetition +, 47
- Productions, 40
- Project, 414
- project management, 414
- properties file, 413
- qualified name, 202
- qualified types, 382
- re-generation, 282
- RealThis
 - advantages, 225
 - class structure, 223
 - control flow, 224
 - disadvantages, 225
 - object collaboration, 224

- object structure, 224
- RealThis object composition pattern, 223
- realThis object composition pattern, 149
- RealThis pattern, 223
 - variant, 226
- RealThis variant, 226
 - advantages, 226
 - class structure, 226
 - control flow, 226
 - disadvantages, 226
 - object structure, 226
 - visitors, 226
- recursion, 46, 47
- reference
 - entity, 157
- regular expression, 41
 - (...), 41
 - lowerChar..upperChar, 41
 - negation ~, 41
 - optional ?, 41
 - repetition *, 41
 - repetition +, 41
- regular expressions, 41
- report, 286
- reporting, 257
- reports, 13, 285, 289, 294
 - hookpoints, 296
 - identifiers, 295
 - Java classes, 296
 - templates, 296
 - variables, 296
- Repositories, 414
- resolution
 - adapting symbols, 208
 - ambiguity, 195
 - bottom-up, 196
 - bottom-up intra model, 197
 - concept, 196
 - inter model, 198
 - intra model, 196, 199
 - symbol, 195
 - top-down, 199
- reuse, 109
- right-hand side, 46
- RTE, 106
- runtime environment, 106
- scannerless, 402
- scannerless parsing, 400
- scope, 15, 16, 56, 158, 167, 205, 208
 - ForStatement, 375
 - MCStatement, 374
 - artifact, 167
 - compilation unit, 167
 - global, 168
- scope graph, 159
- scope tree, 159
- Scopes, 394
- scopes, 39, 115, 205
- Scrum, 5
- SE Project, 414
- semantic action, 69
- semantic predicate, 69, 403
 - cmpToken(n, .), 69
 - cmpTokenRegEx(n, .), 69
 - next(...), 69
 - noSpace(n), 69
 - token(...), 69
- semantic predicates, 39
- Serialization, 183
- serialization, 19
- Serialization Strategy, 183
- settings file, 413
- shadowing scope, 159
- singleton, 223
- SL_COMMENT, 44
- SLF4J, 292
- SMI, 155, 159
- Sonar, 414
- SSELab, 414
- Statechart.ftl, 274, 276
- StatechartStateAttributes.ftl, 274–276
- Statements, 371
- static delegator
 - delegate object, 221
 - do method, 221
 - static host class, 221
 - static method, 221
- static delegator pattern, 91, 221, 281
- static type, 137
- String, 44
- string hook, 266
- subgrammar, 39

- sublanguage, 117
- sublanguages, 110
- suffix
 - Ext, 118
- SVN, 414
- symbol, 56, 157, 159
 - accept, 165
 - getAstNode, 165
 - access modifier, 165
 - AccessModifier, 163
 - name, 165
 - resolution, 195
 - scope, 165
 - shadowed, 158
 - usage, 177
 - visibility, 158, 167
- symbol adapter, 204, 206
- symbol kind, 158, 204
 - adaptation, 158
- symbol management infrastructure, 155
- symbol table, 16, 155, 159, 184, 212
 - List, 184
 - Optional, 184
 - String, 184
 - boolean, 184
 - defaults, 184
 - numeric defaults, 184
 - instantiation, 179
 - quick navigation, 159
 - scope, 184
 - surrogate, 159
 - to collect information, 159
- symbol tables
 - composition, 204
 - language aggregation, 206
 - language inheritance, 205
 - symbol adapter, 206
- Symbols, 352, 393
- symbols, 39, 115
- template
 - addToGlobalVar, 253
 - bindHookPoint, 253
 - changeGlobalVar, 253
 - debug, 253
 - defineGlobalVar, 253
 - defineHookPointWithDefault, 253
 - defineHookPoint, 253
 - error, 253
 - existsHookPoint, 253
 - getGlobalVar, 253
 - include2, 253
 - includeArgs, 253
 - include, 253
 - info, 253
 - requiredGlobalVars, 253
 - requiredGlobalVar, 253
 - signature, 253
 - trace, 253
 - warn, 253
 - adaptation, 262
 - API, 252
 - decoration, 272, 276
 - global variable, 260
 - hook point, 262
 - local variable, 260
 - name, 254
 - parameters, 256
 - qualified name, 254
 - replacing, 265, 272, 275
 - signature, 246, 257
- template engine, 35, 245
- template hook, 266
- template hook pattern, 232
- template method
 - in design pattern, 232
- template path, 265
- template-string hook, 266
- templates, 33
- temporary keyword, 61
- terminal, 83
 - in square brackets, 83
 - mapped to AST, 83
 - named, 83
 - optional, 88
 - semantically relevant, 48, 83, 88
- terminals, 48
- testing, 404
- text model, 4
- token
 - Digits, 345

- Digit, 345
- HexDigit, 345
- Num_Double, 345, 346
- Num_Float, 345
- Num_Int, 345
- Num_Long, 345
- OctalDigit, 345
 - composed, 60
 - splitting, 335
- tokens, 40, 400
- tool provider, 34
- tool smith, 34
- TOP mechanism, 15, 18, 96, 281
- top-down intra model resolution, 199, 201
- Trac, 414
- trace message, 287
- transformation
 - AST, 246
- traversal, 135
- Traverser, 115
- type check, 363
 - expression, 363
 - literal, 363
- type check algorithm, 362
- type checking, 361
 - expression, 361
- type expression, 362
- type inference algorithm, 362, 365, 366
- type safe, 362
- type system, 361, 362
- type variable, 363
- TypeCheck
 - IDerive, 365
 - ISynthesize, 365
- typecheck, 361
- Types, 355

- unidirectional association pattern, 230
- unique error code, 214
- usage error, 286, 287

- V-Model, 5
- variable declaration, 381
- version control, 280
- visibilities, 157
- visibility, 158, 167

- visibility scope, 159
- visit, 135
- visitor, 19
 - endVisit, 136
 - handle, 136
 - traverse, 136
 - visit, 136
 - composition, 149
 - embedding, 145
 - external, 135
 - functional, 135
 - imperative, 135
 - language embedding, 144, 145, 149
 - language inheritance, 144, 145
 - subclassing, 145
- visitor infrastructure, 136
- visitor pattern, 87, 135
- visitors, 115

- warning, 286, 287
- warnings, 285
- well-formedness, 211
- whitespaces, 400
- workflow, 35, 318
- WS, 42, 44

- XP, 5