

RWTH Aachen University
Software Engineering Group

MontiCore 5 Language Workbench

Edition 2017



Bernhard Rumpe,
Katrin Hölldobler

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

Aachener Informatik-Berichte,
Software Engineering

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Band 32



[HR17] K. Hölldobler, B. Rumpe:
MontiCore 5 Language Workbench Edition 2017.
Shaker Verlag, ISBN 978-3-8440-5713-3. Aachener Informatik-Berichte, Software Engineering, Band 32. December 2017.
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 its 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]) demonstrating that the conglomerate of languages that the UML is made of can be substantiated with useful transformation, refinement and refactoring techniques. 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, hard coded modeling tools were not helpful in realizing these techniques. This was the original motivation for MontiCore that can also be found in the foundational theses in [Kra10, Völ11].

Later, it became apparent that UML will be complemented by several 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, 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 Tutorial and Reference Manual the core facilities of MontiCore are described. Extensions are available through various projects either using or enhancing MontiCore. We have a sophisticated technique to generate transformation languages and their transformation engines based on DSLs [HRW15, AHRW17, RRW15, HHRW15, Wei12], we have explored tagging languages [Loo17, MRRW16, GLRR15], various forms of the UML and its derivatives [Sch12, Wor16, Hab16, Rei16, Rot17] and plenty specific languages.

Despite MontiCore is an academic tool to explore modeling and meta-modeling techniques, after 14 years of development, it has reached an extraordinary strength and is thus increasingly used in industrial projects, like energy management [Pin14], 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 comfort, e.g., graphical editing, but advanced functionality for model-based analysis or synthesis of software intensive systems as well as 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, Professor Dr. Christian Berger, Vincent Bertram, Arvid Butting, Anabel Derlam, Imke Drave, Robert Eikermann, Timo Greifenberg, Dr. Hans Grönniger, Dr. Tim Gülke, Dr. Arne Haber, Guido Hansen, Olga Haubrich, Lars Hermerschmidt, Dr. Christoph Herrmann, Gabi Heuschen, Steffen Hillemacher, Oliver Kautz, Carsten Kolassa, Dr. Anne-Therese Körtgen, Thomas Kurpick, Evgeny Kusmenko, Dr. Holger Krahn, Dr. Stefan Kriebel, Achim Lindt, Dr. Markus Look, Professor Dr. Shahar Maoz, Matthias Markthaler, Dr. Klaus Müller, Dr. Pedram Mir Seyed Nazari, Antonio Navarro Pérez, Nina Pichler, Dr. Claas Pinkernell, Dr. Dimitri Plotnikov, Deni Raco, Dr. Holger Rendel, Dr. Dirk Reiss, Dr. Daniel Retkowitz, Dr. Alexander Roth, Dr. Martin Schindler, David Schmalzing, Steffi Schrader, Dr. Frank Schroven, Christoph Schulze, Igor Shumeiko, Brian Sinkovec, Simon Varga, Dr. Steven Völkel, Dr. Ingo Weisemöller, Michael von Wenckstern, and 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, as well as to Sylvia Gunder, who ensures 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, who partially described their contribution directly in this reference manual.

In addition, we would like to thank all reviewer for their intense reviews of certain chapters (chapters are mentioned in brackets): Kai Adam (Ch. 1,9,14,19), Vincent Bertram (Ch. 15,17), Arvid Butting (Ch. 7,9), Imke Drave (Ch. 1,2,3,4,12,16), Robert Eikermann (Ch. 7,8,9,10), Timo Greifenberg (Ch. 3), Guido Hansen (Ch. 4,6,13), Olga Haubrich (Ch. 2,17), Steffen Hillemacher (Ch. 5,12), Oliver Kautz (Ch. 3,6,9,11), Evgeny Kusmenko (Ch. 7), Achim Lindt (Ch. 2,8), Nina Pichler (Ch. 18), Deni Raco (Ch. 7,10), David Schmalzing (Ch. 3,6,11), Christoph Schulze (Ch. 15,19), Igor Shumeiko (Ch. 5,12), Brian Sinkovec (Ch. 9,18), Simon Varga (Ch. 4,6,13), Michael von Wenckstern (Ch. 3,13,14) and Dr. Andreas Wortmann (Ch. 7,9).

To all readers of this manual: 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, 23.12.2017

Katrin Hölldobler, Bernhard Rumpe

¹www.se-rwth.de/publications/

Contents

1	Introduction	1
1.1	MontiCore Language Workbench	2
1.2	Notational Conventions	3
1.3	Textual Modeling	4
1.4	Methodical Considerations: Agile Modelling	5
2	Getting Started	7
2.1	Prerequisites	7
2.2	How to Install MontiCore for Command Line	8
2.2.1	Installation	8
2.2.2	Run MontiCore	8
2.2.3	Compile and Run the Target	9
2.2.4	MontiCore Commandline Arguments	10
2.3	Using MontiCore in Eclipse	11
2.3.1	Setting up Eclipse	11
2.3.2	Importing the Example	12
2.3.3	Running MontiCore	12
2.4	Using MontiCore in IntelliJ IDEA	13
2.4.1	Setting up IntelliJ IDEA	13
2.4.2	Importing the Example	14
2.4.3	Running MontiCore	14
3	Architecture of a Model Processor	17
3.1	Structure of a Model Processor - External View	17
3.2	Internal Architecture of a Generator	18
3.3	Tool Workflow On The Top Level	19
4	MontiCore Grammar for Language and AST Definitions	23
4.1	Lexical Tokens	24
4.1.1	Definition of Tokens using Regular Expressions	25
4.1.2	Actions to Process a Token	26
4.1.3	Predefined Tokens	27
4.2	Productions	29
4.2.1	Terminals	31
4.2.2	Enumeration	31
4.2.3	Nonterminals	32
4.2.4	Interface Nonterminals	32
4.2.5	Extending Nonterminals	33
4.2.6	Abstract Nonterminals	34

4.2.7	Starting Nonterminal	35
4.2.8	Infix Operations and Priorities	35
4.2.9	Restricting the Cardinality of a Nonterminal	37
4.2.10	Symbols and Scopes	37
4.2.11	Passing Code to the ANTLR	38
4.3	Context Conditions for the MCG Language	39
4.4	Semantic Predicates and Actions	44
4.5	Editor assistance for the MCG Language	44
4.6	EBNF for the MCG Language	44
5	Abstract Syntax Tree	51
5.1	Mapping Nonterminals to the AST	51
5.2	Interface and Abstract Nonterminals	52
5.3	Extending Nonterminals	54
5.4	Extending the Abstract Syntax Implementation	55
5.5	Terminals in the AST	57
5.6	Enumerations	58
5.7	ASTNode: A Base Interface for AST Classes	58
5.8	Generated ASTNode Subclasses	61
5.9	Node Construction Using the Node Builder Mill	66
5.10	Handwritten Extension of AST Classes and Node Builders	70
5.10.1	Handwritten Extension of AST Classes	70
5.10.2	Handwritten Extension of AST Builders and Mills	72
6	Parser Generation and Use	75
6.1	Generating a Parser and a Lexer	75
6.2	Interface of the Generated Parser Classes	78
6.3	Executing a Generated Parser	79
7	Language Composition	81
7.1	Introduction to Language Composition	81
7.2	Language Composition at a Glance	84
7.3	Grammar Constructs for Language Composition	87
7.3.1	Component Grammar	88
7.3.2	External Nonterminals	88
7.3.3	Importing and Extending Grammars	90
7.4	Language Inheritance	90
7.4.1	Redefining Productions of Grammars	91
7.4.2	Extending Multiple Grammars	93
7.5	Language Embedding	94
7.6	Composing the Builder Infrastructure	95
7.7	Composing Parsers	96
7.8	Conservative Extension	97
7.8.1	Conservative Extension of the Concrete Syntax	98
7.8.2	Access-Conservative Extension of the Abstract Syntax	99
7.8.3	Modification-Conservative Extension of the Abstract Syntax	100

7.8.4	AST Signatures Causing Java Type Errors	101
8	Visitors for AST Traversal	103
8.1	Visitor Interface of a Language	103
8.2	Predefined Visitor Variants	107
8.2.1	Inheritance Visitor	107
8.2.2	Delegator Visitor	109
8.2.3	Parent Aware Visitor	109
8.3	Visitors for Composed Languages	110
8.3.1	Visitor for Language Inheritance and Extension	110
8.3.2	Visitor for Language Inheritance – an Alternative	113
8.3.3	Visitors for Compositional Language Embedding	114
9	Symbol Management Infrastructure	121
9.1	Introduction to Symbol Table Concepts	121
9.2	Defining Symbols	123
9.2.1	Generated Classes For Symbols	124
9.2.2	RTE Classes For Symbols	125
9.3	Defining Scopes	127
9.3.1	Generated Classes For Scopes	127
9.3.2	RTE Classes For Scopes	128
9.3.3	Artifact Scope and Global Scope	129
9.4	Collaboration between AST, Symbol, and Scope	130
9.5	Using Symbols	130
9.6	Instantiating Symbol Tables	131
9.6.1	RTE Classes For Symbol Table Building	132
9.6.2	Generated Classes For Symbol Table Building	132
10	Context Conditions	133
10.1	Context Condition Infrastructure	134
10.2	Implementation of Context Conditions	137
10.3	Testing Context Conditions	139
10.3.1	Testing a Context Condition on a Valid Model	139
10.3.2	Testing a Context Condition on an Invalid Model	140
11	Design Pattern Used and Invented for MontiCore	143
11.1	Static Delegator Design Pattern	143
11.2	RealThis Object Composition Pattern	145
11.3	Template Hook Pattern	147
12	FreeMarker	149
12.1	The FreeMarker Template Languages	149
12.2	Expressions in FreeMarker	151
12.3	Control Directives in FreeMarker	152
12.4	FreeMarker Drawbacks	154

13 Generator Engine using Flexible Templates	155
13.1 Methodical Considerations	155
13.2 Generator API	157
13.3 Configuring the Generation Process	159
13.4 MontiCore APIs for Templates	162
13.4.1 Shortcuts: Aliases in Templates	162
13.4.2 The Template Controller	163
13.4.3 Logging within a Template	169
13.4.4 Variables in the Templates with GlobalExtensionManagement	169
13.5 Hook Points for Adaptation	171
13.5.1 The Concept of Hook Points	171
13.5.2 Forms of Hook Points	174
13.5.3 Defining Explicit Hook Points in Templates	177
13.5.4 Binding Hook Points	178
13.5.5 Replacing and Decorating Hook Points	179
13.5.6 HookPoint Replacement and Decoration Strategy	180
14 Integrating Handwritten Code	183
14.1 Integration of Handwritten Code	183
14.2 Adaptation of Generated Code by Subclassing	184
14.3 Adaptation of Generated Code using the TOP Mechanism	185
15 Error Handling, Logging and Reporting	189
15.1 Where to find Concrete Help for an Error, Warning, or other Message . . .	189
15.2 Errors, Warnings and Log Messages	189
15.2.1 Errors	190
15.2.2 Warnings and Information	191
15.2.3 Form of Errors, Warnings and Log Messages	191
15.3 The Error and Logging Component	193
15.4 Logging Configurations in MontiCore	195
15.4.1 Selecting one of the given Configurations	196
15.4.2 Using a Custom logback Configuration	197
15.4.3 Initializing the Log within Java	197
15.4.4 Providing a Custom Log Implementation	197
15.5 Reports	198
15.5.1 Where to Find Reports	198
15.5.2 How to Configure Reporting	198
15.5.3 Identifiers contained in the Reports	199
15.5.4 List of the Reports	201
15.6 For Developers: How to Deal with Errors and Warnings	204
16 MontiCore Configuration with Groovy	205
16.1 Generate the Standard Classes	206
16.2 MontiCore Base Class for Groovy Scripts	208
16.2.1 Methods Available within Groovy Scripts	209
16.2.2 Variables Available within Groovy Scripts	210

16.2.3 Available preimported Classes within Groovy Scripts	211
17 Some MontiCore Grammars Explained	213
17.1 Component Grammar MCBasics.mc4	213
17.2 Component Grammar StringLiterals.mc4	215
17.3 Component Grammars for Numbers	217
17.3.1 Component Grammar MCNumbers.mc4	217
17.3.2 Component Grammar MCHexNumbers.mc4	219
17.4 Component Grammar MCBasicTypes1.mc4	220
17.5 Component Grammars for UML Languages	221
17.5.1 Component Grammar UMLSterotype.mc4	221
17.5.2 Component Grammar Cardinality.mc4	222
17.5.3 Component Grammar Completeness.mc4	223
17.5.4 Component Grammar UMLModifier.mc4	223
17.6 Component Grammar MCExpressions.mc4	224
18 Some Example Languages	229
18.1 A Simple Automaton Language	229
18.2 Hierarchical Automaton	235
18.3 A Language for Automata with Invariants	236
18.4 Scannerless Parsing to Handle Complex Tokens	238
18.4.1 Parsing with Whitespaces	239
18.4.2 Temporarily Parsing with Whitespaces	240
18.4.3 Preventing Whitespaces between Tokens	241
18.5 Tip: Testing Grammars and their Models	242
18.6 Questionnaire Language	244
19 Developer's View on MontiCore	247
19.1 MontiCore's GitHub Repository	248
19.1.1 External Developers and Forks	249
19.1.2 MontiCore's Maven Projects	250
19.2 Other Source Code Locations	251
20 Further Reading and Related Work from the SE Group, RWTH Aachen	253
20.1 Agile Model Based Software Engineering	253
20.2 Generative Software Engineering	253
20.3 Unified Modeling Language (UML)	254
20.4 Domain Specific Languages (DSLs)	254
20.5 Software Language Engineering	254
20.6 Modeling Software Architecture & the MontiArc Tool	255
20.7 Compositionality & Modularity of Models	255
20.8 Semantics of Modeling Languages	256
20.9 Evolution & Transformation of Models	256
20.10 Variability & Software Product Lines (SPL)	256
20.11 Cyber-Physical Systems (CPS)	257
20.12 State Based Modeling (Automata)	257
20.13 Robotics	258

Contents

20.14 Automotive, Autonomic Driving & Driver Assistance	258
20.15 Energy Management	258
20.16 Cloud Computing & Enterprise Information Systems	259
List of Figures	261
Listings	265
References	269
Index	283

Chapter 1

Introduction

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

The tools are partially generated by the *language workbench MontiCore* and partially need handcoded extensions. This reference manual explains how to do this.

However, 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: 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 need 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.



Tip 1.1 Where to find MontiCore

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

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

MontiCore also includes a number of plug-ins for Eclipse and thus supposedly has a rather comfortable 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 closed product out-of-the-box for the generation of software as well as 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.

At a glimpse, the features of MontiCore are:

- Modular definition of languages and language fragments
- 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 definition of concrete syntax, abstract syntax, parser and internal representation of models.
- Easy definition of 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

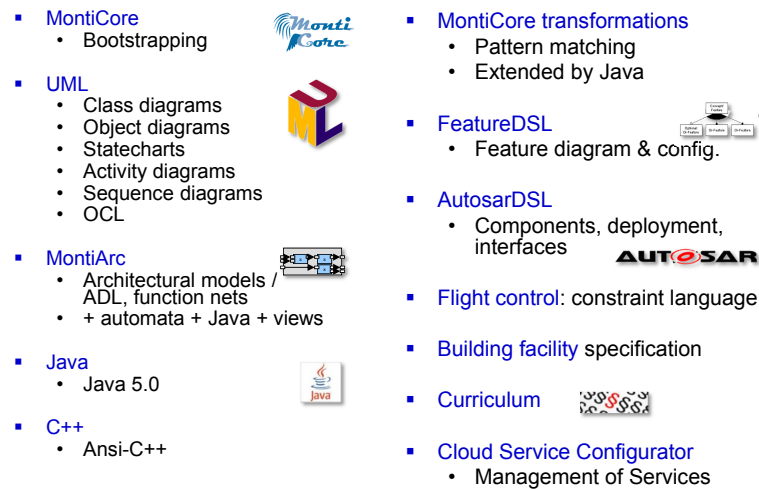


Figure 1.2: Some languages MontiCore provides

- Language definition
- How an abstract syntax (AST) looks like
- Managing symbols and visibility of definitions
- Model composition
- Language components and their composition
- Navigation of the AST with 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 Java class called `Person`.

```

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

```

Java «gen» Person

Listing 1.3: Example in Java

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 to exhibit concrete situations in the tool or product, such as for example the extension of a generated class GroupImpl by handcoded class GroupEIMP, which are both present in the product which is the final target of the development process. We show this like in Figure 1.4

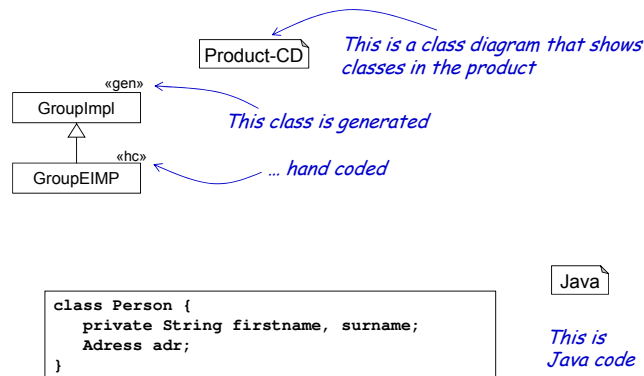


Figure 1.4: Notational conventions

1.3 Textual Modeling

Many experts think that the mental model in the conscious human brain is the most important form of model. 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 reference manual.

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

Diagrams and text will coexist in the future and may be even closely integrated. MontiCore 4 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 Modelling

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 reference manual 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.

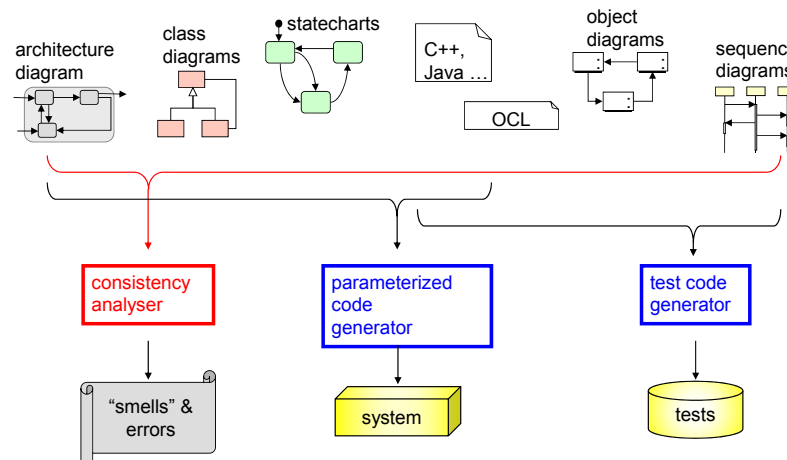


Figure 1.5: Agile use of models for coding and testing

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

1. Introduction

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

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

There are three options to start MontiCore. All come with a larger set of configuration options. For a quick peek as well as for an automatic execution, e.g., within nightly builds, the command line version should be tried out, for example an exemplary automaton DSL. MontiCore also supports different environments:

- MontiCore as command line tool
- MontiCore as Eclipse plugin
- MontiCore as IntelliJ plugin

In the following all three variants are shortly explained together with the most important configuration options. However, a potentially newer explanation can be found at

Codeschnipsel

```
www.monticore.de  
www.monticore.de/gettingstarted/
```

2.1 Prerequisites

- Install the latest JDK (at least version 8 required) provided by Oracle¹.
- 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` variable is set such that the Java compiler is available. 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` .

¹<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

2. Getting Started

Now we have the prerequisites to run MontiCore from the command line. MontiCore as plugin, however, needs some more components installed, as you can see below.

2.2 How to Install MontiCore for Command Line

In a nutshell do the following first steps to use MontiCore as a command line tool:

Prerequisites Install the Java Development Kit and further preparation steps as described in Section 2.1.

Installation Download the MontiCore distribution file, unzip it, and change to the extracted directory as described in Section 2.2.1.

Running MontiCore Execute MontiCore on the provided language definition `Automaton.mc4` as described in Section 2.2.2.

Compiling the Product Compile all the generated and supplied handwritten Java source files as described in Section 2.2.3.

Running the Product Execute the `automaton` tool on an example model `example/PingPong.aut` as described in Section 2.2.3.

2.2.1 Installation

1. Download the MontiCore zip distribution file².
2. Unzip the distribution. It will unzip a directory called `mc-workspace` containing the executable MontiCore CLI (short for command line interface) JAR along with a directory `src` containing handwritten automaton DSL infrastructure, a directory `hwc` containing handwritten code that will be incorporated into the generated code, and a directory `example` containing an example automaton model.

2.2.2 Run MontiCore

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

shell

```
1 java -jar monticore-cli.jar Automaton.mc4 -hcp hwc/
```

The only required argument `Automaton.mc4` denotes the input grammar for MontiCore to process and generate the language infrastructure. The second argument

²<http://www.monticore.de/gettingstarted/>

denotes the path to look for handwritten code that is to be incorporated into the generated infrastructure.

MontiCore will be launched and the following steps will be executed:

- a) The specified grammar will be parsed and processed by MontiCore.
- b) Java source files for the corresponding DSL infrastructure will be generated into the default output directory `out`. This infrastructure consists of:
 - `out/automaton/_ast` containing the abstract syntax representation of the automaton DSL (cf. Chapter 5).
 - `out/automaton/_cocos` containing infrastructure for context conditions of the automaton DSL (cf. Chapter 10).
 - `out/automaton/_od` containing infrastructure for printing object diagrams of the automaton DSL.
 - `out/automaton/_parser` containing the generated parsers which are based on ANTLR (cf. Chapter 6).
 - `out/automaton/_symboltable` containing infrastructure for the symbol table of the automaton DSL (cf. Chapter 6).
 - `out/automaton/_visitor` containing infrastructure for visitors of the automaton DSL (cf. Chapter 9).
 - `out/reports/Automaton` containing reports created during the processing of the automaton grammar.
- c) The output directory will also contain a log file of the executed generation process `monticore.YYYY-MM-DD-HH:mm:ss.log` with the generation time in its name.

2.2.3 Compile and Run the Target

1. Compiling the automaton DSL

- Execute the command:

shell

```
1 javac -cp monticore-cli.jar -sourcepath "src/;out/;hwc/" \
2      src/automaton/AutomatonTool.java
```

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

This will compile all generated classes located in `out` and all handwritten classes located in `src` and `hwc`. Please note that the structure of the handwritten classes follows (though not necessarily) the package layout of the generated code, i.e. there are the following sub directories (Java packages):

2. Getting Started

- `src/automaton` contains the top level language realization for using the generated DSL infrastructure. In this case the class `src/automaton/AutomatonTool.java` constitutes a main class executable for processing automaton models with the automaton DSL (inspect the class and see below for how to execute it).
 - `src/automaton/cocos` contains infrastructure for context condition of the automaton DSL.
 - `src/automaton/prettyprint` contains an exemplary use of the generated visitor infrastructure for processing the parsed model. Here: for pretty printing.
 - `src/automaton/visitors` contains an exemplary analysis using the visitor infrastructure. The exemplary analysis counts the states contained in the parsed automaton model.
 - `hwc/automaton/_ast` contains an exemplary usage of the handwritten code integration mechanism for modifying the AST for the automaton DSL.
 - `hwc/automaton/_symboltable` contains handwritten extensions of the generated symbol table infrastructure.
- Running the automaton DSL tool
 - Execute the command:

`shell`

```
1 java -cp "src/;out/;hwc/;monticore-cli.jar" \  
2     automaton.AutomatonTool example/PingPong.aut
```

Please note again the Unix paths separation with colons.

This will run the automaton DSL tool. The argument `example/PingPong.aut` is passed to the automaton DSL tool as input file. Examine the output on the command line which shows the processing of the example automaton model.

A simple Experiment

The shipped example automaton DSL (all sources contained in `mc-workspace/src` and `mc-workspace/hwc`) can be used as a starting point. 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.

2.2.4 MontiCore Commandline Arguments

The MontiCore generator can be adapted in various ways. For that purpose, MontiCore accepts a number of parameters:

- grammar1 grammar2 ...** the input grammar(s), like `Automaton.mc4`
- o path** Optional output directory for all generated code; defaults to `out`
- mp path1 path2 ...** Optional list of directories or files to be included for reference resolution (equivalent: `-modelPath`)
- hcp path1 path2 ...** Optional list of directories to look for handwritten code to integrate (equivalent: `-handcodedPath`)
- s file.groovy** Optional Groovy script to control the generation workflow (equivalent: `-script`). For further explanation on how to develop a custom Groovy script see Chapter 16.
- g path1 path2** handle all grammars found in that path (equivalent: `-grammars`)
- fp path1 path2** Optional list of directories to look for handwritten templates to integrate (equivalent: `-templatePath`)
- f** Optional parameter that specifies whether the code generation should be enforced, i.e. disable incremental code generation (default is false; equivalent: `-force`)
- d** Optional parameter that specifies whether developer level logging should be used (default is false; equivalent: `-dev`). This option selects a different predefined Groovy script leading to a different Log configuration.
- cl file.xml** Optional logback configuration file to customize the logger, e.g. log level, message format (equivalent: `-customLog`). An explanation on how to write such a logback.xml is available here: <https://logback.qos.ch/manual/>
- h help** (equivalent: `-help`)

When composing grammars, dependent grammars need to be located in the model path using the `-mp` option. The `-g` option is used, when you want to generate code from all grammars that can be found in that directory.

Only the most important parameters can be directly given by arguments. For a deeper control of the generation workflow, including various configuration options, it is possible to use a Groovy script. For example, Groovy can be used to adapt the generation process in such a form, that the generated classes are EMF compatible and then fully integrated with any EMF tool (cf. Chapter 16).

2.3 Using MontiCore in Eclipse

It is possible to use MontiCore as a Maven plugin. For getting started with MontiCore as a Maven plugin using Eclipse, the following is to do:

2.3.1 Setting up Eclipse

Before importing the example project and executing MontiCore as a Maven plugin, please make sure Eclipse is configured properly or configure it as follows:

2. Getting Started

1. Download and install Eclipse (or use an existing one)
2. Open Eclipse
3. Install needed Plugins
 - Help > Install new Software
 - Make sure the M2E (Maven 2 Eclipse) Plugin³ is installed
 - Install the following MontiCore M2E Extension⁴
 - Install MontiCore 4 (or higher) extension
 - Install SE Groovy extension
 - For the Compiler:
 - Install the M2E connector for the Eclipse JDT Compiler⁵
4. Make sure to configure Eclipse to use a JDK instead of an JRE
 - Window > Preferences > Java > Installed JREs

2.3.2 Importing the Example

The shipped example automaton DSL⁶ 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 2.3.3. To import the example do the following:

1. Download and unzip the Automaton example⁶
2. Open Eclipse and select
 - File > Import > Maven > Existing Maven Projects > next
 - Click on the *Browse..* button and use the folder holding the pom.xml from the automaton example.

2.3.3 Running MontiCore

To execute the MontiCore Maven plugin and thus process a given grammar and generate the Java code for the DSL infrastructure, do the following:

- Right click on the project
- Run as > Maven install

³<http://download.eclipse.org/technology/m2e/releases>

⁴ <https://nexus.se.rwth-aachen.de/update-site/>

⁵ <http://download.jboss.org/jbosstools/updates/m2e-extensions/m2e-jdt-compiler/>

⁶<http://www.monticore.de/gettingstarted/Automaton-4.5.3.zip>

MontiCore will be launched and the following steps will be executed:

1. The grammar specified in the pom.xml will be parsed and processed by MontiCore.
2. Java source files for the corresponding DSL infrastructure will be generated into the default output directory `../target/generated-sources/monticore/sourcecode`. This infrastructure consists of:
 - `/automaton/_ast` containing the abstract syntax representation of the automaton DSL (cf. Chapter 5).
 - `/automaton/_cocos` containing infrastructure for context conditions of the automaton DSL (cf. Chapter 10).
 - `/automaton/_od` containing infrastructure for printing object diagrams of the automaton DSL.
 - `/automaton/_parser` containing the generated parsers which are based on ANTLR (cf. Chapter 6).
 - `/automaton/_symboltable` containing infrastructure for the symbol table of the automaton DSL (cf. Chapter 6).
 - `/automaton/_visitor` containing infrastructure for visitors of the automaton DSL (cf. Chapter 9).
 - `/reports/Automaton` containing reports created during the processing of the automaton grammar.
3. The output directory will also contain a log file of the executed generation process `monticore.YYYY-MM-DD-HH:mm:ss.log`.

After installing and executing MontiCore in Eclipse, your workspace should look similar to Figure 2.1.

2.4 Using MontiCore in IntelliJ IDEA

For getting started with MontiCore as a Maven plugin using IntelliJ do the following:

2.4.1 Setting up IntelliJ IDEA

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

2. Getting Started

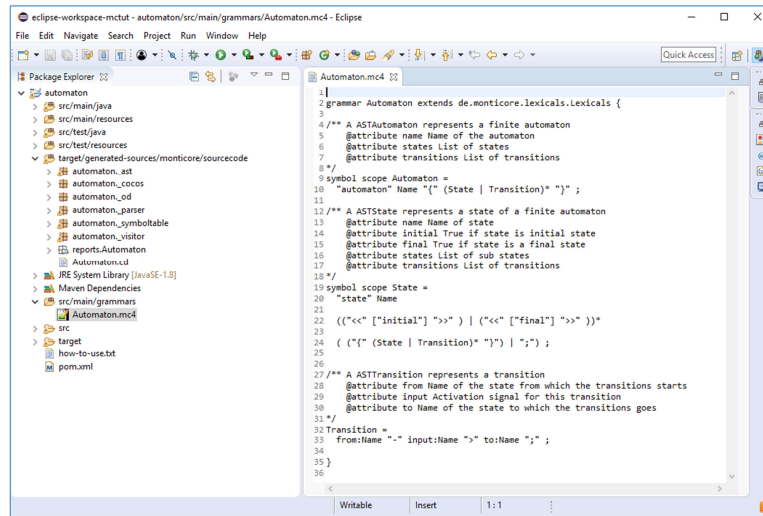


Figure 2.1: Eclipse after importing the example project and executing MontiCore

2.4.2 Importing the Example

The shipped example automaton DSL⁶ can be used as a starting point. Once imported into IntelliJ, 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 do the following:

1. Download and unzip the Automaton Example⁶
2. In the IDE select: File > Open
3. Select the folder holding the pom.xml

2.4.3 Running MontiCore

To execute the MontiCore Maven plugin and thus process a given grammar and generate the Java code for the DSL infrastructure, do the following:

From the Maven Projects menu on the right select Automaton > Lifecycle > install (double click). MontiCore will be launched and the following steps will be executed:

1. The grammar specified in the pom.xml will be parsed and processed by MontiCore.
2. Java source files for the corresponding DSL infrastructure will be generated into the default output directory `../target/generated-sources/monticore/sourcecode`. This infrastructure consists of:
 - `/automaton/_ast` containing the abstract syntax representation of the automaton DSL (cf. Chapter 5).

- /automaton/_cocos containing infrastructure for context conditions of the automaton DSL (cf. Chapter 10).
 - /automaton/_od containing infrastructure for printing object diagrams of the automaton DSL.
 - /automaton/_parser containing the generated parsers which are based on ANTLR (cf. Chapter 6).
 - /automaton/_symboltable containing infrastructure for the symbol table of the automaton DSL (cf. Chapter 6).
 - /automaton/_visitor containing infrastructure for visitors of the automaton DSL (cf. Chapter 9).
 - /reports/Automaton containing reports created during the processing of the automaton grammar.
3. The output directory will also contain a log file of the executed generation process `monticore.YYYY-MM-DD-HH:mm:ss.log`.

After installing and executing MontiCore in IntelliJ IDEA, your workspace should look similar to Figure 2.2.

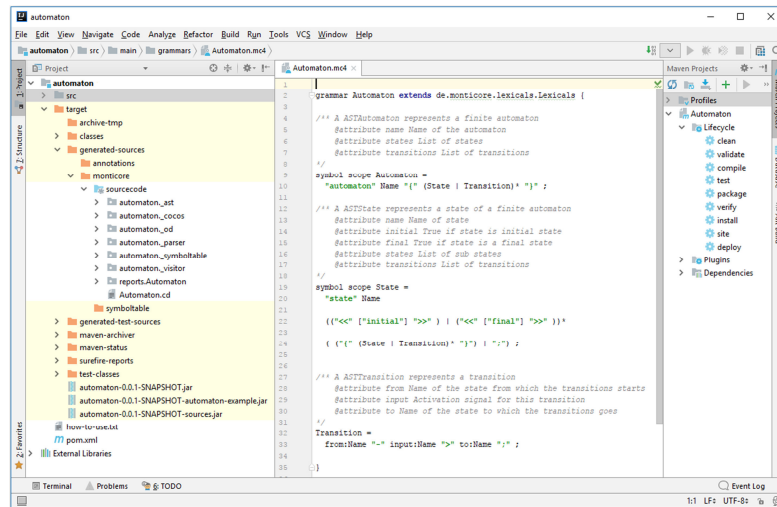


Figure 2.2: IntelliJ IDEA after importing the example project and executing MontiCore

This chapter explained the technical setup for using MontiCore. After installing MontiCore as described above it can be used to develop new modeling languages and generators as described in subsequent chapters. One possibility is of course to start with one of the provided examples and modify it for your own purposes.

Chapter 3

Architecture of a Model Processor

This chapter overviews the standard architecture of MontiCore tool. It shows how the architecture structures the features and capabilities such a tool offers. This is a prerequisite to understand the usage of model-based tool processors, as well as how to define and adapt the processing.

It is notable that MontiCore as well as its components and derivatives cannot only be used for generation, but also for deep analysis, transformation in general or also for interpretation at runtime, but in the following, we concentrate on the widely used generative aspect.

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

3.1 Structure of a Model Processor - External View

There are many forms of model processing. Each first step processes one or more models, then applies some internal transformations and finally produces artifacts. Often, these are partial or even completely executable programs written in a GPL. It is also possible to produce models of another language, documentation, proof obligations to be handled by a model checker or a verifier, websites in html, or overview drawings.

The generated code typically uses some existing frameworks, the operating system and other platform specific code. However, generators need high flexibility and smartness enable incorporation of handwritten code, platform specific adaptations of the generated code, predefined components, and possibly project or even user specific preferences.

Classical compilers embed concepts to manage this within a programming language through imports of external frameworks, compiler pragmas or a macro preprocessor. This is in contrast to model-based generation where the model typically only carries the domain knowledge, while the *parameterized generator* adds technical details and allows to fill adaptation points in form of *generator scripts* and *templates*.

Figure 3.1 shows the external view on such a generator and the artifacts used and produced. Different persons may adopt different roles providing and using parts of the artifacts that occur in a model-based generative project.

3. Architecture of a Model Processor

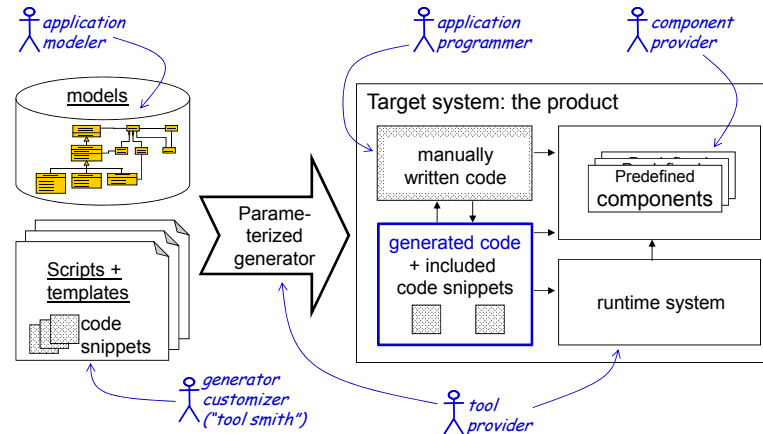


Figure 3.1: Structure of a generator - external view

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

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

3.2 Internal Architecture of a Generator

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

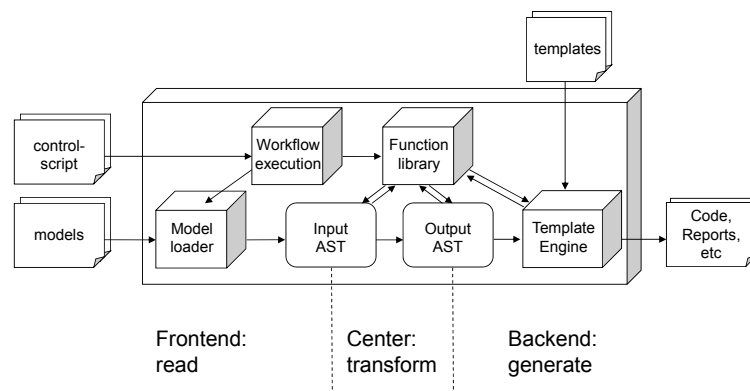


Figure 3.2: Internal architecture of a generator

A *model loader* handles the loading of all needed models and their transformation in an internal 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 data base 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. The second part of the *frontend* contains a library of data structures and functions to check the context conditions on the input models, or load further needed models, to ensure the resulting input AST is well formed.

The *central* part of a generator *transforms* the input AST into an output AST. That may be a rather complex transformational mapping from one kind of models into another kind, or a relatively simple augmentation by additional information. MontiCore provides the capability to translate between different kinds of ASTs including, for example, the translation from any source language to Java. MontiCore also supports to attach specific templates to AST nodes, such that parts of the generator intelligence can be deferred to the templates while selection of the appropriate templates is part of the augmentation of the output AST and decided in the transformation part.

The *backend* of a generator fokusses 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 as well as project-specific templates, which describe the concrete shape of the resulting artifacts. This process is highly configurable and adaptive, e.g. using a *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 that controls the transformations and generation as well as a larger function library that facilitates to build symbol tables, arrange transformations, set specific template configurations, etc.

Figure 3.3 overviews the chapters containing the appropriate information about how to develop each component for your own model parser.

3.3 Tool Workflow On The Top Level

Listing 3.4 shows a Java main method that puts the different modules of a DSL tool in a linear workflow. It defines the tool's main control structure and serves as blueprint for other DSL processing tools. This example uses the Automaton DSL which is also used in larger parts of this report. Details, e.g. on the methods used, can be found in the corresponding chapters respectively the complete example in the MontiCore project.

Line 4 identifies the file of an automaton model from the arguments. In lines 7-10 the parser AutomatonParser (s. Chapter 6) is used to parse the automaton model. Based

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

3. Architecture of a Model Processor

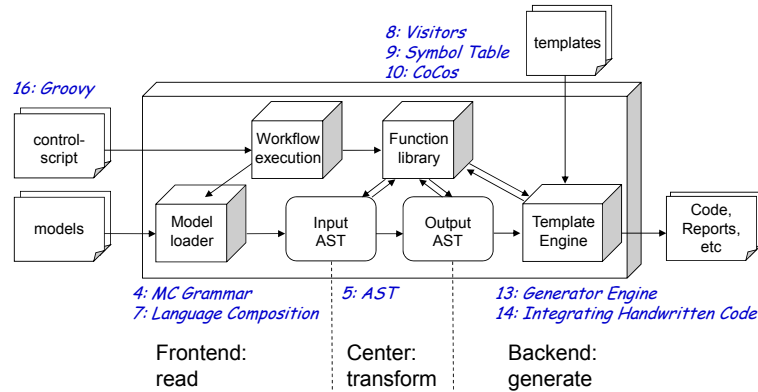


Figure 3.3: Chapter structure of the reference manual

on the resulting AST, the symbol table (s. Chapter 9) is created (l. 13). Lines 16-22 show an example resolution of a state symbol.

Since the parser can only identify *context-free* parsing errors (s. 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 `AutomatonCoCoChecker` object is created, which can be configured with concrete context conditions (ll. 25-30). The `checkAll` method checks all registered context conditions (l. 33). Next, the model is analyzed. Visitors provide a decent infrastructure to traverse the AST and operate on it (s. Chapter 8). Here, the number of states is calculated (l. 38).

Finally, the model is pretty printed (l. 42). It uses MontiCore’s generator engine (cf. Chapter 13). Pretty printing serves several purposes: On the one hand, the resulting model may be easier maintainable or simply stored as documentation. On the other hand, the pretty printing process helps to check, whether the parsing and AST construction was complete and correct. Usually, but not in this example, an executable implementation of the automaton 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 the corresponding chapters.


```

1 public class AutomatonTool {
2     public static void main(String[] args) {
3         // Retrieve the model name
4         String model = args[0];
5
6         // setup the language infrastructure
7         AutomatonLanguage lang = new AutomatonLanguage();
8
9         // parse the model and create the AST representation
10        ASTAutomaton ast = parse(model);
11
12        // setup the symbol table
13        Scope modelTopScope = createSymbolTable(lang, ast);
14
15        // can be used for resolving names in the model
16        Optional<Symbol> aSymbol =
17            modelTopScope.resolve("Ping", StateSymbol.KIND);
18        if (aSymbol.isPresent()) {
19            Log.info("Resolved state symbol \"Ping\"; FQN = "
20                + aSymbol.get().toString(),
21                AutomatonTool.class.getName());
22        }
23
24        // setup context condition infrastructure
25        AutomatonCoCoChecker checker = new AutomatonCoCoChecker();
26
27        // add a custom set of context conditions
28        checker.addCoCo(new StateNameStartsWithCapitalLetter());
29        checker.addCoCo(new AtLeastOneInitialAndFinalState());
30        checker.addCoCo(new TransitionSourceExists());
31
32        // check the CoCos
33        checker.checkAll(ast);
34
35        // Now we know the model is well-formed
36
37        // analyze the model with a visitor
38        CountStates cs = new CountStates();
39        cs.handle(ast);
40
41        // execute a pretty printer
42        PrettyPrinter pp = new PrettyPrinter();
43        pp.handle(ast);
44        System.out.println(pp.getResult());
45    }
46 }

```

Listing 3.4: Example tool for the Automaton DSL

Chapter 4

MontiCore Grammar for Language and AST Definitions

In MontiCore, grammars are the central notation from where 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 explains the MontiCore grammar format. It discusses productions defining the different types of nonterminals and their relations, context conditions that define the well-formedness of a grammar, and additional concepts that allow to further configure the processing of or code generation from MontiCore grammars.

MontiCore grammars describe the *context-free syntax* of languages in a notation based on the Extended Backus Naur Form (EBNF, [ASU86]) and the ANTLR tool [Par13]. In addition, a MontiCore grammar provides convenient constructs for the specification of commonly used *options* and additional *context-sensitive* concepts of languages. As MontiCore creates Adaptive LL(*) parsers, it also supports *semantic predicates*, which make it possible to describe common context free parsable languages. However, MontiCore's main purpose of 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, from a grammar MontiCore derives

- the *parser* for the concrete syntax (cf. Chapter 6),
- the *data structure* for the abstract syntax (AST, cf. Chapter 5),
- the transformation filling the AST when parsing a model, and
- infrastructure to manage *symbols* and *scopes*.

The major part of the frontend of a generator tool is generated using a MontiCore grammar. Furthermore, MontiCore provides *inheritance*, *extension* and *overriding* mechanisms for productions allowing an improved reuse of sublanguages (cf. Chapter 7). However, this chapter focuses on syntax and semantics 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, it may mark a grammar as a *component* or define super-grammars which will be explained in Chapter 7.

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



Listing 4.1: Minimal grammar example

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

- *productions* (like A and B) are the main elements of a grammar and constitute the syntactic specification of the language
- *lexer productions* help describing the small tokens, such as names, values, keywords that are the atoms, which the grammar reads
- *grammar options* which allow to configure the grammar
- specific *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 using itself a MontiCore grammar. To avoid confusion when describing grammars with grammars, we avoid the direct reference to the EBNF-nonterminals in the following explanations, but use examples only.

4.1 Lexical Tokens

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

Typical forms of tokens are

- 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 included as terminals (like `"if"`) directly in the productions and thus need no explicit rules. On the other hand, it is possible to introduce *nonterminals* that stand for a single token or a class of tokens, such as for all numbers, strings or names. The nonterminals defined via lexical productions thus will contribute to the abstract syntax in form of attributes, but not introduce new AST classes.

4.1.1 Definition of Tokens using Regular Expressions

Lexical productions are a simplified form of production 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 a token by a *regular expression*. If not specified through an explicit type in the production (see below), a token results in a value of type `String`. The token is later stored as an attribute of the resulting type in the AST named like the token. If the input matches the regular expression defining this token, the lexer recognizes a nonterminal and produces a 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 definition of lexical rules corresponds to regular expressions, which have the following constituents:

- Constant strings surrounded by single or double quotes, like `'st'` or `"st"`.
- A range of characters given by `lowerChar..upperChar`.
- The `|` separates alternatives.
- Grouping uses parentheses `(and)`.
- The character `~` negates, that means the expression matches what is not part of the not negated expression, i.e., the expression without `~`, for example `~('a' | 'b')`.
- The `+` sign denotes repetition of one or more times and can be added to nonterminals or groups.
- `*` is interpreted as zero or more times.
- `?` specifies zero or one-time.

```
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 definitions more readable and allow reuse of definitions at several places. 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 allow a modular definition of lexicals. Therefore, only `NUM_INT` results in a `String` that is stored in the AST, but the string will contain the fragments.

4.1.2 Actions to Process a Token

Actions (i.e., Java code) can be embedded into lexical definitions to modify the lexer result directly. So actions don't contribute to the definition of the concrete syntax, but extend the parser by mapping concrete syntax to elements stored in the abstract syntax or for completely different things.

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

Listing 4.4: 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 4.4). The last line encloses Java code and avoids that this token is passed to the parser. Thus, it is not necessary to explicitly place the nonterminal `WS` in the grammars. We also refer to ANTLR for a detailed discussion.

In the example in Listing 4.5, 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 4.6 shows how to adapt the 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. The supported predefined types are for example `float`,

```

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

```

Listing 4.5: Lexical production for strings without quotation marks, which are removed in a Java action

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 4.6: Changing the result type of lexicals

If the default methods do not work, we can implement our own conversion method as shown in Listing 4.7. The token Cardinality should be adapted by 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.

```

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 4.7: Add a conversion method for lexical types

4.1.3 Predefined Tokens

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

4. MontiCore Grammar for Language and AST Definitions

- MCBasics,
- MCLiterals and
- Types



Tip 4.8 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.MCLiterals.mc4
5             de.monticore.Cardinality.mc4
6             de.monticore.Completeness.mc4
7             de.monticore.MCNumbers.mc4
8             de.monticore.MCHexNumbers.mc4
9             de.monticore.StringLiterals.mc4
```

Files

For an inclusion in a grammar use, e.g., `de.monticore.MCBasics` or include the directory in the grammar path. A detailed description can be found in Chapter 17.

Table 4.9: Predefined tokens of the MCBasics and MCLiterals component grammars.

Token	Value Type	Defined in
WS	-	MCBasics
SL_COMMENT	-	MCBasics
ML_COMMENT	-	MCBasics
NEWLINE	¹	MCBasics
Name	String	MCBasics
Num_Long	long	MCLiterals
DecimalIntegerLiteral	long	MCLiterals
HexIntegerLiteral	long	MCLiterals
DecimalDoublePointLiteral	float	MCLiterals
DecimalFloatingPointLiteral	double	MCLiterals
Char	char	MCLiterals
String	String ²	MCLiterals

When including the available basic grammars MCBasics and MCLiterals, we can use a number of predefined tokens given as described in Table 4.9.³ All tokens (cf. Table 4.9,

¹fragment, not a complete token

²delimiters are removed

³Please note that Types does not define tokens, but only parser nonterminals. MCLiterals also defines additional parser nonterminals.

column 1) are stored as `String` but the `MCLiterals` grammar also provides nonterminals (cf. Table 4.9, column 2) that use these tokens, but their AST classes also provide a method `getValue()` that converts the stored `String` to a better 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 (cf. Section 5.7)

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

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

The `Types` grammar defines nonterminals that provide different kinds of data types. For example, primitive ones such as `int` or `boolean`, and reference types, arrays, generics or the definitions for widely reusable types `ImportStatement` and `QualifiedName`. See Table 4.10 for the list of the most interesting nonterminals.

Table 4.10: Predefined nonterminals of `Types` grammar.

Nonterminal	Meaning
<code>Type</code>	Interface for all forms of types
<code>ReferenceType</code>	Interface
<code>QualifiedName</code>	Sequence of Names, sparated by <code>"."</code>
<code>PrimitiveType</code>	<code>"boolean"</code> , etc.
<code>WildcardType</code>	Single type argument, using wildcard <code>"?"</code>

4.2 Productions


A production consists of a left-hand side that defines a new nonterminal (e.g. `A`) and the right-hand side, i.e., the body of a production describing how the nonterminal is defined. Listing 4.11 shows some simple examples for productions. They are rather similar to lexical productions. In addition, productions support recursion (thus becoming really context-free and not just regular) and a number of techniques to control how to map the productions to the AST classes. The mapping is discussed in Chapter 5 and partially introduced here.

The body of a production is composed of terminals and nonterminals, both of which can be part of alternatives, be optional or occur multiple times. The `MontiCore` grammar allows

- constant strings surrounded by double quotes, e.g., `"st"`,

4. MontiCore Grammar for Language and AST Definitions

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



Listing 4.11: Some production examples

- a range of characters given by `lowerChar..upperChar` (cf. Listing 4.7),
- the `|` separates alternatives,
- grouping uses parentheses `(and)`,
- `+` to express repetition, i.e., one or more times and can be added to nonterminals and groups,
- `*`, which means zero or more times,
- `?`, which means 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 repetition of the left nonterminal `NT` being separated by the right terminal `T`.

Furthermore, *nonterminals* allow to structure parsing including mutual recursion (and only a slight restriction on mutual left recursion). The main differences to the lexical productions are the possibility to use arbitrary other 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 nonterminals and terminals describing the attributes where they will be stored in the AST.
- The ampersand `&` can be attached to name-nonterminals like `Name&` to allow using keywords as names. For example, this is useful for qualified names in package declarations to allow keywords such as "abstract" within a package declaration. Thus, `package de.mc.abstract` is a valid package within a class diagram even though `abstract` is also a keyword in the class diagram language.
- References to other nonterminals can be attached using `@`, like `NT1@NT2`. 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.

The following describes further details.

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). In case a terminal is semantically relevant, e.g. like a terminal in an alternative, there are two options to mark it as relevant: Naming it or surrounding it with square brackets (cf. Listing 4.12). In the first case, the given name of the terminal is used as attribute. In the latter case, a boolean stores whether the terminal occurred in the model. The mapping of relevant terminals to the AST is discussed in Chapter 5.

```

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

MCG

Listing 4.12: Augmentation of terminals for storage in the AST

4.2.2 Enumeration

If the language describes an alternative of multiple terminals resulting in boolean attributes, there is a possible alternative for the abstract syntax. Instead of mapping each terminal to a boolean attribute, an Integer with constants mimicking an enumeration can be used. To use this variant, the alternatives are enclosed in square brackets (Listing 4.13). In this case it is necessary to add a name in front of the square brackets. As shown, each terminal can be named, e.g., PRIVATE: "-", which will result in a constant of the chosen name. 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 it is allowed to add a name that is equivalent to the one that would be derived automatically.

```

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

MCG

Listing 4.13: A choice of alternate terminals is stored as integer

Instead of using a list of keywords (i.e. relevant terminals), it is also possible to use an enumeration nonterminal. As shown in Listing 4.14, the MontiCore grammar language allows to define enumerations directly by using the keyword `enum` followed by a name and a body consisting of a list of constants separated as alternatives. This enumeration nonterminal can be used like any other nonterminal.

```

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

```

MCG

Listing 4.14: 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 indicated by $*$ or $+$ (at least one). Nonterminals are always mapped to the AST (cf. Chapter 5). For every nonterminal there is a class generated for the AST. 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      "}";

```

MCG

Listing 4.15: Automatic naming of unnamed nonterminals

Both, nonterminals and terminals, can be explicitly named by adding a name and a colon. In case no explicit name is given, the name of the composition is derived from the nonterminal name by lower casing the first letter. Thus, `state:State` is equivalent to `State` in Listing 4.15 which reduces the developer's writing effort.

There is also a convenient way of defining comma separated lists. This is a frequently occurring pattern for which MontiCore provides the following construct $y: (A \mid \mid ", ") +$ that is equivalent to $y:A(", " y:A)*$. On the left (i.e. the A), there must be a single nonterminal, while on the right a terminal is needed. The definition of the nonterminal `QualifiedName` is given as $(Name \mid \mid ".")+$ and demonstrates the usage of this feature. Accordingly, $y: (A \mid \mid ", ") *$ is equivalent to $(y:A(", " y:A)*)?$, that also allows an empty repetition.

4.2.4 Interface Nonterminals

A unique feature is the possibility to define interface nonterminals. An example of an interface nonterminal is given in Listing 4.17. Here, the production `I` defines an interface nonterminal and the productions `A` and `B` implement the interface `I`. In the production `C`, interface `I` is used like a normal nonterminal on the production's right-hand side. Semantically, an interface production is equivalent to a production having its implementing productions as alternatives on the right-hand side as shown in Listing 4.18. Interfaces can be used to extend languages and thus are one of the core concepts of language composition (cf. Chapter 7). The main and really important 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.



Tip 4.16 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 below) do.

```

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

```

MCG

Listing 4.17: An interface nonterminal and several nonterminals implementing it

```

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

```

MCG

Listing 4.18: Alternative to interface nonterminal in Listing 4.17 accepting the same concrete syntax, but I knows A and B

Furthermore, productions of interface nonterminals can have a body defining a signature for all nonterminals implementing the interface nonterminal. That is, an interface nonterminal I can define the nonterminals and relevant terminals that all nonterminals implementing I have to be a valid implementation. Considering the example in Listing 4.19, 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 that implement I. However, as demonstrated, the concrete syntax and order of the elements is not prescribed by I, which allows flexible adaptation of the concrete syntax of nonterminals implementing an interface. So, interface bodies are mainly a mechanism to add functionality to the abstract syntax nodes described in Chapter 5.

4.2.5 Extending Nonterminals

The MontiCore grammar format allows to extend the production of an already defined nonterminal by additional alternatives without modifying the original definition. Listing 4.20

```

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 4.19: Interface nonterminal defining its signature

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. For parsing, extension B is equivalent to adding the extending nonterminal B as new alternative to the extended nonterminal A. We chose a definition inverse to EBNF in order to provide an object-oriented solution, where subclasses can extend their superclasses without any changes in the definition of the superclass. This is a second core mechanism to extend languages and reuse grammars. Listing 4.21 shows an equivalent alternative to Listing 4.20 that accepts the same concrete syntax.

```

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

```

Listing 4.20: Extending the production of a nonterminal

```

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

```

Listing 4.21: Equivalent alternative to extension in Listing 4.20 accepting the same concrete syntax, but A knows and thus is coupled to B

4.2.6 Abstract Nonterminals

An abstract nonterminal is similar to an interface nonterminal, but it is introduced by the keyword `abstract`. Abstract nonterminals can bundle nonterminals as shown in the example in Listing 4.23. 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. The nonterminal `AutomatonElement` can now be used on a production's right-hand side to allow both, states and transitions. Listing 4.24 shows an alternative to Listing 4.23 that accepts the same concrete syntax.

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 mark extension points more clearly. That means, the grammar is designed to be extended later by the addition of further nonterminals that extend the abstract ones. In contrast to interface productions, it is possible to add methods using



Tip 4.22 When to use Nonterminal Extension

Extension on nonterminals (like A in Listing 4.20) shares characteristics with class extension. On the concrete syntax that 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 usually not used explicitly in the concrete syntax at all, but plays a role in the abstract syntax.

If A is meant for extension, it is usually better to make it an interface, but when a default version of the concrete syntax exists, then this needs to be defined in an ordinary nonterminal and then extended in sub-nonterminals.

```

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

```

MCG

Listing 4.23: Abstract production in a grammar

AST rules (cf. Chapter 5). As abstract nonterminals are mapped to abstract classes, a production can only extend one abstract production but implement many interfaces.

```

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

```

MCG

Listing 4.24: Alternative to abstract nonterminal in Listing 4.23 accepting the same concrete syntax

4.2.7 Starting Nonterminal

The start (axiom) of a MontiCore grammar is by default the first nonterminal defined within the grammar. However, in case a different nonterminal should act as the starting nonterminal of the language, this can be explicitly stated. To mark a nonterminal as the start, it is introduced by the keyword `start`, as shown in Listing 4.25. This is especially helpful, when the starting nonterminal is inherited.

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 a possibility to attach priorities to infix operations and if necessary also the keyword `<rightassoc>`, which marks the infix operation

4. MontiCore Grammar for Language and AST Definitions

```
1 grammar Automaton3 extends InvAutomaton, Expression {  
2  
3   start Automaton;  
4  
5   // ...  
6 }
```

Listing 4.25: Explicitly setting a top-level nonterminal that is inherited with `start`

as a right associative. Figure 4.26 shows an example of a typical expression language using an excerpt of Java.

```
1 interface Expression;  
2  
3 CallExpression implements Expression <240> =  
4     Expression Arguments;  
5  
6 MultExpression implements Expression <180> =  
7     leftExpression:Expression  
8     (   multiplicativeOp:"*"   
9         |   multiplicativeOp:"/"   
10        |   multiplicativeOp:"%"   
11        )  
12     rightExpression:Expression;  
13  
14 AddExpression implements Expression <170> =  
15     leftExpression:Expression  
16     (   additiveOp:"+"   
17        |   additiveOp:"-"  
18        )  
19     rightExpression:Expression;  
20  
21 AssignmentExpression implements Expression <60> = <rightassoc>  
22     leftExpression:Expression  
23     (   assignment:"="   
24        |   assignment:"+="      // more alternatives ...   
25        )  
26     rightExpression:Expression;
```

Listing 4.26: Example for an expression language using priorities for its infix operations

MontiCore allows to define the `Expression` nonterminal as an interface and then allows to subsequently add alternatives. Usually many alternatives are infix, but by far not all of them and thus should get a priority in the form of an enclosed integer `<170>`. In our example `a+b*c` would parse as `a+(b*c)`, because `180>170`. Many of the non-infix alternatives do not need such a priority, but it may be helpful for extension. However, the priority of prefix operators, like `!"` also influences the parsing order. For example `!"a && b"` would wrongly be parsed as `!"(a && b)"`, if `!"` has lower precedence than `"&&"`.

The advantage of this way of writing down expressions is twofold: (1) the grammar is small

and simple and thus fairly readable. (2) As discussed in Chapter 5, the abstract syntax will also appear in its natural form. For a deeper discussion of how to deal with parsing of infix expressions, that naturally occur to be left recursive, we refer to the standard literature. In former approaches, left recursion had to be reformulated as a right recursion, which would lead to a strange AST (the need for its rearrangement after parsing it, respectively).

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

The good news is that MontCore can handle left recursion within its productions. ANTLR can already handle direct left recursion. MontCore has expanded this feature for left recursions that include an interface and implement this interface through many alternatives as shown in Figure 4.26. This is not general mutual left recursion, but sufficient to allow extensibility at the fine-grained level MontCore is designed for.



Tip 4.27 Left-Recursion is Possible

The good news is that MontCore can handle left recursion within its productions. ANTLR can already handle direct left recursion.

MontCore has expanded this feature for left recursion that include an interface nonterminal and many implementing alternatives.

This is an important advantage in MontCore, because language embedding and extension sometimes introduce mutual recursion only across several language components, i.e. when an already defined nonterminal `Expression` shall be extended with more variants of expressions in an extending grammar.

4.2.9 Restricting the Cardinality of a Nonterminal

In the next Chapter 5, we generally introduce the `ast` rules. Normally, they do not affect the concrete syntax, but there is an exception that we want to mention here already. With a statement like the one given in Listing 4.29 (see Fig. 4.28) it is possible to constrain the number of occurrences of the nonterminal on the right-hand side by having a minimum and maximum number. Line 6 shows such a constraint. Any natural numbers are allowed.

4.2.10 Symbols and Scopes

Textual languages use explicit names for referencing any kind of language entities (methods, classes, attributes, states, signals, etc.) that have been defined elsewhere. For an efficient management of names and the referenced entities, symbols and a symbol management infrastructure is helpful.

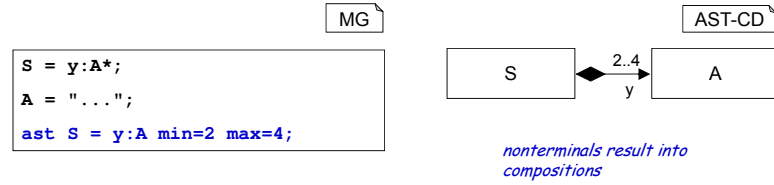


Figure 4.28: Constraining the cardinality of a nonterminal, when parsing

```

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

```

Listing 4.29: Constraining the cardinality of a nonterminal

For a detailed explanation of the definition and usage of symbols and 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. With the keyword `symbol` attached to a production, there has to be a nonterminal `Name` on the right-hand side, which introduces a new symbol. The keyword `scope` attached to a production indicates that the nonterminal introduces a new scope that includes exactly that AST node and all its subnodes. All symbols defined in these subnodes are restricted in their visibility to this scope.

Furthermore, a nonterminal like `Name` on 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 referenced (e.g. `Name@State`). See Chapter 9 for details.

4.2.11 Passing Code to the ANTLR

Rarely, it is necessary to directly specify additional Java code for the underlying ANTLR parser. For this purpose, the MontiCore grammar allows to add Java code as shown in the following example (Listing 4.30).

It is possible to add any piece of code (methods, variables etc.) 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 4.30 counts the number of LT (<) of type parameters or type arguments and can be used for checking the correct number of brackets. The method `incCounter()` allows to increase the counter by the value passed to the method.

Similar to the 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 4.31. An additional

```

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 4.30: Add Java code to the parser

method `capitalize()` is added to the lexer. A String passed to this method is returned with its first letter capitalized.

```

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

```

Listing 4.31: Add Java code to the lexer

These insertions 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 qualified. Furthermore, it has proven useful to keep the Java code in the grammars as little as possible and e.g. delegate all more complex functions to other Java classes.

The `parserjava` and `lexerjava` statements will be mapped to two different classes respectively objects and therefore cannot directly be called from each other. However, the function `getCompiler()` can be used from 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. So controlling the lexer to switch modes for lexing is not easy / possible from the parser.

4.3 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, 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] ?` optional output of A
- `[A|B]` alternative A or B
- `[placeholder=A|B]` named alternative

Naming

<i>CoCo</i> <i>Expl.</i>	0xA4003 (error) The grammar name <code>[grammarName]</code> 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 <code>[package]</code> of the grammar must not differ from the package of the grammar file.
<i>CoCo</i> <i>Expl.</i>	0xA4005 (warning) The name <code>[name]</code> used for the nonterminal <code>[nonterminalName]</code> referenced by the production <code>[productionName]</code> should start with a lower-case letter.
<i>CoCo</i> <i>Expl.</i>	0xA4006 (error) The production <code>[productionName]</code> must not use the attribute name <code>[attributeName]</code> for different nonterminals.
<i>CoCo</i> <i>Expl.</i> <i>Hint</i>	0xA4018 (error) The production <code>[productionName]</code> must not use the keyword <code>[keyword]</code> 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 <code>[productionName]</code> must not use an alternative of keywords without naming it.
<i>CoCo</i> <i>Expl.</i>	0xA4024 (error) The production <code>[productionName]</code> extending the production <code>[extendedProductionName]</code> must not use the name <code>[name]</code> for the nonterminal <code>[nonterminalName]</code> as <code>[extendedProductionName]</code> already uses this name for the nonterminal <code>[extendedProductionsNonterminalName]</code> .
<i>CoCo</i> <i>Expl.</i>	0xA4025 (error) The overriding production <code>[productionName]</code> must not use the name <code>[name]</code> for the nonterminal <code>[nonterminalName]</code> as the overridden production uses this name for the nonterminal <code>[overriddenProductionsNonterminalName]</code> .

Implements/Extends

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

Existence

<i>CoCo</i> <i>Expl.</i>	0xA2025 (error) The nonterminal <code>[nonterminalName]</code> must not be defined by more than one production.
<i>CoCo</i> <i>Expl.</i>	0xA2030 (error) The production <code>[productionName]</code> must not reference the <code>[nonterminal interface nonterminal]</code> <code>[referencedName]</code> because there exists no defining production for <code>[referencedName]</code> .
<i>CoCo</i> <i>Expl.</i>	0xA2031 (error) The production <code>[productionName]</code> must not use the nonterminal <code>[referencedName]</code> because there exists no production defining <code>[referencedName]</code> .
<i>CoCo</i> <i>Expl.</i>	0xA0276 (error) The external nonterminal <code>[name]</code> must not be used in a grammar not marked as a grammar component.
<i>CoCo</i> <i>Expl.</i>	0xA0277 (error) The abstract nonterminal <code>[name]</code> 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 <code>[name]</code> must not be used without nonterminals implementing it in a grammar not marked as a grammar component.
<i>CoCo</i> <i>Expl.</i> <i>Hint</i>	0xA4014 (error) Duplicate enum constant: <code>[enumConstant]</code> . The constants of enumerations must be unique within an enumeration.
<i>CoCo</i> <i>Expl.</i>	0xA4015 (error) The lexical production <code>[productionName]</code> must not allow the empty token.
<i>CoCo</i> <i>Expl.</i>	0xA4016 (error) The lexical production <code>[productionName]</code> must not reference the nonterminal <code>[tokenName]</code> because there exists no lexical production defining <code>[tokenName]</code> .
<i>CoCo</i> <i>Expl.</i>	0xA4017 (error) The lexical production <code>[productionName]</code> must not reference the nonterminal <code>[tokenName]</code> because <code>[tokenName]</code> is defined by a production of another type than lexical. Lexical productions may only reference nonterminals defined by lexical productions.
<i>CoCo</i> <i>Expl.</i>	0xA4020 (error) There must not exist more than one AST rule for the nonterminal <code>[nonterminalName]</code> .

<i>CoCo</i> <i>Expl.</i>	0xA4021 (error) There must not exist an AST rule for the nonterminal [nonterminalName] because there exists no production defining [referencedName].
<i>CoCo</i> <i>Expl.</i>	0xA4021 (error) There must not exist an AST rule for the enum nonterminal [nonterminalName].
<i>CoCo</i> <i>Expl.</i>	0xA4028 (error) 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].

Overriding

<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 enum production [productionName] must not be overridden.

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. Inheritance may not be cyclic.

4. MontiCore Grammar for Language and AST Definitions

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:

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

Files

4.4 Semantic Predicates and Actions

Semantic predicates and actions are Java expressions embedded in curly braces and followed by a question mark. They allow to check constraints on passed model elements or inject Java code into the parser, e.g. counters to count opening and closing brackets. The syntax and semantics of semantic predicates and actions have been adopted from ANTLR, therefore we don't include an example here and refer to the ANTLR documentation available at [Par13] instead. This construct is not very often necessary and potentially only needs to be used by experts.

Due to the restricted capabilities of the token lexer to read its input and select the right token, it is, however, sometimes useful to use semantic predicates to help MontiCore to select the correct parsing form. See Section 18.4.3 for examples.

4.5 Editor assistance for the MCG Language

The MontiCore language is a purely ASCII (UTF-8) based language and thus, grammars can be modified by any editor. However, MontiCore also provides a specific editor with the usual assistance, such as highlighting, auto completion and an integration into the Eclipse and IntelliJ platforms. The editor can be used out of the box.



Tip 4.32 The MontiCore Grammar Editor

A syntax assisting editor including syntax highlighting, auto completion, etc. for the grammar language can be found here:

```
1 Repository: Monticore/monticore github
2 Directory:  monticore-editor
```

Files

4.6 EBNF for 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, Version June, 22nd, 2017.
3  */
4
5  MCGrammar ::=
6    ('package' QualifiedName ';' )?
7    (MCImportStatement )* ('component')?
8    'grammar' Name ('extends' (GrammarReference || ',' )+ )?
9    '{'
10   (Prod | ASTRule | GrammarOption | Concept | StartRule)*
11   '}' ;
12
13  MCImportStatement ::=
14  'import' QualifiedName ('.' '*' )? ';' ;
15
16  GrammarReference ::= QualifiedName ;
17
18  Prod ::=
19    LexProd
20    | ClassProd
21    | InterfaceProd
22    | AbstractProd
23    | ExternalProd
24    | EnumProd
25    ;
26
27  StartRule =
28    'start' RuleReference ';' ;
29
30  //////////////////////////////////////# Productions
31
32  LexProd ::=
33    ('fragment')?
34    ('token' ) Name
35    '=' (LexAlt || '|' )+
36    ':' ('{' Action '}' )?
37    (Name ('->' QualifiedName ':' '{' Action '}' )? )? )? ';' ;
38
39  ClassProd ::=
40    SymbolDefinition* Name
41    (
42      'extends' (RuleReference || ',' )+
43      | 'implements' (RuleReference || ',' )+
44      | 'astextends' (GenericType || ',' )+
45      | 'astimplements' (GenericType || ',' )+
46    ) *
47    ('{' Action '}' )?
48    '=' (Alt ('|' Alt ) * ) )? ';' ;
49
50  InterfaceProd ::=

```

4. MontiCore Grammar for Language and AST Definitions

```
51     'interface' SymbolDefinition* Name
52     (
53         'extends' (RuleReferenceWithPredicates || ',' )+
54         | 'astextends' (GenericType || ',' )+
55     ) * ('=' (Alt || '|') + )? ';' ;
56
57 AbstractProd ::=
58     'abstract' SymbolDefinition* Name
59     (
60         'extends' (RuleReferenceWithPredicates || ',' )+
61         | 'implements' (RuleReferenceWithPredicates || ',' )+
62         | 'astextends' (GenericType || ',' )+
63         | 'astimplements' (GenericType || ',' )+
64     ) * ('=' (Alt || '|') + )? ';' ;
65
66 ExternalProd ::=
67     'external' SymbolDefinition* Name GenericType? ';' ;
68
69 EnumProd ::=
70     'enum' Name '=' (Constant || '|') + ';' ;
71
72 Alt ::= "<rightassoc>"? RuleComponent* ;
73
74 RuleComponent =
75     NonTerminalSeparator
76     | Block
77     | NonTerminal
78     | Terminal
79     | ConstantGroup
80     | SemanticpredicateOrAction
81     ;
82
83 NonTerminalSeparator ::=
84     (Name ':' )? '(' Name ('&')? '||' String ')' ('*' | '+' ) ;
85
86 Block ::=
87     '(' (Alt ('|' Alt ) * ) ')' (('?' | '*' | '+') )? ;
88
89 Option ::=
90     'options' '{' OptionValue '}' ;
91
92 OptionValue ::=
93     Name '=' Name ';' ;
94
95 NonTerminal ::=
96     (Name ':' )? Name ('@' Name)? ('&' | ('?' | '*' | '+') ) * ;
97
98 Terminal ::=
99     (Name ':' )? String (('?' | '*' | '+') ) * ;
100
101 Constant ::=
102     (Name ':' )? String ;
```

```

103
104 ConstantGroup ::=
105     (Name ':' )? '[' (Constant || '|' )+ ']' (('?' | '*' | '+'))? ;
106
107 SemanticpredicateOrAction ::=
108     ('{' ExpressionPredicate '}' '?'
109      |
110      '{' Action '}' ) ;
111
112 LexAlt ::=
113     (LexComponent ) * ;
114
115 LexComponent ::=
116     LexBlock
117     | LexCharRange
118     | LexChar
119     | LexString
120     | LexActionOrPredicate
121     | LexNonTerminal
122     | LexSimpleIteration
123     ;
124
125 LexBlock ::=
126     ('~')? '(' (LexAlt || '|' )+ ')' (('?' | '*' | '+'))? ;
127
128 LexCharRange ::=
129     ('~')? Char '..' Char ;
130
131 LexChar ::= ('~')? Char ;
132
133 LexString ::= String ;
134
135 LexActionOrPredicate ::=
136     ('{' ExpressionPredicate '}' '?' ) ;
137
138 LexNonTerminal ::= Name ;
139
140 LexSimpleIteration ::=
141     (LexString | LexChar ) ('?' | '*' | '+') ;
142
143 LexOption ::=
144     'options' '{' Name '=' Name ';' '}' ;
145
146
147 //##### AST Rules
148
149 ASTRule ::=
150     'astrule' Name
151     (
152         'astextends' (GenericType || ',' )+
153         | 'astimplements' (GenericType || ',' )+
154     ) *

```

4. MontiCore Grammar for Language and AST Definitions

```
155     ('='
156     (Method | AttributeInAST ) *
157     )? ';' ;
158
159 Method ::=
160     'method' ('public'| 'private'| 'protected')?
161     ('final')? ('static')?
162     GenericType Name '(' (MethodParameter || ',' ) * ')'
163     ('throws' (GenericType || ',' ) + )?
164     '{'
165     Action
166     '}' ;
167
168 MethodParameter ::= GenericType Name ;
169
170 AttributeInAST ::=
171     (Name ':' )? GenericType Card? ;
172
173 Card ::=
174     ('*'
175     |
176     'min' '=' Num_Int ('max' '=' (Num_Int | '*' ) )?
177     |
178     'max' '=' (Num_Int | '*' ) ) ;
179
180 //##### Grammar Options
181
182 GrammarOption ::=
183     'options' '{' (FollowOption | AntlrOption ) * '}' ;
184
185 FollowOption ::=
186     'follow' Name Alt ';' ;
187
188 AntlrOption ::=
189     Name ('=' (Name | String) )? ;
190
191 //##### Concepts
192
193 Concept ::=
194     'concept' Name MConcept ;
195
196
197 // ##### External Productions
198
199 Action ::= ... ;
200
201 ExpressionPredicate ::= ... ;
202
203 MConcept ::= ... ;
204
205 // ##### Types and References
206
```

```

207 GenericType ::=
208     QualifiedName ('<' (GenericType || ',')+ '>' )? ( '[' ']' )* ;
209
210 RuleReference ::= Name ("<Num_Int">)? ;
211
212 RuleReferenceWithPredicates ::= SemanticpredicateOrAction Name ;
213
214 QualifiedName ::= (Name || '.')+ ;
215
216 // ##### Symbol Infrastructure
217
218 SymbolDefinition ::=
219     "symbol" ( "(" ("Name") " )? | "scope" ;

```

Listing 4.33: EBNF of the MontiCore grammar MCG

**Tip 4.34 The MontiCore Grammar**

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

<pre> 1 Repository: MontiCore/monticore github 2 Directory: monticore-generator/src/main/grammars/ 3 Files: de.monticore.grammar.Grammar.mc4 4 de.monticore.grammar.Grammar_WithConcepts.mc4 </pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Files</div>
---	--

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, a lexer, a 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 generates also builders and other helpful classes derived from a grammar. For a given grammar all generated files 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 the 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 classes.	<code>_ast</code>
Visitor	For traversing the AST (see Chapter 8)	<code>_visitor</code>
Context Condition Infrastructure	Environment to implement Context Conditions (See Chapter 10)	<code>_cocos</code>
EBNF and CD Representation of the Grammar	Help understanding the grammar and AST structure	<code>_report</code>

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**. In the AST nonterminals that are marked with a *** or *+* result in lists. 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.

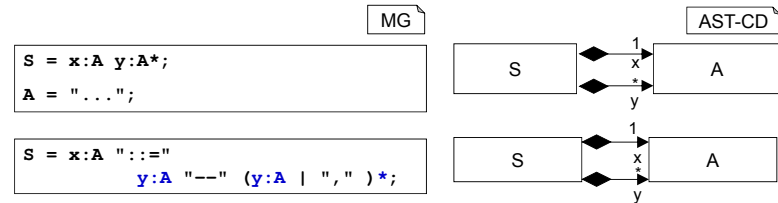


Figure 5.2: Sequences of nonterminals in the AST

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 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.

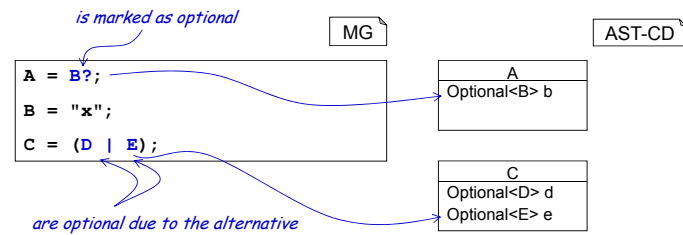


Figure 5.3: Optional nonterminals

The storage in lists is efficient, but abstracts from certain information that might still be relevant. In pathological cases like $(A|A)$, or $(A|B) *$ the alternative taken respectively the order in which *As* and *Bs* 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*. Considering 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 4.17 and 4.18 are equivalent for the concrete syntax, their AST differs 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 (cf. Figure 5.5). Usually other nonterminals implement or extend those nonterminals, which will be explained in Section 5.3.

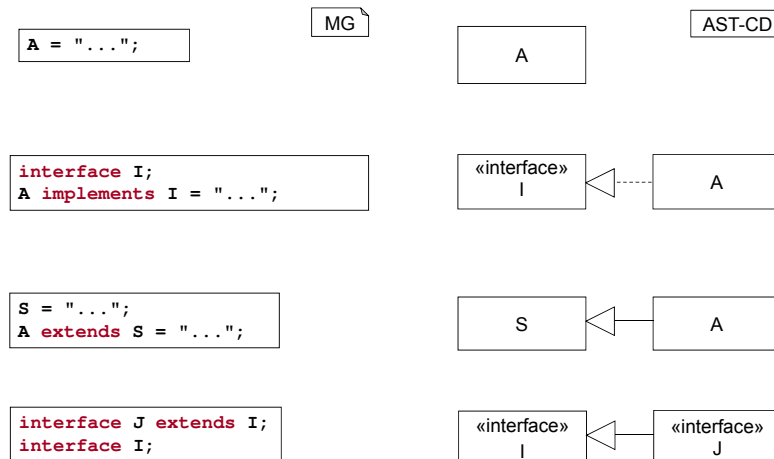


Figure 5.4: How interfaces in a grammar map to the abstract syntax

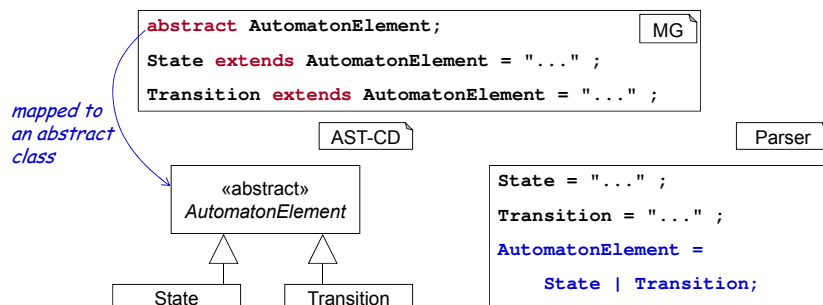


Figure 5.5: How abstract nonterminals in a grammar map to abstract classes



Tip 5.6 Interface and Abstract Nonterminals

Interfaces and abstract nonterminals help to structure the grammar 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).

And they help to structure the abstract syntax, represented in a set of classes and interfaces. So a good structure of the AST is based on a good structure of the grammar.

5.3 Extending Nonterminals

On the AST the extension mechanism for nonterminals is straightforward, as it directly maps to extension of the according AST classes (cf. Figure 5.7 and 5.4). Interface nonterminals map to Java interfaces. AST classes created for nonterminals that implement an interface nonterminal, implement the corresponding Java interface. Therefore, the AST differs from its concrete syntax equivalent, i.e., an alternative (cf. Section 4.2.4). 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 4.3), but implemented by normal and abstract nonterminals.

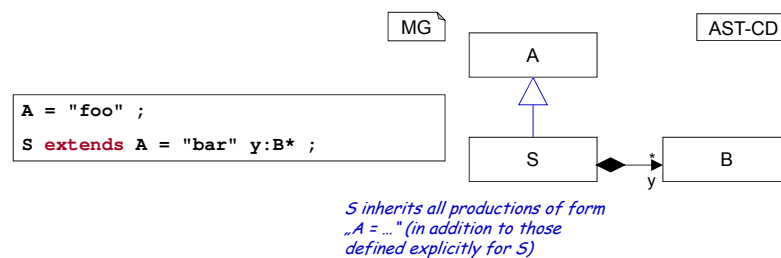


Figure 5.7: Productions extending other productions

The MCG grammar format allows to add 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. Keyword `astextends` 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 followed by a Java class respectively Java interface on the right. They are directly realized in the structure in the generated AST (cf. Figure 5.8 and 5.9).

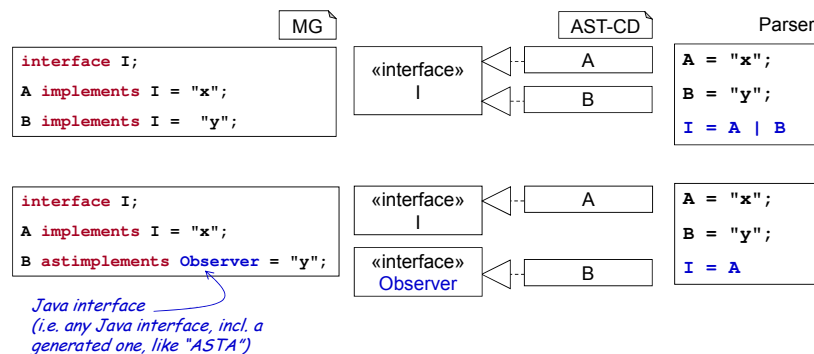


Figure 5.8: Implements in abstract and concrete syntax

In the `astimplements` statement any Java interface, e.g. `Observer`, but also interfaces derived from the grammar, eg. `ASTI`, can be used. However, some restrictions apply.

For instance, given `B astimplements IF`, the Java interface `IF` enforces its methods to be 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 `ast` statement or (3) a handwritten version of class `ASTB` is provided that implements these methods (see Section 5.10).

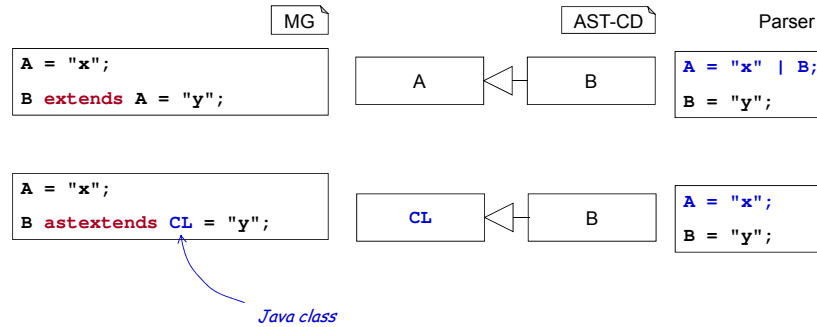


Figure 5.9: Inheritance in abstract and concrete syntax

The Java class `CL` in statement `B astextends CL` also obeys restrictions. Because Java only allows single inheritance, then the `astextends` statement lets `CL` replace the normal superclass, which is directly or transitively a subclass of `ASTNode`. Therefore, `CL` must be a subclass of `ASTNode` or at least implement interface `ASTNode`. Thus, it is not possible to use foreign classes, such as `Observable`.

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

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

5.4 Extending the Abstract Syntax Implementation

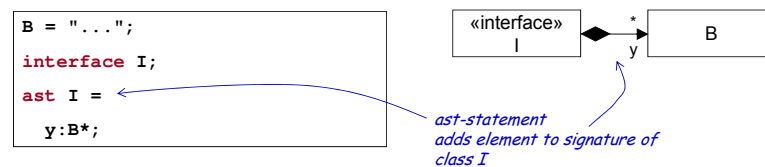


Figure 5.10: Extending the AST structure

AST rules start with the keyword `ast` 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 `ast` (cf. Figure 5.10).

5. Abstract Syntax Tree

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

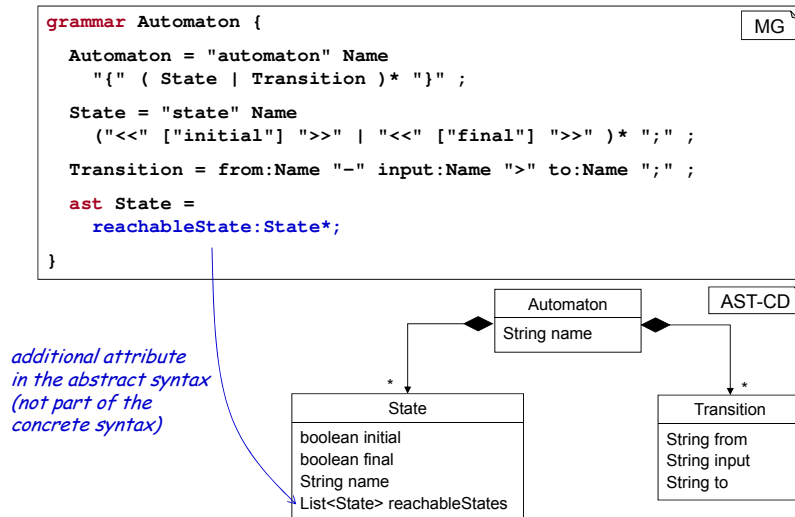


Figure 5.11: Adding attributes in the AST with the `ast` statement

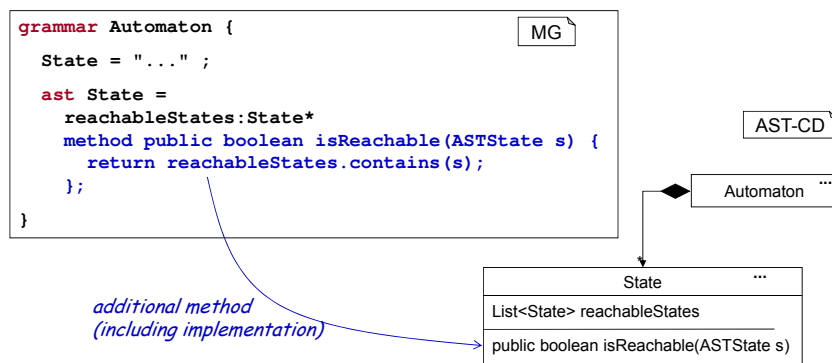


Figure 5.12: Adding methods in the AST with the `ast` statement

In AST rules, nonterminals can be used the same way as they are in productions (cf. Figure 5.11). They result in additional attributes 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 to override the type chosen by the attribute derivation.

Please note that the mechanism for a handcoded extension of the AST allows roughly the


Tip 5.13 AST Rules, like `astextends` and `ast`

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 `ast` 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 `ast` are visible in the public signature of a generated AST class.

same effects and 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.

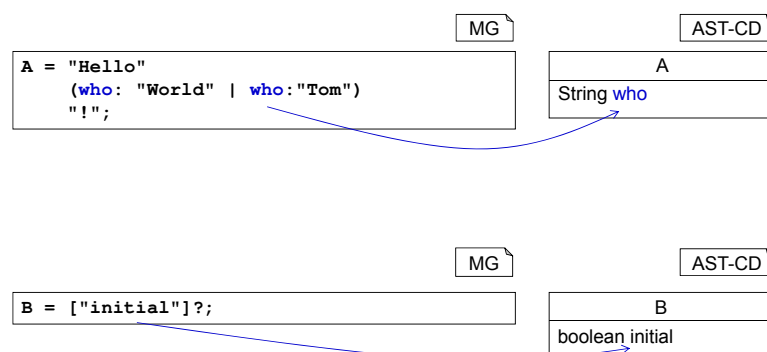


Figure 5.14: Terminals included in the AST

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.

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 decapitalized. 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"`.

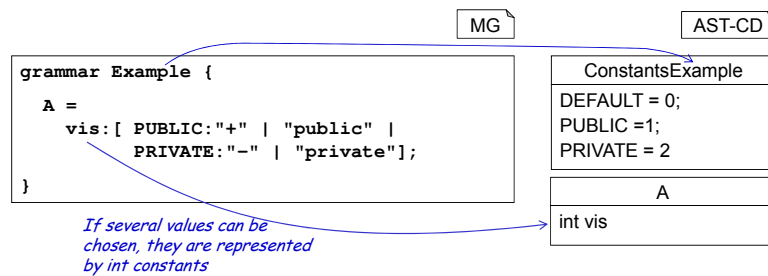


Figure 5.15: Choice of one of several values stored as `int`

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.

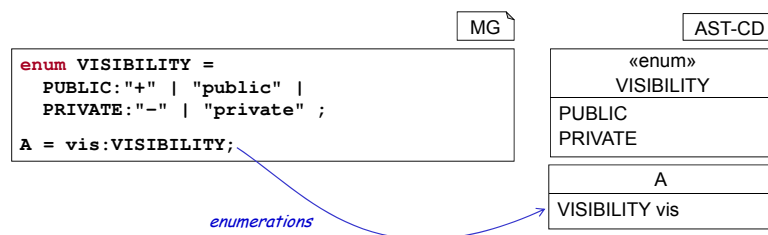


Figure 5.16: Explicit definition of an enumeration

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 to compare AST nodes, stores comments from the original file and the source position, from where the AST node and its substructure has been parsed. Furthermore, it provides functions that allow to deal with the symbol and scope structure. This will be discussed in Chapter 9.

The ASTNode interface shown in Figure 5.17 provides the common signature of 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    Optional<SourcePosition> get_SourcePositionStartOpt();
18    void set_SourcePositionStart(SourcePosition start);
19    void set_SourcePositionStartAbsent();
20    void set_SourcePositionStartOpt(Optional<SourcePosition> value);
21
22    // Managing the attached source position (end)
23    boolean isPresent_SourcePositionEnd();
24    SourcePosition get_SourcePositionEnd();
25    Optional<SourcePosition> get_SourcePositionEndOpt();
26    void set_SourcePositionEnd(SourcePosition end);
27    void set_SourcePositionEndAbsent();
28    void set_SourcePositionEndOpt(Optional<SourcePosition> value);
29
30    // Managing attached comments
31    boolean add_PreComments(Comment precomment);
32    // in total ~30 methods for managing List<Comment> pre
33
34    boolean add_PostComments(Comment postcomment);
35    // in total ~30 methods for managing List<Comment> post
36
37    // Symbol infrastructure management
38    // When the node is embedded in a scope:
39    boolean isPresentEnclosingScope();
40    Scope getEnclosingScope();
41    Optional<? extends Scope> getEnclosingScopeOpt();
42    void setEnclosingScope(Scope scope);
43    void setEnclosingScopeAbsent();
44    void setEnclosingScopeOpt(Optional<Scope> value);
45
46    // When the node defines a symbol:
47    boolean isPresentSymbol();
48    Symbol getSymbol();

```

5. Abstract Syntax Tree

```

49 Optional<Symbol> getSymbolOpt();
50 void setSymbol(Symbol s);
51 void setSymbolAbsent();
52 void setSymbolOpt(Optional<Symbol> value);
53
54 // When the node itself defines a new scope for its children:
55 boolean isPresentSpannedScope();
56 Scope getSpannedScope();
57 Optional<? extends Scope> getSpannedScopeOpt();
58 void setSpannedScope(Scope scope);
59 void setSpannedScopeAbsent();
60 void setSpannedScopeOpt(Optional<Scope> value);
61 }

```

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. 3f and ll. 7ff). By default the `deepEquals` methods ignore the order of children occurring in the AST, but the comparison with a precise check of the order can also be used.

The source position is internally stored as an optional and thus the usual six methods for their management are provided. Because it consists of start and end, we have in total 12 methods (ll. 15ff and ll. 23ff). 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. Furthermore, ASTNode objects contain the necessary infrastructure to manage symbols and scopes that the nodes are potentially defining or enclosed in (ll. 39ff, ll. 47ff, and ll. 55ff). See Chapter 9 for details.

Figure 5.18 shows a part of the signature and data structure of all ASTNodes.

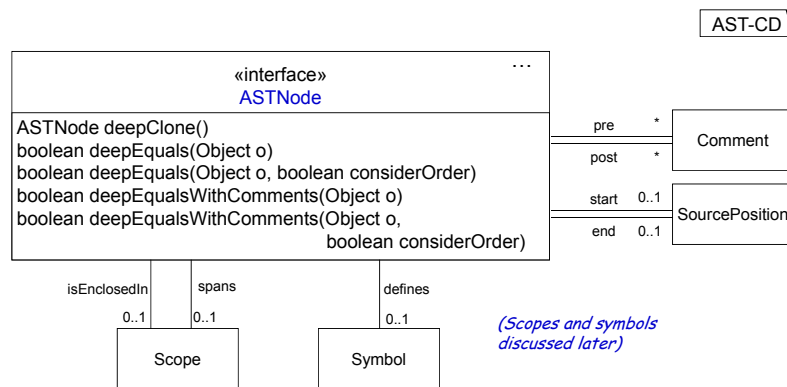


Figure 5.18: Common interfaces of AST classes

For a default implementation of several functions provided by ASTNode objects, the MontiCore runtime environment also provides an abstract subclass `ASTCNode` that implements

also 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 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 assess 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 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 automaton3._ast;
2
3 public class ASTState extends ASTCNode implements ASTAutomaton3Node
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    // Constructors (but we use builders):
14    protected ASTState();
15    protected ASTState(
16        String name,
17        Optional<ASTNatLiteral> prio,
18        List<ASTState> states,
19        List<ASTTransition> transitions,

```

5. Abstract Syntax Tree

```
20     boolean initial,  
21     boolean r__final);  
22  
23     // Visitor management:  
24     void accept(Automaton3Visitor visitor);  
25     void accept(MCBasicsVisitor visitor);  
26     void accept(MCLiteralsVisitor 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 `ASTAutomaton3Node` 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 below described get and set functions. Two constructors (l. 13ff.) exist, but are also protected, because any object of that class is to be defined using the also generated builders (see Section 5.9).

For each directly or indirectly included grammar the node also provides an appropriate `accept` method (l. 23ff.), 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 Automaton grammar relies on `MCBasics` and `MCLiterals`.

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, but the attributes are obviously deferred to the implementing classes. In Listing 5.20 the second part of class `ASTState` is shown.

```
1  
2     // Treatment of children and semantically relevant terminals:  
3     String getName();  
4     void setName(String name);  
5  
6     // Optional attributes have six methods:  
7     boolean isPresentPrio();  
8     ASTNatLiteral getPrio();  
9     Optional<ASTNatLiteral> getPrioOpt();  
10    void setPrio(ASTNatLiteral prio);  
11    void setPrioAbsent();  
12    void setPrioOpt(Optional<ASTNatLiteral> value);  
13  
14    boolean isInitial();  
15    void setInitial(boolean initial);  
16  
17    boolean isFinal();  
18    void setFinal(boolean r__final);  
19 }
```

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

For each nonterminal with a mandatory occurrence 1 a get and set method are generated to allow access to the attribute. Lines 3f 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 are 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 six 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 to question the presence (l. 7 in Listing 5.20) and another method allows to set the value to absent (l. 11). Please also note that there are two get methods for an optional attribute: getPrioOpt to retrieve the optional value and the partial method getPrio, which issues an error message and raises an exception, if the value is absent. getPrio thus must be guarded by isPresentPrio.

Line 14 shows, how optional, but semantically relevant terminals are managed. And line 17 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.

We show the signature for the Transition* attributes 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<Object> collection);
7  boolean retainAllTransitions(Collection<Object> collection);
8
9  boolean containsTransition(Object element);
10 boolean containsAllTransitions(Collection<Object> 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)

```

5. Abstract Syntax Tree

```
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 equalsTransition(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 subobjects. 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 Modelling Framework) Integration is Available

MontiCore is a standalone tool infrastructure.

But if desired, the Monticore generator can generate signatures for the AST node classes in such a from 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 automaton3._ast;
2
3 public class ASTState extends de.monticore.emf._ast.ASTECNode
4     implements ASTAutomaton3Node
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             Automaton3Package.ASTState_States);
13     protected List<ASTTransition> transitions =
14         new EObjectContainmentEList<ASTTransition>(
15             ASTTransition.class, this,
16             Automaton3Package.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    );
    }

```

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 AutomatonMill created for the example language for finite automata (cf. Section 18.1). 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 automaton._ast;
2
3  public class AutomatonMill {
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 finally 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 finally 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 nonterminal `State`.

In general, these methods do not differ from the methods generated for class `ASTState`, but have one important difference: Where the method in class `ASTState` has 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).addTransition(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 automaton3._ast;
2
3 public class ASTStateBuilder extends
4     ASTNodeBuilder<ASTStateBuilder> {
5     // Setting an attribute
6     ASTStateBuilder setName(String s);
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    ASTStateBuilder setPrioOpt(Optional<ASTNatLiteral> value);
16
17    // Setting a List valued attribute
18    ASTStateBuilder setTransitionList(
19        List<ASTTransition> transitions);
20    ASTStateBuilder clearTransitions();
21    ASTStateBuilder addTransition(ASTTransition element);
22    ASTStateBuilder addAllTransitions(
23        Collection<? extends ASTTransition> collection);
24    ASTStateBuilder removeTransition(Object element);
25    ASTStateBuilder addTransition(int index, ASTTransition element);
26    // ... in total ~30 methods to handle the transition list
27
28    ASTStateBuilder setStateList(List<ASTState> states);
29    ASTStateBuilder addState(ASTState element);
30    // ... in total ~30 methods to handle the state list
31
32    // Inherited methods for attributes from ASTNodeBuilder
33    // (first for the Optionals)
34    ASTStateBuilder set_SourcePositionStart(SourcePosition start);
35    ASTStateBuilder set_SourcePositionStartAbsent();
36    ASTStateBuilder set_SourcePositionStartOpt(
37        Optional<SourcePosition> value);
38
39    ASTStateBuilder set_SourcePositionEnd(SourcePosition end);
40    ASTStateBuilder set_SourcePositionEndAbsent();
41    ASTStateBuilder set_SourcePositionEndOpt(

```

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.

5. Abstract Syntax Tree

```
42         Optional<SourcePosition> value);
43
44     ASTStateBuilder setEnclosingScope(Scope enclosingScope);
45     ASTStateBuilder setEnclosingScopeAbsent();
46     ASTStateBuilder setEnclosingScopeOpt(Optional<Scope> value);
47
48     ASTStateBuilder setSpannedScope(Scope spannedScope);
49     ASTStateBuilder setSpannedScopeAbsent();
50     ASTStateBuilder setSpannedScopeOpt(Optional<Scope> value);
51
52     ASTStateBuilder setSymbol(Symbol symbol);
53     ASTStateBuilder setSymbolAbsent();
54     ASTStateBuilder setSymbolOpt(Optional<Symbol> value);
55
56     // Inherited methods for attributes from ASTNodeBuilder
57     // (the Lists of Comments)
58     ASTStateBuilder set_PreCommentList(List<Comment> comments);
59     ASTStateBuilder add_PreComment(Comment element);
60     // ... in total ~30 methods to handle the pre comments
61
62     ASTStateBuilder set_PostCommentList(List<Comment> comments);
63     ASTStateBuilder add_PostComment(Comment element);
64     // ... in total ~30 methods to handle the post comments
65
66
67     // Finally constructing the object
68     ASTState build();
69 }
```

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. 18 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. 34 the signature for setting and manipulating the inherited attributes is shown. To avoid name clashes, the inherited methods partially have an underscore in their name.

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.

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      Optional<ASTNatLiteral> getPrioOpt();
9      ASTNatLiteral getPrio();    // partial
10     boolean isPresentPrio();
11
12     // Some inherited retrieval methods for
13     // attributes from ASTNodeBuilder
14     Optional<Symbol> getSymbolOpt();
15     Symbol getSymbol();    // partial
16     boolean isPresentSymbol();
17
18     // Some retrievers for the Transition* attribute
19     boolean containsTransitions(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`.

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; 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 build 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.

Also 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. But 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

To extend 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`.

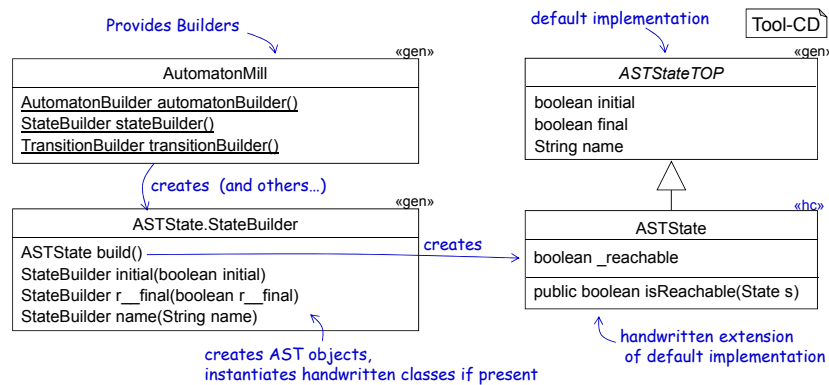


Figure 5.29: Example: Handwritten AST class `ASTState` injected into the parsing process



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 `C`, which now becomes generated as `ClsTOP`. `Cls` should inherit from `ClsTOP`, but also could ignore the generated class.

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

This approach brings numerous advantages:

- 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 need not 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. A builder mill uses 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:

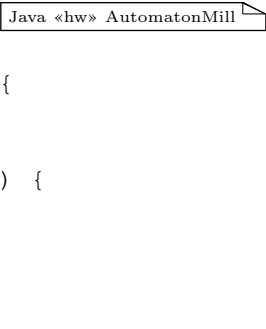
```
1 package automaton._ast;
2
3 public class AutomatonMill {
4 // ... only the protected elements
5
6 // attributes store individual builder mills for each node
7 // (but all may be the same instance)
8 protected static AutomatonMill mill;
9 protected static AutomatonMill millASTAutomaton;
10 protected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

Java «gen» AutomatonMill

Listing 5.31: Internal structure of the AutomatonMill

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

```
1 package automaton._ast;
2
3 public class AutomatonMill extends AutomatonMillTOP {
4
5     @Override
6     protected ASTTransitionBuilder _transitionBuilder() {
7         return new MyTransitionBuilder();
8     }
9
10 }
```



Listing 5.32: Handcoded extension of the `AutomatonMill`

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

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.

6.1 Generating a Parser and a Lexer

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                                     Java «MontiCore» ParserGenerator
2 public static void generateFullParser (ASTMCGrammar astGrammar,
3                                     IterablePath handcodedPath,
4                                     File targetDir)

```

Listing 6.2: Method signature used to generate a parser

If the parser generator is used within Java, the method `generateFullParser` 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 AST representation of the grammar that describes a modeling language whose models should be parsed (see Tip 6.7),
2. 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, and
3. the directory in which the generated parser will be created, which is freely selectable. For instance, it can be `"gen/"` or `"out/"`.

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

```

1 // Create the AST                                     Java «hw» GenerateAutomatonParser
2 String filename = "Automaton.mc4";
3 ASTMCGrammar ast;
4 ast = new Grammar_WithConceptsParser()
5         .parseMCGrammar(filename).get();
6
7 // Initialize symbol table
8 // (using imported grammars from the model path)
9 ModelPath modelPath = new ModelPath(Paths.get("monticore-cli.jar"));
10 MCGrammarSymbolTableHelper.initializeSymbolTable(ast, modelPath);
11
12 // Hand coded path
13 IterablePath handcodedPath = IterablePath.empty();
14
15 // Target directory
16 File outputDir = new File("gen");
17
18 // Generate the parser
19 ParserGenerator.generateFullParser(ast, handcodedPath, outputDir);

```

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

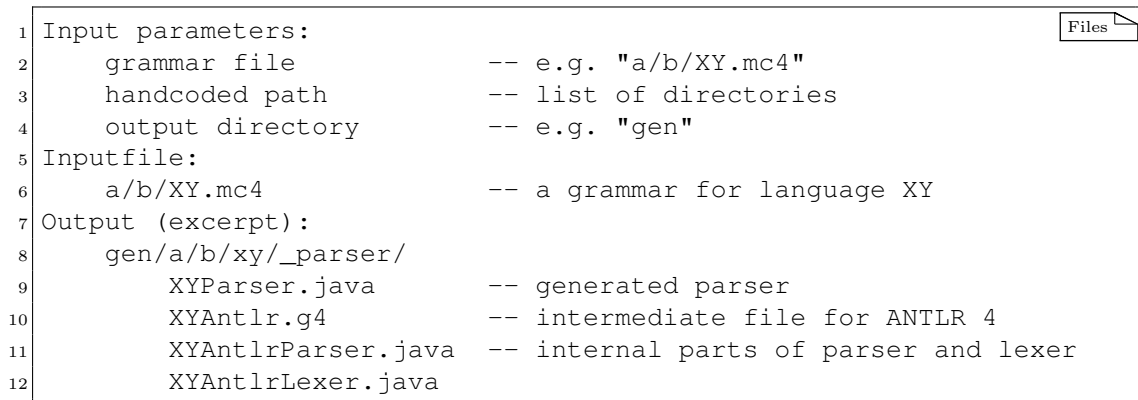
The code block in line 3ff. loads the grammar that describes the language. When the AST is built, it represents only the currently processed grammar without its super grammars. Super grammar information is added in subsequent steps.

The statements in line 9 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 10. While constructing the symbol table, all the dependent grammars are loaded and included.

The statement in line 19 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`-file¹ 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 ANTLR4 file is created that is used in the following steps as an input for the ANTLR4 tool to create a parser and a lexer.

Step two: In the second step the ANTLR4 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).

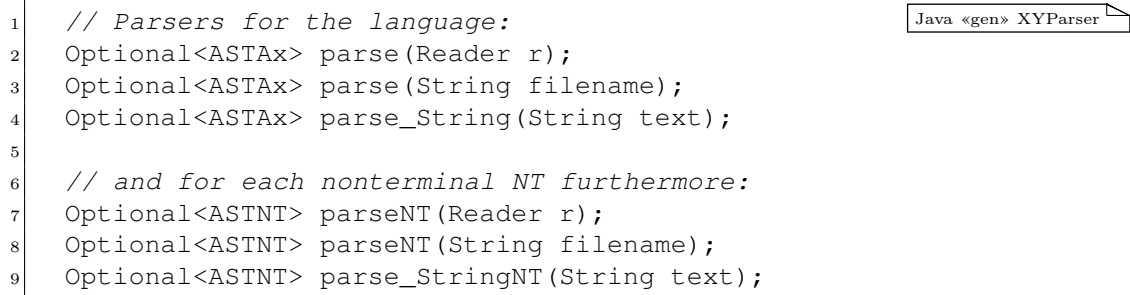
¹This file is not detailed here, please refer to [Par13] for further information

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 ANTLR4 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

The generated parser class `XYParser` contains three methods (cf. line 2-4 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. line 7-9 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 r);
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 r);
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).

The parse methods are called as normal methods. Internally a parser uses a factory 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 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, but instead the MontiCore runtime environment (RTE, marked as «RTE») is. The MontiCore runtime environment is packaged in the provided MontiCore jar as well.

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

As a first example, the Automaton 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 AutomatonParser p = new AutomatonParser();
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 Automaton parser reads a file (or string) and constructs the AST corresponding to the model of the Automaton 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 Automaton 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-generator/src/main/grammars/
3 File: -- grammar describing how MontiCore grammars look like
4       de.monticore.grammar.Grammar_WithConcepts.mc4
5 Directory: monticore-generator/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 Optional<ASTMCGrammar> ast = Optional.empty();
2 String filename = "automaton/Automaton.mc4";
3 Grammar_WithConceptsParser p =
4     new Grammar_WithConceptsParser();
5
6
7 // Create the AST
8 ast = p.parseMCGrammar(filename);

```

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 larger proliferation of software languages for many different domains far beyond software development will appear. 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.

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 participating 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 and no change of the language components themselves into a composed language.

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 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 7.1 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 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 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 part 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	8.3.1	8.3.3	8.3
Symbol Management	9	-	-	-
Context Conditions	10	-	-	-
Backend (Generator)	(13)	-	-	-

More details can be found in the respective research results, such as [MSN17, Rot17, Wei12, Sch12, Völ11, Kra10, HMSNRW16a, MSNRR16, HLMSN⁺15, 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. The following consequently is a high-level overview:

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.

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 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 and 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 `DelegatorVisitor` (see Chapter 8), which is available for each language and allows to compose visitors that have been individually developed for sublanguages. The `DelegatorVisitor` 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. But, 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, don't believe, that their composition is an easy task. We do not even really know well at the moment, 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 a nonterminal is not defined here, but marks 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.

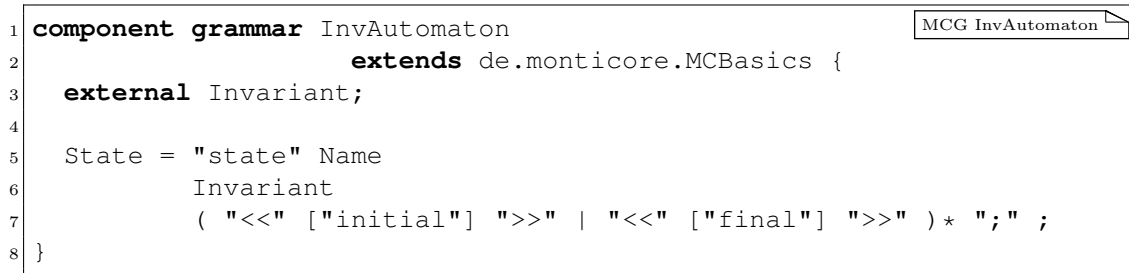
interface nonterminals (and to some extent `abstract nonterminals`) enable to structure monolithic grammars, but can also be used to mark extension points of a grammar component. When importing a grammar component, additional alternatives can be added to interfaces.

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.

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

```

Listing 7.2: 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 7.2). An external nonterminal defines an extension point in the grammar, i.e., it can be used like all other nonterminals 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 7.2, `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 (empty) interface that needs to be implemented when the language is completed:

```

1 package invautomaton._ast;
2
3 public interface ASTInvariantExt
4             extends ASTNode, ASTInvAutomatonNode {
5
6     // ... only clone and equals signatures are given
7 }

```

Java «gen» ASTInvariantExt

Listing 7.3: 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 7.2 leads to the Java interface `ASTInvariantExt` shown in Listing 7.3.

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 Automaton2 extends InvAutomaton {
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 }

```

MCG Automaton2

Listing 7.4: 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 7.5.

```

1 package automaton2._ast;
2
3 public class ASTInvariant extends ASTCNode
4     implements ASTInvariantExt, ASTAutomaton2Node {
5
6     protected Optional<ASTLogicExpr>
7         logicExpr = Optional.empty();
8     protected boolean mINUS;
9 }

```

Java «gen» ASTInvariant

Listing 7.5: Implementation of the Invariant nonterminal

The nonterminal `Invariant` is now mapped to a normal AST class with the speciality, 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.

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 `"-g"` or the `"-mp"` options.



Technical Info 7.6 How Grammars are Imported and Extended

When a grammar shall be extended, the compiler searches the grammar path (specified with the `"-g"` or the `"-mp"` options). These paths need 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 Maven [Fou17a] 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 7.7). In this example, the language `HierarchicalAutomaton` extends the language `Automaton`. This way, the new grammar `HierarchicalAutomaton` inherits all productions defined in the original grammar `Automaton`. 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 HierarchicalAutomaton extends Automaton { MCG HierarchicalAutomaton
2
3   // keep the old start
4   start Automaton;
5
6   // redefine a nonterminal
7   State = "state" Name
8     ( "<<" ["initial"] ">>" | "<<" ["final"] ">>" ) *
9     ( ";" | "{" (State | Transition)* "}" );
10 }
```

Listing 7.7: Language inheritance: One grammar extending another and redefining an inherited nonterminal

The new parser normally 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 repeat the original nonterminal using the `start` statement as shown above.

7.4.1 Redefining Productions of Grammars

A production for nonterminal NT is redefined by (a) either defining a new production for the same nonterminal NT (cf. Listing 7.7) 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 shadows the original production for NT. The original is completely replaced. This is the case for the nonterminal `State` as the grammar `Automaton` already defined the nonterminal `State`, but the grammar `HierarchicalAutomaton` has a production redefining this nonterminal.

For the abstract syntax, there will be a new `ASTState` class defined that implements 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 7.8).

```

1
2 package hierarchicalautomaton._ast;
3
4 public class ASTState extends automaton._ast.ASTState
5     implements ASTHierarchicalAutomatonNode {
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 }

```

Java «gen» ASTState

Listing 7.8: The new ASTState class extends the old ASTState class and serves as 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 7.8 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 original

and extended language. 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 4.3). 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.

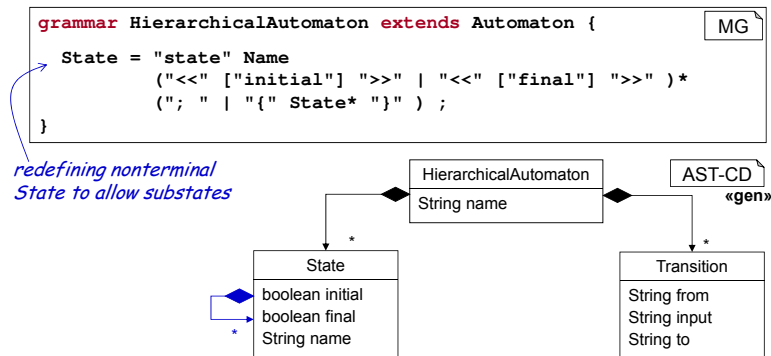


Figure 7.9: Language inheritance

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 later 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 Multiple 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 7.10). Here, the grammar `Automaton3` extends the grammars `InvAutomaton` and `Expression`. Different from the example in Listing 7.4, we this time reuse an existing language for invariants.

```
1 grammar Automaton3 extends InvAutomaton, Expression { MCG Automaton3
2
3   // LogicExpr is defined in grammar Expression and now
4   // bound to the external NT
5   Invariant = LogicExpr;
6
7 }
```

Listing 7.10: 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 extend 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 `InvAutomaton` and `Expression` define the same nonterminal, the definition in grammar `InvAutomaton` 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 are belonging 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 abstract syntax, context conditions or any other elements of these languages before.

We can assume that this is the case for the languages `Expression` and `InvAutomaton` 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 `InvAutomaton` (Listing 7.2) 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 5 of Listing 7.10, the extension point `Invariant` of the imported grammar `InvAutomaton` 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 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 *builder mill*, called `G1Mill`, as discussed in Section 5.9. The builder 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 builder 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.

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 builder mill `G1Mill` has to be initialized accordingly. For thus purpose, each builder 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 form `G1Builder.sBuilder()` deliver `g2._ast.ASTSBuilder` objects.

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 program start.

If the extension of nonterminal `S` is AST-conservative (see Section 7.8), 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 new attributes are all 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

A parser is either directly called or a parse object is created using the static methods provided by a parser class. Each sublanguage `G1` has its own parser in class `G1Parser`. If `G2` extends `G1`, then a class `G2Parser` exists that provides a `G2` parser as well as a parser for each nonterminal of `G1` and `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). But this is hidden from the developer of functions for a sublanguage like `G1`.

In particular, MontiCore provides the parsing functions in such a way that all parsing functions for a sublanguage `G1` are still available in class `G1Parser` even when embedded in a composed language `G2`. The static delegator that implements `G1Parser`, however, is reconfigured in such a way that (1) internally the parsing functions of the composed language `G2` are used and (2) the resulting AST is also built from the AST classes of the composed language `G2`, which, however, reuses `G1` nonterminals if the nonterminals are not redefined.

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 the functionality using parser, builder, and the later discussed visitors of a sublanguage, need not to be aware of its embedding 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

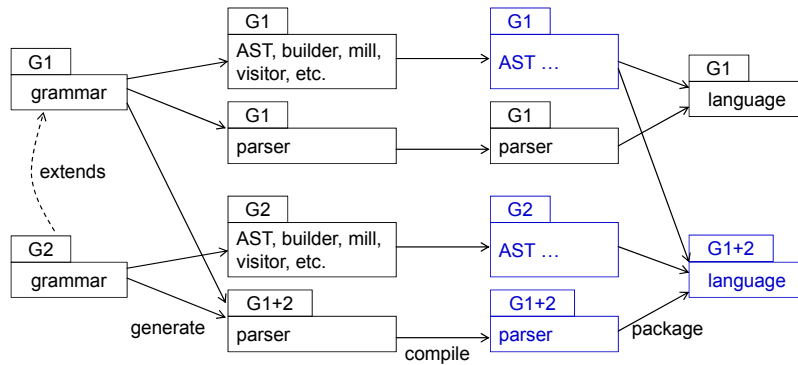


Figure 7.11: Composition of a language is executed as late as possible: late binding

precaution necessary to ensure that type incompatibilities do not prevent compilation of the composed AST classes.

Figure 7.11 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 didn't 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 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 any grammar.

```

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

7.8.1 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.

```

1  grammar LG* extends LG1 {
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 }
```

MCG fragments

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).

²Let us denote the language, i.e. the set of words, of a grammar L by $Sem(L)$

```

1 grammar LG* extends LG1 {
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 }

```

MCG fragments

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, MontiCore 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.

7.8.2 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 7.8.4.

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 {
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.

```

MCG fragments

```

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 7.8.4, how that can be handled.

Changing the order of the nonterminals, like in LG21, also is generally access-conservative, because the AST does not store the order.

7.8.3 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 und thus work closely together. However, there are two important differences: Builders and cardinalities of nonterminals.

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.

7.8.4 AST Signatures Causing Java Type Errors

The considerations in Sections 7.8.1 and 7.8.2 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	M_{LG2} = Decimal;	3	N_{LG2} = "one" M_{LG2} ;	4	P_{LG2} = "some" M_{LG2}^* ;	5	Q_{LG2} = "optional" $M_{LG2}^?$;	6	}
					// ok		// does not compile		// does not compile		

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` 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 {
2      M = Decimal;
3      N = "one" M;
4      P = "some" MStar;
5      MStar = M*;
6      Q = "optional" MOpt;
7      MOpt = M?;
8  }
```

MCG fragments

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 {
2      M = Decimal;
3      P = "some" m2:M*;
4      Q = "optional" m2:M?;
5  }
```

MCG fragments

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 by Robert Heim

Processing a model requires to implement operations on the AST of the model. Since different operations often 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 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.

The original visitor pattern [GHJV94] describes the traversal algorithm as part of the data structure. For AST processing this prohibits adjusting traversal in specific AST visitors as they all share the same (generated) AST implementation. Hence, a variant 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 imperative¹.

The following sections describe MontiCore's visitor infrastructure and how it enables agile development of concrete visitors to process ASTs.

8.1 Visitor Interface of 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 interface that consists of a default traversal algorithm for the AST and *visit* methods serving as hook points for custom processing of specific AST nodes.

¹The described visitor infrastructure bases on [HMSNRW16b] where a detailed elaboration can be found.

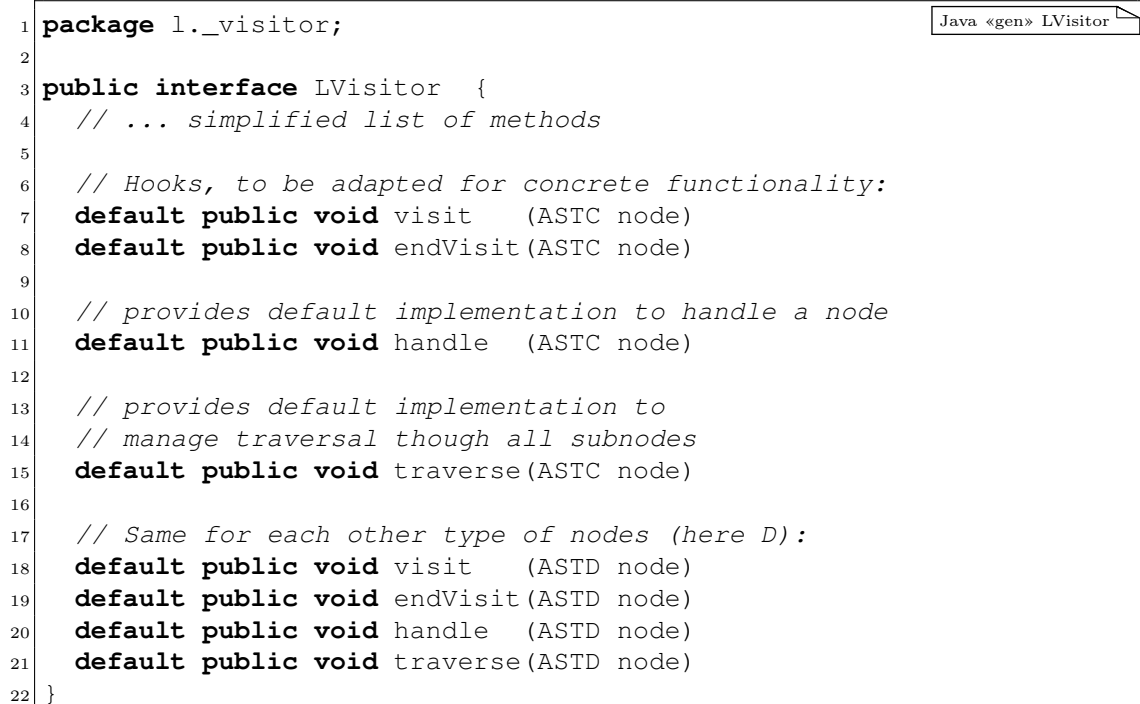
8. Visitors for AST Traversal

The *visitor interface* of a language L is an interface that defines 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. order of 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.

Listing 8.1 shows an excerpt of the generated interface for language L with nonterminals C and D :

```
1 package l._visitor;
2
3 public interface LVisitor {
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 subnodes
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 8.1: Signature of a Visitor for language L

Interface types as well as abstract classes of the AST usually do not specify their own children in a complete form (only their specific realization classes do). Hence, the visitor interfaces provides the same methods but omits the `traverse` method.

All methods introduced for the visitor interface define default implementations. Hence, the language engineer only overrides the default implementations of interest to implement a visitor. Most commonly these are some of the `visit` and `endVisit` methods. Their



Tip 8.2 Standard Traversal of Language L

For a given language L, it is common to use the `LVisitor` or the (later explained) `LInheritanceVisitor` to process the AST.

It is rarely required to adjust the default depth-first traversal. Usually, it is sufficient to override the `visit` methods.



Tip 8.3 Four Visitors for each Language L

MontiCore generates four visitors for each language L. They are explained in the following and are generated to these locations:

```
1 Directory: out/
2 Files:    l._visitor.LVisitor.java
3           l._visitor.LInheritanceVisitor.java
4           l._visitor.LDelegatorVisitor.java
5           l._visitor.LParentAwareVisitor.java
```

All visitors are meant for extension by subclassing. If the visitor is an interface, it provides default implementations for its methods.

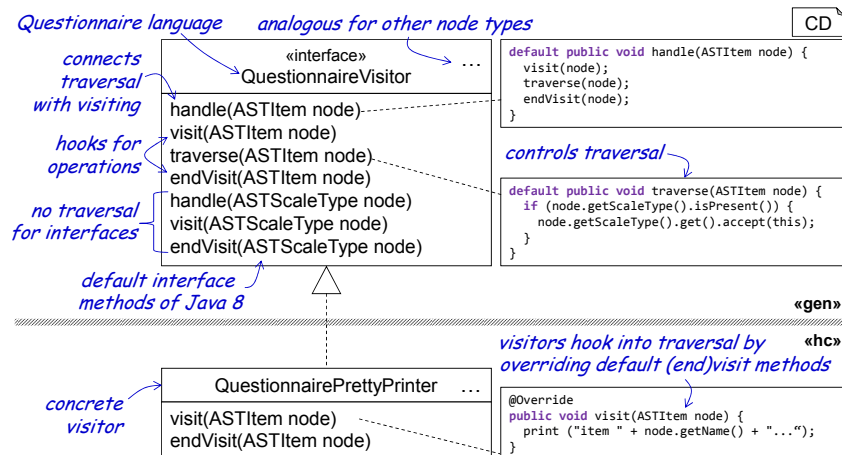


Figure 8.5: Visitor for the `Questionnaire` language and an hardcoded usage

default implementations do nothing while `handle` and `traverse` provide the default traversal algorithm for the AST.

It is worth noticing that the standard visitor interface implements the `handle` method as shown in Listing 8.6 (shown in a slightly simplified representation, ignoring composability for now).

Figure 8.5 shows an example of the visitor for the `Questionnaire` language described in Section 18.6. A concrete visitor (cf. `QuestionnairePrettyPrinter` at the bottom of Figure 8.5) implements the visitor interface and inherits the default methods including



Technical Info 8.4 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 runtime type. For example, a variable could be typed by an interface type, but its value might be of a class type that implements the interface. Here, the static type is the interface type, but the dynamic type during runtime is the concrete class 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 means that the method implementation of the dynamic runtime type of `this` is selected (this enables overriding methods since the more specific overriding version is executed during runtime). However, the method selection does not take into account the dynamic type of the other arguments. This should not be confused with *method overloading* which describes the method selection based on the static type information of arguments available during compile time. The latter choice does not change even if at runtime an argument of a more specific type is given.

During traversal, a visitor gets a node to traverse. Here, the node does only provide the static type information of its children (e.g., a child could be of an interface type). However, the visitor requires knowledge about the dynamic node type of the children to determine how to traverse them. It must call the appropriate handle method for the dynamic type of each child. One solution is to use type introspection (so called *reflection*), but it is (a) slow and (b) breaks the type-safety of the implementation.

It is possible to simulate dynamic dispatching of *one* argument (in addition to the `this`-reference) by adding `accept(LVisitor)` methods within each AST node of language `L`. Instead of directly calling the `handle` method of a node's child, the child's `accept` method is called with `this` (the visitor instance) as argument (cf. Figure 8.5). The `accept` method then simply calls back the appropriate `handle` method by using the child's specific type. This is possible since the child itself does know its specific type. Note, that the actual `handle` method is selected based on method overloading using the static type information available in the child. This pattern falls short during 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 other infrastructure is required. More details can be found in other literature (e.g., [HMSNRW16b]).

the traversal algorithms. By overriding `visit` or `endVisit` methods it can hook into relevant parts during the traversal and implement operations. Executing a concrete visitor is as simple as calling the `handle` method of the visitor 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 (s. Technical Info 8.4).

```

1 public interface LVisitor {
2     // for each kind of nodes C:
3     default public void handle(ASTC node) {
4         visit(node);
5         traverse(node);
6         endVisit(node);
7     }
8 }

```

Java «gen» LVisitor

Listing 8.6: Simplified presentation of a handle operation (omitting composability)

8.2 Predefined Visitor Variants

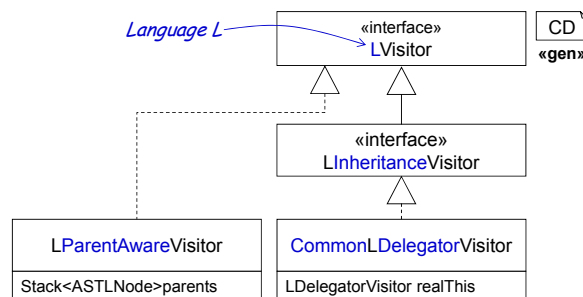


Figure 8.7: Generated visitors for language L extend/implement the default visitor

There exist several extensions to the default simple visitors with different options of re-definition (and complexity). Figure 8.7 summarizes the variants. At the top is the simple visitor interface and its extensions are described in the following.

8.2.1 Inheritance Visitor



Tip 8.8 Inheritance Visitors should be Preferred

It is common that a concrete visitor implements the `LInheritanceVisitor` 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 visitor is generated and we recommend to use it. This is robust if the language evolves.

The default simple visitor 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. An example could be counting occurrences of the super type.

To this effect, the inheritance visitor (s. Figure 8.10) does not only visit nodes in their most specific type (here `ASTRange`) but also calls the `visit` methods of all their super types. In Figure 8.10 and also the more complete Listing 8.9, the `handle` method for `ASTRange` also visits the node in its super type `ASTScaleType` and the other supertypes as defined by the derived AST of the language (top of the figure).

```

1  public interface QuestionnaireInheritanceVisitor {
2      // handle for ASTRange calls four visit methods, because
3      // ASTRange extends/implements
4      //      ASTScaleType, ASTQuestionnaireNode, ASTNode:
5      default public void handle(ASTRange node) {
6          visit((ASTNode) node);
7          visit((ASTQuestionnaireNode) node);
8          visit((ASTScaleType) node);
9          visit(node);
10         traverse(node);
11         endVisit(node);
12         endVisit((ASTScaleType) node);
13         endVisit((ASTQuestionnaireNode) node);
14         endVisit((ASTNode) node);
15     }
16 }
17

```

Java «gen» QuestionnaireInheritanceVisitor

Listing 8.9: Implementation of an inheritance visitor `handle` method

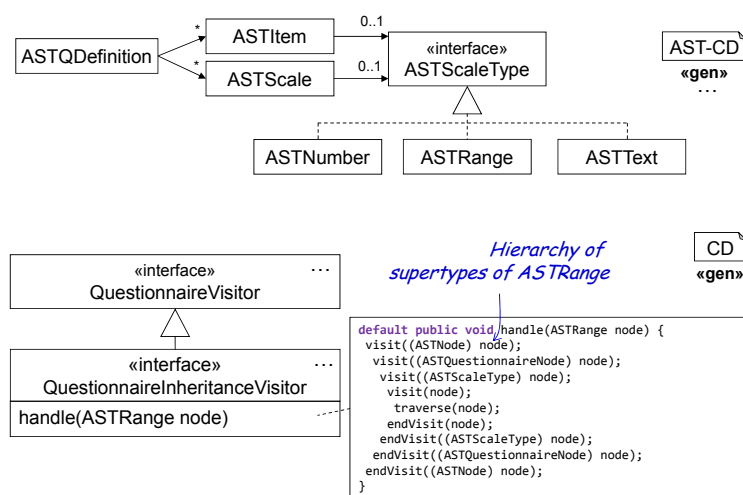


Figure 8.10: Inheritance visitor calls `visit` hooks for all super types

8.2.2 Delegator Visitor

For a language composition (s. Chapter 7) the *delegator visitor* enables reuse of existing visitor implementations of the original languages. To provide such visitor composition all visitors must implement the *realThis* pattern to support *composition of control* (cf. Section 11.2 and [HMSNRW16b]). Because this leads to a complex interaction structure of visitors and visited ASTs, this will be discussed on a detailed example in Section 8.3.



Tip 8.11 Delegator Visitors are used for Composed Languages

Because of the double dispatch that retains static typing, a visitor of an original language needs to be mapped on an AST of a new language that inherits the original language. This is required. The double dispatch cannot be added for new types that are not part of the given (compiled) super languages.

When composing languages, a delegator visitor of the new language wraps the original visitors of the original languages and composes several visitor objects.

As illustrated in Figure 8.7, for a language *L* the generated delegator visitor implementation is given in class *LDelegatorVisitor*. This class includes all relevant implementations for handling the delegation patterns and can be reused by subclassing. It also provides the implementation for the *realThis* pattern used to compose several visitors. See Section 8.3 for composition details.

8.2.3 Parent Aware Visitor

When processing an AST node it is sometimes convenient to have information about its parent and ancestor nodes available. Since the *ASTNode*'s API does not provide such accessors, a dedicated *parent aware* visitor is generated for a given MontiCore grammar. The *ParentAwareVisitor* is an abstract class implementing the language's visitor interface. It keeps track of the parent information and provides an extended API to access it: *getParent* and *getParents*. Both are intended to be invoked from inside the *visit* methods. While *getParent* returns the direct parent of the given node, the *getParents* returns a list of all ancestors beginning with the direct parent. The *getParent* returns an *Optional* value since the root node has no parent.

Figure 8.12 shows the generated *LParentAwareVisitor* for the grammar *L*. The *MyVisitor* is an exemplary concrete visitor that makes use of the parent information by using the *getParent* method from within the overridden *visit (ASTC c)* method.

In order to maintain the ancestors' information, the generated *ParentAwareVisitor* uses a *stack*. Each time before traversing the children of a node, the current node is put onto the stack. Within a *visit* method the direct parent can then be accessed using the *getParent* method that returns the top node from the stack. After traversing the children the current node is removed again.

During runtime the stack is altered at the beginning and end of each *traverse* method. The object diagram (s. Figure 8.12) shows a snapshot of the parents information when

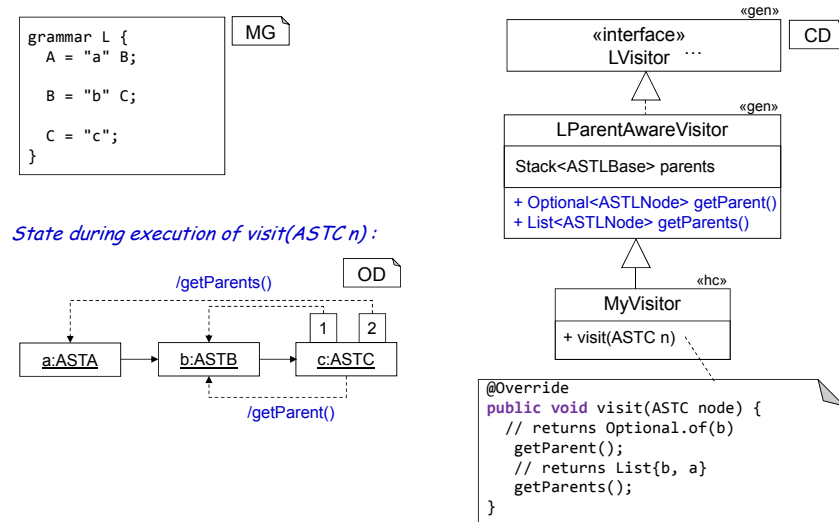


Figure 8.12: Additional API methods of the parent aware visitor

visiting `c:ASTC` node. Here `getParent` returns the direct parent node `b:ASTB` as an instance of `ASTLNode`, and `getParents` returns the list with both ancestors `b` and `a`.

8.3 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 section 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 a case of language aggregation nothing needs to be done. This is different from *language embedding* or *inheritance* where the language elements of the embedded/inherited language are part of the host language. Therefore, these cases are discussed below based on examples.

8.3.1 Visitor 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 almost directly. 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 nonterminal `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 are violated. This needs to be carefully clarified examining the manually implemented behavior of `V1`. At least it is not a type problem,

```

1 grammar Automaton5 extends de.monticore.MCBasics {
2
3   Automaton = "automaton" Name "{" AutElement* "}";
4
5   // The interface allows extension
6   interface AutElement;
7
8   State implements AutElement = "state" Name ";";
9
10  Transition implements AutElement =
11    from:Name "-" input:Name ">" to:Name ";";
12 }

```

Listing 8.13: Automaton language with interface nonterminal `AutElement` used for extension

```

1 grammar Automaton6 extends Automaton5 {
2   start Automaton;
3
4   TransitionWithOutput implements AutElement =
5     from:Name "-" input:Name "/" output:Name ">" to:Name ";";
6 }

```

Listing 8.14: Adding transitions with output to the Automaton5 language of Listing 8.13

because all attributes defined in the original class `lg1._ast.ASTN` are inherited to the new subclass `lg2._ast.ASTN`.

Second, a visitor `V1` for language `LG1` normally cannot be directly applied to the new language `LG2`, because new nonterminals occur, that the old visitor `V1` is not aware of. `V1` has no specific `visit` method for those new nonterminals and does not know how to traverse the new child nodes. Therefore, visitor `V1` needs either to be *subclassed* by the new visitor `V2` or needs to be *embedded* in a new visitor `V2` using the delegation visitor technique. The delegation visitor technique is more general, because it allows to compose several base language visitors instead of only extending a single one [HMSNRW16a].

We demonstrate visitor extension through subclassing in this subsection on the following example for automata and give a variant in the following subsection. In Listing 8.14, l. 1 and 2 the `Automaton6` language conservatively extends `Automaton5` from Listing 8.13. Listing 8.14 then introduces a new form of transition that has an additional output (l. 4). The new transition implements the interface production `AutElement`, enabling to use complex transitions whenever an instance of `AutElement` is required.

The resulting AST data structure is illustrated in Figure 8.15. The AST node `ASTAutElement` is implemented by the new `ASTTransitionWithOutput` node type of `Automaton6`. MontiCore also produces the interface `Automaton6Visitor` for the new language. Generated visitor interfaces extend the visitor interfaces of all superlanguages to inherit their default implementations. Here, the visitor interface `Automaton6Visitor` extends `Automaton5Visitor` and adds default implementations for the new node type

ASTTransitionWithOutput. Using this inheritance relation all default implementations of the original language's visitor interface are reused. Hence, when implementing a visitor for the new language the aggregated default implementations are available. Consequently, the pretty printer for the new language (Automaton6PrettyPrinter) can be implemented by extending the Automaton5PrettyPrinter of the original language (l. 2 of Listing 8.16). It thereby reuses the hardcoded pretty printing for all nodes of the original language and only adds a pretty printing method for transitions with output.

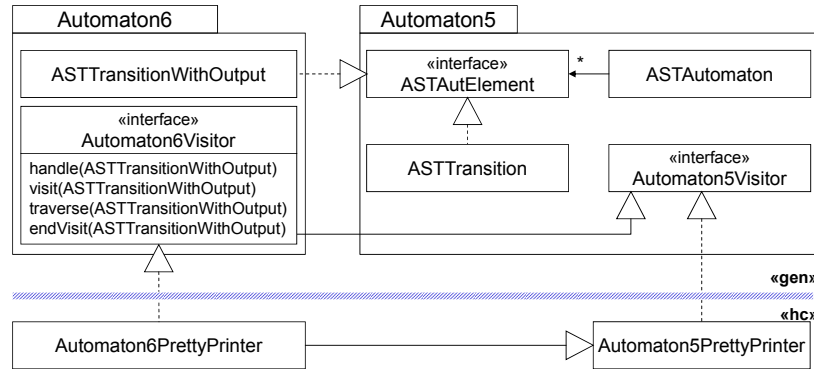


Figure 8.15: Overview of visitor classes for Automaton6

Visitors of the original language are unaware of new AST node classes introduced in inheriting languages. For example, the pretty printer of the Automaton5 language is able to handle ASTTransition nodes, but does not know about ASTTransitionWithOutput.

But ASTTransitionWithOutput is a subclass of the ASTAutElement type and thus still handed to the handle(ASTAutElement) method of the Automaton5PrettyPrinter. ASTAutElement is the most specific type double dispatching in the visitor will identify. Hence, handle(ASTAutElement) would be executed. Furthermore, traversal of children would be omitted, because the visitor's traverse(ASTAutElement) method doesn't know about any children. This is counter intuitive.

```

1 public class Automaton6PrettyPrinter
2     extends Automaton5PrettyPrinter
3     implements Automaton6Visitor {
4     @Override
5     public void visit(ASTTransitionWithOutput node) {
6         print(node.getFrom());
7         print(" - " + node.getInput() + " / " + node.getOutput() + " > ");
8         print(node.getTo());
9         println(";");
10    }
11 }

```

Listing 8.16: A visitor of the new language reuses an original language's visitor

```

1 grammar Automaton15 extends de.monticore.MCBasics {
2
3   Automaton = "automaton" Name "{" (State|Transition)* "}";
4
5   State = "state" Name ";";
6
7   Transition =
8     from:Name "-" input:Name ">" to:Name ";";
9 }

```

Listing 8.17: Automaton language without explicit extension point

Consequently, MontiCore suggests to not directly apply a visitor on any language extension, because the Java compiler's typechecker does not prevent this. MontiCore introduces the convention to only run a visitor on its own language, unless embedded or otherwise adapted to the composed language. This means that a concrete visitor implementation must be adjusted to be reused on an inheriting language, even if nothing changes in the visitor. This, however, is easy and requires minimal glue code. For example, reusing the pretty printer without adding anything is as easy as defining a class similar to Listing 8.16, but with an empty class body. Still the original visitor of the superlanguage would be extended to the sublanguage, because it also implements the visitor interface of the sublanguage.

Please note that defining visitors for *language inheritance* should not be confused with the inheritance visitor. These mechanisms are orthogonal. The latter is used to call visit methods that are defined on super classes of an AST class and can be applied within a language, but also across extended languages.

8.3.2 Visitor for Language Inheritance – an Alternative

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 form, such that the body of nonterminal does not provide surprises (e.g. unset attributes or semantically relevant terminals) for visitors of the original language.

The following example (cf. Listing 8.17) is very similar to the last one. Automaton15 accepts the same models as Automaton5, but instead of explicitly providing an interface for extension, we redefine the Transition nonterminal, such that transitions with output become possible. 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 form of an optional output.

The resulting AST is different from the previous example Automaton6. In particular, two versions of ASTTransition classes exist:

```

1 Directory: out/
2 Generated Files: automaton15/_ast/ASTTransition.java
3                  automaton16/_ast/ASTTransition.java

```

```

1 grammar Automaton16 extends Automaton15 {
2
3   start Automaton;
4
5   Transition =
6     from:Name "-" input:Name ("/" output:Name)? ">" to:Name ";";
7
8 }

```

Listing 8.18: Conservative extension of transitions from Automaton15 of Listing 8.17

```

1 public class Automaton16PrettyPrinter
2     extends Automaton15PrettyPrinter
3     implements Automaton16Visitor {
4
5   @Override
6   public void visit(automaton16._ast.ASTTransition node) {
7     print(node.getFrom() + " - " + node.getInput());
8     if(node.isPresentOutput()) {
9       print(" / " + node.getOutput());
10    }
11    print(" > " + node.getTo());
12    println(";");
13 }

```

Listing 8.19: A visitor of the new language reuses an original language's visitor

The version '16 of `ASTTransition` (cf. Listing 8.18) is furthermore a subclass of version '15. And thus the pretty printer for `Automaton16` is again implemented as visitor and reuses the 15er version of the pretty printer through subclassing (Listing 8.19).

The main difference can be seen in line 7 of Listing 8.19, which accounts for the additional optional attribute output. Line 5 of Listing 8.19 shows an important detail, namely it is a `visit` method for the new version of the `ASTTransition` class. To avoid ambiguities for the parser as well as confusion by the reader, 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.

8.3.3 Visitors for Compositional Language Embedding

Language embedding, especially when multiple languages are involved, enforces composition of existing and reusable visitors based on the composition form of the language. Compositional visitors are often used when composing context condition checks that have been defined on individual sublanguages.

For visitor composition, `MontiCore` combines the visitor pattern with the `realThis` object composition pattern described in Section 11.2. This leads to a powerful, but also complex

structure. It is described in the following based on the pretty printing of a composed language Automaton3, which is already defined in Listing 7.10 on page 94.

We assume that two pretty printers are given in classes ExpressionPrettyPrinter and InvAutomatonPrettyPrinter. Both implement the normal visitor interface. However, they are intentionally prepared for composition. That means the `realThis` pattern is implemented by all visitors (see Listing 8.20):

```

1 public class InvAutomatonPrettyPrinter
2     implements InvAutomatonVisitor {
3
4     InvAutomatonVisitor realThis = this;
5
6     @Override
7     public void setRealThis(InvAutomatonVisitor realThis) {
8         this.realThis = realThis;
9     }
10
11     @Override
12     public InvAutomatonVisitor getRealThis() {
13         return realThis;
14     }
15     // ... more methods
16 }

```

Listing 8.20: The `realThis` implementation of a compositional visitor

Furthermore, to become compositional all methods are implemented in such a way, that they do not use `this` (neither explicitly or implicitly), but use `getRealThis` instead. For example if attributes need to be shared, `getRealThis().getAttribute()` is used. As a comfortable alternative, one can externalize all state of the visitors into a separate object of class `PrettyPrinterIndenter`. In the example, `PrettyPrinterIndenter` does the printing and manages indentation. This class is independent of any AST and can thus be reused in all pretty printers. It is noteworthy, that all generated visitor classes and interfaces implement the `realThis` pattern by default.

The example of language embedding for visitors in Figure 8.21 is based on the example in Listing 7.10. The `ASTInvariant` of the Automaton3 language implements the `ASTInvariantExt` interface of the InvAutomaton language and stores an object of class `ASTLogicExpr`. The visitor interface of the new language extends both original language visitor interfaces to implement the delegator pattern.

Figure 8.21 also shows, that neither the original languages nor their handwritten pretty printers have to be adapted (assuming that they are implementing the `realThis` pattern). The composition of two languages in a *third* grammar Automaton3, however, leads to the definition of *four* visitors:

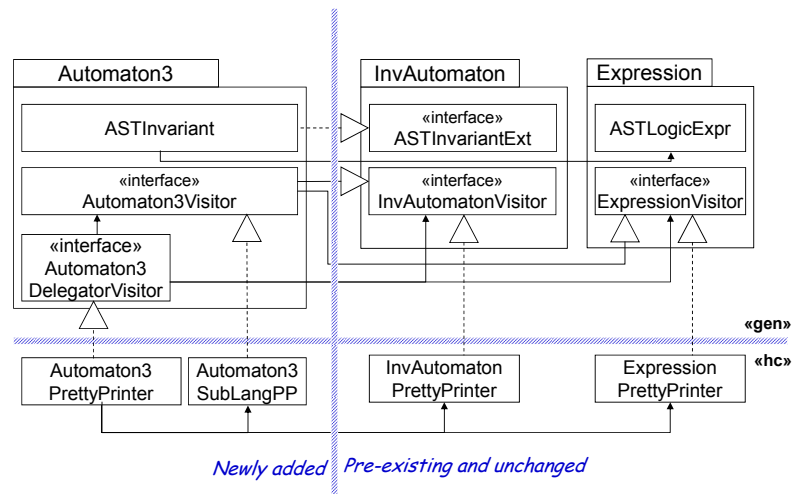


Figure 8.21: Overview of visitors for Automaton3

- Two original visitors are reused directly.
- A rather small visitor for the new nonterminals of grammar Automaton3 is defined in Automaton3SubLangPP.
- The composition is carried out in a subclass Automaton3PrettyPrinter of the Automaton3DelegatorVisitor, whose only purpose is to delegate to the correct pretty printer.

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 8.22 How to Compose Visitors according to their Languages

Composition is technically challenging, but does not enforce much encoding:

1. Use the `realThis` pattern for all visitors as basis for composition
2. Don't care about `handle` or `traversal` at language borders. The composite will do.
3. If visitors have local state that should be shared, either use that state only through `realThis`, or externalize the state into an external object and share its reference.
4. Build a subclass of the generated `DelegatorVisitor` of the composed language and provide the visitors of the sublanguages.

Our composition visitors do not rely on reflection and are type preserving.

Class `Automaton3DelegatorVisitor` provides setters for visitors of all (potentially transitively inherited) original languages. MontiCore uses knowledge of the AST model by generating this delegator visitor in such a way that it delegates all `handle`, `traverse`, and `visit` calls to the concrete original visitor of the language that the current node belongs to. So the delegator visitor makes heavy reuse of all these methods. On the other hand it is important that all visitors implement the `realThis` pattern and thus delegate back to the composite. Please note that even if there is no explicit use of `this` in a handwritten visitor, it inherits methods that use `getRealThis` heavily.

In the example the `Automaton3SublangPP` visitor in Listing 8.23 implements the `realThis` pattern. It also reacts on `ASTInvariant` nodes (lines 5 and 10) to print an additional comment:

```

1  public class Automaton3SublangPP implements Automaton3Visitor {
2
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     }
13
14     // ... more methods
15 }

```

Listing 8.23: The implementation of the pretty printer for the `Automaton3` sublanguage (with only one nonterminal)

Please note that the delegator visitor only delegates the `visit`, `handle` and `traverse` methods with the `ASTInvariant` signature to this visitor, although there would be a lot more of those methods inherited. The same is true for other visitors: Only the nonterminals that are explicitly defined in that language are delegated to the respective visitor.

Subclasses of `Automaton3DelegatorVisitor` basically delegate each `visit`, `handle` and `traverse` call to exactly one of its sub-visitors (cf. Listing 8.24). This delegation behavior is adjustable by extending this delegator visitor and overriding the corresponding methods. Overriding one of those methods in the subclass of the delegator has precedence over the implementation on the sub-visitors. This is shown in Listing 8.24 in line 10.

```

1                                     Java «hw» Automaton3PrettyPrinter
2 public class Automaton3PrettyPrinter
3                                     extends Automaton3DelegatorVisitor {
4
5     // -----
6     // Here we can override any visit, endVisit, ... method
7     // Which gets precedence over any (by default) delegated method:
8
9     @Override
10    public void visit(ASTAutomaton node) {
11        out.println("/* print by composed Automaton3PrettyPrinter */");
12        out.println("automaton " + node.getName() + " {}");
13        out.indent();
14    }
15 }

```

Listing 8.24: Composing the infrastructure for several visitors with option to override behavior delegation

Finally, all visitors are composed using the delegator visitor like in Listing 8.25.

```

1 protected PrettyPrinterIndenter out;                                     Java «hw» Automaton3PrettyPrinter
2
3 public Automaton3PrettyPrinter(PrettyPrinterIndenter o) {
4     out = o;
5
6     // ... configured with three sublanguage visitors
7     setInvAutomatonVisitor(new InvAutomatonPrettyPrinter(o));
8     setExpressionVisitor(new ExpressionPrettyPrinter(o));
9     setAutomaton3Visitor(new Automaton3SublangPP(o));
10 }

```

Listing 8.25: Composing the three visitors through delegation and giving them the same shared state

The statements from lines 7-9 instantiate the sublanguage visitors and directly add them to the composite. Object *o* is handed over to all sub-visitors to share the same state.

The composition always needs to compose visitors of all participating sublanguages. If we omitted a statement then all nodes of that sublanguage as well as all child AST nodes are not visited anymore. For example, omitting the statement in line 9 would lead to a disappearance of the expression part. Omitting the statement in line 7 would give an empty result, because already the top level node of class *Automaton3* would be ignored.

Finally, Listing 8.26 demonstrates how to use the composed visitor.

```
1 // Common storage for all pretty printers
2 PrettyPrinterIndenter ppi = new PrettyPrinterIndenter();
3
4 // The composite visitor
5 Automaton3PrettyPrinter acpp = new Automaton3PrettyPrinter(ppi);
6
7 // run the visitor
8 ast.accept(acpp);
9 System.out.println(ppi.getResult());
10
```

Listing 8.26: The composed visitors can be used as if it is only one monolithic component

In line 3 the shared state is instantiated. Line 6 creates the composite. In l. 9 the visitor is applied to the model and in l. 10 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 composite visitor, when each visitor reuses the realThis pattern for the state as well.

The blueprint shows, that for n languages composed in a $(n+1)$ th grammar, $n+1$ visitors for sublanguages composed are by one more visitor for the overall language. This sums up to $n+2$ visitor objects. Composed visitors are again usable in a composition for even larger languages. Composition of visitors scales up in hierarchies of language compositions.

Chapter 9

Symbol Management Infrastructure

co-authored by Pedram Mir Seyed Nazari

This chapter gives a lightweight introduction to the management of symbols within models, respectively the generated infrastructure for symbol management.

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 only 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 enclosed spanning tree.

The symbol management infrastructure (SMI) is based on the thesis [MSN17] and its predecessor [Völ11]. The chapter therefore only gives a lightweight overview of the available mechanisms and concepts and refers to these two theses for further details.

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. 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).

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.

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*.

A *symbol* is an abstraction of the model entity and contains obviously the name that is defined, but also some 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, a name usage refers to, so-called *symbol tables* are employed. In general, 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.

A *scope* typically is defined by a nonterminal with a body that has a form of brackets and nonterminals that may introduce names. The brackets for the visible containment structure and the nonterminals may introduce new names that are then potentially restricted to the enclosing scope. Please note, that a nonterminal may introduce several scopes, but in practice it is sufficient to assist one scope, because otherwise the grammar could be slightly refactored.

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 is eventually spread among one or several sources, for example for a class the list of attributes including inherited attributes.
3. It acts as a *surrogate* for the entity definition in other models that use the symbol (via its name), thus preventing to having to load the complete model, but the symbol table only.

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, also a link to the definition is 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.

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: method name, modifiers, return type and parameter 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.



Tip 9.1 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.

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 has a size 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.

Scopes contain symbol definitions, but also import and export symbols. Each symbol is defined in exactly one scope.

The SMI of MontiCore provides reasonable defaults to facilitate developing of language-specific symbol tables. Figure 9.2 gives an overview of the main elements (conceptually and technically) of the SMI, which are introduced in the remainder of this chapter based on concepts of the Automaton language (see Section 18.1) as well as concepts of the Java language.

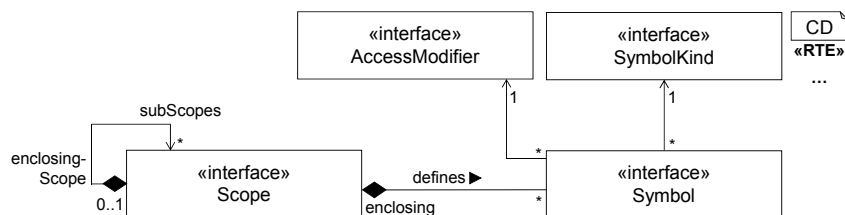


Figure 9.2: Overview of the main concepts of SMI

We might add the observation that when symbols are imported in heterogeneous language environments, they might change their kind, name or other essential information. For example a state `Ping` of an automaton might become an enum value `Ping` or alternatively a boolean method `isInPing()` in Java. An attribute `a` of a class diagram may also map to two symbols `getA()` and `setA(A)` when mapped to a Java interface.

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 Automaton grammar, shown in Listing 9.3 that also contains a reduced form of expressions and statements.

```

1 grammar Automaton extends de.monticore.MCNumbers {
2     symbol scope Automaton =
3         "automaton" Name "{" (State | Transition)* "}" ;
4
5     symbol scope State =
6         "state" Name
7         (( "<<" ["initial"] ">>" ) | ( "<<" ["final"] ">>" )) *
8         ( ( "{" (State | Transition | Counter)* "}" ) | ";" ) ;
9
10    Transition =
11        source:Name@State "-" input:Name ( "|" Statement )? ">"
12            target:Name@State ";" ;
13
14    symbol Counter = "counter" Name "=" Decimal ";" ;
15
16    interface Statement ;
17    Print implements Statement = "?" Name ;
18    Increment implements Statement = "++" Name ;
19 }

```

Listing 9.3: Automaton with counters and transition statements

The grammar in Listing 9.3 identifies three kinds of symbols: each `State` has a name (l. 5), a `Counter` has a name (l. 14) and the `Automaton` also has a name (l. 2). Therefore, each of the respective nonterminals has the keyword `symbol` attached.

The `symbol` can only be attached to nonterminals that introduce a *name* on the right-hand side, which technically means that nonterminal `Name` or equivalently `name:Name` is used. `Name` has to be mandatory, i.e., neither optional nor in a list. Furthermore, it must not have a different name, i.e., `b:Name` would not work.

9.2.1 Generated Classes For Symbols

If the keyword `symbol` is used, the generator creates an additional subdirectory (package) `_symboltable` with several classes that allow to manage symbols. For each kind of symbol, we get three classes (here demonstrated on the nonterminal `State`):

StateSymbol.java shall contain the relevant information about the symbol. The generator includes the *name* and the *kind* as well as a link to the defining *AST node*.

StateKind.java defines the symbol kind. Each symbol in the grammar automatically introduces a new kind. The symbol kind allows to distinguish different symbols, even if they have the same name.

StateResolvingFilter.java is used to find `State` symbols (and will be discussed later).


```

1 package automaton._symboltable;
2
3 public class StateKind implements SymbolKind {
4     private static final String NAME =
5         "automaton._symboltable.StateKind";
6
7     public String getName();
8     public boolean isKindOf(SymbolKind kind);
9 }

```

Java «gen» StateKind

Listing 9.4: Generated implementation of class StateKind

StateKind as shown in Listing 9.4 is a lightweight class mainly used to identify kinds. It can be used as generated. StateSymbol, in contrast, is a candidate that needs a specific extension with additional attributes that contain the above-mentioned *relevant* information. What the relevant information of a symbol is, highly depends on the form of symbol and where and how it shall be used. Attributes for example also need a type, while methods even need a signature. In the above example we could store the initial value of a counter in the counter symbol for potential re-initialization.

```

1 package automaton._symboltable;
2
3 public class StateSymbol extends CommonSymbol {
4
5     public static final StateKind KIND;
6
7     // the are the adapted methods:
8     public StateSymbol(String name);
9     public Optional<ASTState> getStateNode();
10
11 }

```

Java «gen» StateSymbol

Listing 9.5: Generated implementation of class StateSymbol

Extension of generated classes can again, e.g., be achieved via the TOP mechanism described in Section 14.3 or by building subclasses.

9.2.2 RTE Classes For Symbols

Figure 9.6 (top part) depicts three classes and interfaces that are provided by the SMI runtime environment.

The Symbol 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`. Additionally, every symbol has a kind (e.g., kind *method*, *state*, etc.) represented by the SymbolKind interface. Listing 9.7 shows the signature that a Symbol class always provides.

9. Symbol Management Infrastructure

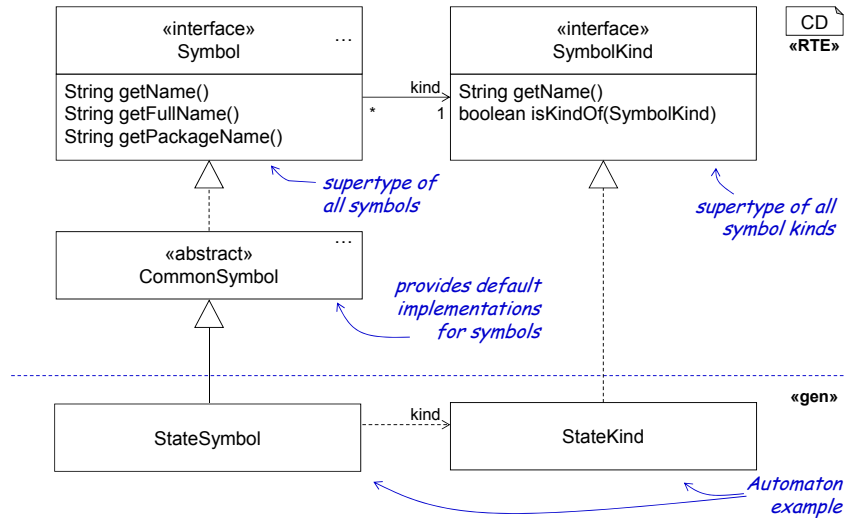


Figure 9.6: Technical symbol classes provided by SMI plus language specific classes

```

1 public interface Symbol {
2     // the name:
3     String getName();
4     String getPackageName();
5     String getFullName();
6
7     // the kind:
8     SymbolKind getKind();
9     boolean isKindOf(SymbolKind kind);
10
11     // connection to the AST defining node:
12     void setAstNode(ASTNode node);
13     Optional<ASTNode> getAstNode();
14     SourcePosition getSourcePosition();
15
16     // a symbol knows the scope it is defined in:
17     Scope getEnclosingScope();
18     void setEnclosingScope(MutableScope scope);
19 }

```

Java «gen» Symbol

Listing 9.7: Interface of all Symbol classes

As depicted in Listing 9.7, the generated classes extend the RTE classes accordingly.

Each symbol has an unchangeable name and kind. The package name however is optional and may be empty. If the AST defining the symbol is available, then a navigation from symbol to AST is possible. Furthermore, each symbol is defined within a scope (see discussion in Section 9.3), i.e. its enclosing scope.

CommonSymbol from Figure 9.6 provides default implementations for a symbol based on concepts of widespread languages such as Java. As a general rule, Symbol should

not be implemented directly. Instead, `CommonSymbol` can be subclassed and adapted by overriding its methods. A language-specific symbol should provide all essential information about the model element it denotes and that way facilitate its processing.

The bottom part of Figure 9.6 illustrates the symbol classes by the example of a state symbol. The class `StateSymbol` implements a state symbol and provides useful information, e.g., if the state is initial (`isInitial()`) or final (`isFinal()`). What we regard as relevant information depends on the form of use of the symbols. In the `StateSymbol` case, we might for example be interested to know, which input the state can react to.

The state kind is represented by the class `StateKind`. Since every state symbol has the same kind, a static constant `StateKind KIND` in `StateSymbol` should be defined which simplifies usage of the resolution mechanism.

The AST itself is not changed. However, the `ASTNode` interface provides three methods `setSymbol`, `getSymbol` and `symbolIsPresent` to handle navigation from an AST node to a related symbol.

9.3 Defining Scopes

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 form of the language. The generator, therefore, is only able to provide a general infrastructure for symbol management, but it is expected that the scope management is extended by handwritten code or other configuration mechanisms to deal with individual forms of visibility.

The keyword `scope` attached to a nonterminal production introduces a type of scope. Each time, the nonterminal is instantiated in the AST, an according scope instance is built as well. We identify two types of scopes in the example grammar in Listing 9.3, namely the `Automaton` (l. 2) and the `State` scope (l. 5). Although it is often the case that scopes defines symbols as well, in the example above it is just a coincidence. Anonymous scopes would be possible as well. Java statement blocks are such an example. The keyword `scope` can therefore be attached to any nonterminal, but it only makes sense, if there actually are symbols that can be enclosed in a sub-scope.

9.3.1 Generated Classes For Scopes

If a nonterminal is marked with keyword `scope`, then a scope class is generated. Listing 9.8 shows the a part of the `StateScope` class.

The generated scope class itself, however, only delegates to the inherited methods. Thus, only the constructors need to be generated and the rest is handled by the RTE implementation.

```

1 package automaton._symboltable;
2
3 public class StateScope extends CommonScope {
4     public StateScope() {
5         super();
6     }
7     public StateScope(boolean isShadowingScope) {
8         super(isShadowingScope);
9     }
10    public StateScope(Optional<MutableScope> enclosingScope) {
11        super(enclosingScope, true);
12    }
13 }

```

Java «gen» StateScope

Listing 9.8: Generated implementation of class StateScope

9.3.2 RTE Classes For Scopes

Listing 9.9 shows the most important signatures that the CommonScope provides and actually implements. The methods allow to add new symbols and most importantly also allow to resolve symbols based on either the name or the name and extra information. Actually resolution is a highly complex process that depends on visibility as well as the hierarchical structure of scopes that is normally structurally equivalent to the AST, but also allows importing symbols from foreign scopes when desired.

```

1 public class CommonScope implements MutableScope {
2
3     // Symbols:
4     public void add(Symbol symbol);
5     public void remove(Symbol symbol);
6     public <T extends Symbol> Optional<T> resolve(...);
7     public <T extends Symbol> Collection<T> resolveMany(...);
8
9     // Scopes:
10    public Optional<MutableScope> getEnclosingScope();
11    public List<MutableScope> getSubScopes();
12    public void addSubScope(MutableScope subScope);
13 }

```

Java «RTE» CommonScope

Listing 9.9: Signature of MutableScopes

Scopes within a model can be *structured hierarchically* which leads to a *scope tree*. The methods `getEnclosingScope` and `getSubScopes` (see association in Figure 9.2) allow to access the enclosing scope and the subscopes, respectively. The only exception to this rule is the `GlobalScope`, which also exists predefined in the RTE. It resembles the top-level scope and can be used to manage symbol resolution across artifacts, i.e., in language aggregation.

Figure 9.10 shows the structure for scopes provided by the SMI. In case of a shadow-

ing scope (where local names shadow imported names), `isShadowingScope` returns true. To configure whether a scope is a shadowing or visibility scope, the constructor `CommonScope(boolean isShadowingScope)` is provided. Symbols are visible from outside their enclosing scope, if they are exported which is stated via method `exportsSymbols`. By default, this holds true for scopes. The reason is that, for example, a static field `f` defined in a class (scope) `C` can be accessed through `C.f`.

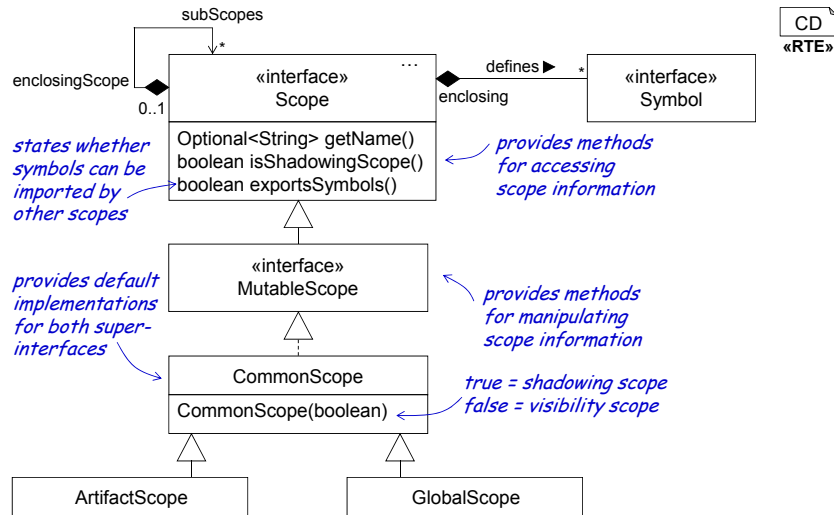


Figure 9.10: Technical scope classes provided by SMI

The interface `MutableScope` depicted in Figure 9.10 provides methods for manipulating a scope, such as via method `setEnclosingScope`. Moreover, `MutableScope` enables to customize the resolution process. This is only required if the default resolution is not sufficient. Please refer to [MSN17] for a detailed discussion on this.

Since `CommonScope` implements `MutableScope`, each scope is mutable. This is in particular important for the symbol table creation.

9.3.3 Artifact Scope and Global Scope

Due to a modular structure of the many artifacts that are typically involved in a development project, the SMI provides two standard scopes the *artifact* and the *global scope*.

The *artifact scope* represents the scope of the artifact (or compilation unit). It is the top scope of a model and by default a shadowing scope. Artifact scopes can contain additional information such as a package name and import statements. This information is, among others, important for name qualifying and inter-model references. In SMI, artifact scopes are realized in the `ArtifactScope` class which subclasses `CommonScope` (see Figure 9.10).

The *global scope* is the root of the scope graph, thus it is the direct or indirect enclosing scope of all scopes. Its direct subsopes are artifact scopes since they are the top scopes of the models. That way, the global scope “connects” models and enables inter-model

references. 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. `GlobalScope` also extends the `CommonScope` class (see Figure 9.10).



Tip 9.11 Navigation from Symbol Usage to Symbol Definition

There are several ways to establish navigation from 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 makes sense to establish a direct link in 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 completely bypassed.

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 form of handwritten Java code (see Chapter 14).

9.4 Collaboration between AST, Symbol, and Scope

Figure 9.12 illustrates the relation between AST nodes, symbols and scopes. 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 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. Hence, a symbol can exist without a corresponding AST node, which is usually the case when an external symbol table is loaded without loading the full model.

Both, AST nodes and symbols, can be defined in a scope, namely the enclosing scope. For example, `ASTState` and `StateSymbol` are defined in either an `AutomatonScope` or a `StateScope`.

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 state names are used in transitions. Actually in complex situations

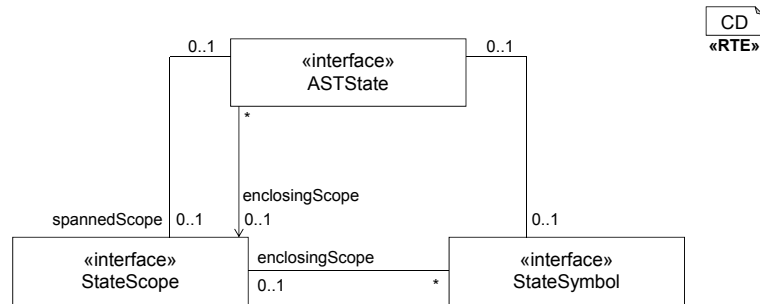


Figure 9.12: Relationship between AST, symbol, and scope

additional information may be necessary. Method overloading, e.g., can only be resolved, because method name and argument types are available.

In a grammar it is possible to extend nonterminal `Name` by a suffix, explaining to which kind of symbol the name refers to. In Listing 9.3, lines 10f a transition has a `source State` and a `target State`. Therefore, additional methods are generated to retrieve the symbols which the names are referring to in Listing 9.13.

```

1 public class ASTTransition ...
2 {
3     // additionally generated methods, because of
4     // Transition = source:Name@State ... target:Name@State
5
6     // Retrieving the symbol:
7     public Optional<StateSymbol> getSourceSymbol();
8     public Optional<StateSymbol> getTargetSymbol();
9
10    // Directly navigation to the definition of the symbol in the AST
11    // (if the defining model is loaded):
12    public Optional<ASTState> getSourceNode();
13    public Optional<ASTState> getTargetNode();
14 }

```

Listing 9.13: Extended signature of ASTTransitions

As a convenient shortcut, it is also possible to directly retrieve the `ASTNode`, where the mentioned symbol is defined, in case the defining model is loaded. In the automaton example, states are defined within the same model and, therefore, navigation is always possible.

9.6 Instantiating Symbol Tables

While the infrastructure can be relatively well generated, the actual instantiation of the symbols as well as the establishment of all links described above needs additional code that can be executed after the parsing process is done. Furthermore, if names are used in

a model, where the corresponding referenced symbols neither exists within the model nor within imported symbol tables, respective error messages have to be defined.

Because of the wide variety of possibilities, how to define and look up symbols, it is generally expected that handwritten extensions are necessary. Only in relatively simple and straightforward cases, such as flat scopes, or hierarchical scopes with standard visibility, the generated scope classes as well as the provided RTE classes can be used directly. For detailed discussions, see [MSN17].

9.6.1 RTE Classes For Symbol Table Building

To build up a model's symbol table, SMI provides the interface `SymbolTableCreator` and its default implementation `CommonSymbolTableCreator`. The symbol table creator is employed in combination with the *Visitor Pattern* (see Chapter 8) to build the symbol table.

Symbol table creators for specific languages should subclass `CommonSymbolTableCreator` and additionally provide a `createFromAST` method with an appropriate `ASTNode` type as parameter. This method is the entry point to create the symbol table starting from the top AST node.

9.6.2 Generated Classes For Symbol Table Building

For each grammar, MontiCore generates five classes that deal with the overall management of symbol tables:

AutomatonLanguage contains the configuration for the automaton language, namely the parser, the creator for symbol tables, the source, where to load other models, and the list of filters necessary to resolve symbols in a scope.

AutomatonModelLoader knows where to find additional models un which external symbols can be found. In the automaton example, external symbols are not used and, therefore, nothing happens here.

AutomatonModelNameCalculator is a helper for symbol resolution.

AutomatonSymbolMill establishes the builder concept for symbols very similar to the builder concept for the AST nodes.

AutomatonSymbolTableCreator contains a generated version of method `createFromAST`. It realizes a visitor for automaton AST nodes, that uses traversal and visit methods to identify all places where symbols are defined or used and allows to handle them respectively.

Again, the generated classes can and should be extended by handwritten code to actually implement the appropriate symbol infrastructure management.

Chapter 10

Context Conditions

co-authored by 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 forms 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 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 AST only. Also, context conditions can again be checked after transforming the AST to ensure that the AST still represents a valid model.

For example, the Automaton language (cf. Section 18.1) 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 MontiCore's infrastructure to implement context conditions for a DSL. Given a grammar (such as the Automaton grammar in Section 18.1), 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 Automaton language this results in three generated interfaces, namely `AutomatonASTStateCoCo`, `AutomatonASTTransitionCoCo` and `AutomatonASTAutomatonCoCo`, each defining a check signature for the corresponding AST node.

For example, MontiCore generates the interface `AutomatonASTStateCoCo` that defines the method signature of the `check` method for nodes of type `ASTState` (s. 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

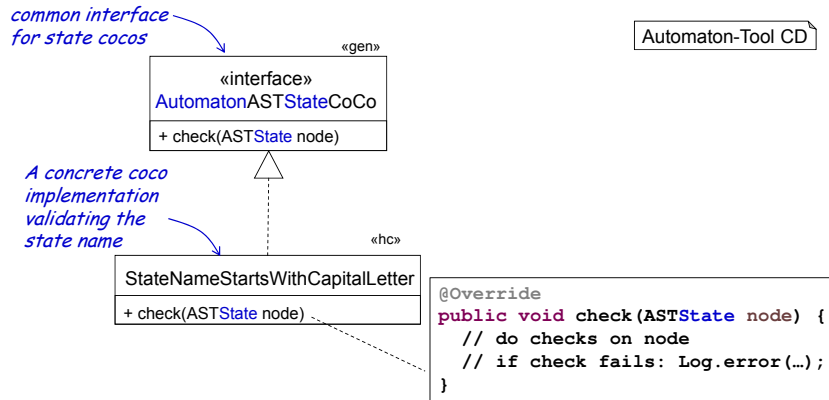


Figure 10.1: CoCo infrastructure for nonterminals

`StateNameStartsWithCapitalLetter` implements the former mentioned interface for state nodes (s. Figure 10.1). Listing 10.6 shows the full 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.

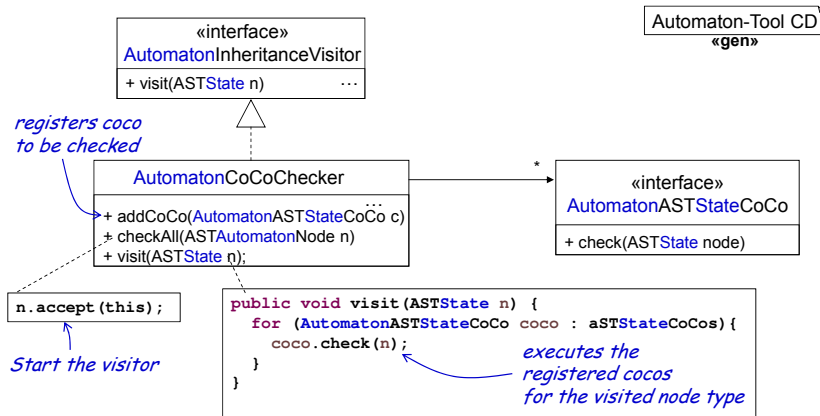


Figure 10.2: The generated CoCo checker

As an example, the Automaton language's checker `AutomatonCoCoChecker` is shown in Figure 10.2. Internally, the checker uses the visitor pattern (cf. Chapter 8) to traverse a given AST node. It implements the `visit` methods to invoke the corresponding registered context conditions on the specific nodes. Hence, in case of the Automaton language

```

1 public class AutomatonCoCoChecker
2     implements AutomatonInheritanceVisitor {
3
4     public void checkAll (ASTAutomatonNode node) {
5         node.accept (getRealThis());
6     }
7
8     private Collection<AutomatonASTStateCoCo> aSTStateCoCos
9         = new HashSet<>();
10
11     public void addCoCo (AutomatonASTStateCoCo coco) {
12         aSTStateCoCos.add (coco);
13     }
14
15     @Override
16     public void visit (ASTState node) {
17         for (AutomatonASTStateCoCo coco : aSTStateCoCos) {
18             coco.check (node);
19         }
20     }
21
22     // ... analog infrastructure for other AST elements ...
23 }

```

Listing 10.3: Implementation of the AutomatonCoCoChecker class

the AutomatonCoCoChecker implements the AutomatonInheritanceVisitor interface. Listing 10.3 shows the implementation of the checkAll method and the methods regarding the states.

Listing 10.4 shows a usage example of the AutomatonCoCoChecker. 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.



Tip 10.5 Fixed Sets of Context Conditions

A language typically has a fixed set of standard context conditions. As a best practice a language implementation should provide a configured checker to language users.

In case of the Automaton language, the handwritten class named AutomatonCoCos implements the method getCheckerForAllCoCos that creates an AutomatonCoCoChecker object, configures it by adding the common context conditions and then returns it. Language users can use this method to obtain a configured checker to check a model for well-formedness.

```

1 // setup context condition infrastructure
2 AutomatonCoCoChecker checker = new AutomatonCoCoChecker();
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 AutomatonCoCoChecker and check the context conditions

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 (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 (s. 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. This ensures that 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

```

1 public class StateNameStartsWithCapitalLetter
2 implements AutomatonASTStateCoCo {
3
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                    "0xAUT02 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

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. 7). Having the enclosing scope, one can try to resolve the source state using its name (`node.getFrom()`, ll. 9f). If a state with the particular name does not exist, this leads to an error (ll. 12f).

```

1 public class TransitionSourceExists
2 implements AutomatonASTTransitionCoCo {
3
4     @Override
5     public void check(ASTTransition node) {
6         checkArgument(node.isPresentEnclosingScope());
7
8         Scope enclosingScope = node.getEnclosingScope();
9         Optional<StateSymbol> sourceState =
10             enclosingScope.resolve(node.getFrom(), StateSymbol.KIND);
11
12        if (!sourceState.isPresent()) {
13            // Issue error...
14            Log.error(
15                "0xAUT03 Source state of transition missing.",
16                node.getSourcePositionStart());
17        }
18    }
19 }

```

Listing 10.8: Using the symbol table in a context condition

```

1 public class TransitionSourceExistsTest {
2
3     // setup the language infrastructure
4     AutomatonLanguage lang = new AutomatonLanguage();
5     AutomatonParser parser = new AutomatonParser();
6
7     @BeforeClass
8     public static void init() {
9         // replace log by a sideeffect free variant
10        LogStub.init();
11    }
12
13    @Before
14    public void setUp() throws RecognitionException, IOException {
15        Log.getFindings().clear();
16    }
17 }

```

Listing 10.9: Initial setup to test 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 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. Two reusable objects are initialized and stored as attributes in l. 3f, because no test modifies them. The `init` method (ll. 8) replaces the normal `Log` 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. 13) – 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
8      Scope modelTopScope = createSymbolTable(lang, ast);
9
10     // setup context condition infrastructure & check
11     AutomatonCoCoChecker checker = new AutomatonCoCoChecker();
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 138). First of all, the model is parsed directly from a string in l. 3f (cf. Chapter 6) and its symbol table is created (s. Chapter 9) in line 8. The context condition is instantiated and added to a checker (ll. 11). Next, the checker is executed on the model (l. 14). Finally, the test verifies that no errors or warnings occurred (l. 16).

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 parsed and its symbol table is created (ll. 3f). This model uses a state `Blubb` that had not been defined.

A checker is configured with the context condition under test and executed on the invalid model. This example expects exactly one error (l. 18) with a given text (l. 19). Checking that all expected findings occurred 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      Scope modelTopScope = createSymbolTable(lang, ast);
10
11     // setup context condition infrastructure & check
12     AutomatonCoCoChecker checker = new AutomatonCoCoChecker();
13     checker.addCoCo(new TransitionSourceExists());
14
15     checker.checkAll(ast);
16
17     // we expect two errors in the findings
18     assertEquals(1, Log.getFindings().size());
19     assertEquals("0xAUT03 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 (0xAUT03), because error messages are relatively often modified.

Chapter 11

Design Pattern Used and Invented for MontiCore

Design pattern [GHJV94] are a helpful concept 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 pattern 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 objects and symbols 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 s 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. 3ff.). The 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.

11. Design Pattern 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. By default Slf4jLog
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 }
```

Java «RTE» Log

Listing 11.1: A static delegator method

```
1 public class LogStub extends Log {
2
3     protected LogStub() { }
4
5     // Initialize the LogStub as Log
6     public static void init() {
7         Log.setLog(new LogStub());
8     }
9
10    // The customized behaviour
11    protected void doInfo(String msg, String logName) {
12        // adapted implementation
13    }
14 }
```

Java «RTE» LogStub

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 14 of Listing 11.1) now delegates to the protected method

`doInfo` (cf. line 11ff.) 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.

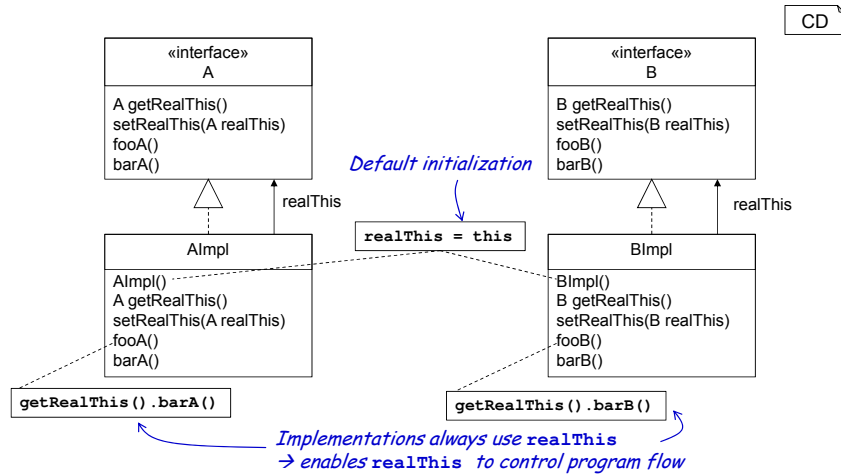
Limitations are also coming with the use of static methods. E.g. web frameworks forbid static methods. If parallel processes want their own individual instantiations, then conflicts between otherwise independent processes may occur and must be managed like described in [Rum17].

In contrast to the delegation pattern in [GHJV94], this pattern is a conjunction of a singleton and delegating methods. Thus, the static methods need not use the same object to delegate to. Instead, there can be multiple objects stored and handled internally.

11.2 RealThis Object Composition Pattern

When generating code, it is often convenient to generate independent artifacts that are, however, tightly integrated during runtime. This is in particular necessary when parts of classes originate from different generators or a class functionality shall be partially handcoded and generated. To allow separated artifacts but integrated runtime elements in Java, individual classes need to be defined, but their instantiated objects need to be closely composed.

C# provides partial classes for that 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 variability in the composition is possible).

Figure 11.3: Components use `realThis` instead of `this` to enable close collaboration

The core idea of the *realThis* pattern is to compose objects that collaborate closely and behave as if they are one single object. As Figure 11.3 shows, the objects to be composed implement a specific program flow. They never use `this` explicitly or implicitly (i.e., by calling another instance method directly, `this` would implicitly be used). Instead the object holds an attribute called `realThis` that references the `this` object instance by default. All explicit and implicit `this` usages are replaced by `realThis`. When delegating everything to `realThis`, the original program execution stays viable since `realThis` references `this` by default. For customization, getters and setters enable to change the `realThis` reference to another object. A composing object, hence, can set itself to be in control (s. Figure 11.4). For composing A and B while keeping C in control of the program flow the respective `realThis` of the composed objects a and b is set to the CImpl instance. The composed object of type C then delegates to the specified `realThis` and thereby hands over control over the program flow. Figure 11.5 shows the impact on the program flow by example. Here, the original behavior is overridden to match a new program flow: In this runtime program flow c gets back in control of the program flow after calling `fooA()`. This enables c to override AImpl's original behavior.

This inversion of control principle can be interpreted as an injection of the `this` reference at runtime to compose functionality. As mentioned, the visitor infrastructure of MontiCore is built on this pattern to enable visitor composition (see Chapter 8).

In general, the *realThis* pattern benefits from enabling late composition of functionality while users would not recognize that they work with a composed object. A drawback is the overhead required during implementation since every explicit and implicit `this` usage must be correctly identified and replaced by `realThis`. Also, during runtime, such composition incorporates many object instances and an increased method delegation stack occurs (cf. Figure 11.5). Furthermore, to overcome the single inheritance limitation of Java, the implementation of each class should be interface driven, meaning, that the signature of all methods should be defined in an interface of the class and the `realThis` attribute should be of the interface type (s. Figure 11.3). This enables a composing object to extend

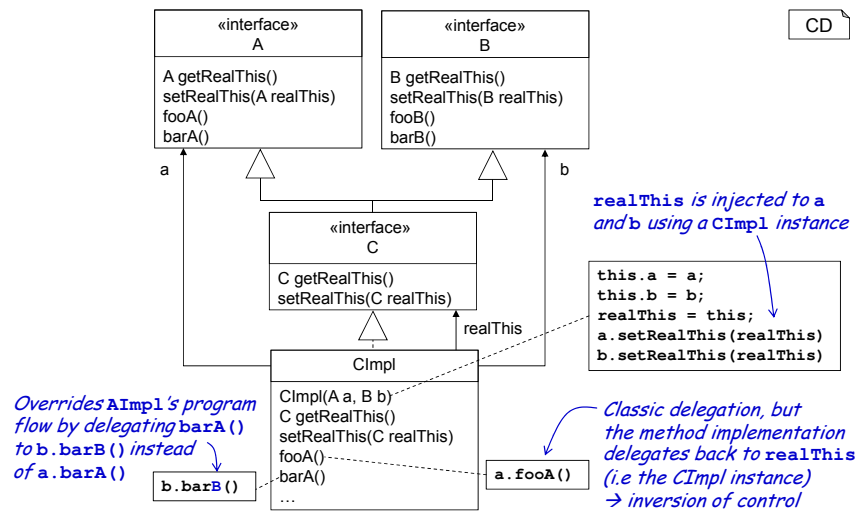


Figure 11.4: Composing objects using the `realThis` approach

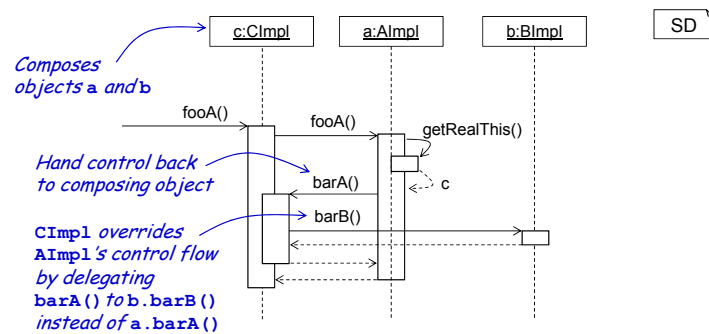


Figure 11.5: Sequence diagram showing the runtime program flow in a `realThis` composition

all interfaces of the composed objects (s. Figure 11.4). Consequently, the composing object can provide itself as `realThis` for each of the composed objects. Thereby, controlling multiple composed objects by one single composing object is possible.

11.3 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 11.6:

- 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).

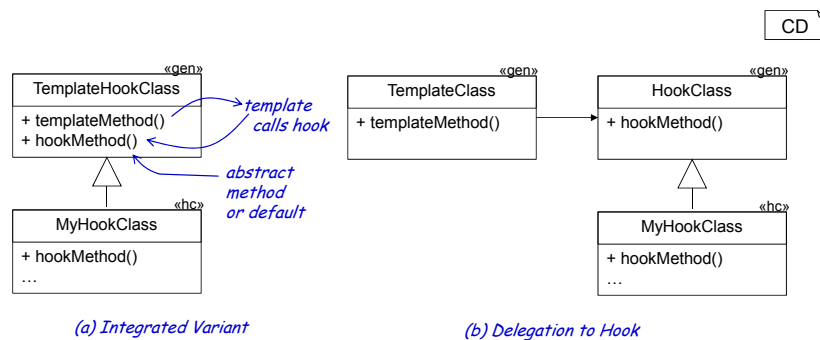


Figure 11.6: Template hook pattern variants for integration of handwritten and generated code

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]. It 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 this technique of templates and hooks to FreeMarker templates using the hook point mechanisms explained in Section 13.5.

Chapter 12

FreeMarker

co-authored by 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 [Fre17], 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 4 currently uses FreeMarker's version 2.3.23¹. FreeMarker's version history is available online².

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 the tutorial [Fre17].

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.

¹Status Nov 2017

²http://freemarker.org/docs/app_versions.html

```

1 <!-- we assume variable ast is pointing towards an AST object
2     containing class information (this is a FreeMarker comment)
3 --#>
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;
2
3 /* Definition of a standard PersonFactory (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}`,
- directives, such as `<#if ...>`, and
- comments like `<!-- ... --#>`


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.

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 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 [Fre17]. 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 12.5. 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
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 Drawbacks

FreeMarker unfortunately has drawbacks: It is completely untyped and heavily relying on reflection, which can lead to bad forms of errors during runtime. The availability of the robust generator framework, however, and that it runs in tools not end-products, were two arguments to still integrate it into MontiCore. Freemarker's template errors still occur at development time and thus are not shipped to end users. Furthermore, we have added some precautions e.g. using a signature statement that add some type safety.

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 [Fre17] 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.
- Several APIs that MontiCore provides for the usage within FreeMarker templates, including AST access, calls of other templates, or adaptation of the call structure between template.

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, many times pretty printing the AST is not sufficient, but additional outputs that are not available in the input should be produced. 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 decent 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 normal 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 classdiagram 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 us 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*:

- 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 almost always is 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 file that should be written. The latter do not directly produce output, but concentrate on controlling the AST traversal and call other templates that create the output. 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 Chapter 16) or of course directly within Java.

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 of comments, the file I/O object, the paths, where templates or handcoded files can be found, the output directory, and more. See Section 13.3 for a detailed description.

The `GeneratorEngine` offers several `generate` (and `generateNoA`) methods with varying signatures. 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.

13. Generator Engine using Flexible Templates

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 allows 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 depend 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 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) and the `generateNoA` method (ll. 27) 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. 1. Lines 3-5 define a special form of comments and switch the tracing of templates on. If tracing is on, the contribution of a template is marked by the template name within the comments. 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
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 Chapter 16). A special configuration class, namely `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    FileReaderWriter      fileHandler;              // ...
12    FreeMarkerTemplateEngine freeMarkerTemplateEngine; // exist
13
14    // and a configuration method
15    TemplateController getNewTemplateController(String templateName);
16 }
```

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, an `out` directory is used by default in the current path. (Also compare this to the `-o` argument when calling the CLI discussed in Section 2.2.4).
- `defaultFileExtension`: Files that shall have a default extension do get this string as suffix of their filename. The default is used when no explicit file extension is given as argument, when executing 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 Section 5.10, `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 handwritten templates are considered during code generation if the path containing the templates is listed in the `additionalTemplatePaths` variable. This 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 model name is printed in a comment if tracing is on. This is helpful if the artifacts are generated from one model, but other models exist for other artifacts. 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 variable that may be defined or changed in one `generate` call, and accessed or changed in a subsequent call. If not used with care that may unfortunately also be a source of multiple definition of global variables (error 0xA0122).

- `fileHandler`: The file handler is used to read and write files as well as to check, which handcoded files are existing. This basic operations are recorded with the reporting described in Section 15.5. It is recommended to use the standard objects in all occasions. The default for an unset configuration attribute is an instance of class `FileReaderWriter`.
- `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 functionality within the templates. The available Java data structures are defined in Chapter 5, but custom extensions are of course 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 four standard objects 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, because a list of aliases provides shortcuts.

Listing 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")}`.

```

1 // Aliases provided for templates
2
3 // Include and read signature commands, see Section 13.4.2
4 include(template)          -> tc.include(template)
5 includeArgs(tpl,par...)    -> tc.includeArgs(tpl,par...)
6 signature(par...)          -> tc.signature(par...)
7
8 // Warnings, errors, infos ..., see Section 13.4.3
9 error(message)             -> Log.error(message)
10 warn(message)              -> Log.warn(message)
11 info(message, logger)      -> Log.info(message, logger)
12 debug(message, logger)     -> Log.debug(message, logger)
13 trace(message, logger)     -> Log.trace(message, logger)
14
15 // Global variable management, see Section 13.4.4
16 defineGlobalVar(name, value) -> glex.defineGlobalVar(name, value)
17 defineGlobalVar(name)        -> glex.defineGlobalVar(name)
18 defineGlobalVars(name...)    -> glex.defineGlobalVars(name...)
19 changeGlobalVar(name, value) -> glex.changeGlobalVar(name, value)
20 addToGlobalVar(name, value)  -> glex.addToGlobalVar(name, value)
21 getGlobalVar(name)           -> glex.getGlobalVar(name)
22 requiredGlobalVar(name)      -> glex.requiredGlobalVar(name)
23 requiredGlobalVars(name...)  -> glex.requiredGlobalVars(name...)
24
25 // Hook point management, see Section 13.5
26 bindHookPoint(name, hp)      -> glex.bindHookPoint(name, hp)
27 defineHookPoint(name)        -> glex.defineHookPoint(tc, name)
28 defineHookPoint(name, ast)    -> glex.defineHookPoint(tc, name, ast)
29 existsHookPoint(name)        -> glex.existsHookPoint(name)

```

Listing 13.4: Shortcuts: Aliased functions available in templates

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.

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 Template Controller

The `TemplateController` provides methods for typical template operations such as the inclusion of sub templates or the instantiation of further auxiliary classes as helpers.

13. Generator Engine using Flexible Templates

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 `"tpl4.F.ftl"`, and also without default extension, like `"tpl4.F"`.

It is also possible to use a pathname of the underlying operating system, such as `"tpl/F.ftl"` in Unix. However, `"tpl/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, the templates are on the outer loop.

```
1 class TemplateController {
2
3     StringBuilder include(String          templateName,
4                          ASTNode       ast);
5     StringBuilder include(List<String> templateNames,
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 }
```

Java TemplateController

Listing 13.6: Include methods provided by the `TemplateController tc`

Please note that the `include` command 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. 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", ast.getOneChild()) }
2  ${include("my.Template") }
3  ${include(["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 Arguments

A second group of two `include` methods uses a slightly different approach of 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. The template called has the variables `glex`, `tc` and `ast` set only. `ast` remains identical to the `ast` of the calling template if not explicitly set.

The list of further arguments is only implicitly passed to the called template. To make this implicit list explicit and accessible through variable names, the `signature` method is used inside the called template. `signature` 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 it is wrong. Unfortunately, no typechecking happens.

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, string `prefix` to variable `"text1"`, and postfix to `"text2"`. In the second part, the content of variable `ast` is bound to `n` and `i` becomes 42. Unfortunately, the variable names need to be enclosed in quotation marks!

```
1 // Call of a template
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 Template Execution Results to Files

The write methods, e.g., the one in the Listing 13.11 (ll. 2) processes 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).

The file is opened, content is written, and the file is closed. Other versions use the default extension (ll. 5) or an already defined `Path` object (ll. 8).

```

1 class TemplateController {
2     void write(String templateName,
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`

13. Generator Engine using Flexible Templates

The `writeArgs` versions (ll. 12-16) 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 }
```

Java TemplateController

Listing 13.13: Further methods provided by the `TemplateController tc`

- `getTemplatename` (l. 3) allows to retrieve the name of the template (allows e.g. to rename templates without changing content).

- `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 3 assumes a constructor without arguments, while the version in line 7 allows to set the arguments of a constructor). Resulting objects may be used as *helpers*.
- `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 this API (described in Section 15.3) is directly available through aliasing within templates. Hence, this example is intended for template developers, who use logging information in templates.

```

1 // error and warning go to stdout
2 ${error("0x12345 A critical error occurred.")}
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 all subsequent 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);
```

Java GlobalExtensionManagement

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` only replaces the value of an already existing global variable.

`hasGlobalVar` 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.

`addToGlobalVar` assumes the variable name contains a list and adds its argument 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", [])`.

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, downwards from calling templates to called ones, but also upwards to the caller. 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. additional functionality is desired, the modifier shall be adapted, or additional annotations shall be attached to 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 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 something 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. Figure 13.18 demonstrates the effects of decoration with hook points. 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 not be changed. Only their execution is adapted.

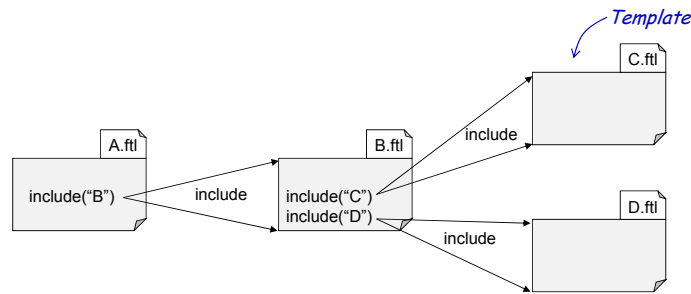
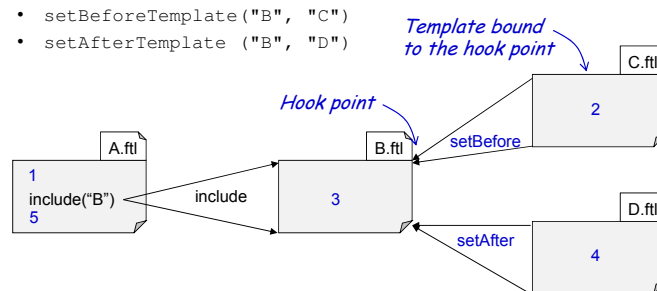
Figure 13.17: Hierarchical include structure induced by the `include` commands

Figure 13.18: Decoration before and after a template (execution order is 1..5)

Hook points can be *explicitly defined* within a template (see Section 13.5.3), but also each *template itself* (by its qualified name) acts as an implicitly defined hook point. Figure 13.20 shows the effect of an explicit definition and binding, while Figure 13.19 shows the effect of a *template replacement*, where the template name is used as hook point name.

An even more fine grained replacement is possible: We use the paired combination of *template and an AST object* it acts on, as hook point name (i.e. more precisely "*identifier*") and allow its individual decoration or replacement.

Therefore, MontiCore provides the following mechanisms:

- A hook point can be used to *replace* an existing template (see Figure 13.19).
- Existing templates can be *decorated* with additional hook points that are executed before or after the original template (see Figure 13.18).

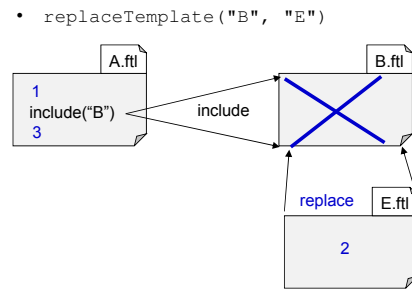


Figure 13.19: External replacement of a template

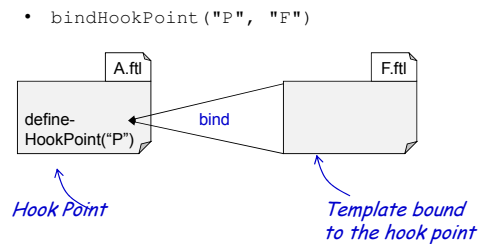


Figure 13.20: Defining an explicit hook point and binding it

- 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 13.20).

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 body 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`), 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 13.19 and 13.20. In these figures templates E and F cannot be decorated or replaced. However, the decoration of template B in Figure 13.21 remains active, when B itself is replaced.

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.

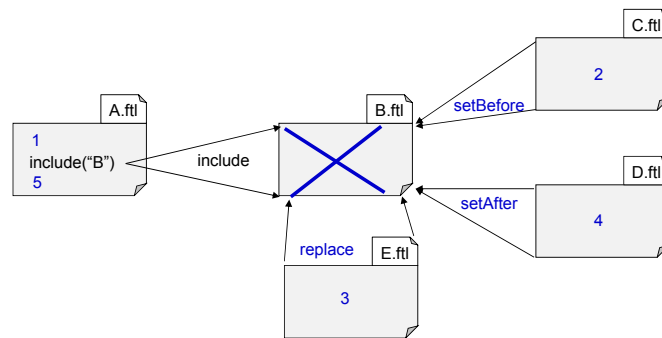


Figure 13.21: External template replacement keeps its decoration (execution order is 1..5)

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. They 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.

13.5.2 Forms of Hook Points

So far we have discussed filling hook points with templates. However, MontiCore provides also the following forms of hook point values that allow to inject a normal string, an inline template or even Java code. A hook point is filled by one (or if allowed more) *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.

Each of these forms of hooks is defined by a subclass of `HookPoint`. Its signature is shown in Listing 13.23.


Tip 13.22 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 the order of execution is then not easy to understand.

```

1 package de.monticore.generating.templateengine;
2
3 public abstract class HookPoint {
4
5     public abstract
6     String processValue(TemplateController controller,
7                        ASTNode ast);
8
9     public abstract
10    String processValue(TemplateController controller,
11                      List<Object> args);
12
13    public abstract
14    String processValue(TemplateController controller,
15                      ASTNode node,
16                      List<Object> args);
17 }

```

Java HookPoint

Listing 13.23: 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 Figures Listing 13.24-Listing 13.26.

```

1 public class StringHookPoint extends HookPoint {
2     // constructor of StringHookPoint
3     public StringHookPoint(String value);
4 }

```

Java StringHookPoint

Listing 13.24: 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 }
```

Java TemplateHookPoint

Listing 13.25: Constructors for TemplateHookPoint

```
1 public class TemplateStringHookPoint extends HookPoint {
2     // constructors of TemplateStringHookPoint
3     public TemplateStringHookPoint(String statement)
4
5                                     throws IOException;
6 }
```

Java TemplateStringHookPoint

Listing 13.26: Constructor of TemplateStringHookPoint

glex provides access to global variables 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` are to be mentioned.

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 the 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. It receives all necessary information through the arguments, i.e., variables `tc` and `ast`. `tc` can provide additional infos, e.g., to access 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. Through its definition, it gets a *name*. 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 13.27. In this listing the hook point with the name "`<JavaBlock>?TemplateName:member`" is called. If the hook point should work on the same ast node then the ast argument can also be omitted.

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  // to find out, whether a hook point is bound (if necessary)
9  ${glex.existsHookPoint("<JavaBlock>?TemplateName:member")}
10
11 // shortcuts with the same effect
12 ${defineHookPoint("<JavaBlock>?TemplateName:member", ast) }
13 ${defineHookPoint("<JavaBlock>?TemplateName:member") }

```

Listing 13.27: 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. 12f.

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

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),
3. the type of the ast where the hook point is applied to, 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 defined in the `ClassImpl` template that requires a `Class` ast, returns an arbitrary set of `JavaBlock` statements, and is defined to add class *members* can be named with `<JavaBlock>*ClassContent:member`.

An often used hook point name, however, uses a different convention, namely

```
1  ${glex.defineHookPoint("JavaCopyright") }
```

FTL

that expects a comment containing some copyright information for each generated Java class.

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 `GlobalExtensionManagement` provides the method `bindHookPoint`.

```
1  class GlobalExtensionManagement {
2      String defineHookPoint(TemplateController controller,
3                           String hookName,
4                           ASTNode ast);
5      String defineHookPoint(TemplateController controller,
6                           String hookName);
7
8      void bindHookPoint(String hookName, HookPoint hp);
9      void bindStringHookPoint(String hookName, String content);
10     void bindTemplateHookPoint(String hookName, String tpl);
11
12     boolean existsHookPoint(String hookName);
13 }
```

Java GlobalExtensionManagement

Listing 13.28: Methods of the `GlobalExtensionManagement` class for hook point management

The class `GlobalExtensionManagement` is not only responsible for managing global variables, but also for hook points in templates. Listing 13.28 describes the methods it provides for this purpose.

The effect of the methods called from templates, namely `defineHookPoint` 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 within a template. The code examples in Listing 13.29 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 13.29: 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, when executing it. 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 13.30.

```
1 class 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 }
```

Java GlobalExtensionManagement

Listing 13.30: 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 three argument version of `replaceTemplate` 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.

**Tip 13.31 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.

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 repeated here in detail:

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` : 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.

`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 13.32 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) 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>>`.

Chapter 14

Integrating Handwritten Code

co-authored by 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*, in short *hc*, and the result *handwritten code*. 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. Thus, any handcoded parts need to be located in separate artifacts. Each artifact is either completely handcoded or generated, but not a mixture.



Tip 14.1 Keep Generated and Handwritten Code Separate

Keep generated and handwritten code in separate artifacts.

Ideally separate the artifacts in different directory hierarchies, such that cleaning of all generated artifacts is simple.

And do not put any generated artifacts under version control.

A good generator aims for extensibility of the the resulting code. This can be accomplished by 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 a 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.

For that purpose, many classes (including all AST-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 14.2).

This approach 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.

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.

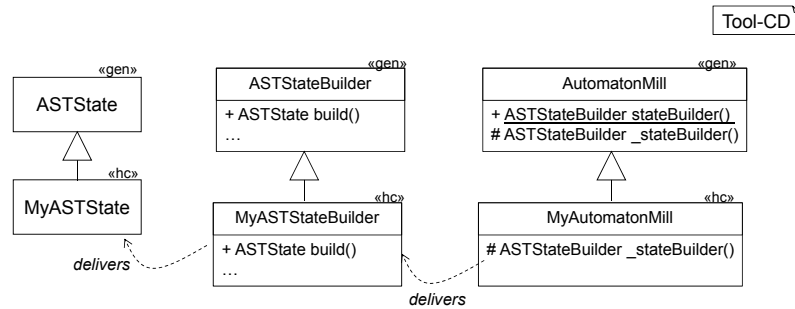


Figure 14.2: How a given class can be extended by building a subclass



Tip 14.3 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. However, when the superclass is not re-generated (in the same form), the handcoded subclass becomes erroneous and does not compile anymore.

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.

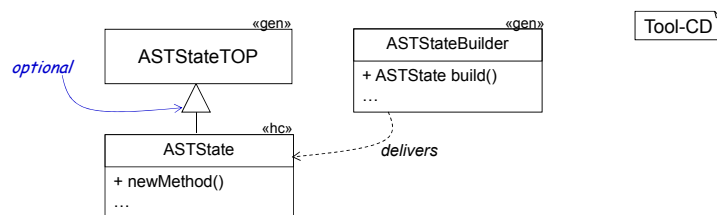


Figure 14.4: How a given class can be replaced by a handwritten class

Figure 14.4 demonstrates this. Here a handwritten class `ASTState` is existent in the path for handcoded files (argument `-hcp`) that replaces the generated `ASTState` class. The 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 serve as superclass for `ASTState`. The developer has the following options:

- The handwritten class extends the generated class (as shown).
- The handwritten class is a copied version of the original class with specific modification, but does not extend the TOP class.
- 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.

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.

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 (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.



Tip 14.5 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.
- Do not forget to initialize additional attributes.
- Rerun the generator.

This introduces a handcoded version of the class and thus allows versioning and modification.

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 14.6 demonstrates how a class and its builder are replaced using the TOP mechanism.

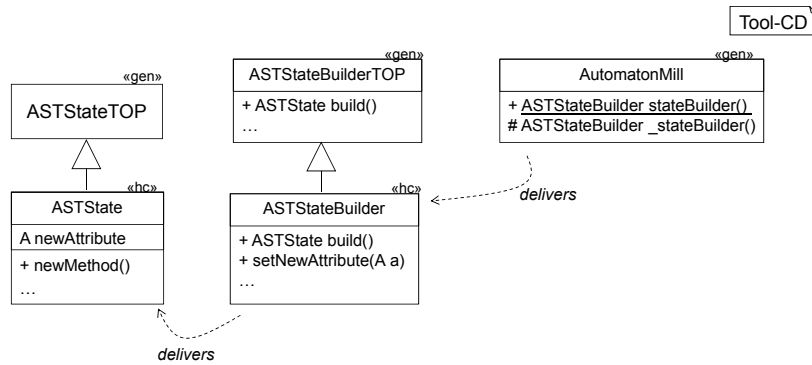


Figure 14.6: How a given class and its builder can be replaced by handwritten versions

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 replaced individually, while the generator for other classes does not need to take notice at all. Figure 14.7 demonstrates possible resulting structures.

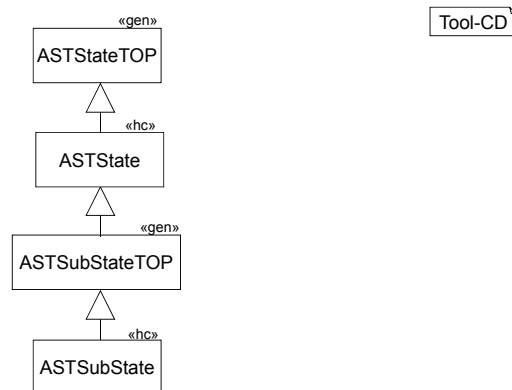


Figure 14.7: How a given class and its subclass can be replaced by a handwritten class

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 by 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.

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  www.monticore.de           // Explanations to errors
2                               // and potential help
```

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 to avoid internal errors. In case you got one, please send it to bugreport@monticore.de including the source model, configuration, MontiCore-version and other potentially interesting information. It will not be generally visible, but used for improvement among MontiCore 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 result 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 " $0 \times [0-9A-Z]^5$ ". 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 they 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 `out/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 oder 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 as a central delegator for issuing errors, warnings, and log messages.

```

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. `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.

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

```
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 int getErrorCount()
17    public static boolean isFailQuickEnabled()
18 }
```

Java «RTE» Log

Listing 15.2: Logging API in class Log

to be respectively issued using the debug log methods in ll. 10. 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 using 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 many models  
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 were issued and if so, fail quick is applied and the application is terminated then immediately.

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 [QOS17b] 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 [QOS17a]. 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 default 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: `out/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: `out/monticore.detailed.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

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 [QOS17b, QOS17a].

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, but throws an `Exception` (e.g. to be caught by `JUnit`). 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`.

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.

¹This is usable in make, shell or CLI environments

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 out/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(aPath, reportManagerFactory)   
2 // ...  
3 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 `ReportManager`, 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)

```

Groovy

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");

```

Java

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!CDCClass(5,2)
3 @age!CDAttribute(7,4)
4 @_!Modifier
5 @_!Modifier(!2)
6 @_!Modifier(3,4!2)

```

Reporting

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 `CDCClass` 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.

15. Error Handling, Logging and Reporting

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)
```

Reporting

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
```

Reporting

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 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.

13_SymbolTable.txt contains the content of the symbol table after the execution of the generation process. This information support generator developers in understanding the content and structure of the symbol table.

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_ObjectDiagram.txt contains the AST structure in 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 [Gro17] 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) [Gro17] which can be graphically presented by yEd [yWo17] for example. An excerpt of such a diagram is shown in Figure 15.10

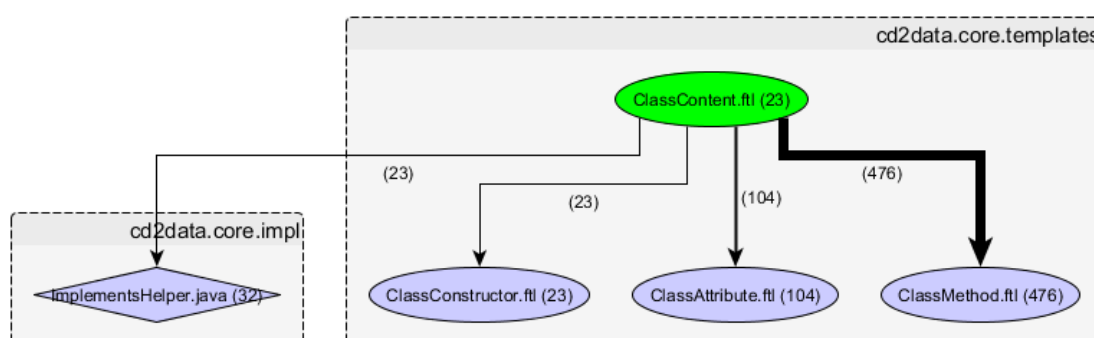


Figure 15.10: Relationship between artifacts (templates and Java, excerpt)

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.

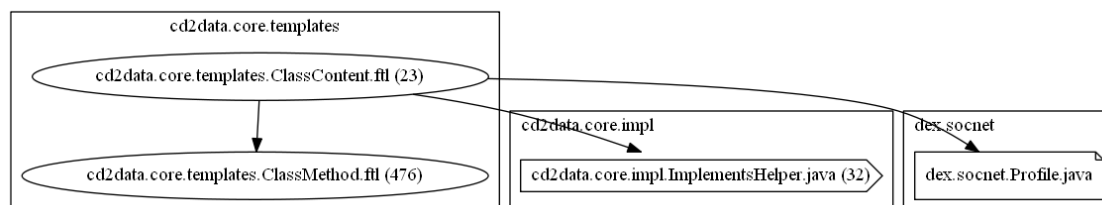


Figure 15.11: Relationship between template artifacts (excerpt)

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 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 `17_InputOutputFiles.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. 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) automatically 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 (see Section 15.4) is designed to support the aforementioned steps of handling exceptions.

Chapter 16

MontiCore Configuration with Groovy

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 Groovy script is executed on the base class `MontiCoreScript`, providing a set of available methods (cf. Listing 16.2). MontiCore provides two predefined Groovy scripts that can be used to execute the generator: `monticore_emf.groovy` and `monticore_noemf.groovy`. Both Groovy scripts implement the same generation process explained later in this chapter. The only difference between those two scripts is that `monticore_emf` generates EMF-compatible AST classes while `monticore_noemf` generates clean MontiCore AST classes. By default, the MontiCore CLI uses the `monticore_noemf` whereas the MontiCore Maven plugin uses the `monticore_emf` Groovy script. As described in Chapter 2, the user can choose between those two predefined Groovy scripts provided by MontiCore as well as develop a custom Groovy script. The 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 an 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 ten actions:

- M1** Initialize generation (incremental generation, reporting, global scope)
- 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
- M6** Generate parser using ANTLR
- M7** Decorate CD-AST with methods

M8 Generate symbol and scope classes for models of the grammar

M9 Generate ast classes, visitor and context condition infrastructure

M10 Write reports to files

Listing 16.1 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 the incremental generation, enables reporting and creates the global scope. 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 super-grammars, that also have to be loaded (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 written to a file as part of the reports created by MontiCore. 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, in M7 the class diagram AST is enriched with further methods for the AST class generation in preparation for M9. In M8 the symbol and scope classes are generated that allow defining a symbol table for the processed grammar. Finally, in M9 the AST classes as well as the visitor and context condition infrastructure are generated. At the end, in M10 all reports are written into files.

One can see that the workflow is rather straight forward (see points M1 to M10 above). Simplicity in the controlling scripts is favored here. Thus, adaptation of the control workflow can be done by substituting the control script through a handwritten one. This 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 16.1 are part of the MontiCore component library and described in Section 16.2.

16.1 Generate the Standard Classes

Listing 16.1 depicts the Groovy script `monticore_noemf.groovy` that is used by the MontiCore generator to generate standard classes. It generates the whole model processing infrastructure such as lexer, parser, AST classes, context conditions, visitors and symbol table infrastructure. It is the default script used by the MontiCore CLI. To use this script via the MontiCore Maven plugin, it can be passed as argument using the parameter `-s` oder `-script`.

```

1
2 // M1 basic setup and initialization:
3 // initialize incremental generation; enabling of reporting; create
  global scope
4 IncrementalChecker.initialize(out)
5 InputOutputFilesReporter.resetModelToArtifactMap()
6 globalScope = createGlobalScope(modelPath)
7 Reporting.init(out.getAbsolutePath(), reportManagerFactory)
8
9 while (grammarIterator.hasNext()) {
10   input = grammarIterator.next()
11   if (force || !IncrementalChecker.isUpToDate(input, out, modelPath,
12     templatePath, handcodedPath)) {
13     IncrementalChecker.cleanUp(input)
14
15     // M2: parse grammar
16     astGrammar = parseGrammar(input)
17
18     if (astGrammar.isPresent()) {
19       astGrammar = astGrammar.get()
20
21       // start reporting
22       grammarName = Names.getQualifiedName(astGrammar.getPackageList
23         (), astGrammar.getName())
24       Reporting.on(grammarName)
25       Reporting.reportParseInputFile(input, grammarName)
26
27       // M3: populate symbol table
28       astGrammar = createSymbolsFromAST(globalScope, astGrammar)
29
30       // M4: execute context conditions
31       runGrammarCoCos(astGrammar, globalScope)
32
33       // M5: transform grammar AST into Class Diagram AST
34       astClassDiagram = transformAstGrammarToAstCd(glex, astGrammar,
35         globalScope, handcodedPath)
36       astClassDiagramWithST = createSymbolsFromAST(globalScope,
37         astClassDiagram)
38
39       // write Class Diagram AST to the CD-file (*.cd)
40       storeInCdFile(astClassDiagramWithST, out)
41
42       // M6: generate parser and wrapper
43       generateParser(glex, astGrammar, globalScope, handcodedPath,
44         out)
45     }
46   }
47 }
48
49 for (astGrammar in getParsedGrammars()) {
50   reportGrammarCd(astGrammar, globalScope, out)
51 }

```

16. MontiCore Configuration with Groovy

```
46 astClassDiagram = getCDOOfParsedGrammar(astGrammar)
47
48 // M7: decorate Class Diagram AST
49 decorateCd(glex, astClassDiagram, globalScope, handcodedPath)
50
51 // M8: generate symbol table
52 generateSymbolTable(astGrammar, globalScope, astClassDiagram, out,
53     handcodedPath)
54
55 // M9 Generate ast classes, visitor and context condition
56 generate(glex, globalScope, astClassDiagram, out, templatePath)
57
58 // M10: flush reporting
59 Reporting.flush(astGrammar)
60 }
```

Listing 16.1: Groovy script used to generate the standard result

Moreover, MontiCore is able to generate AST classes that are compatible to the Eclipse Modelling Framework (EMF) [SBPM08]. The Groovy script used to generate EMF compatible classes is called `monticore_emf.groovy` and can be used by the MontiCore CLI via the command line parameter `-s` or `-script`. The workflow is identical to the standard script except where specific methods are used that realize the compatibility. It is the default Groovy script used by the MontiCore Maven plugin.

16.2 MontiCore Base Class for Groovy Scripts

Listing 16.2 depicts the method signatures that are provided by the base class shipped with MontiCore and can be used within the Groovy script. All 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 the typical import mechanism, variable and method declarations known from Java can be used to implement a custom Groovy script. In the following subsections, methods of the base class, predefined variable and preimported classes are briefly described.

```
1 Optional<ASTMCGrammar> parseGrammar(Path grammar)
2 List<ASTMCGrammar> parseGrammars(IterablePath grammarPath)
3 generateParser(GlobalExtensionManagement glex,
4     ASTMCGrammar grammar, GlobalScope symbolTable,
5     IterablePath handcodedPath, File outputDirectory)
6 generateParser(GlobalExtensionManagement glex,
7     ASTMCGrammar grammar, GlobalScope symbolTable,
8     IterablePath handcodedPath, File outputDirectory,
9     boolean embeddedJavaCode, Languages lang)
10 generateSymbolTable(ASTMCGrammar astGrammar,
11     GlobalScope symbolTable, ASTCDCompilationUnit astCd,
12     File outputDirectory, IterablePath handcodedPath)
```

```

14  ASTMCGrammar createSymbolsFromAST(GlobalScope globalScope,
15      ASTMCGrammar ast)
16  ASTCDDCompilationUnit createSymbolsFromAST(
17      GlobalScope globalScope, ASTCDDCompilationUnit ast)
18  runGrammarCoCos(ASTMCGrammar ast, GlobalScope scope)
19  ASTCDDCompilationUnit transformAstGrammarToAstCd(
20  ASTCDDCompilationUnit transformAstGrammarToAstCd(
21      GlobalExtensionManagement glex, ASTMCGrammar astGrammar,
22      GlobalScope symbolTable, IterablePath targetPath)
23  reportGrammarCd(ASTMCGrammar astCd, GlobalScope globalScope,
24      File outputDirectory)
25  decorateCd(GlobalExtensionManagement glex,
26      ASTCDDCompilationUnit astClassDiagram,
27      GlobalScope symbolTable, IterablePath targetPath)
28  generate(GlobalExtensionManagement glex,
29      GlobalScope globalScope,
30      ASTCDDCompilationUnit astClassDiagram, File outputDirectory,
31      IterablePath templatePath)
32  decorateEmfCd(GlobalExtensionManagement glex,
33      ASTCDDCompilationUnit astClassDiagram,
34      GlobalScope symbolTable, IterablePath targetPath)
35  generateEmfCompatible(GlobalExtensionManagement glex,
36      GlobalScope globalScope,
37      ASTCDDCompilationUnit astClassDiagram,
38      File outputDirectory, IterablePath templatePath)
39  GlobalScope createGlobalScope(ModelPath modelPath)

```

Listing 16.2: Methods available in the Groovy scripts

16.2.1 Methods Available within Groovy Scripts

The provided Groovy scripts use several methods that are predefined by the MontiCore base class. 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.

createGlobalScope(...) uses the model path received as input parameter to create the global scope. The global scope is used to resolve grammars, e.g., super grammars of the processed grammars.

parseGrammar(...) parses the grammar received as input parameter and creates the corresponding AST. Returns the created AST.

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.

transformAstGrammarToAstCd(...) translates the grammar AST to a class diagram AST for further processing. Returns the created class diagram AST.

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.

reportGrammarCd(...) writes the given class diagram to a file located in the reporting folder for reporting purposes.

decorateCd(...) enriches the class diagram AST with further methods for the AST classes generated for the processed grammar. The AST classes are generated by the `generate(...)` method.

generateSymbolTable(...) generates symbol and scope classes as well as a symbol table creator class for the processed grammar.

generate(...) generates the AST classes, the visitor interfaces and classes, and classes to define and check context conditions for the language defined by the processed grammar.

decorateEmfCd(...) similar to `decorateCd(...)` but creates methods for EMF compatibility as well.

generateEmfCompatible(...) similar to `generate(...)`, but creates classes and methods for EMF compatibility as well.

16.2.2 Variables Available within Groovy Scripts

The provided Groovy scripts use 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 super 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.

boolean force created from the force parameter. It indicates whether the generation is always enforced, or there will be first a check, whether regeneration is necessary at all.

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.

16.2.3 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 those 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.

IncrementalChecker offers methods to check whether generated files are up to date.

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

Some MontiCore Grammars Explained

MontiCore provides a number of grammars and libraries. Their usage supports language engineering. This chapter shortly describes some of the available grammars. This chapter therefore serves two purposes: First, it explains some of the available grammars and non-terminals for a reuse and extension. Second, it also demonstrates how to write a reusable, and hopefully also well engineered grammar. For that purpose, we discuss the nonterminals and their arrangements in productions as well as some best practices for production definition.



Tip 17.1 Basic Grammars

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.MCNumbers.mc4
7              de.monticore.MCHexNumbers.mc4
8              de.monticore.StringLiterals.mc4
9              de.monticore.UMLStereotype.mc4
10             de.monticore.UMLModifier.mc4
11             de.monticore.MCBasicTypes1.mc4
12             de.monticore.expressions.MCExpressions.mc4
```

Files

17.1 Component Grammar MCBasics.mc4

Lexicals are very basic entities of a language, such as names, whitespaces and comments. Several of these token never show up in the AST.

The header defines the package and that the `MCBasics` grammar is a component grammar that depends on the no other grammar.

17. Some MontiCore Grammars Explained

```
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' ) : { _channel = HIDDEN; };
```

MCG MCBasics

Comments come in the 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 fully shown here):

```
1 token SL_COMMENT =
2     "//" ( ~('\n' / '\r' ) ) * :
3     { _channel = HIDDEN; // ...
4     };
5
6 token ML_COMMENT =
7     "/*" ( { _input.LA(2) != '/' } ? '*' |
8           NEWLINE | ~('*' | '\n' | '\r' ) ) *
9     "*/" :
10    { _channel = HIDDEN; // ...
11    };
12 }
```

MCG MCBasics

The semantic predicate `_input.LA(2) != '/'` in the regular expression allows '`/*`' to be parsed only, when the second character in the input differs from '`/`'. This is a compact form to exclude closing patterns on the token level.

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 package de.monticore;
2
3 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 =
2     source:CharToken;
3
4 ast CharLiteral =
5     method public char getValue() {
6         return de.monticore.MCLiteralsDecoder.decodeChar(getSource());
7     }
8 ;
9
10 token CharToken
11     = '\'' (SingleCharacter|EscapeSequence) '\''
12     : {setText(getText().substring(1, getText().length() - 1));};
13
14 fragment token HexDigit
15     = '0'..'9' | 'a'..'f' | 'A'..'F' ;
16
17 fragment token OctalDigit
18     = '0'..'7' ;
19
```

17. Some MontiCore Grammars Explained

```
20  fragment token SingleCharacter
21      = ~ ('\\');
22
23  fragment token EscapeSequence
24      = '\\\' ('b' | 't' | 'n' | 'f' | 'r' | '"' | '\'' | '\\\'')
25          | OctalEscape | UnicodeEscape;
26
27  fragment token OctalEscape
28      = '\\\' OctalDigit
29          | '\\\' OctalDigit OctalDigit
30          | '\\\' ZeroToThree OctalDigit OctalDigit;
31
32  fragment token UnicodeEscape
33      = '\\\' 'u' HexDigit HexDigit HexDigit HexDigit;
34
35  fragment token ZeroToThree
36      = '0'..'3' ;
```

While the token `CharToken` would be sufficient for a use, it is convenient to embed the pure string in an AST object that provides additional functionality, but more important that can be targeted by the generated visitors. It is therefore a matter of taste, whether the token is reused directly or the embedding nonterminal `CharLiteral`.

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  ast StringLiteral =
5      content:String
6      method public String getValue() {
7          if(content == null) {
8              content =
9                  de.monticore.MCLiteralsDecoder.decodeString(getSource());
10         }
11         return content;
12     };
13
14  token StringToken
15      = '"' (StringCharacters)? '"'
```

MCG StringLiterals

```

16      : {setText(getText().substring(1, getText().length() - 1));};
17
18      fragment token StringCharacters
19          = (StringCharacter)+;
20
21      fragment token StringCharacter
22          = ~ ('"' | '\\') | EscapeSequence;

```

The main difference to a single character is that the decoded string is stored in a buffer 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 sync.

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 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 hexadecimals, octals or binary numbers. And finally they could be integers or longs.

The following two grammars demonstrate, how to define numbers and extend the way how numbers are encoded in subgrammars. 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 package de.monticore;
2
3 component grammar MCNumbers extends de.monticore.MCBasics {
4
5     interface Number;
6
7     ast Number =
8         method public String getSource()
9             { throw new UnsupportedOperationException(
10                 "0xFF230 Method not implemented"); }
11         method public int getValueInt()
12             { throw new UnsupportedOperationException(
13                 "0xFF231 Method not implemented"); }

```

17. Some MontiCore Grammars Explained

```
14     method public long getValue()  
15     { throw new UnsupportedOperationException(  
16         "0xFF232 Method not implemented"); }  
17 ;
```

The interface methods have 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 `Number`.

One nonterminal implementing this interface parses exactly the `Decimals` and thus produces only positive numbers (≥ 0):

```
1  Decimal implements Number =  
2      source:DecimalToken;  
3  
4  ast 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' ;
```

MCG MCNumbers

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  ast Integer =  
5      method public int getValueInt() {  
6          int a = Integer.parseInt(getDecimalpart());
```

MCG MCNumbers

```

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 tokens 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 package de.monticore;
2
3 component grammar MCHexNumbers extends
4     de.monticore.MCNumbers, de.monticore.MCBasics {
5
6     Hexadecimal implements Number =
7         source:HexadecimalToken;
8
9     ast Hexadecimal =
10         method public int getValueInt() {
11             return Integer.parseInt(getSource().substring(2),16);
12         }
13         method public long getValue() {
14             return Long.parseLong(getSource().substring(2),16);
15         }
16     ;
17
18     token HexadecimalToken
19         = '0' ('x' | 'X') HexDigit HexDigit*;
20
21     fragment token HexDigit
22         = '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 hexadecimals:

```
1  HexInteger implements Number = MCG MCHexNumbers
2      (negative:["-"])? hexadecimalpart:HexadecimalToken;
3
4  ast 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 Grammar `MCBasicTypes1.mc4`

This grammar defines very simple Java compliant types. The scope of this grammar is to ease the reuse of type structures in Java-like sublanguages. The grammar contains qualified names, import statement, primitive types from Java, void, and reference types, but does not include types with dimensions, generics, and type parameters.

`QualifiedName` represents a possibly qualified name in the AST. The different parts of a qualified name are separated by `'.'`; they are stored in a list of `Strings`.

The `ImportStatement` represents the import for any form of artifact. We generally assume that artifacts, including various forms of models as well as code use and therefore import each other.


```

1 package de.monticore;
2
3 QualifiedName =
4     part:(Name || ".")+;
5
6 ast QualifiedName =
7     method public String getQName(){
8         return de.se_rwth.commons.Names.getQualifiedName(
9             this.getPartList());
10    };
11
12 ImportStatement =
13     "import" QualifiedName ("." Star:["*"])? ";" ;
14
15 ast ImportStatement =
16     method public String getQName(){
17         return getQualifiedName().getQName();
18    };

```

In addition to the productions defining the new nonterminals, this grammar adds a hand-written function `getQName()` to the AST, which reconstructs the full name as a string. Furthermore, `isStar()` can be used to identify, whether a star import has been defined.

For all other type definitions, we refer directly to the grammar `MCBasicTypes1` or any other grammar defining types.

17.5 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.

17.5.1 Component Grammar `UMLStereotype.mc4`

The grammar uses a Java action to check whether the closing parenthesis are written without spaces in between. This is unfortunately necessary, because generic types use single parenthesis and it is therefore not wise to introduce a token `">>"` directly. A short

17. Some MontiCore Grammars Explained

semantic predicate, that calls the predefined `noSpace` method can be used to glue the two previous tokens together and achieve the same effect.

```
1 package de.monticore;
2 component grammar UMLStereoType extends de.monticore.StringLiterals {
3     StereoType =
4         "<<" value:(StereoValue || ",")+ ">" ">" {noSpace()}? ;
5         // It is not possible to define ">>" in one string.
6         // Parsing generic types like "List<List<String>>"
7         // is then not working.
8
9     StereoValue =
10         Name& ("=" text:StringLiteral)?;
11
12     ast StereoType =
13         method public boolean contains(String name) ...
14         method public boolean contains(String name, String value) ...
15         method public String getValue(String name) ...
16
17     ast StereoValue =
18         content:String
19         method public String getValue() { ...
20 }
```

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.

17.5.2 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 `ast` statement.

```
1 package de.monticore;
2 component grammar Cardinality extends
3     de.monticore.MCBasics,
4     de.monticore.MCNumbers
5 {
6     Cardinality =
7         "["
8         ( many:["*"] {_aNode.setLowerBound(0);_aNode.setUpperBound(0);}
9         | lowerBoundLit:Decimal
10         { _aNode.setLowerBound(
11             _aNode.getLowerBoundLit().getValueInt();
12             _aNode.setUpperBound(_aNode.getLowerBound()); }
13         ( ".." (
```

```

14         upperBoundLit:Decimal
15         ( {_aNode.setUpperBound(
16             _aNode.getUpperBoundLit().getValueInt();})
17         |
18         noUpperLimit:["*"] {_aNode.setUpperBound(0);} ) )?
19     ) "]" ";
20
21     ast Cardinality =
22         lowerBound:int
23         upperBound:int;
24 }

```

Please note that 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. The code is executed directly when parsing and is used to calculate some additional attributes defined using the *ast* statement. Alternatively would it have been possible to embed that code in some extra methods, e.g. like the *getValue()* methods used in the *MCNumbers* grammar.

17.5.3 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.

While white spaces in general should not be included, it is again necessary to separate the brackets in the incomplete case to avoid a clash with similar symbols from other languages.

```

1 package de.monticore;
2 component grammar Completeness {
3     Completeness =
4         [complete:"(c)"]
5         | ("(" [incomplete:"..."] ")")
6         | [incomplete:"(..., ...)"]
7         | [complete:"(c,c)"]
8         | [rightComplete:"(...,c)"]
9         | [leftComplete:"(c,...)"];
10 }

```

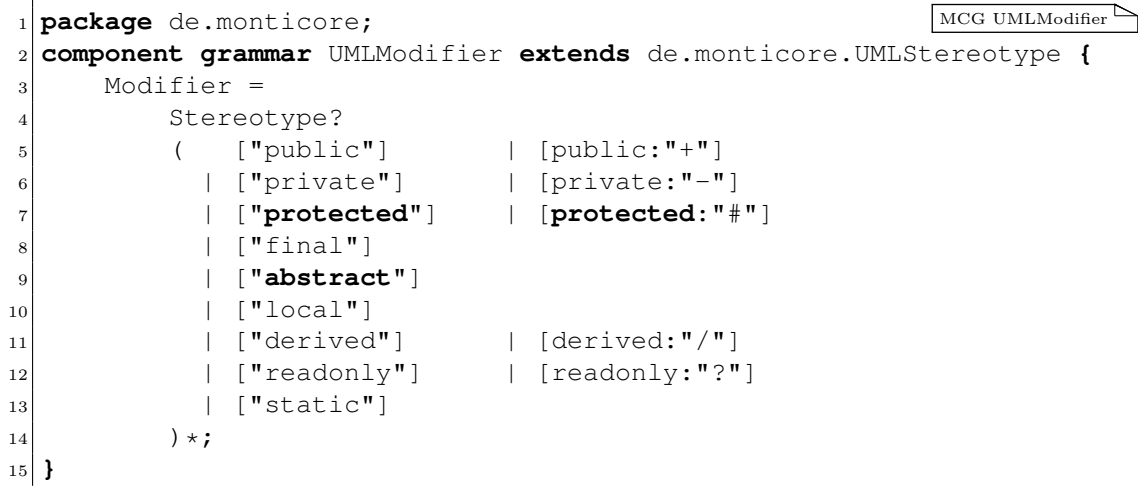
MCG Completeness

17.5.4 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 nonterminal

Modifier three 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 de.monticore.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 an 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 advantage that a visitor could act on a modifier directly, but also the disadvantage that more classes would be generated.

17.6 Component Grammar MCExpressions.mc4

The forthcoming `MCExpression` grammar is very specifically inspired from the Java programming language. It contains almost all elements that Java provides, with the exception of creating new objects or defining anonymous classes.

For reference the grammar is completely included here. The grammar definition starts with the definition of interface `Expression` and subsequently implements this interface with a variety of expressions in different priorities. Please see Section 4.2.8 for discussion of priorities and how to use and extend the grammar with new forms of expressions as well as the removal of unwanted forms of expressions.

```

1 package de.monticore;
2
3 component grammar MCExpressions extends de.monticore.types.Types {
4
5 interface Expression;

```

These forms of expressions mainly deal with objects (applying this or super) as well as type casting and method call:

```

1 ThisExpression implements Expression <280> =
2   Expression "." ["this"];
3
4 SuperExpression implements Expression <270> =
5   Expression "." "super" SuperSuffix;
6
7 GenericInvocationExpression implements Expression <260> =
8   Expression "." PrimaryGenericInvocationExpression;
9
10 TypeCastExpression implements Expression <230> =
11   "(" Type ")" Expression;
12
13 InstanceofExpression implements Expression <140> =
14   Expression "instanceof" Type;
15
16 PrimaryThisExpression implements Expression <320>
17   = "this";
18
19 PrimarySuperExpression implements Expression <330>
20   = "super";
21
22 ClassExpression implements Expression <360>
23   = ReturnType "." "class";
24
25 PrimaryGenericInvocationExpression implements Expression <370>
26   = TypeArguments GenericInvocationSuffix;
27
28 GenericInvocationSuffix
29   = ["super"] SuperSuffix
30     | ["this"] Arguments
31     | Name Arguments;
32
33 SuperSuffix
34   = Arguments
35     | "." TypeArguments? Name Arguments?;
36
37 Arguments
38   = "(" (Expression || ",")* ")";

```

In the following block a number of applications, for example reading attributes, indexing arrays, calling methods is defined:

17. Some MontiCore Grammars Explained

```
1 QualifiedNameExpression implements Expression <290> =
2   Expression "." Name;
3
4 ArrayExpression implements Expression <250> =
5   Expression "[" indexExpression:Expression "]" ;
6
7 CallExpression implements Expression <240> =
8   Expression Arguments;
```

Suffixes and prefixes generally have a high priority. Please note that due to the given priorities, `-v++` parses as `-(v++)`.

```
1 SuffixExpression implements Expression <220> =
2   Expression
3     (   suffixOp:"++"
4       |   suffixOp:"--"
5     );
6
7 PrefixExpression implements Expression <210> =
8   (   prefixOp:"+"
9     |   prefixOp:"- "
10    |   prefixOp:"++"
11    |   prefixOp:"--"
12  )
13   Expression;
14
15 BooleanNotExpression implements Expression <200> =
16   "~" Expression;
17
18 LogicalNotExpression implements Expression <190> =
19   "!" Expression;
```

Then follow a large set of infix operations in their usual priorities. Please note that if priorities are identical, then left associative parsing is applied. Thus, `a*b*c` is represented in the AST as `(a*b)*c`.

```
1 MultExpression implements Expression <180> =
2   leftExpression:Expression
3     (   multiplicativeOp:"*"
4       |   multiplicativeOp:"/"
5       |   multiplicativeOp:"%"   )
6     rightExpression:Expression;
7
8 AddExpression implements Expression <170> =
9   leftExpression:Expression
10    (   additiveOp:"+"
11      |   additiveOp:"- "      )
12   rightExpression:Expression;
13
```

```

14 ComparisonExpression implements Expression <150> =
15     leftExpression:Expression
16         (   comparison:"<="
17             |   comparison:">="
18             |   comparison:">"
19             |   comparison:"<"           )
20     rightExpression:Expression;
21
22 IdentityExpression implements Expression <130> =
23     leftExpression:Expression
24         (   identityTest:"=="
25             |   identityTest:"!="       )
26     rightExpression:Expression;
27
28 BinaryAndOpExpression implements Expression <120> =
29     leftExpression:Expression "&" rightExpression:Expression;
30
31 BinaryXorOpExpression implements Expression <110> =
32     leftExpression:Expression "^" rightExpression:Expression;
33
34 BinaryOrOpExpression implements Expression <100> =
35     leftExpression:Expression "|" rightExpression:Expression;
36
37 BooleanAndOpExpression implements Expression <90> =
38     leftExpression:Expression "&&" rightExpression:Expression;
39
40 BooleanOrOpExpression implements Expression <80> =
41     leftExpression:Expression "||" rightExpression:Expression;
42
43 ConditionalExpression implements Expression <70> =
44     condition:Expression "?" trueExpression:Expression
45         ":" falseExpression:Expression;
46
47 AssignmentExpression implements Expression <60> = <rightassoc>
48     leftExpression:Expression
49         (   assignment:"="
50             |   assignment:"+="
51             |   assignment:"-="
52             |   assignment:"*="
53             |   assignment:"/="
54             |   assignment:"&="
55             |   assignment:"|="
56             |   assignment:"^="
57             |   assignment:">>="
58             |   assignment:">>>="
59             |   assignment:"<="
60             |   assignment:"%="       )
61     rightExpression:Expression;

```

The grammar has been deliberately designed in such a way that operators of same priority are stored within the same AST class as an explicit additional attribute. E.g. assignment

17. Some MontiCore Grammars Explained

is an attribute with according get/set- methods of type string containing one of the above defined operators.

Unfortunately the shifting expressions that use "<<" and ">>" again produce a challenges and need special handling with a look into the input chain of the parser:

```
1 ShiftExpression implements Expression <160> = MCG MCExpressions
2     // It is not possible to define "<<", ">>" or ">>>"
3     // in one string. Parsing generic types like
4     // "List<List<String>>" is then not working.
5     leftExpression:Expression
6     (   shiftOp:"<" "<" {noSpace()}?
7         {_aNode.setShiftOp("<<");}
8     |   shiftOp:">" ">" {noSpace()}? ">" {noSpace()}?
9         {_aNode.setShiftOp(">>>");}
10    |   shiftOp:">" ">" {noSpace()}?
11        {_aNode.setShiftOp(">>");}
12    )
13    rightExpression:Expression;
```

In the final block, the grammar defines a number of prime expressions, such as bracket enclosed expressions and simple variable names:

```
1 BracketExpression implements Expression <310> MCG MCExpressions
2     = "(" Expression ")";
3
4 NameExpression implements Expression <350>
5     = Name;
6
7 LiteralExpression implements Expression <340>
8     = Literal;
9 }
```

It is an advantage of MontiCore that the expression language can almost be defined in its natural form, defining both the parsing strategy and the AST. This is possible because MontiCore allows a form of mutual recursive definitions of left recursive productions, where an interface nonterminal (here: Expression) and a variety of implementing nonterminals are allowed.

Chapter 18

Some Example Languages

co-authored by Robert Heim

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 various parts of the manual.

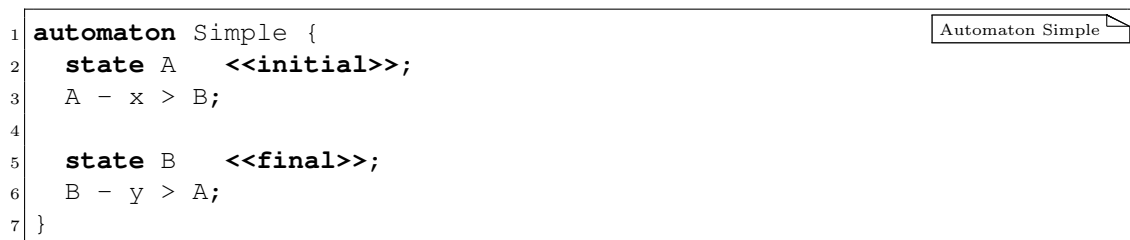
The main purpose is to demonstrate some features of the MontiCore language workbench. For a detailed and more complete overview of existing languages and tools behind that languages please see the forthcoming MontiCore language and tool book.

18.1 A Simple Automaton Language

The SAutomaton language (S is short for Simple) is a 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 18.1 shows a simple automaton having two states and two transitions and 18.2 shows an automaton describing how a ping-pong game is played. Graphical versions are displayed in Figure 18.3.

```
1 automaton Simple {  
2   state A    <<initial>>;  
3   A - x > B;  
4  
5   state B    <<final>>;  
6   B - y > A;  
7 }
```



Listing 18.1: 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 18.2: Example model for the Automaton language

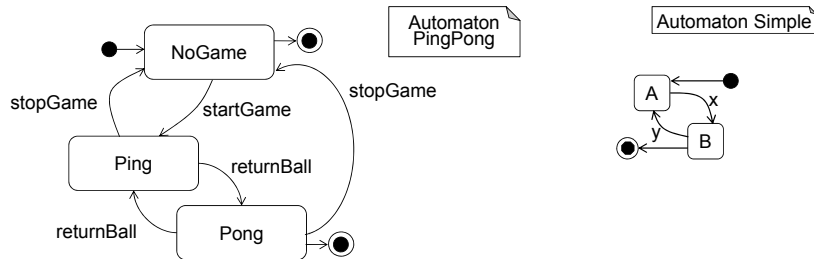


Figure 18.3: Two automata example models

As can be seen in these examples an automaton consists of a basic block (l. 2 and 14 in Listing 18.2), states (l. 3ff in Listing 18.2) and transitions (l. 7ff in Listing 18.2). 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 18.4.

```

1
2 grammar SAutomaton extends de.monticore.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 18.4: MontiCore grammar for the SAutomaton language

The grammar does not have a package declaration and immediately starts with the start nonterminal of the grammar. The grammar extends the base grammar MCBasics, which 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)* "}"
2
3 State = "state" Name (("«initial»" ) | ("«final»" ))* ";"
4
5 Transition = Name "-" Name ">" Name ";"

```

Listing 18.5: EBNF of the SAutomaton language

To understand the AST, it suffices to use the grammar shown in Listing 18.6 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 18.7. 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
2 grammar SAutomaton extends de.monticore.MCBasics {
3     Automaton = Name (State | Transition)* ;
4     State      = Name (["initial"] | ["final"])* ;
5     Transition = from:Name input:Name to:Name ;
6 }

```

Listing 18.6: MontiCore grammar for the SAutomaton language

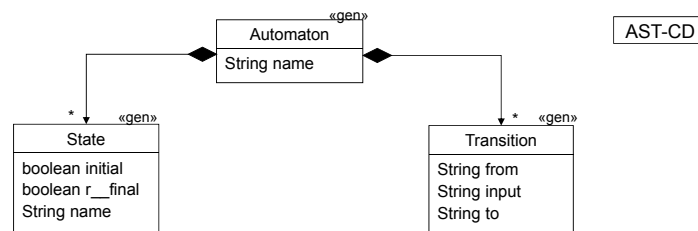


Figure 18.7: AST of the Simple Automaton language

For a deeper look into the class structure of the AST, the connection with the runtime classes provided by MontiCore are shown in Figure 18.8.

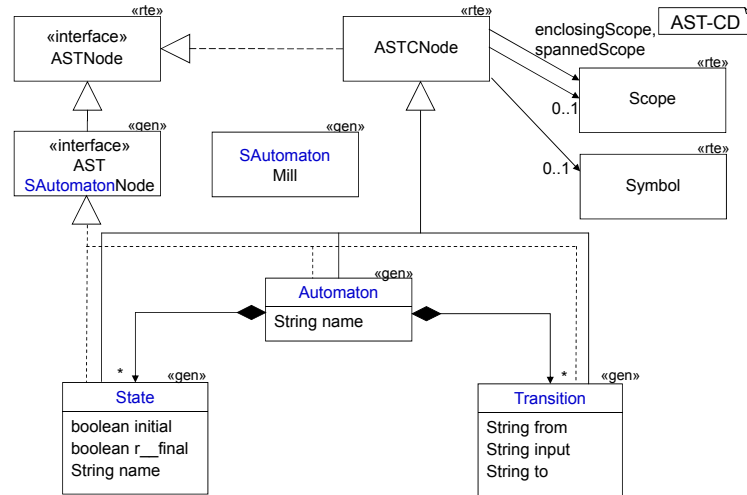


Figure 18.8: AST of the SAutomaton language extended by runtime classes

For the SAutomaton language the usual four forms of visitors are generated as depicted in Figure 18.9. 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 context condition checker, which is explained in Chapter 10 and later on in this chapter.

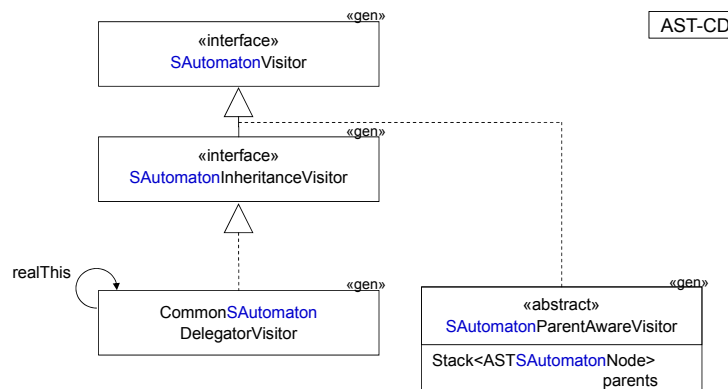


Figure 18.9: Visitors for the Automaton language

Two of the nonterminals defined in the SAutomaton grammar in Listing 18.4 are marked as symbols: Automaton and State. Furthermore, Automaton is marked as scope spanning. Transition sources and targets are references to states. Therefore, a symbol table

infrastructure is generated. This includes symbol, symbol kind and scope classes as well as a basic symbol table creator and a language class.

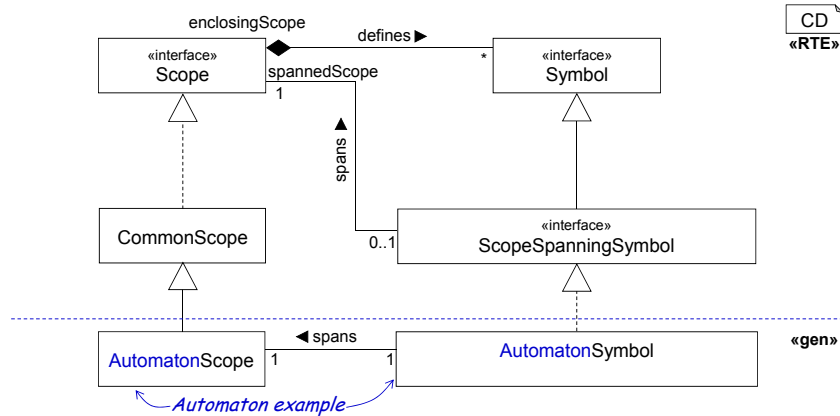


Figure 18.10: Symbol table structure of the Automaton language

The essentials of the generated symbol table infrastructure are shown in Figure 18.10. As described in Chapter 9 for every nonterminal X marked as a symbol the classes $XSymbol$, $XKind$ and $XResolvingFilter$ are generated. Thus, for the $SAutomaton$ language the following classes are generated:

- AutomatonSymbol and StateSymbol,
- AutomatonKind and StateKind, and
- AutomatonResolvingFilter and StateResolvingFilter

In addition, for every nonterminal Y marked as a scope a $YScope$ class is generated. Hence, for the nonterminal Automaton an AutomatonScope class is generated. Finally, for a language X the infrastructure to create and manage the symbols is generated: $XLanguage$, $XModelloader$, $XModelNameCalculator$, $XSymbolMill$ and $XSymbolTableCreator$.

1	Directory:	out/sautomaton/_symboltable	Files
2	Files:	SAutomatonLanguage	
3		SAutomatonModelloader	
4		SAutomatonModelNameCalculator	
5		SAutomatonSymbolMill	
6		SAutomatonSymbolTableCreator	
7		AutomatonSymbol	
8		AutomatonKind	
9		AutomatonResolvingFilter	
10		AutomatonScope	
11		StateSymbol	
12		StateKind	
13		StateResolvingFilter	

Listing 18.11: Summary of files generated for the symbol table of $SAutomaton$

Thus, for the SAutomaton language SAutomatonLanguage, SAutomatonModelloader, SAutomatonModelNameCalculator, SAutomatonSymbolMill and SAutomatonSymbolTableCreator are generated. Listing 18.11 summarizes the files generated for the symbol table of the SAutomaton language.

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 SAutomaton 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.

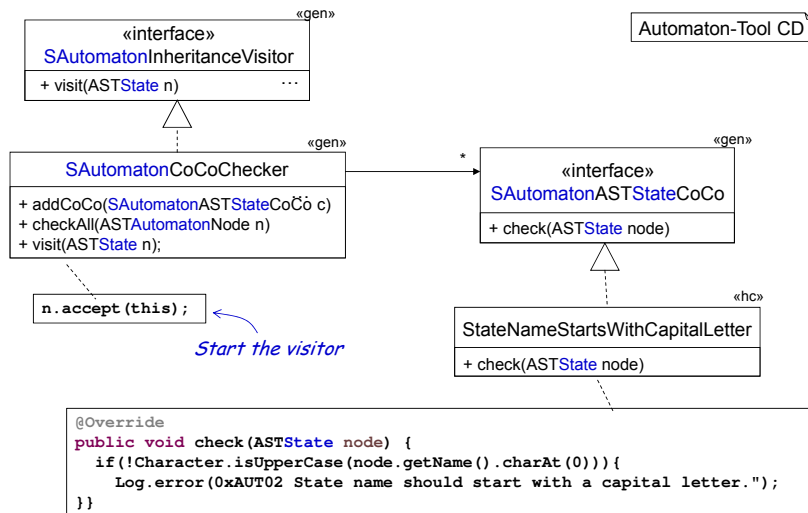


Figure 18.12: Defining context conditions for the SAutomaton language

For defining well-formedness rules MontiCore generated 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 18.12 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 18.12). The checker is implemented as a visitor. It traverses the AST and checks all context conditions suitable for the visited nodes. For the SAutomaton language the files listed in Listing 18.13 are generated for defining context conditions.

```

1 Directory:  out/sautomaton/_cocos
2 Files:      SAutomatonCoCoChecker
3             SAutomatonASTAutomatonCoCo
4             SAutomatonASTStateCoCo
5             SAutomatonASTTransitionCoCo
6             SAutomatonASTSAutomatonNodeCoCo

```

Listing 18.13: Summary of files generated for context conditions of SAutomaton

18.2 A Language for Hierarchical Automata

In the previous section the language SAutomaton for simple automata was explained and two example automata were considered. The next automaton in Listing 18.14 is not a valid model of the SAutomaton language, but is an extended version of the automaton in Listing 18.2. Here, the automaton has an additional state InGame that contains the two states Ping and Pong and their transitions. SAutomaton 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 18.14: Example model for the hierarchical Automaton language

As this language is an extension of the SAutomaton language, the grammar of HAutomaton (H for Hierarchical) extends the grammar of SAutomaton (l. 1 of Listing 18.15). Thereby all nonterminals of SAutomaton are inherited. HAutomaton uses the same starting nonterminal as SAutomaton (cf. in l. 6). As states can be hierarchical in HAutomaton, 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. 8).

```

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

```

Listing 18.15: MontiCore grammar for the HAutomaton language

The resulting AST structure is depicted in Figure 18.16. The AST classes of `SAutomaton` are reused for `HAutomaton`. For the redefined `State` nonterminal a new AST class `State` is generated that extends the `State` class of `SAutomaton`. As for `SAutomaton` the typical four types of visitors are generated. The grammar defines no symbols or scopes itself, thus no symbol or scope classes are generated. Instead, the symbols of `SAutomaton` can be reused for `HAutomaton`. However, as states are hierarchical in `HAutomaton` the symbol table creation needs to be reimplemented. For the new state nonterminal of `HAutomaton` a context condition interface is generated as well as a context condition checker.

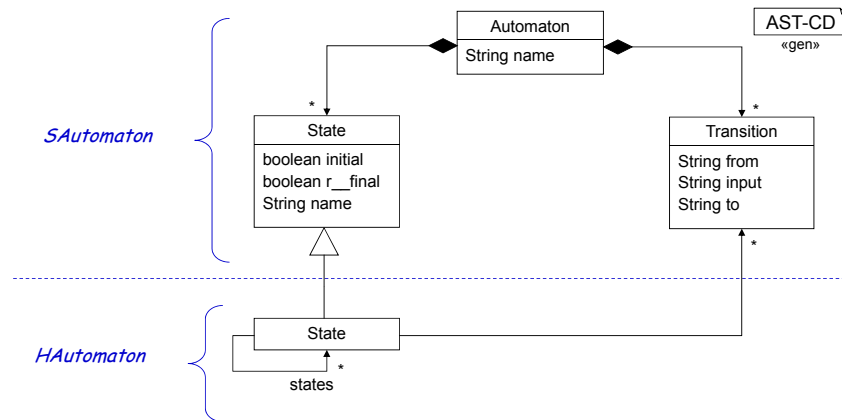


Figure 18.16: AST of the hierarchical Automaton language

18.3 A Language for Automata with Invariants

Another example of an automaton is shown in Listing 18.17. In comparison to the simple automata of `SAutomaton` states have invariants. A possible and flexible grammar for this kind of automata is shown in Listing 18.18.


```

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 18.17: Example model for the Automaton language with invariants

This grammar is called `IAutomatonComp` (I for Invariant and Comp for component) and defines states that have invariants. However, it does not yet define any syntax for invariants but uses an external nonterminal (cf. l. 7). Thus, this grammar is meant for extension and must be marked as a component (cf. l. 2). An extension of this grammar called `IAutomaton` (I for invariant) that defines a syntax for invariants is shown in Listing 18.19. `IAutomaton` overrides the external nonterminal `Invariant` and thereby defines a syntax for it. The resulting AST structure is shown in Figure 18.20.

```

1
2 component grammar IAutomatonComp extends de.monticore.MCBasics {
3
4   Automaton =
5     "automaton" Name "{" (State | Transition)* "}" ;
6
7   external Invariant;
8
9   State = "state" Name
10     Invariant ( "<<" ["initial"] ">>" | "<<" ["final"] ">>" ) * ";" ;
11
12   Transition =
13     from:Name "-" input:Name ">" to:Name ";" ;
14 }

```

Listing 18.18: Grammar Component for `IAutomatonComp` that defines state with invariants

For `IAutomatonComp` MontiCore generates the AST classes for `Automaton`, `State` and `Transition` and an interface `Invariant` for the external nonterminal. For `IAutomaton` MontiCore generates an interface for the interface nonterminal `LogicExpr`, AST classes for the nonterminals `Truth`, `Not` and `Var` as well as an AST class for the overriding nonterminal `Invariant`. The AST class for `Invariant` implements the interface `Invariant` of generated for `IAutomatonComp`.

```

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

```

Listing 18.19: Grammar for automata with state invariants

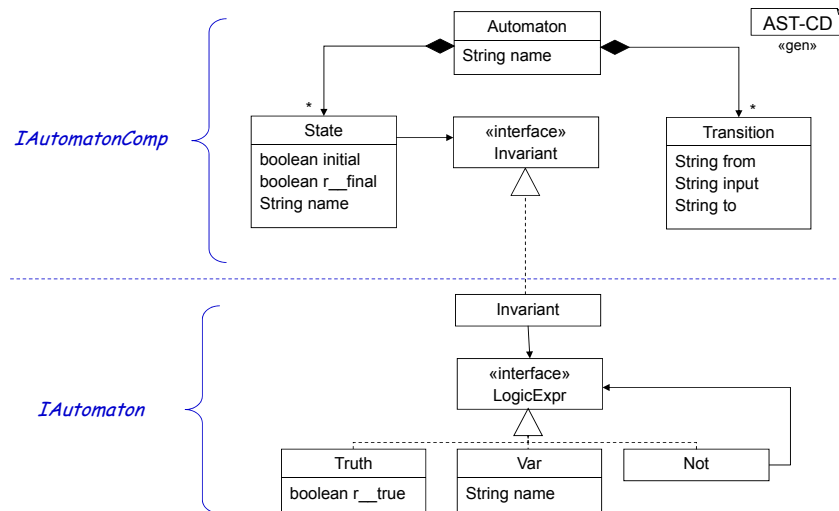


Figure 18.20: AST of the Automaton language with Invariants

As for SAutomaton the typical four types of visitors are generated for IAutomatonComp as well as for IAutomaton. Both grammar define no symbols or scopes, thus no symbol 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.

18.4 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 among several possible solutions in the following two sections.

18.4.1 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' ) ;

```

MCG Scannerless

Listing 18.21: 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 ;

```

MCG Scannerless

Listing 18.22: 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. Because `Expression` is recursively defined, 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.

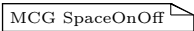
This form of grammar definition almost does not use the lexer and thus is called almost *scannerless*. It puts the burden of token aggregation to the parser and thus slows down the parsing process by a not neglectable factor. However, it exploits the parsing capabilities to full context-free parsing. Not only whitespaces are managed by the parser, but many other tokens are separated into their character sequences and managed by the parser directly.

18.4.2 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 18.23: 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 a number of preprocessed token 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 that do the switching. Unfortunately, these tokens need to be present in the input stream, which makes the process only usable for certain special cases. The below excerpt shows an impractical example for the ">>" operator:

```

1  ShiftExpression implements Expression <160> =
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 18.24: 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 this case.

18.4.3 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> =
2      leftExpression:Expression
3      (   shiftOp2:"<" "<" {noSpace()}?
4      |   shiftOp3:">" ">" {noSpace()}? ">" {noSpace()}?
5      )
6      rightExpression:Expression;
7
8  C = Name "." NoWSLast2 Name NoWSLast2 ;
9
10 NoWSLast2 = {noSpace()}? ;

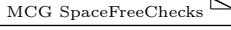
```

Listing 18.25: 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 nonterminal `C`.

The definition of method `noSpace()` needs expert knowledge about ANTLR4, which can e.g. be found on ANTLR4's website². However, the following definition of `noSpace()` works for all forms of token and can be used without any modification. Because of its general nature, it is provided by the MontiCore parser.

```
1  concept antlr {
2      parserjava {
3          // checks that there is no space between the last two token
4          // this function is used after the two tokens that are combined
5          public boolean noSpace() {
6              org.antlr.v4.runtime.Token t1 = _input.LT(-1);
7              org.antlr.v4.runtime.Token t2 = _input.LT(-2);
8              // token are on same line
9              // and columns differ exactly length of earlier token (t2)
10             return ((t1.getLine()==t2.getLine()) &&
11                     (t1.getCharPositionInLine()==
12                      t2.getCharPositionInLine()+t2.getText().length()));
13         }
14     }
15 }
```



Listing 18.26: Function `noSpace` disallows spaces between the last two tokens

18.5 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.

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 18.28.

²<http://www.antlr.org/>



Tip 18.27 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.

```

1 package test;
2 import static org.junit.Assert.*;
3 import de.monticore.scannerless._ast.*;
4 import de.monticore.scannerless._cocos.*;
5 import de.monticore.scannerless._symboltable.*;
6 import de.monticore.scannerless._parser.*;
7
8 public class CheckScannerlessTests {
9
10     // setup the language infrastructure
11     ScannerlessParser parser = new ScannerlessParser() ;
12
13     @BeforeClass
14     public static void init() {
15         // replacing log by a side effect free variant
16         LogStub.init();
17         Log.enableFailQuick(false);
18     }
19
20     @Before
21     public void setUp() {
22         Log.getFindings().clear();
23     }
24 }

```

Listing 18.28: Setting up a JUnit test infrastructure

A typical testing class, such as `CheckScannerlessTests` 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 18.29 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 }

```

Java CheckScannerlessTests

Listing 18.29: 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.

18.6 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 extends de.monticore.MCLiterals { MCG Questionnaire
2
3   symbol scope QDefinition = "questionnaire" Name "{"
4     ( Item | Scale ) *
5   "}";
6
7   symbol Item = "item" Name question:String (scale:Name | ScaleType);
8
9   symbol Scale = "scale" Name ScaleType;
10
11  interface ScaleType;
12
13  Range implements ScaleType = "range" "["
14    minTitle:String? min:IntLiteral
15    ".."
16    max:IntLiteral maxTitle:String?
17  "]" ;
18
19  Number implements ScaleType = "number";
20  Text implements ScaleType = "text" maxCharacters:IntLiteral?;
21
22  Select implements ScaleType = "select" "{"
23    options:SelectOption+
24  "}";
25
26  SelectOption = id:IntLiteral ":" title:String;
27 }
```

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 `MCLiterals` and therefore extends it.

As defined by production rule `QDefinition`, each questionnaire model starts with the keyword `questionnaire` (l. 3) followed by its name and an arbitrary count of `Items` and `Scales` enclosed by curly brackets (ll. 3-5).

An `Item` (l. 7) 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. 9) 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. 11). 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. 13-17) 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. 19) 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. 20) 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. 22-24). The keyword `select` introduces the modeling element and encloses the available options in curly brackets. Each `SelectOption` (l. 26) 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 19

Developer's View on MontiCore

authored by Robert Heim

In this chapter an overview of where to find MontiCore's source files is given. MontiCore is open source and was originally developed by the Software Engineering Group of RWTH Aachen University. MontiCore imports other sources, partly as open source. Consequently, the source code mentioned in this report is not fully located in one single location.

In general, Java is MontiCore's default programming language. Apache Maven [Fou17a] is a de facto standard to manage module dependencies and the build infrastructure of Java projects. Following this standard, MontiCore's source code is structured in Maven's default project layout [Fou17c]. One project object model (POM) [Fou17b] specifies one Maven project. The POM is located in a file called `pom.xml` in the root folder of a Maven project. The most important folders are explained in the following. On the top level the folder `src` separates source code from generated code. The latter is located in the `target` folder. The `src` folder contains a `main` folder for the productive source code and a `test` folder containing all 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, grammars (`src/main/grammars`) and models (`src/main/models`) are separated. Please note, that by default only the compiled source code (`target` folder) and resources of the `src/main/resources` folder are included when Maven packages the final jar. Also, the `target` folder must never be part of a version control system since it is generated.

Currently, MontiCore uses two project management tools (SSELab and GitHub). The SSELab project [Mon17a] is private. All source code specific to MontiCore that is not yet open source is located in its SVN repository [Mon17b]. Besides that a main purpose of the SSELab project is its ticket system for planning development cycles and to track bugs. However, it also contains MontiCore's Release Notes [Mon17c] as well as source code and documentation of outdated MontiCore versions. Additionally, some languages that are not yet open source are located in the `mclang` folder [Mon17d].

**Tip 19.1 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 *Maven project* [Fou17a] 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 Maven project is reusable in other Maven projects by defining a dependency on the deployed artifact. Most management projects consist of multiple Maven projects to structure modules of complex source code. Maven projects can be nested. E.g., MontiCore as well as its runtime are available as single jars that bundle the different specific modules.

19.1 MontiCore's GitHub Repository

MontiCore's source code itself is open source and located in the MontiCore GitHub repository¹. There are several branches which have their own purpose:

master contains the latest stable source code. On each release of MontiCore the master branch is updated from the dev branch.

dev is the developers branch on which developers actually work day to day.

ticketX other branches may be created in case huge changes or experimental features are implemented. To keep track of features the name of such a branch should match a ticket in MontiCore's ticket system. Such development should not interfere with the dev branch and, hence, is developed in its own branch until it is stable. Then, the branch is merged into the dev branch and the ticket branch can be deleted.

¹<https://github.com/MontiCore/monticore>



Tip 19.2 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 // Language-projects in development
11 Repository: https://sselab.de/lab2/private/svn/MontiCore/
12
13 // Packaged components, sources, snapshots etc.
14 Nexus:      https://nexus.se.rwth-aachen.de/
15
16 // Help for regular builds, continuous integration, etc.
17 CI:         http://ci.se.rwth-aachen.de/
18
19 // Results of daily checks for smells, etc.
20 Sonar:      http://metric.se.rwth-aachen.de/
21
22 // Management of open issues, enhancements, etc.
23 Tickets:    https://sselab.de/lab2/private/trac/MontiCore/
24
25 // Release notes about important changes, etc.
26 Release Notes:
27             https://sselab.de/lab2/private/intwiki/MontiCore/
28             index.php?title=Release_Notes_MontiCore

```

The *github* project is public. It contains the core project and a number of languages. The *SSELab* repository is private and contains languages still under development. Access can be granted upon request. The *Nexus* website provides convenient download of the newest versions of the required resources and allows a nice integration into maven project organisation. *Sonar* and *Jenkins* CI identify common problems early, automatically execute tests and build deployable artifacts.

The MontiCore project team internally uses the Trac ticket system to manage open issues.

19.1.1 External Developers and Forks

External developers should only work on the dev branch. It is also suboptimal to start too many branches. As a best practice, side developments should fork² the repository and work in their own fork. When finished, they must merge all new changes from MontiCore's dev

²<https://help.github.com/articles/fork-a-repo/>

branch and then create a merge request on the dev branch. MontiCore's core developers will review and possibly accept it to incorporate the work.

19.1.2 MontiCore's Maven Projects

The MontiCore GitHub Repository³ consists of several Maven 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. Here, the Groovy-based MontiCoreScript and all MontiCore configuration options are defined as well.

monticore-runtime provides everything required to execute MontiCore or a MontiCore-based DSL. This includes the Generator Engine (s. 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 and types as well as some helpers to work with these grammars (e.g., pretty printers).

The other Maven projects complement the core MontiCore functionalities:

monticore-cli provides MontiCore's Command Line Interface by basically wrapping the MontiCore generator script with a CLI.

monticore-editor contains core infrastructure supporting textual (`texteditor-core`) and graphical (`graphicaleditor-core`) model editors as well as the editor for the MontiCore Grammar (`grammar-editor`).

monticore-emf-runtime contains the runtime environment for EMF compatible MontiCore applications.

monticore-maven contains the Maven plugin⁴ that enables executing MontiCore as part of the Maven build lifecycle⁵.

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).

³<https://github.com/MontiCore/monticore>

⁴<https://nexus.se.rwth-aachen.de/monticore/latest/monticore-maven/monticore-maven-plugin/index.html>

⁵<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

19.2 Other Source Code Locations

Since the GitHub repository only includes source code specific to MontiCore, other locations are still relevant, too:

se-commons Located in an SSELab project⁶ the se-commons provide helpers that are common across multiple projects of the chair. The project itself is private, but the packaged libraries are world accessible through the Nexus⁷. The parts used by MontiCore are:

se-commons-logging contains the Log's source code as described in Section 15.3.

se-commons-groovy contains helpers such as the GroovyRunner and GroovyInterpreter to interpret Groovy scripts (cf. Chapter 16).

se-commons-utilities contains utility classes such as SourceCodePosition, CLIArguments, and the Names helper.

cd4analysis The CD4Analysis language is located as language within the MontiCore SSELab project⁸. It mainly focuses on class diagrams for early activities of the development project. It contains full assistance for associations, composition, qualifiers, etc., but does not define methods. There is a code generator available, that ensures consistency of the defined model, storage, etc.

automaton The Automaton language is located as language within the MontiCore SSELab project⁹. In the provided version, it serves mainly as teaching and toy example.

javaDSL The JavaDSL language is located as language within the MontiCore SSELab project¹⁰. The javaDSL is fully compatible with the Java 8 standard. It contains a complete grammar, symbol table infrastructure, context conditions, etc. The language is meant for either complete integration into a larger language or selection of individual parts, and probably particular Statement or Expression.

javaLib The JavaLib is located within the MontiCore SSELab project¹¹.

⁶<https://sselab.de/lab1/user/project/se-commons/>

⁷<https://nexus.se.rwth-aachen.de/>

⁸<https://sselab.de/lab2/private/svn/MontiCore/trunk/mclang/cd4analysis/>

⁹<https://sselab.de/lab2/private/svn/MontiCore/trunk/mclang/automaton/>

¹⁰<https://sselab.de/lab2/private/svn/MontiCore/trunk/mclang/javaDSL/>

¹¹<https://sselab.de/lab2/private/svn/MontiCore/trunk/mclang/javaLib/>

Chapter 20

Further Reading and Related Work from the SE Group, RWTH Aachen

20.1 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.” Modeling will be used in development projects much more, 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] 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 reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

20.2 Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivative 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]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. 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].

20.3 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 two books [Rum16] and [Rum12] 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 [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. 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].

20.4 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] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR⁺07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK⁺11]. 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.

20.5 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]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. 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] and the backend [RRRW15]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. 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.

20.6 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] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR⁺11] using deltas [HRRS11, HKR⁺11] and evolution on deltas [HRRS12]. [GHK⁺07] and [GHK⁺08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

20.7 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] and algebraically underpinned in [HKR⁺07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. 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.

20.8 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 [MRR11e] compares 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 [MRR11e] 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.

20.9 Evolution & Transformation of Models

Models are the central artifact 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], 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.

20.10 Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations.

Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK⁺08] 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 to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

20.11 Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] 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]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

20.12 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] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] 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 [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP⁺11].

20.13 Robotics

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 ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR⁺13] framework allows robotics experts and laymen to model robotic assembly tasks.

20.14 Automotive, Autonomic Driving & 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⁺08]. [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. 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.

20.15 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⁺11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

20.16 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.

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
2.1	Eclipse after importing the example project and executing MontiCore	14
2.2	IntelliJ IDEA after importing the example project and executing MontiCore	15
3.1	Structure of a generator - external view	18
3.2	Internal architecture of a generator	18
3.3	Chapter structure of the reference manual	20
4.28	Constraining the cardinality of a nonterminal, when parsing	38
5.2	Sequences of nonterminals in the AST	52
5.3	Optional nonterminals	52
5.4	How interfaces in a grammar map to the abstract syntax	53
5.5	How abstract nonterminals in a grammar map to abstract classes	53
5.7	Productions extending other productions	54
5.8	Implements in abstract and concrete syntax	54
5.9	Inheritance in abstract and concrete syntax	55
5.10	Extending the AST structure	55
5.11	Adding attributes in the AST with the <code>ast</code> statement	56
5.12	Adding methods in the AST with the <code>ast</code> statement	56
5.14	Terminals included in the AST	57
5.15	Choice of one of several values stored as <code>int</code>	58
5.16	Explicit definition of an enumeration	58
5.18	Common interfaces of AST classes	60

5.29	Example: Handwritten AST class <code>ASTState</code> injected into the parsing process	71
7.9	Language inheritance	93
7.11	Composition of a language is executed as late as possible: late binding	97
8.5	Visitor for the <code>Questionnaire</code> language and an handcoded usage	105
8.7	Generated visitors for language <code>L</code> extend/implement the default visitor	107
8.10	Inheritance visitor calls <code>visit</code> hooks for all super types	108
8.12	Additional API methods of the parent aware visitor	110
8.15	Overview of visitor classes for <code>Automaton6</code>	112
8.21	Overview of visitors for <code>Automaton3</code>	116
9.2	Overview of the main concepts of SMI	123
9.6	Technical symbol classes provided by SMI plus language specific classes	126
9.10	Technical scope classes provided by SMI	129
9.12	Relationship between AST, symbol, and scope	131
10.1	CoCo infrastructure for nonterminals	135
10.2	The generated CoCo checker	135
11.3	Components use <code>realThis</code> instead of <code>this</code> to enable close collaboration	146
11.4	Composing objects using the <code>realThis</code> approach	147
11.5	Sequence diagram showing the runtime program flow in a <code>realThis</code> composition	147
11.6	Template hook pattern variants for integration of handwritten and generated code	148
13.17	Hierarchical include structure induced by the <code>include</code> commands	172
13.18	Decoration before and after a template (execution order is 1..5)	172
13.19	External replacement of a template	173
13.20	Defining an explicit hook point and binding it	173
13.21	External template replacement keeps its decoration (execution order is 1..5)	174
14.2	How a given class can be extended by building a subclass	185
14.4	How a given class can be replaced by a handwritten class	185

14.6	How a given class and its builder can be replaced by handwritten versions .	187
14.7	How a given class and its subclass can be replaced by a handwritten class .	187
15.10	Relationship between artifacts (templates and Java, excerpt)	203
15.11	Relationship between template artifacts (excerpt)	203
18.3	Two automata example models	230
18.7	AST of the Simple Automaton language	231
18.8	AST of the SAutomaton language extended by runtime classes	232
18.9	Visitors for the Automaton language	232
18.10	Symbol table structure of the Automaton language	233
18.12	Defining context conditions for the SAutomaton language	234
18.16	AST of the hierarchical Automaton language	236
18.20	AST of the Automaton language with Invariants	238

Listings

1.3	Example in Java	4
3.4	Example tool for the Automaton DSL	21
4.1	Minimal grammar example	24
4.2	Lexical productions for SimpleName and SimpleString	25
4.3	Lexical productions for Numbers using token fragments	26
4.4	Lexical productions for white spaces	26
4.5	Lexical production for strings without quotation marks, which are removed in a Java action	27
4.6	Changing the result type of lexicals	27
4.7	Add a conversion method for lexical types	27
4.11	Some production examples	30
4.12	Augmentation of terminals for storage in the AST	31
4.13	A choice of alternate terminals is stored as integer	31
4.14	Explicit definition of an enumeration	32
4.15	Automatic naming of unnamed nonterminals	32
4.17	An interface nonterminal and several nonterminals implementing it	33
4.18	Alternative to interface nonterminal in Listing 4.17 accepting the same con- crete syntax, but I knows A and B	33
4.19	Interface nonterminal defining its signature	34
4.20	Extending the production of a nonterminal	34
4.21	Equivalent alternative to extension in Listing 4.20 accepting the same con- crete syntax, but A knows and thus is coupled to B	34
4.23	Abstract production in a grammar	35
4.24	Alternative to abstract nonterminal in Listing 4.23 accepting the same con- crete syntax	35
4.25	Explicitly setting a top-level nonterminal that is inherited with start	36
4.26	Example for an expression language using priorities for its infix operations	36
4.29	Constraining the cardinality of a nonterminal	38
4.30	Add Java code to the parser	39
4.31	Add Java code to the lexer	39
4.33	EBNF of the MontiCore grammar MCG	45
5.17	Signature of the ASTNode superclass of all AST nodes	59
5.19	Signature of the generated AST class to represent states: part 1	61
5.20	Attribute management signature of a generated AST class: part 2	62
5.21	Signature for a List attribute in a generated AST class: part 3	63
5.22	Comparison and cloning in a generated AST class: part 4	64

5.24	EMF version of the <code>ASTState</code> class signature	65
5.25	Signature of the builder mill for all <code>Automaton</code> AST classes	66
5.26	Signature of the <code>Builder</code> for <code>State</code> objects: part 1	67
5.27	Retrieving methods for a <code>Builder</code> class: part 2	69
5.31	Internal structure of the <code>AutomatonMill</code>	72
5.32	Handcoded extension of the <code>AutomatonMill</code>	73
6.1	Location of the <code>MontiCore</code> parser generator	75
6.2	Method signature used to generate a parser	76
6.3	Java code creates a parser for automata (using its grammar)	76
6.4	List of files produced during the generation of a parser	77
6.5	Methods that can be used for parsing	78
6.6	Various forms of parsing	79
6.8	Where to find the <code>MontiCore</code> grammar grammar	80
7.2	Example of a grammar component with an external nonterminal	88
7.3	External nonterminals are mapped to interfaces in the AST	89
7.4	Language embedding with binding the external nonterminal	89
7.5	Implementation of the <code>Invariant</code> nonterminal	90
7.7	Language inheritance: One grammar extending another and redefining an inherited nonterminal	91
7.8	The new <code>ASTState</code> class extends the old <code>ASTState</code> class and serves as substitute	92
7.10	Language embedding: Filling extension points	94
8.1	Signature of a <code>Visitor</code> for language <code>L</code>	104
8.6	Simplified presentation of a <code>handle</code> operation (omitting composability) . .	107
8.9	Implementation of an inheritance visitor <code>handle</code> method	108
8.13	<code>Automaton</code> language with interface nonterminal <code>AutElement</code> used for ex- tension	111
8.14	Adding transitions with output to the <code>Automaton5</code> language of Listing 8.13	111
8.16	A visitor of the new language reuses an original language's visitor	112
8.17	<code>Automaton</code> language without explicit extension point	113
8.18	Conservative extension of transitions from <code>Automaton15</code> of Listing 8.17 . .	114
8.19	A visitor of the new language reuses an original language's visitor	114
8.20	The realThis implementation of a compositional visitor	115
8.23	The implementation of the pretty printer for the <code>Automaton3</code> sublanguage (with only one nonterminal)	117
8.24	Composing the infrastructure for several visitors with option to override behavior delegation	118
8.25	Composing the three visitors through delegation and giving them the same shared state	118
8.26	The composed visitors can be used as if it is only one monolithic component	119
9.3	<code>Automaton</code> with counters and transition statements	124
9.4	Generated implementation of class <code>StateKind</code>	125
9.5	Generated implementation of class <code>StateSymbol</code>	125

9.7	Interface of all Symbol classes	126
9.8	Generated implementation of class StateScope	128
9.9	Signature of MutableScopes	128
9.13	Extended signature of ASTTransitions	131
10.3	Implementation of the AutomatonCoCoChecker class	136
10.4	Configure the AutomatonCoCoChecker and check the context conditions	137
10.6	Implementation of a context condition for State objects	138
10.8	Using the symbol table in a context condition	138
10.9	Initial setup to test a context condition	139
10.10	Testing a context condition on a valid model	140
10.11	Testing a context condition on an invalid model	141
11.1	A static delegator method	144
11.2	Customized static delegator method	144
12.1	Principle of FreeMarker: Copy the template content, execute FreeMarker commands, and inject their results into the output	150
12.2	Result when applying the template	150
12.5	FreeMarker conditional	153
12.6	FreeMarker switch statement	153
12.7	FreeMarker loop	154
12.8	Example for a FreeMarker loop	154
12.9	Extended form of a FreeMarker loop	154
13.1	Signature of a generate and generateNoA method	158
13.2	How the GeneratorEngine can be used	159
13.3	Configuration options of the GeneratorSetup class	160
13.4	Shortcuts: Aliased functions available in templates	163
13.6	Include methods provided by the TemplateController tc	164
13.7	Examples for including sub-templates within a template	165
13.8	The includeArg methods provided by the TemplateController tc	165
13.9	Examples for using signature	166
13.11	Write methods provided by the TemplateController tc	167
13.13	Further methods provided by the TemplateController tc	168
13.14	Logging examples from within templates	169
13.15	Methods to manage global variables with glex	170
13.16	Manipulating global variables from within a template	171
13.23	Signature that HookPoints provide	175
13.24	Constructor for StringHookPoints	175
13.25	Constructors for TemplateHookPoint	176
13.26	Constructor of TemplateStringHookPoint	176
13.27	Methods to define a hook point in a template	177
13.28	Methods of the GlobalExtensionManagement class for hook point man- agement	178
13.29	Example: setting a hook point	179
13.30	GlobalExtensionManagement for hook point management	180

15.2	Logging API in class Log	194
15.3	Example for controlling fail quick	195
15.4	How to enable reporting	198
15.5	How to stop and start reporting	199
15.6	Additional information reported in 08_Detailed.txt	199
15.7	Exemplaric object identifiers in reports	199
15.8	Object identifiers in reports	200
15.9	Representation of various entities	200
16.1	Groovy script used to generate the standard result	207
16.2	Methods available in the Groovy scripts	208
18.1	Simple automaton in text format	229
18.2	Example model for the Automaton language	230
18.4	MontiCore grammar for the SAutomaton language	230
18.5	EBNF of the SAutomaton language	231
18.6	MontiCore grammar for the SAutomaton language	231
18.11	Summary of files generated for the symbol table of SAutomaton	233
18.13	Summary of files generated for context conditions of SAutomaton	235
18.14	Example model for the hierarchical Automaton language	235
18.15	MontiCore grammar for the HAutomaton language	236
18.17	Example model for the Automaton language with invariants	237
18.18	Grammar Component for IAutomatonComp that defines state with invariants	237
18.19	Grammar for automata with state invariants	238
18.21	Whitespaces made explicit in the productions	239
18.22	Whitespaces explicitly used in productions	239
18.23	Whitespaces temporarily explicit in the production using a state switch in the lexer	240
18.24	Switching whitespaces on and off temporarily in productions	241
18.25	Disallowing spaces between last two token	241
18.26	Function noSpace disallows spaces between the last two tokens	242
18.28	Setting up a JUnit test infrastructure	243
18.29	Some JUnit tests	244

References

- [AHRW17] 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.
- [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.
- [Bec15] Kent Beck. *JUnit Pocket Guide*. O’Reilly, 2015.
- [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.
- [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.
- [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.

-
- [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.
- [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⁺11] 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.
- [Fou17a] Apache Software Foundation. Maven, 2017. <http://maven.apache.org/> [Online; accessed 15-December-2017].
- [Fou17b] Apache Software Foundation. Maven Documentation - POM, 2017. <http://maven.apache.org/pom.html> [Online; accessed 15-December-2017].
- [Fou17c] Apache Software Foundation. Maven Documentation - Standard Directory Layout, 2017. <http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html> [Online; accessed 15-December-2017].
- [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.
- [Fre17] FreeMarker website. <http://freemarker.org/>, 2017.
- [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⁺08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

- [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 domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.
- [GKR⁺07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.
- [GKR⁺08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.
- [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.
- [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.
- [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 13-November-2017].
- [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.

-
- [Gro17] GraphML Working Group. The graphml file format, 2017. <http://graphml.graphdrawing.org/> [Online; accessed 13-November-2017].
- [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⁺15] 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.
- [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.
- [HMSNRW16a] 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.
- [HMSNRW16b] 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)*, pages 67–82. Springer, 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.
- [HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.
- [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.
 - [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.
 - [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.
 - [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*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.
- [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.
- [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 Stefan 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.
- [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.
- [MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.
- [Mon17a] MontiCore. MontiCore’s Privat SSELab Project, 2017. <https://sselab.de/lab1/user/project/MontiCore/> [Online; accessed 15-December-2017].
- [Mon17b] MontiCore. MontiCore’s Private SVN Repository, 2017. <https://sselab.de/lab2/private/svn/MontiCore/trunk/> [Online; accessed 15-December-2017].
- [Mon17c] MontiCore. MontiCore’s Release Notes, 2017. <https://sselab.de/lab2/private/svn/MontiCore/index.php> [Online; accessed 15-December-2017].
- [Mon17d] MontiCore. Repository of MontiCore’s Private Languages, 2017. <https://sselab.de/lab2/private/svn/MontiCore/trunk/mclang/> [Online; accessed 15-December-2017].
- [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. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.
- [MRR11e] 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.

- [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.
- [MRR14] 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.
- [MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2017. To appear.
- [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.
- [QOS17a] QOS.ch. Logback project, 2017. <http://logback.qos.ch/> [Online; accessed 13-November-2017].
- [QOS17b] QOS.ch. Simple logging facade for java, slf4j, 2017. <http://www.slf4j.org/> [Online; accessed 13-November-2017].
- [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. Shaker Verlag, 2017.

- [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.
- [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.
- [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.
- [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.
- [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.
- [yWo17] yWorks. yEd Graph Editor, 2017. <http://www.yworks.com/en/products/yfiles/yed/> [Online; accessed 13-November-2017].
- [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

`${...}`, 150
_ast, 9
_channel, 26
_cocos, 9
_od, 9
_parser, 9
_symboltable, 9
_visitor, 9
-cl, 197
-customLog, 197
-dev, 197
-d, 197
-g, 90
-handcodedPath, 160
-hcp, 184, 185
-mp, 90
-out, 184
-script, 205, 206
-s, 205, 206
-templatePath, 161
.now, 152
01_Summary.txt, 201
02_GeneratedFiles.txt, 201
03_HandwrittenCodeFiles.txt, 201
04_Templates.txt, 201
05_HookPoint.txt, 201
06_Instantiations.txt, 201
07_Variables.txt, 201
08_Detailed.txt, 201
09_TemplateTree.txt, 202
10_NodeTree.txt, 202
11_NodeTreeDecorated.txt, 202
12_TypesOfNodes.txt, 202
13_SymbolTable.txt, 202
14_Transformations.txt, 202
15_ArtifactGml.gml, 203
16_ArtifactGv.gv, 203
17_InputOutputFiles.txt, 204
18_ObjectDiagram.txt, 202
</#directive>, 153
<#-- ... --#>, 150
<#assign name=value>, 153
<#case ...>, 153
<#compress>, 154
<#default>, 153
<#directive parameters>, 153
<#else>, 153, 154
<#elsif ...>, 153
<#if ...>, 150, 153
<#items ...>, 154
<#list ...>, 154
<#switch ...>, 153
<rightassoc>, 35
?date, 151
ASTAutomaton3Node, 62
ASTAutomatonBuilder, 66
ASTCNode, 60–62, 69
ASTConstantsGr, 58
ASTNodeBuilder
 build, 67
ASTNodeBuilder, 67
 EnclosingScope, 67
 PostComments, 67
 PreComments, 67
 SourcePositionEnd, 67
 SourcePositionStart, 67
 SpannedScope, 67
 Symbol, 67
 deepClone, 67
 deepEqualsWithComments, 67
 deepEquals, 67
 equalAttributes, 67
 equalsWithComments, 67
ASTNode, 55, 58–61, 127, 132
 EnclosingScope, 59
 PostComments, 59

- PreComments, 59
- SourcePositionEnd, 59
- SourcePositionStart, 59
- SpannedScope, 59
- Symbol, 59
- deepClone, 59
- deepEqualsWithComments, 59
- deepEquals, 59
- equalAttributes, 59
- equalsWithComments, 59
- ASTStateBuilder, 66–69
- ASTStateTOP, 71, 185
- ASTState, 61, 62, 64, 71, 130
- ASTTransitionBuilder, 66
- AddExpression, 226
- Aliases, 163
- Arguments, 225
- ArrayExpression, 225
- ArtifactScope, 129
- AssignmentExpression, 226
- AutElement, 111
- Automaton15, 113
- Automaton16, 113
- Automaton3PrettyPrinter, 118
- Automaton3Tool, 119
- Automaton3, 115
- Automaton5, 111
- Automaton6PrettyPrinter, 112
- Automaton6, 111
- AutomatonASTAutomatonCoCo, 134
- AutomatonASTStateCoCo, 134
- AutomatonASTTransitionCoCo, 134
- AutomatonCoCoChecker, 135
- AutomatonElement, 53
- AutomatonInheritanceVisitor, 136
- AutomatonLanguage, 132
- AutomatonMillTOP, 73
- AutomatonMill, 66, 72, 73
- AutomatonModelLoader, 132
- AutomatonModelNameCalculator, 132
- AutomatonParser, 19, 79
- AutomatonScope, 130
- AutomatonSymbolMill, 132
- AutomatonSymbolTableCreator, 132
- Automaton, 62, 91, 123, 127, 135, 231
- BinaryAndOpExpression, 226
- BinaryOrOpExpression, 226
- BinaryXorOpExpression, 226
- BooleanAndOpExpression, 226
- BooleanNotExpression, 226
- BooleanOrOpExpression, 226
- BracketExpression, 228
- CD4A, 4
- CD4Code, 156
- CD, 4
- CallExpression, 225
- Cardinality, 27, 222
- CharLiteral, 215
- CharToken, 215
- CheckScannerlessTests, 243
- ClassExpression, 225
- Cloneable, 61
- CodeHookPoint, 175, 176
- CommonScope, 127–129
- CommonSymbolTableCreator, 132
- CommonSymbol, 125, 126
- ComparisonExpression, 226
- Completeness, 223
- ConditionalExpression, 226
- Counter, 123
- DecimalToken, 218
- Decimal, 218
- EObjectContainmentEList, 65
- EnclosingScope, 59, 67
- Expression, 93, 94, 225, 228
- Ext, 89
- FileReaderWriter, 161
- FreeMarkerTemplateEngine, 161
- GenerateAutomatonParser, 76
- GeneratorEngine, 157, 159, 168
- GeneratorSetup, 157, 159, 160, 164
- GenericInvocationExpression, 225
- GenericInvocationSuffix, 225
- GlobalExtensionManagement, 162, 170, 178–180, 210
- GlobalExtensionMangement, 161
- GlobalScope, 128, 130
- Grammar_WithConcepts.mc4, 79
- Grammar_WithConceptsParser, 79
- HAutomaton, 235
- HIDDEN, 240
- Hash, 151

- HexInteger, 220
- HexadecimalToken, 219
- Hexadecimal, 219
- HierarchicalAutomaton, 91
- HookClass, 148
- HookPoint, 174
- IAutomatonComp, 237
- IAutomaton, 237
- IdentityExpression, 226
- ImportStatement, 29, 221
- IncrementalChecker, 211
- Increment, 123
- InputOutputFilesReporter, 211
- InstanceofExpression, 225
- Integer, 218
- InvAutomatonPrettyPrinter, 115
- InvAutomaton, 94, 115
- Invariant, 88, 89, 237
- IterablePath, 210
- JAVA_HOME, 7
- JavaCopyright, 178
- LCoCoChecker, 135
- LDelegatorVisitor, 104, 109, 116
- LInheritanceVisitor, 104, 107
- LParentAwareVisitor, 104, 109
- LVisitor, 104
- List, 63
- LiteralExpression, 228
- Log.init(), 197
- Log.initDEBUG(), 197
- Log.initWARN(), 197
- LogStub.init(), 197
- LogStub, 139, 141, 144, 193, 198
- LogicExpr, 89, 94, 237
- LogicalNotExpression, 226
- Log, 143, 144, 162, 163, 169, 193, 197, 211
- MCBasicTypes1, 221
- MCBasics, 28, 95, 213
- MCExpressions, 225
- MCHexNumbers, 219
- MCLiterals, 28
- MCNumbers, 217
- ML_COMMENT, 214
- ModelPath, 210
- Modifier, 224
- MontiCoreReports, 210
- MontiCoreScript, 198, 205, 208–210
- MultExpression, 226
- MutableScope, 128, 129
- MyTransitionBuilder, 73
- NEWLINE, 214
- NameExpression, 228
- Names, 211
- Name, 52, 63, 124, 131, 214
- Not, 237
- Number, 217
- Observer, 54
- Optional, 63
- PATH, 7
- ParserGenerator, 75
- PingPong, 230
- PostComments, 59, 67
- PreComments, 59, 67
- PrefixExpression, 226
- PrimaryGenericInv.Expression, 225
- PrimarySuperExpression, 225
- PrimaryThisExpression, 225
- QualifiedNameExpression, 225
- QualifiedName, 29, 221
- QuestionnaireTool, 104
- Questionnaire, 244
- ReportManager, 198
- Reporting.flush(), 199
- Reporting.off(), 199
- Reporting.on(), 199
- Reporting.reportToDetailed(), 199
- Reporting, 199, 211
- SAutomaton, 229, 230
- SL_COMMENT, 214
- Scannerless, 239
- Sequence, 151
- ShiftExpression, 228
- SimpleName, 25
- SimpleString, 25
- Slf4jLog, 193
- SourcePositionEnd, 59, 67
- SourcePositionStart, 59, 67
- SpaceOnOff, 240
- SpannedScope, 59, 67
- StateKind.java, 124
- StateKind, 125

StateNameStartsWithCapitalLetter, `_ast`, 51
137
StateResolvingFilter.java, 124
StateScope, 127, 130
StateSymbol.java, 124
StateSymbol, 125, 130
Statement, 123
State, 89, 111, 113, 231
StereoValue, 222
Stereotype, 222
StringHookPoint, 175
StringLiterals, 215
StringLiteral, 216
StringToken, 216
String, 52
SuffixExpression, 226
SuperExpression, 225
SuperSuffix, 225
SymbolKind, 125
SymbolTableCreator, 132
Symbol, 59, 67, 125, 126
TOP mechanism, 148
TemplateClass, 148
TemplateController, 162, 163, 168
TemplateHookClass, 148
TemplateHookPoint, 166, 175, 176
TemplateStringHookPoint, 175, 176
ThisExpression, 225
TransitionSourceExists, 138
TransitionWithoutOutput, 111
Transition, 111, 113, 231
Truth, 237
TypeCastExpression, 225
Types, 28, 29
UMLModifier, 224
UMLStereotype, 222
UML, 1
WSOff, 241
WSOn, 241
WS, 214
WhiteSpace, 239, 240
XYAntlr.g4, 77
XYAntlrLexer, 77
XYAntlrParser, 77
XYParser, 77
\${...}, 150
`_ast`, 51
`_automatonBuilder`, 72
`_channel`, 240
`_cocos`, 51
`_parser`, 51
`_reachable`, 71
`_report`, 51
`_stateBuilder`, 72
`_symboltable`, 124
`_transitionBuilder`, 72
`_visitor`, 51
abstract, 62
accept, 62, 106
addCoCo, 135
addSubScope, 128
addToGlobalVar, 163, 170
additionalTemplatePaths, 160
assign, 170
astextends, 54
astextend, 55
astimplements, 54
ast, 55, 158, 162, 165
automatonBuilder, 66
automaton, 251
bindHookPoint, 163, 178, 181, 182
bindStringHookPoint, 178
bindTemplateHookPoint, 178
bind, 173
boolean, 29
build, 67, 68
cd4analysis, 251
changeGlobalVar, 163, 170
checkAll, 135, 136
check, 134
commentEnd, 161
commentStart, 161
component, 87, 88
createFromAST, 132
createGlobalScope, 208, 209
createSymbolsFromAST, 208, 209
debug, 163, 169, 192, 193
decorateCd, 208, 209
decorateEmfCd, 208, 209
deepClone, 59, 64, 65, 67
deepEqualsWithComments, 59, 64, 65, 67

- deepEquals, 59, 60, 64, 65, 67
- defaultFileExtension, 160
- defineGlobalVars, 163
- defineGlobalVar, 163, 170
- defineHookPoint, 163, 177, 178
- dev, 248
- doInfo, 143
- eGet, 65
- eIsSet, 65
- eNotify, 65
- eSet, 65
- eUnset, 65
- enableFailQuick, 193, 195
- endVisit, 104
- enum, 31, 58
- equalAttributes, 59, 64, 65, 67
- equalsWithComments, 59, 64, 67
- error, 163, 169, 192, 193
- existsHandwrittenFile, 168, 169
- existsHookPoint, 163, 177, 178
- extends, 34, 91
- external, 87, 88
- flush, 199
- ftl, 149
- generateEmfCompatible, 208, 209
- generateFullParser, 76
- generateNoA, 157, 158
- generateParser, 208, 209
- generateSymbolTable, 208, 209
- generate, 157, 158, 208, 209
- getAstNode, 125
- getEnclosingScope, 125, 128
- getErrorCount, 193
- getFullName, 125
- getGlobalVar, 163, 170
- getKind, 125
- getName, 125
- getPackageName, 125
- getParents, 109
- getParent, 109
- getQName(), 221
- getSourcePosition, 125
- getSource, 217
- getStateNode, 125
- getSubScopes, 128
- getSymbol, 127
- getTemplatename, 168
- getValue(), 29
- getValue, 217
- glex, 161, 162, 165, 170, 210
- grammarIterator, 210
- handCoded, 71
- handcodedPath, 71, 160, 210
- handle, 104, 105
- hasGlobalVar, 170
- hwc, 9
- import, 87
- includeArgs, 163, 165, 166
- include, 163, 164
- info, 143, 163, 169, 192, 193
- init(), 144
- init, 95
- instantiate, 168, 169
- interface, 62, 87
- int, 29
- isFailQuickEnabled, 193, 195
- isKindOf, 125
- isReachable, 71
- isShadowingScope, 129
- isStar(), 221
- javaDSL, 251
- javaLib, 251
- master, 248
- mc4, 76
- method, 56
- monticore-cli, 250
- monticore-editor, 250
- monticore-emf-runtime, 250
- monticore-generator, 250
- monticore-grammar, 250
- monticore-maven, 250
- monticore-runtime, 250
- monticore-templateclassgen., 250
- monticore.YYYY-MM-DD-HH:mm:ss.log,
9
- monticore_emf.groovy, 205
- monticore_noemf.groovy, 205, 206
- name:Name, 124
- noSpace(), 242
- null, 63, 69
- outputDirectory, 160
- out, 9

- parseGrammars, 208, 209
- parseGrammar, 208, 209
- parseNT, 78
- parse_StringNT, 78
- parse_String, 78
- parse, 78
- pom.xml, 247
- processValue, 175
- reachableStates, 56
- realThis, 146
- replaceTemplate, 180–182
- replacetemplate), 173
- reportGrammarCd, 208, 209
- reportManagerFactory, 198, 210
- reportToDetailed, 199
- requiredGlobalVars, 163, 170
- requiredGlobalVar, 163, 170
- resolveMany, 128
- resolve, 128
- runGrammarCoCos, 208, 209
- scope, 127
- se-commons-groovy, 251
- se-commons-logging, 251
- se-commons-utilities, 251
- se-commons, 251
- setAfterTemplate, 173, 180–182
- setAstNode, 125
- setBeforeTemplate, 173, 180–182
- setEnclosingScope, 125
- setSymbol, 127
- signature, 159, 163, 165, 175
- spaceOnFlag, 240
- src/main/grammars, 247
- src/main/java, 247
- src/main/models, 247
- src/main/resources, 247
- src/test/java, 247
- src/test/resources, 247
- src, 9
- start, 35, 91
- stateBuilder, 66
- symbolIsPresent, 127
- symbol, 123, 124
- target, 247
- tc, 162, 164, 165, 168
- templatePath, 210
- trace, 163, 169, 192, 193
- tracing, 161
- transformAstGrammarToAstCd, 208, 209
- transitionBuilder, 66
- traverse, 104, 105
- varName??, 152
- visit, 104, 135
- warn, 163, 169, 192, 193
- writeArgs, 167
- write, 167
- abstract, 34
- abstract nonterminal, 34, 53
- abstract syntax, 23, 51, 85
- abstract syntax (AST), 18
- abstract syntax tree, 19
- action, 26
- adaptation
 - subclass, 184
 - TOP mechanism, 185
- Agile Model-Based Development, 6
- agile modelling, 5
- Ansi-C++, 2
- ANTLR, 23, 37, 38, 44, 77, 206
 - concept, 38
 - lexerjava, 38
 - parserjava, 38
- arrays, 29
- artifact, 81, 183
 - generated, 183
 - handcoded, 183
- artifact scope, 129
- ASCII, 44
- AST, 19, 51
 - extension, 55
 - handwritten extension, 70
 - spanning tree, 121
- AST classes, 51
- AST-access-conservative, 99
- AST-conservation, 92
- AST-conservative, 99, 110
- AST-modification-conservative, 99, 100
- Automaton DSL, 19
- autonomous driving simulation, 2
- AutoSar, 2

- axiom, 35
- backend, 87
- black-box reuse, 81
- body, 29
- builder, 66, 85, 95, 184, 186
 - AST-conservative extension, 95
 - handwritten extension, 72
 - initialization, 95
- builder mill, 66, 95
 - handwritten extension, 72
 - initialization, 95
- builder mills, 184
- building facilities, 2
- cardinality, 37
- CD4Analysis, 251
- Char, 28
- checker, 135
- CI, 247
- Class Diagrams for Analysis, 4
- CLI, 206
- cloud service, 2
- code hook, 174
- comma separated lists, 32
- command line, 7, 8
- Command Line Interface, 250
- comparison, 60
- Compiling, 9
- component grammar, 83
- composition, 81
 - abstract syntax, 85
 - backend, 87
 - builder, 85
 - builder mill, 95
 - concrete syntax, 84
 - conservative, 83
 - context conditions, 86
 - late binding, 82
 - objects with `realThis`, 145
 - parser, 85, 96
 - parser CS-conservative, 96
 - scopes, 86
 - semantic, 82
 - symbols, 86
 - visitors, 86
- concept, 38
- conceptual distance, 155
- concrete syntax, 84
- conservative, 83
- conservative extension, 97
- console output, 192
- context condition, 133
 - check, 135
 - implemenation, 134
 - implementation, 137
 - symbol table use, 137
 - tests, 139
- context condition infrastructure, 134
- context condition interface, 134
- context conditions, 86
- context-free grammar, 121, 133
- context-free parser, 238
- context-free syntax, 23
- context-sensitive restrictions, 133
- continuous integration, 247
- CS-AST-compliance, 92
- CS-conservation, 92
- CS-conservative, 98
- data structure, 23
- debug message, 191
- DecimalDoublePointLiteral, 28
- DecimalIntegerLiteral, 28
- decorator hook point, 171
- delegation pattern, 145
- DelegatorVisitor, 86
- design pattern, 143
 - RealThis object composition, 145
 - static delegator, 143
 - template hook, 147
 - visitor, 103
- directory structure, 184
- dispatching in visitors, 106
- double dispatching, 106
- DSL, 81
- DSLs, 1
- dynamic dispatching, 106
- dynamic type, 106
- EBNF, 44, 231
- Eclipse, 7, 11

- editor, 44
- Eclipse Modelling Framework, 65, 208
- editor, 250
 - auto completion, 44
 - highlighting, 44
- EMF, 65, 208, 250
- encapsulation, 81
- energy management, 2
- enumeration, 31, 58
 - mapped to AST, 58
- error, 204
 - by incorrect usage, 189
 - internal, 189, 190
- error handling, 189
- error management, 169
- error message, 137
- errors, 189
- exception, 204
- exp?functionname, 151
- explicit hook point, 171
- expression problem, 106
- extension, 33
 - AST, 55
 - AST-access-conservative, 99
 - AST-conservative, 92, 99
 - AST-modification-conservative, 99, 100
 - conservative, 92, 97
 - CS-AST-compliance, 92
 - CS-conservative, 92, 98
 - Java type errors, 101
 - multiple grammars, 93
- Extreme Programming, 5
- fail quick, 169, 195
- feature diagram DSL, 2
- flight control, 2
- fragment token, 26
- free modification, 92
- FreeMarker, 149, 155
 - Boolean, 151
 - Date, 151
 - Hash, 151
 - Number, 151
 - Sequence, 151
 - String, 151
 - control directives, 152
 - control language, 149
 - data types, 151
 - drawbacks, 154
 - expressions, 149, 151
- FreeMarker template language, 149
- generation gap, 148
- generator, 17
 - API, 157
 - architecture, 18
 - backend, 19
 - central part, 19
 - engine, 159
 - frontend, 18
 - setup, 159
 - workflow, 19
- generator engine, 149, 155
- generator scripts, 17
- generics, 29
- git, 248
- GitHub, 248, 250
- github, 249
- GitLab, 248
- global scope, 129
- grammar, 23, 51
 - Cardinality, 222
 - Completeness, 223
 - MCBasicTypes1, 221
 - MCBasics, 213
 - MCExpressions, 225
 - MCHexNumbers, 219
 - StringLiterals, 215
 - UMLModifier, 224
 - UMLStereoType, 222
 - import, 87
 - 0xA4*** messages, 40
 - component, 23
 - context conditions, 39
 - derivates, 23
 - editor, 44
 - error messages, 40
 - syntax, 23
 - testing, 242
- grammar component, 88
- grammar concepts, 24
- grammar format, 23

- grammar options, 24
- Grammar.mc4, 49
- Grammar_WithConcepts.mc4, 49
- grammars, 1
- Graphviz, 202
- Groovy, 157, 198, 205, 206
- Groovy script, 19
- handcoded files, 185
- handcoded path, 184
- handcoding, 183
- handwritten class, 185
 - detection, 186
- handwritten code, 183
- handwritten extension, 70
- HexIntegerLiteral, 28
- hook
 - code, 174
 - string, 174
 - template, 174
 - template-string, 174
- hook method
 - in design pattern, 147
- hook point, 19, 171
 - binding, 171
 - decoration, 179
 - decorator, 171
 - explicit, 171, 177
 - implicit, 171
 - name, 171
 - naming convention, 177
 - parameters, 166
 - replacing, 173, 179
 - value, 171
- how to deal with
 - errors, 204
 - exceptions, 204
- human brain, 2
- identifiers, 121
- implements, 33
- implicit hook point, 171
- import, 83
- incremental compilation, 82
- infix, 35
 - associativity, 35
 - priority, 35
 - priority <180>, 35
- information, 190
- information message, 191
- inheritance, 90
- inheritance visitor, 113
- integration
 - handwritten and generated code, 147
- IntelliJ, 7, 13
 - editor, 44
- interface, 32
- interface nonterminal, 32, 52
- internal error, 190
- Java, 2, 121, 247
- Java classpath
 - configuring, 186
- JavaDSL, 251
- JavaLib, 251
- JDK, 7
- Jenkins, 249
- JUnit, 5, 139, 242
- kind, 122
 - model entity, 121
- language
 - extension, 33
- language aggregation, 81, 83
 - visitor, 110
- language components, 82
- language composition, 81, 82, 84
- language embedding, 82, 83
 - visitor, 110, 114
- language extension, 82, 83
- language inheritance, 82, 83, 90, 113
 - visitor, 110
- language library, 83
- language parser, 23
- language workbench, 1, 2
- left recursion, 30, 37
- left-hand side, 29
- lexer, 24, 238
- lexer productions, 24
- lexerjava, 38
- lexical production
 - action, 26

- conversion, 27
 - result type, 26
- lexical rules, 25
- library, 83
- log file, 192
- logback, 196
- logback configuration, 11
- logging, 11, 169
 - debug, 169, 193
 - enableFailQuick, 193
 - error, 169, 193
 - getErrorCount, 193
 - info, 169, 193
 - isFailQuickEnabled, 193
 - trace, 169, 193
 - warn, 169, 193
 - severity level, 169
- logging API, 193
- logging APIs, 195
- logging component, 193
- logs, 189
- M2E, 12
- main control, 19
- make, 186
- Management Project, 248
- Maven, 11, 247, 250
- maven, 186
- Maven 2 Eclipse, 12
- Maven Project, 248
- MCG, 44
- message form
 - debug, 192
 - error, 192
 - info, 192
 - trace, 192
 - warn, 192
- meta-tool, 1
- method body
 - template, 156
- mill, 66
 - composition, 66
 - handwritten extension, 72
- ML_COMMENT, 28
- model, 17
- model loader, 19
- model parser, 18
- model processor, 17
- model transformation, 17
- Modeling in the large, 1
- modelling language, 17
- modularity, 81
- MontiCore, 1, 2
 - adaptation, 75
 - command line, 7
 - configuration, 205
 - Continuous Integration, 247
 - design pattern, 143
 - Download, 8
 - Eclipse plugin, 7
 - error configuration, 189
 - features, 2
 - generator engine, 149
 - grammar, 79
 - grammar editor, 44
 - grammars, 213
 - IntelliJ plugin, 7
 - libraries, 213
 - Maven, 11
 - Nexus, 247
 - parameters, 10
 - Release Notes, 247
 - reports, 189
 - Repositories, 247
 - RTE, 79
 - run, 8
 - runtime environment, 79
 - Sonar, 247
 - Ticket system, 247
- Name, 28
- name, 121
- name definition, 121, 122
- name usage, 121, 122
- NEWLINE, 28
- Nexus, 247, 249
- node builder mill, 66
- nonterminal, 23, 32, 51
 - * mapped to AST, 51
 - + mapped to AST, 51
 - ? mapped to AST, 52
- external, 87, 88

- interface, 87
- abstract, 34, 53, 62
- cardinality, 37
- enumeration, 31
- extension, 33, 88, 91
- extension mapped to AST, 54
- interface, 32, 52, 62
- list, 63
- mapped to AST, 51
- optional, 63
- overriding, 87
- redefining, 91
- token mapped to AST, 52
- nonterminals, 25
- Notational Conventions, 3
- Num_Long, 28
- object composition, 145
- OID, 199
- open source, 248
- package structure, 184
- parameter
 - cl file.xml, 10
 - customLog file.xml, 10
 - d, 10
 - dev, 10
 - f, 10
 - force, 10
 - fp path-list, 10
 - g path-list, 10
 - grammars path-list, 10
 - h, 10
 - handcodedPath path-list, 10
 - hcp path-list, 10
 - help, 10
 - modelPath path-list, 10
 - mp path-list, 10
 - o path, 10
 - s file.groovy, 10
 - script file.groovy, 10
 - templatePath path-list, 10
- parameterized generator, 17
- parser, 23, 85
- parser generator, 75
- parserjava, 38
- POM, 247
- Pretty printing, 20
- pretty printing, 155
- PrimitiveType, 29
- production, 23, 29
 - (...), 29
 - enumeration, 31
 - lowerChar..upperChar, 29
 - naming n:NT, 30
 - no keywords Name&, 30
 - optional ?, 29
 - redefining, 91
 - rekursion NT, 29
 - repetition *, 29
 - repetition +, 29
- productions, 24
- Project, 248
- project management, 247
- project object model, 247
- re-generation, 186
- realThis object composition pattern, 114
- RealThis pattern, 145
- realThis pattern, 109
- recursion, 29
- reference
 - entity, 121
- reference types, 29
- ReferenceType, 29
- regular expression, 25
 - (...), 25
 - lowerChar..upperChar, 25
 - negation ~, 25
 - optional ?, 25
 - repetition *, 25
 - repetition +, 25
- regular expressions, 25
- report, 190
- reports, 9, 189, 192, 198
 - hookpoints, 200
 - identifiers, 199
 - Java classes, 200
 - templates, 200
 - variables, 200
- Repositories, 247
- reuse, 81

- right-hand side, 29
- RTE, 79
 - scope, 127
- runtime environment, 79
- scannerless, 240
- scannerless parsing, 238
- scope, 37, 122, 127
 - artifact, 129
 - compilation unit, 129
 - global, 129
- scope graph, 122
- scope tree, 122
- scopes, 23, 86
- Scrum, 5
- SE Project, 248
- semantic action, 44
- semantic predicate, 44, 241
- semantic predicates, 23
- singleton, 145
- SL_COMMENT, 28
- SLF4J, 195
- SMI, 121, 123
- Sonar, 249
- SSELab, 247–249
- static delegator
 - delegate object, 143
 - do method, 143
 - static host class, 143
 - static method, 143
- static delegator pattern, 66, 143, 184
- static type, 106
- String, 28
- string hook, 174
- sublanguage, 88
- sublanguages, 82
- suffix
 - Ext, 89
- super-grammar, 23
- SVN, 247, 248
- symbol, 37, 122
 - usage, 130
- symbol kind, 122
 - adaptation, 123
- symbol management infrastructure, 121
- symbol table, 121, 122, 134
 - instantiation, 131
 - quick navigation, 122
 - surrogate, 122
 - to collect information, 122
- symbols, 23, 86
- template
 - addToGlobalVar, 163
 - bindHookPoint, 163
 - changeGlobalVar, 163
 - debug, 163
 - defineGlobalVars, 163
 - defineGlobalVar, 163
 - defineHookPoint, 163
 - error, 163
 - existsHookPoint, 163
 - getGlobalVar, 163
 - includeArgs, 163
 - include, 163
 - info, 163
 - requiredGlobalVars, 163
 - requiredGlobalVar, 163
 - signature, 163
 - trace, 163
 - warn, 163
 - adaptation, 171
 - API, 162
 - decoration, 179
 - global variable, 169
 - hook point, 171
 - local variable, 169
 - name, 164
 - parameters, 166
 - qualified name, 164
 - replacing, 173, 179
 - signature, 156, 166
- template engine, 19, 155
- template hook, 174
- template hook pattern, 147
- template method
 - in design pattern, 147
- template path, 173
- template-string hook, 174
- templates, 17
- terminal, 31, 57
 - in square brackets, 57

- mapped to AST, 57
 - named, 57
 - optional, 63
 - semantically relevant, 31, 57, 63
- testing, 242
- text model, 4
- tokens, 24, 238
- tool provider, 18
- tool smith, 18
- TOP mechanism, 71, 185
- Trac, 249
- trace message, 191
- transformation
 - AST, 156
- traversal, 103
- unique error code, 137
- usage error, 190
- UTF-8, 44
- V-Model, 5
- version control, 184
- visibilities, 122
- visibility, 127
- visit, 103
- visitor
 - endVisit, 103
 - handle, 103
 - traverse, 103
 - visit, 103
 - composition, 109, 114
 - embedding, 111
 - external, 103
 - functional, 103
 - imperative, 103
 - language embedding, 110, 114
 - language inheritance, 110
 - subclassing, 111
- visitor interface, 103
- visitor pattern, 62, 103
- visitors, 86
- warning, 190, 191
- warnings, 189
- well-formedness, 133
- whitespaces, 239
- WildcardType, 29
- workflow, 19, 205
- WS, 26, 28
- www.monticore.de, 7
- XP, 5