SE | RWTH AACHEN UNIVERSITY

RWTH Aachen University
Software Engineering Group

MontiCore Language Workbench and Library Handbook

Edition 2021

Bernhard Rumpe,
Katrin Hölldobler,
Oliver Kautz

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

Aachener Informatik-Berichte, Band 48
Software Engineering

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

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 48

Bernhard Rumpe, Katrin Hölldobler, Oliver Kautz
RWTH Aachen University

MontiCore Language Workbench and Library Handbook

Edition 2021

Shaker Verlag
Düren 2021

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Printed in Germany.

ISBN 978-3-8440-8010-0
ISSN 1869-9170

Shaker Verlag GmbH • Am Langen Graben 15a • 52353 Düren
Telefon: 02421 / 99 0 11 - 0 • Telefax: 02421 / 99 0 11 - 9
Internet: www.shaker.de • E-Mail: info@shaker.de

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

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

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

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

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

We would like to thank all current and former members of our group as well as all students and apprentices who helped to develop MontiCore in its current shape. Namely, we would like to thank Kai Adam, Daoud Ali, Vassily Aliseyko, Professor Dr. Christian Berger, Vincent Bertram, Miriam Boß, Arvid Butting, Joel Charles, Manuela Dalibor, Anabel Derlam, Niklas Dienstknecht, Imke Drave, Robert Eikermann, Christoph Engels, Arkadii Gerasimov, Dr. Timo Greifenberg, Dr. Hans Grönniger, Dr. Tim Gülke, Dr. Arne Haber, Guido Hansen, Olga Haubrich, Malte Heithoff, Dr. Lars Hermerschmidt, Dr. Christoph Herrmann, Gabi Heuschen, Steffen Hillemacher, Nico Jansen, Hendrik Kausch, Christian Kirchhof, Carsten Kolassa, Dr. Anne-Therese Körtgen, Thomas Kurpick, Evgeny Kusmenko, Dr. Holger Krahn, Dr. Stefan Kriebel, Achim Lindt, Dr. Markus Look, Daniel Maibach, Professor Dr. Shahar Maoz, Matthias Markthaler, Dr. Dan Matheson, Dr. Judith Michael, Joshua Mingers, Dr. Klaus Müller, Dr. Pedram Mir Seyed Nazari, Antonio Navarro Pérez, Lukas Netz, Mathias Pfeiffer, Nina Pichler, Dr. Claas Pinkernell, Dr. Dimitri Plotnikov, Deni Raco, Professor Dr. Jan Ringert, Dr. Holger Rendel, Dr. Dirk Reiss, Dr. Daniel Retkowitz, Dr. Alexander Roth, Dr. Martin Schindler, David Schmalzing, Steffi Schrader, Dr. Frank Schroven, Dr. Christoph Schulze, Igor Shumeiko, Brian Sinkovec, Sebastian Stüber, Simon Varga, Dr. Steven Völkel, Louis Wachtmeister, Dr. Ingo Weisemöller, Dr. Michael von Wenckstern, and Professor Dr. Andreas Wortmann. The individual contributions to MontiCore and its derivatives resulted in numerous publications[1]. Special thanks go to Marita Breuer and Galina Volkova, who maintain and extend MontiCore, and in particular to Sylvia Gunder and Sonja Müßigbrodt, who ensure that all financial and project activities supporting our language workbench project MontiCore are running perfectly.

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

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

Aachen, 27.03.2021

Bernhard Rumpe, Katrin Hölldobler, Oliver Kautz

[1] www.se-rwth.de/publications/

Contents

1 Introduction to Tool Generation **1**
1.1 MontiCore Language Workbench . . . 1
1.2 Notational Conventions . . . 3
1.3 Textual Modeling . . . 4
1.4 Methodical Considerations: Agile Modeling . . . 5

2 Getting Started with MontiCore **7**
2.1 Prerequisites: Installing the Java Development Kit . . . 7
2.2 Install and Use the MontiCore Command Line Interface . . . 8
2.2.1 Installation . . . 9
2.2.2 Inspect the Example Grammar . . . 9
2.2.3 Run MontiCore . . . 12
2.2.4 Compile the Target . . . 22
2.2.5 Run the Tool . . . 28
2.3 Using MontiCore via Gradle From the Command Line . . . 29
2.4 Using MontiCore in Eclipse . . . 29
2.4.1 Setting up Eclipse . . . 30
2.4.2 Importing the Example . . . 30
2.4.3 Running MontiCore . . . 30
2.5 Using MontiCore in IntelliJ IDEA . . . 31
2.5.1 Setting up IntelliJ IDEA . . . 31
2.5.2 Importing the Example . . . 31
2.5.3 Running MontiCore . . . 32

3 Architecture of a Model Processor **33**
3.1 Structure of a Model Processor - External View . . . 33
3.2 Internal Architecture of a Generator - Component View . . . 34
3.3 Tool Workflow . . . 36

4 MontiCore Grammar for Language and AST Definitions **39**
4.1 Lexical Tokens for the Scanner . . . 40
4.1.1 Definition of Tokens using Regular Expressions . . . 41
4.1.2 Actions to Process a Token . . . 42
4.1.3 Predefined Tokens in Importable Grammars . . . 43
4.2 Productions in the Grammar . . . 46
4.2.1 Terminals . . . 48
4.2.2 Enumeration . . . 49
4.2.3 Nonterminals . . . 50
4.2.4 Interface Nonterminals: implements . . . 51

4.2.5 Extending Nonterminals: extends . . . 52
4.2.6 Abstract Nonterminals . . . 52
4.2.7 Starting Nonterminal . . . 54
4.2.8 Infix Operations and Priorities . . . 54
4.2.9 Restricting the Cardinality of a Nonterminal . . . 55
4.2.10 Symbols and Scopes . . . 56
4.2.11 Passing Code to the ANTLR Parser . . . 57
4.2.12 Annotations for Nonterminals and Grammars . . . 58
4.2.13 Predefined Nonterminals in Importable Grammars . . . 59
4.3 Additional Control Directives in the MCG Language . . . 59
4.3.1 Splitting Tokens . . . 60
4.3.2 Local Keywords: Avoid handling Keywords as Tokens . . . 61
4.4 Context Conditions for the MCG Language . . . 62
4.5 Semantic Predicates and Actions . . . 69
4.6 EBNF of the MCG Language . . . 70

5 Abstract Syntax Tree 77
5.1 Mapping Nonterminals to the AST . . . 78
5.2 Interface and Abstract Nonterminals . . . 79
5.3 Extending Nonterminals: astimplements, astextends . . . 79
5.4 Extending the Abstract Syntax Implementation . . . 81
5.5 Terminals in the AST . . . 83
5.6 Enumerations . . . 84
5.7 ASTNode: A Base Interface for AST Classes . . . 85
5.8 Generated ASTNode Subclasses . . . 86
5.9 Node Construction Using the Node Builder Mill . . . 91
5.10 Handwritten Extension of AST Classes and Node Builders . . . 96
5.10.1 Handwritten Extension of AST Classes: TOP-Mechanism . . . 96
5.10.2 Handwritten Extension of AST Builders and Mills . . . 97

6 Parser Generation and Use 101
6.1 Generating a Parser and a Lexer, as done in MontiCore . . . 101
6.2 Interface of the Generated Parser Classes . . . 104
6.3 Executing a Generated Parser . . . 106

7 Language Composition 109
7.1 Introduction to Language Composition . . . 109
7.2 Language Composition at a Glance . . . 113
7.3 Grammar Constructs for Language Composition . . . 116
7.3.1 Component Grammar . . . 117
7.3.2 External Nonterminals . . . 117
7.3.3 Importing and Extending Grammars . . . 119
7.4 Language Inheritance . . . 120
7.4.1 Redefining / Overriding Productions of Grammars . . . 121
7.4.2 Extending the Implementation Structure of a Nonterminal . . . 123
7.4.3 Extending Multiple Inherited Grammars . . . 124

7.5 Language Embedding . . . 125
7.6 Composing the Builder Infrastructure . . . 126
7.7 Composing Parsers . . . 127
7.8 Composition of Visitors and Context Conditions . . . 128
7.9 Conservative Extension . . . 129
7.9.1 Conservative Extension of the Concrete Syntax . . . 129
7.9.2 Access-Conservative Extension of the Abstract Syntax . . . 131
7.9.3 Modification-Conservative Extension of the Abstract Syntax . . . 132
7.9.4 AST Signatures Causing Java Type Errors . . . 133

8 Visitors for AST Traversal 135
8.1 Visitor Infrastructure for a Language . . . 136
8.1.1 Traverser Interface and Implementing Class . . . 136
8.1.2 Visitor2 Interface . . . 140
8.1.3 Handler Interface . . . 141
8.1.4 Inheritance Handler for Explicit Visit of Supertypes . . . 142
8.2 Visitors for Composed Languages . . . 144
8.2.1 Visitor Infrastructure for Language Inheritance and Extension . . . 145
8.2.2 Visitor for Language Inheritance with Overriding Nonterminal . . . 147
8.2.3 Visitors for Compositional Language Embedding . . . 149

9 Symbol Management Infrastructure 155
9.1 Introduction to Symbol Table Concepts . . . 157
9.1.1 Symbols . . . 157
9.1.2 Scopes . . . 158
9.2 Defining Symbols . . . 160
9.2.1 Runtime (RTE) Classes For Symbols . . . 162
9.2.2 Generated Classes For Symbols . . . 164
9.2.3 Defining Additional Symbol Attributes via `symbolrule` . . . 166
9.3 Defining Scopes . . . 167
9.3.1 Artifact Scope and Global Scope . . . 167
9.3.2 Runtime Environment Classes for Scopes . . . 168
9.3.3 Generated Classes For Scopes . . . 172
9.3.4 Defining Scope Attributes and Methods via `scoperule` . . . 175
9.4 Collaboration between AST, Symbol, and Scope . . . 176
9.5 Using Symbols . . . 177
9.6 Instantiating Symbol Tables . . . 179
9.6.1 Phase 1: Symbols and Scope Skeletons . . . 179
9.6.2 Phase 2+: Filling Symbols with Value . . . 182
9.7 Loading and Storing Symbol Tables . . . 182
9.7.1 Stored Symbol Tables . . . 183
9.7.2 RTE Classes For Symbol Table Persistence . . . 184
9.7.3 Generated Classes for Symbol Storage and Their Adaptation . . . 187
9.7.4 Loading Symbol Tables . . . 190
9.7.5 Storing Symbol Tables . . . 191
9.7.6 Realizing Custom Serialization Strategies . . . 192

9.8 Resolving Symbols in Scopes . . . 195
9.8.1 How to Use Symbol Resolution . . . 195
9.8.2 Concept Of Symbol Resolution . . . 196
9.8.3 Generated Implementation for Symbol Resolution . . . 199
9.8.4 Customizing Symbol Resolution . . . 201
9.9 Visitors Also Handle Symbol Tables . . . 203
9.10 Symbol Tables in Composed Languages . . . 204
9.10.1 Symbol Management Infrastructure for Language Inheritance . . . 205
9.10.2 Symbol Management Infrastructure for Language Aggregation . . . 206
9.10.3 Symbol Adapters . . . 206
9.10.4 Resolving for Adapted Symbols . . . 208

10 Context Conditions 211
10.1 Context Condition Infrastructure . . . 212
10.2 Implementation of Context Conditions . . . 214
10.3 Testing Context Conditions . . . 216
10.3.1 Testing a Context Condition on a Valid Model . . . 217
10.3.2 Testing a Context Condition on an Invalid Model . . . 218

11 Design Patterns Used and Invented for MontiCore 221
11.1 Static Delegator Design Pattern . . . 221
11.2 RealThis Object Composition Pattern . . . 223
11.3 Attribute and Association Access Pattern . . . 227
11.3.1 Attribute Access Pattern . . . 227
11.3.2 Association Access Pattern . . . 230
11.3.3 The Extended Builder Pattern . . . 231
11.4 Template Hook Pattern . . . 232
11.5 Mill Pattern to Assist Composition . . . 233
11.6 Multiple Interface Composition Pattern . . . 236

12 FreeMarker 237
12.1 The FreeMarker Template Languages . . . 237
12.2 Expressions in FreeMarker . . . 238
12.3 Control Directives in FreeMarker . . . 240
12.4 FreeMarker Add Ons . . . 242

13 Generator Engine using Flexible Templates 245
13.1 Methodical Considerations . . . 245
13.2 Generator API . . . 247
13.3 Configuring the Generation Process . . . 250
13.4 MontiCore APIs for Templates . . . 252
13.4.1 Shortcuts: Aliases in Templates . . . 252
13.4.2 The TemplateController . . . 254
13.4.3 Logging within a Template . . . 259
13.4.4 Variables in the Templates with GlobalExtensionManagement . . . 260
13.5 Hook Points for Adaptation . . . 262
13.5.1 The Concept of Hook Points . . . 262

13.5.2 Forms of Hook Points . . . 266
13.5.3 Defining Explicit Hook Points in Templates . . . 268
13.5.4 Binding Hook Points . . . 270
13.5.5 Replacing and Decorating Hook Points . . . 272
13.5.6 HookPoint Replacement and Decoration Strategy . . . 272
13.5.7 A HookPoint Replacement and Decoration Example . . . 274

14 Integrating Handwritten Code 279
14.1 Integration of Handwritten Code . . . 279
14.2 Adaptation of Generated Code by Subclassing . . . 280
14.3 Adaptation of Generated Code using the TOP Mechanism . . . 281

15 Error Handling, Logging and Reporting 285
15.1 Where to find Concrete Help for an Error, Warning, or other Message . . . 285
15.2 Errors, Warnings and Log Messages . . . 286
15.2.1 Errors . . . 286
15.2.2 Warnings and Information . . . 287
15.2.3 Form of Errors, Warnings and Log Messages . . . 288
15.3 The Error and Logging Component . . . 289
15.4 Logging Configurations in MontiCore . . . 292
15.4.1 Selecting one of the given Configurations . . . 292
15.4.2 Using a Custom logback Configuration . . . 293
15.4.3 Initializing the Log within Java . . . 293
15.4.4 Providing a Custom Log Implementation . . . 294
15.5 Reports . . . 294
15.5.1 Where to Find Reports . . . 294
15.5.2 How to Configure Reporting . . . 294
15.5.3 Identifiers contained in the Reports . . . 295
15.5.4 List of the Reports . . . 297
15.6 For Developers: How to Deal with Errors and Warnings . . . 300

16 MontiCore Use and Configuration from CLI or Gradle 303
16.1 MontiCore from Commandline . . . 304
16.1.1 How to Call the CLI . . . 304
16.1.2 Parameters of the CLI . . . 305
16.2 Embedding the CLI in a Makefile Build Process . . . 306
16.3 MontiCore Used via Gradle Plugin . . . 310
16.3.1 Defining a MontiCore Task . . . 311
16.3.2 Compilation and Packaging . . . 314
16.3.3 Defining external Dependencies . . . 314
16.3.4 Example Build Script . . . 315
16.4 MontiCore in Maven . . . 317
16.5 MontiCore Workflow Configuration with Groovy . . . 318
16.5.1 The Standard Groovy Generation Script . . . 319
16.5.2 MontiCore Base Class for Groovy Scripts . . . 321
16.5.3 Methods Available within Groovy Scripts . . . 323

16.5.4 Variables Available within Groovy Scripts . . . 324
16.5.5 Available preimported Classes within Groovy Scripts . . . 325

17 Example MontiCore Grammars 327
17.1 Component Grammar MCBasics.mc4 . . . 328
17.2 Component Grammar StringLiterals.mc4 . . . 329
17.3 Component Grammars for Numbers . . . 331
17.3.1 Component Grammar MCNumbers.mc4 . . . 331
17.3.2 Component Grammar MCHexNumbers.mc4 . . . 333
17.4 Component Grammars for UML Languages . . . 334
17.4.1 Component Grammar UMLStereotype.mc4 . . . 335
17.4.2 Component Grammar Cardinality.mc4 . . . 335
17.4.3 Component Grammar UMLModifier.mc4 . . . 336
17.4.4 Component Grammar Completeness.mc4 . . . 337
17.4.5 Component Grammar MCCommon.mc4 . . . 338

18 Expression and Type Language Components 339
18.1 Literals as Basis for Expressions . . . 340
18.1.1 MCLiteralsBasis . . . 341
18.1.2 MCCommonLiterals . . . 341
18.1.3 MCJavaLiterals . . . 345
18.2 Expressions in various Variants . . . 346
18.2.1 ExpressionsBasis . . . 347
18.2.2 CommonExpressions . . . 348
18.2.3 BitExpressions . . . 349
18.2.4 AssignmentExpressions . . . 349
18.2.5 JavaClassExpressions . . . 350
18.3 Symbols . . . 352
18.3.1 BasicSymbols . . . 352
18.3.2 OOSymbols . . . 354
18.4 Types: From Simple To Generic . . . 355
18.4.1 MCBasicTypes . . . 356
18.4.2 MCCollectionTypes . . . 357
18.4.3 MCSimpleGenericTypes . . . 358
18.4.4 MCFullGenericTypes . . . 358
18.4.5 MCArrayTypes . . . 359
18.5 Using Base Grammars . . . 359
18.6 Type Checking in MontiCore Languages . . . 361
18.6.1 Types in a Symbol Table: SymTypes . . . 362
18.6.2 Using Type Checks: the Type Check API . . . 363
18.6.3 How the Type Check is Configuered . . . 365

19 Statement Language Components 371
19.1 MCStatementsBasis . . . 372
19.2 MCVarDeclarationStatements . . . 373
19.3 MCArrayStatements . . . 374

19.4 MCCommonStatements . . . 374
19.5 MCReturnStatements . . . 376
19.6 MCAssertStatements . . . 376
19.7 MCSynchronizedStatements . . . 377
19.8 MCExceptionStatements . . . 377
19.9 MCLowLevelStatements . . . 378
19.10 MCFullJavaStatements . . . 379

20 The JavaLight Language 381
20.1 Sublanguage Hierarchy of JavaLight . . . 381
20.2 Nonterminals of JavaLight . . . 382
20.2.1 Methods, Constructors, and Attributes . . . 383
20.2.2 Java Annotations . . . 386
20.2.3 Java-Specific Array Initialization . . . 387

21 Some Demonstrating Example Languages 389
21.1 A Simple Automata Language . . . 389
21.2 Hierarchical Automata . . . 396
21.3 A Language for Automata with Invariants . . . 398
21.4 Scannerless Parsing to Handle Complex Tokens . . . 400
21.4.1 Parsing with Whitespaces . . . 400
21.4.2 Temporarily Parsing with Whitespaces . . . 402
21.4.3 Preventing Whitespaces between Tokens . . . 403
21.5 Tip: Testing Grammars and their Models . . . 404
21.6 ColoredGraph Language . . . 405
21.7 Questionnaire Language . . . 408

22 Developer's View on MontiCore 413
22.1 MontiCore's GitHub Repository . . . 414
22.2 For External Developers: How to Contribute . . . 416
22.3 MontiCore's Gradle Projects . . . 416
22.4 Further Source Code Locations . . . 417

23 Further Reading and Related Work 419

List of Figures 429

Listings 433

References 441

Index 463