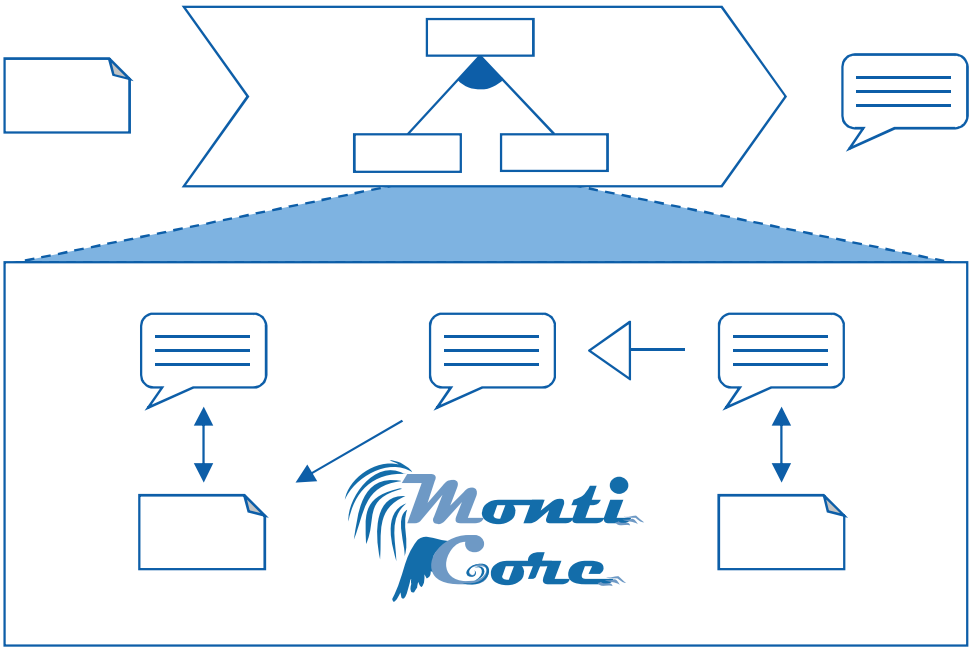
SE | RWTH AACHEN UNIVERSITY

Arvid Butting

Systematic Composition of Language Components in MontiCore



Aachener Informatik-Berichte,
Software Engineering

Band 53

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

Systematic Composition of Language Components in MontiCore

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

vorgelegt von

Arvid Butting, M.Sc. RWTH
aus Willich

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

Tag der mündlichen Prüfung: 12.07.2022

Aachener Informatik-Berichte, Software Engineering

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

Band 53

Arvid Butting
RWTH Aachen University

Systematic Composition of Language Components in MontiCore

Shaker Verlag
Düren 2023

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Zugl.: D 82 (Diss. RWTH Aachen University, 2022)

Printed in Germany.

ISBN 978-3-8440-8936-3
ISSN 1869-9170

Shaker Verlag GmbH • Am Langen Graben 15a • 52353 Düren
Phone: 0049/2421/99011-0 • Telefax: 0049/2421/99011-9
Internet: www.shaker.de • e-mail: info@shaker.de

Eidesstattliche Erklärung

I, Arvid Butting

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

Hiermit erkläre ich an Eides statt

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

Abstract

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

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

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

Kurzfassung

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

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

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

Danksagung

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

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

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

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

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

Aachen, Juli 2022
Arvid Butting

Contents

1 Introduction **1**
- 1.1 Research Question & Objectives . . . 3
- 1.2 Main Results and Structure of Thesis . . . 5

2 Foundations **9**
- 2.1 Software Language Engineering . . . 9
  - 2.1.1 Software Languages . . . 9
- 2.2 The MontiCore Language Workbench . . . 13
  - 2.2.1 MontiCore Grammars . . . 15
  - 2.2.2 Abstract Syntax Tree Data Structure . . . 21
  - 2.2.3 Traversing the Abstract Syntax . . . 23
  - 2.2.4 Context Conditions . . . 25
  - 2.2.5 Identifying Artifacts in the File System . . . 27
  - 2.2.6 Instantiating the Language Infrastructure . . . 27
  - 2.2.7 Integration of Handwritten Code . . . 27
  - 2.2.8 Language Composition . . . 29
- 2.3 Software Product Line Engineering . . . 32
  - 2.3.1 Variability in Software and Software Product Lines . . . 33
  - 2.3.2 Software Reuse . . . 35
  - 2.3.3 Feature Diagrams . . . 36

3 Method for the Systematic Composition of Language Components in MontiCore **41**

4 Generating Kind-Typed Symbol Table Infrastructures **45**
- 4.1 Concept of Kind-Typed Symbol Tables . . . 48
  - 4.1.1 Relationships between Symbols, Scopes, and AST Nodes . . . 49
  - 4.1.2 Defining Names via Symbols . . . 50
  - 4.1.3 Capturing Name Visibility with Scopes . . . 51
  - 4.1.4 Providing Access to a Model's Symbol Table with Artifact Scopes 54
  - 4.1.5 Bridging the Gap Between Models with Global Scopes . . . 55
  - 4.1.6 Using Model Elements through Names . . . 56
  - 4.1.7 Type Definitions and Type Expressions . . . 58
  - 4.1.8 Symbol Resolution . . . 62

4.1.9 Symbol Table Traversal . . . 69
4.1.10 Symbol Table Instantiation . . . 70
4.2 Annotating Grammars with Symbol Table Information . . . 73
4.2.1 Indicate that a Nonterminal Defines a Symbol Kind . . . 73
4.2.2 Indicate that a Nonterminal Spans a Scope . . . 75
4.2.3 Indicate that a Nonterminal Uses the Name of a Symbol . . . 76
4.2.4 Providing Symbol Kind Attributes . . . 77
4.2.5 Providing Scope Attributes . . . 79
4.3 Implementation of the Typed Symbol Table Infrastructure . . . 80
4.3.1 Implementation of Language Mills in MontiCore . . . 81
4.3.2 Implementation of Scopes in MontiCore . . . 82
4.3.3 Implementation of Artifact Scopes in MontiCore . . . 90
4.3.4 Implementation of Global Scopes in MontiCore . . . 92
4.3.5 Implementation of Symbol Resolvers in MontiCore . . . 94
4.3.6 Customizing Symbol Resolution . . . 95
4.3.7 Realization of Symbols in `Symbol` Classes . . . 96
4.3.8 Instantiating Symbol Tables with Scopes Genitors . . . 97
4.3.9 Instantiating Symbol Tables of Composed Languages with Scopes Genitor Delegators . . . 99
4.4 Discussion . . . 99
4.5 Related Work . . . 101

5 Infrastructure for Loading and Storing Symbol Tables 105
5.1 Serialization in General . . . 107
5.1.1 Serialization and Deserialization . . . 107
5.1.2 Serialization Strategies . . . 108
5.1.3 Serialization with Intermediate Structure . . . 111
5.2 Concept for Symbol Table Persistence . . . 112
5.2.1 Overview of Symbol Table Persistence . . . 112
5.2.2 Organization of Persisted Files . . . 114
5.2.3 Concept for Symbol Table Serialization and Deserialization . . . 116
5.3 JSON Infrastructure . . . 123
5.3.1 JSON Abstract Syntax Model . . . 124
5.3.2 Serialization Infrastructure . . . 128
5.3.3 Deserialization Infrastructure . . . 130
5.4 Realization of Loading and Storing of Symbol Tables in MontiCore . . . 132
5.4.1 Commonalities of Symbol DeSers in the `ISymbolDeSer` Interface 133
5.4.2 Commonalities of Scope DeSers in the `IDeSer` Interface . . . 134
5.4.3 The `JsonDeSers` Class . . . 137
5.4.4 `Symbols2Json` Classes for Traversing Symbol Tables . . . 137
5.4.5 `SymbolDeSer` Classes with Serialization Strategies for Symbols . 140

5.4.6 `ScopeDeSer` Classes with Serialization Strategies for Scopes . . . 141
5.4.7 Loading and Storing Symbol Tables via the Global Scope 143
5.4.8 Integrating Loading of Symbol Tables into Symbol Resolution . . . 145
5.4.9 Supporting Storing of Symbol Tables for Model Processing 146
5.5 Customizing the Persistence of Symbol Tables in MontiCore 147
5.5.1 Providing a Serialization Strategy for a Symbol Attribute 147
5.5.2 Omitting Serialization of Symbols of a Certain Kind 148
5.5.3 Realizing Serialization of an Additional Scope Attribute 149
5.5.4 Load ASTs together with Symbol Tables 151
5.5.5 Load Symbol Tables of a Single Language Only 152
5.5.6 Load Symbols as Instances of their Subkinds 152
5.5.7 Load Symbols as Instances of their Super Kinds 153
5.6 Discussion 153
5.7 Related Work 154

6 Using Typed Symbol Tables for Language Composition 157
6.1 Language Inheritance in the Typed Symbol Table Infrastructure 157
6.1.1 Language Inheritance of Scopes 157
6.1.2 Language Inheritance of Symbol Table Creation 159
6.1.3 Language Inheritance of Symbol Table Persistence 161
6.1.4 Reconfiguration via Mills 162
6.2 Adapting between Symbol Kinds 163
6.2.1 Concept for Symbol Adapters 164
6.2.2 Finding Symbol Adapters during Symbol Resolution 165
6.2.3 Combination of Symbol Adapters and Symbol Persistence 167
6.3 Importing Symbols from Java with Class2MC 170
6.4 Aggregation of Languages 173
6.4.1 Aggregation through Shared Grammar 173
6.4.2 Aggregation through Unifying Grammar 174
6.4.3 Aggregation through Resolvers 174
6.4.4 Aggregation through Symbol Files 175
6.5 Discussion 176
6.6 Related Work 176

7 Language Components 179
7.1 Language Component Models 181
7.2 MontiCore Language Component Diagrams 184
7.3 Concept for Identifying Artifacts of Language Components 186
7.3.1 Address Artifacts of a Language Component 186
7.3.2 Artifact Analysis 187
7.3.3 Building Self-Contained Language Component Archives 188

7.4 Realization of Language Components . . . 190
7.4.1 The MLC Language . . . 190
7.4.2 Tool for Processing MLC Models . . . 195
7.5 Discussion . . . 197
7.6 Related Work . . . 199

8 The MontiCore Feature Diagram Language Family 201
8.1 The Feature Diagram Language . . . 203
8.2 The Feature Configuration Languages . . . 207
8.3 The Feature Diagram Analysis Tool . . . 208
8.4 Composing Feature Models with Domain Models . . . 209
8.4.1 Internal Feature Realizations . . . 210
8.4.2 Referring to Feature Realizations . . . 211
8.4.3 Mapping to Feature Realizations . . . 212
8.5 Discussion . . . 213
8.6 Related Work . . . 214

9 Engineering Feature-Oriented Language Product Lines with MontiCore 215
9.1 Concept of a Feature-Oriented Language Product Line . . . 217
9.1.1 Engineering a Language Product Line . . . 217
9.1.2 Roles Involved in Language Product Lines . . . 219
9.1.3 Describing the Composition of Language Components . . . 221
9.1.4 Language Variant Derivation . . . 224
9.2 Realizing Language Product Lines in MontiCore . . . 226
9.2.1 The Language Product Line Language . . . 227
9.2.2 The Composition Infrastructure . . . 229
9.3 Discussion . . . 231
9.4 Related Work . . . 237

10 Application-Based Evaluation 241
10.1 Application of the STI . . . 242
10.2 Performance of Json Infrastructure . . . 243
10.3 Application of Loading and Storing Symbol Tables . . . 244
10.4 Application of Language Composition via Symbol Tables . . . 246
10.5 Application of MontiCore Language Components . . . 248
10.6 Application of the Feature Diagram Language Family . . . 249
10.7 Evaluation of the LCPL . . . 251

11 Conclusion 253
11.1 Summary . . . 253
11.2 Potential for Future Work . . . 254

Bibliography **257**

List of Figures **275**