Generator-Composition for Aspect-Oriented Domain-Specific Languages

Dissertation

Dipl.-Inform. Reiner Jung

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2016

Kiel Computer Science Series (KCSS) 2016/4 v1.0 dated 2016-07-01

ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via `https://www.informatik.uni-kiel.de/kcss`

Published by the Department of Computer Science, Kiel University

Software Engineering Group

Please cite as:

▷ Reiner Jung *Generator-Composition for Aspect-Oriented Domain-Specific Languages.* Number 2016/4 in Kiel Computer Science Series. Department of Computer Science, 2016. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Jung:2016,
  author    = {Reiner Jung},
  title     = {Generator-Composition for Aspect-Oriented Domain-Specific Languages},
  publisher = {Department of Computer Science, Kiel University},
  year      = {2016},
  number    = {2016/4},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering, Kiel University}
}
```

© 2016 by Reiner Jung

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter: Prof. Dr. Wilhelm Hasselbring
Kiel University

2. Gutachter: Prof. Dr. Ralf H. Reussner
Karlsruhe Institute for Technology

Datum der mündlichen Prüfung: 1. Juli 2016

Abstract

Software systems are complex, as they must cover a diverse set of requirements describing functionality and the environment. Software engineering addresses this complexity with Model-Driven Engineering (MDE). MDE utilizes different models and metamodels to specify views and aspects of a software system. Subsequently, these models must be transformed into code and other artifacts, which is performed by generators.

Information systems and embedded systems are often used over decades. Over time, they must be modified and extended to fulfill new and changed requirements. These alterations can be triggered by the modeling domain and by technology changes in both the platform and programming languages. In MDE these alterations result in changes of syntax and semantics of metamodels, and subsequently of generator implementations.

In MDE, generators can become complex software applications. Their complexity depends on the semantics of source and target metamodels, and the number of involved metamodels. Changes to metamodels and their semantics require generator modifications and can cause architecture and code degradation. This can result in errors in the generator, which have a negative effect on development costs and time. Furthermore, these errors can reduce quality and increase costs in projects utilizing the generator.

Therefore, we propose the generator construction and evolution approach GECO, which supports decoupling of generator components and their modularization. GECO comprises three contributions: *(a)* a method for metamodel partitioning into views, aspects, and base models together with partitioning along semantic boundaries, *(b)* a generator composition approach utilizing megamodel patterns for generator fragments, which are generators depending on only one source and one target metamodel, *(c)* an approach to modularize fragments along metamodel semantics and fragment functionality. All three contributions together support modularization and evolvability of generators.

Preface

by Prof. Dr. Wilhelm Hasselbring

Model-driven software engineering aims at leveraging higher abstractions into software development via two essential approaches:

▻ Domain specific languages

▻ Model transformation and code generation

Despite great success in some application domains, model-driven software engineering often faces the challenge of maintaining model transformation code over a long life time. Particularly for transformation code with a long life time, it would be beneficial to have a well-structured, modular software architecture according to the basic software engineering principles. To address the resulting challenges, Reiner Jung invents the so-called GECO approach to improve the maintainability and potential reuse of model-transformation and code-generation code. GECO introduces so-called generator fragments to be combined via aspect-oriented modeling and programming techniques.

Besides the conceptual work, this work contains a significant experimental part and an empirical evaluation, based on the open-source GECO implementation.

This thesis is a good read and I recommend it to anyone interested in software modularization for improved maintainability and potential software reuse.

Wilhelm Hasselbring
Kiel, July 2016

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Approach Overview and Contributions 3
1.3 Document Structure 5

I Foundations **9**

2 Type Systems **11**
2.1 Typing Overview 12
2.2 Simple Types 17
2.3 Recursive Types 23
2.4 Sub-Typing and Sub-Classing 25
2.5 Type Inference 28
2.6 Summary 28

3 Modeling **29**
3.1 Models and Metamodels 30
3.2 Model Transformation 39
3.3 Domain-Specific Languages 42
3.4 Model Traceability 50
3.5 Model Weaving 53
3.6 Megamodels 57
3.7 Transformation Languages 62

4 Graph and Hypergraph-Based Metrics **67**
4.1 Hypergraphs and Graphs 68
4.2 Hypergraph Size 69
4.3 Hypergraph Complexity 69

4.4 Hypergraph Coupling 70
4.5 Graph Cohesion 70

II Generator Composition for Aspect-Oriented DSLs 73

5 Contribution and Research Design 75
5.1 Research Scope 75
5.2 Research Questions 76
5.3 Research Plan and Summary of Results 76

6 Syntax and Semantics of Metamodels 81
6.1 Syntactical Properties of Metamodels 82
6.2 Use Cases for Metamodels 83
6.3 Contextual Metamodel Patterns 88
6.4 Semantics of References 92
6.5 Typing in Metamodels 93
6.6 Expressions in Metamodels 97
6.7 Separating Concerns and Views 100

7 Composition of Generators 109
7.1 Basic Generator Megamodel Patterns 109
7.2 Generator Fragment Combination 119
7.3 Computing Target Model Join Points 122
7.4 Achieving Model Traceability 127

8 Generator Fragment Design 129
8.1 Essential Generator Modules 130
8.2 Type System Mapping 136
8.3 Mapping of Expression Semantics 145
8.4 Model Traceability 151

III Evaluation 155

9 Experiment Design 157

9.1 Assessment of Approach Qualities 157
9.2 Quantitative Evaluation 159
9.3 Expert Interviews 170
9.4 Case Studies 174

10 Prototype Implementations **179**
10.1 Generator Composition API 179
10.2 Fragment Composition Tooling 190
10.3 Realizing Typing in Xtext 208
10.4 Instrumentation Aspect Language 218

11 Experimental Evaluation **239**
11.1 Information System Case Study 239
11.2 Control System Case Study 248
11.3 Semi-Structured Expert Interviews 264
11.4 Evaluation Summary 269

12 Related Work **273**
12.1 Aspect-Oriented and View-Based Modeling 273
12.2 Aspect-Oriented Code Generation 276
12.3 Generator Construction Approaches 281
12.4 Reuse of Transformations 284
12.5 Modularization of Transformations 287

IV Conclusion and Future Work **293**

13 Conclusion **295**
13.1 Contributions 295
13.2 Experimental Findings 297
13.3 Prototypical Application of GECO 299

14 Future Work **301**
14.1 Evaluation of Metamodel Partitioning 301
14.2 Generator Modernization with GECO 302
14.3 Technical Aspects of Megamodel Patterns 302

V Appendix 305

A Interview Development and Evaluation 307
A.1 Interview Guide 307
A.2 Analysis of the Interviews 310

B Tooling 319
B.1 Fragment Framework 319
B.2 Fragment Composition Tooling 319
B.3 Entropy Analysis of Models and Code 320

C Generator Evolution 323
C.1 Identified Revisions of MENGES DSL 323
C.2 MENGES and CoCoME Replication Package 328

Bibliography 333

List of Figures

2.1 Minimal boolean typed expression language 16
2.3 Derivation tree for an expression of the minimal boolean language 17
2.4 Typing rules for structured records 20
2.5 Typing rules for variant types 21
2.6 Typing rules for reference types 22
2.7 Typing rules for arrays 23
2.8 Iso-recursive types 24

3.1 Example of a partial metamodel for a supermarket software system 31
3.2 The functions of the graph morphism (f_E, f_N), depicted together with the edge and node sets E_G, E_H and N_G, N_H, and their source and target functions (s_G, t_G, s_H, t_H), of the two graphs G and H 34
3.3 Example of an instance model for the exemplary metamodel from Figure 3.1 37
3.4 Comparison of compiler phases and DSL code processing 48
3.5 The basic AMW weaving metamodel 55
3.6 Generic Composer depicted as a megamodel of models and transformations 57
3.7 Generic transformation megamodel 60
3.8 Different simplified transformation megamodel notations . . . 61
3.9 Weaving transformation megamodel 62

6.1 Minimal metamodel of a type system 94
6.2 An illustrative example of a metamodel for expressions 98
6.3 Example metamodel comprising the four different roles of metamodels 101

7.1 Matrix of eight basic candidate patterns involving three metamodels . . . 111
7.2 Matrix of basic candidate patterns involving four metamodels . 113
7.3 Five megamodel patterns for base and aspect metamodels with their respective transformations and target metamodels 117
7.4 Illustration of generator fragment composition for pattern P2 and P3 . . . 120
7.5 Illustration of generator fragment composition for P3 integrating a legacy base model generator T_{BM}, including a helper transformation to T_{Ref} to weave in references . . . 122
7.6 Illustration of the generation of target model references based on a component and a monitoring declaration . . . 124

8.1 Functional decomposition of a transformation . . . 131
8.2 An example automaton transformed into C code . . . 135
8.3 Example of a variant type and its equivalent utilizing the composite pattern . . . 140

9.1 Generator megamodel excerpt including the main transformations of the generator . . . 176

10.1 Subtyping rules . . . 195
10.2 Class property lookup rules . . . 195
10.3 Type derivation of properties . . . 203
10.4 Typing rules for logical and compare expressions . . . 205
10.5 Typing rules expressions . . . 206
10.6 Graphical representation of the GECO composition language generator . . . 208
10.7 Taxonomy of types in a type system . . . 209
10.8 IAL mapping metamodel excerpt depicting the central mapping concepts . . . 220
10.9 Semantic rules for the pointcut syntax rule . . . 226
10.10 Semantic rules for location query related syntax rules . . . 227
10.11 Typing rules for node paths . . . 229
10.12 Typing rules for base model object property constraints 231

10.13 Typing rules for constraint elements 232
10.14 Typing rule for the parameter query 233

11.1 Initial version of the megamodel for the CoCoME generator . . . 243
11.2 Architecture of the $T_{Behavior}$ generator 248
11.3 Development of module count and lines of code for G_{old} and G_{new} in the 14 evaluation steps 258
11.4 Development of complexity and coupling for G_{old} and G_{new} in the 14 evaluation steps 258
11.5 Development of complexity coupling ratio for G_{old} and G_{new} . . 259
11.6 Node and edge count Δ for both generators 260
11.7 Complexity and coupling Δ for both generators 261

Listings

2.1 Function subtyping example . . . 26
8.1 Two base type mapping functions used in the Instrumentation Record Language (IRL) [JHS13] implemented in Xtend . 137
8.2 Construction of Java literals in the IRL generator . . . 146
8.3 Example increment method used to ensure correct type overflow semantics for byte . . . 150
8.4 Minimal interface declaration for a trace model handler expressed in Xtend . . . 153
10.1 Generic generator fragment interface for model-to-model transformations . . . 184
10.2 Example of a class declaration with multiple target models, derived from the MENGES case study . . . 184
10.3 Generic weaver interfaces for model weaving, supporting an aspect model or alternatively two models for pointcut and advice . . . 185
10.4 Interface of the abstract class for the integration of ATL transformations in GECO . . . 186
10.5 Minimal interface declaration for a trace model provider expressed in Xtend . . . 188
10.6 Grammar infrastructure . . . 196
10.7 Metamodel declaration . . . 197
10.8 Metamodel declaration . . . 198
10.9 Weaver and Generator Fragment . . . 199
10.10 Weaver Fragment . . . 199
10.11 Aspect Model . . . 200
10.12 Generator Fragment . . . 201
10.13 Selector notation for metamodels . . . 202
10.14 Comparison and logical operators . . . 204
10.15 Basic selector constraint rules . . . 206

10.16 Declaration of trace models 207
10.17 Early realization of base types from the MENGES grammar (called primitive types) 211
10.18 Base types of the IRL 212
10.19 Excerpt of the IRL runtime module class depicting a method used to register the TypeGlobalScopeProvider 213
10.20 Xtext grammar rules for type representation 214
10.21 Xtext grammar rules for properties and functions 215
10.22 Type resolution realized with Xtend 217
10.23 Start rule and facility rules for event type and application model import of the IAL 221
10.24 Rules used to declare and configure aspects 222
10.25 Syntactic rules for the specification of advices in the Instrumentation Aspect Language (IAL) 223
10.26 Grammar rules for value expressions 224
10.27 Syntactic rule for a pointcut declaration 225
10.28 Location query syntax 226
10.29 Example composite location query 227
10.30 Grammar rules for path navigation 228
10.31 Syntactic rules for base model object property constraints . 230
10.32 Syntactic rules expressing access to base model attributes and nodes 230
10.33 Operation query syntax 233
10.34 The literals of the IAL 233
10.35 Excerpt of the generator template containing the standard declarations of an instrumentation aspect class 236
10.36 Excerpt of the generator template realizing a monitoring method 237

List of Acronyms

API Application Programming Interface

AMW ATLAS Model Weaver

ANTLR ANother Tool for Language Recognition

ASG Abstract Syntax Graph

AST Abstract Syntax Tree

ATL ATLAS Transformation Language

AODSL Aspect-Oriented Domain Specific Language

AOM Aspect-Oriented Modeling

AOP Aspect-Oriented Programming

CoCoME Common Component Modeling Example

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

CSS Cascading Style Sheets

CST Concrete Syntax Tree

CSV Comma Separated Values

DFG German Research Foundation

DOM Document Object Model

DSL Domain-Specific Language

DTL Data Type Language

EBNF Extended Backus–Naur Form

EJB Enterprise Java Bean

EMF Eclipse Modeling Framework

EMOF Essential Meta Object Facility

FBL Function Block Language

FDAF Formal Design Analysis Framework

GeKo Generic Composer

GPL General Purpose Language

GQM Goal Question Metric

HOT Higher-Order Transformation

HTML Hypertext Markup Language

IAL Instrumentation Aspect Language

IRL Instrumentation Record Language

ISO International Organization of Standardization

JPA Java Persistence API

JPM Join Point Metamodel

JSF Java Server Faces

JVM Java Virtual Machine

KLighD The Kiel Integrated Environment for Layout (KIELER) Lightweight Diagrams

KIELER The Kiel Integrated Environment for Layout

KobrA Komponentenbasierte Anwendungsentwicklung (engl. component-based application development)

LL Formal grammar which is parsed from left to right using the left most derivation

LR Formal grammar which is parsed from left to right using the right most derivation

LOC Lines of Code

LQN Layered Queueing Networks

MDA Model-Driven Architecture

MDE Model-Driven Engineering

MDSD Model-Driven Software Development

MENGES Model-Based Design Methods for a new Generation of electronic Railway Control Centers

MOF Meta Object Facility

MVM Multi-View Modeling

OCL Object Constraint Language

OMG Object Management Group

OOP Object-Oriented Programming

OSM Orthographic Software Modeling

PCM Palladio Component Model

PHP PHP Webpage Scripting Language

PLC Programmable Logic Controller

QVT Query/View/Transformation

RAM Reusable Aspect Models

SAM Source Aspect Metamodel

SBM Source Base Metamodel

SDF Syntax Definition Formalism

SIB Service Independent Building Block

SLG Service Logic Graph

SMM Structured Metrics Metamodel

ST Structured Text

SUM Single Underlying Model

TAM Target Aspect Metamodel

TBM Target Base Metamodel

TRM Traceability Metamodel

UML Unified Modeling Language

URI Uniform Resource Identifier

VCS Version Control System

XMDD Extreme Model-Driven Development

XMI Extensible Markup Language (XML) Metadata Interchange

XML Extensible Markup Language

XPath XML Path Language

XSD XML Schema Definition

XSLT Extensible Stylesheet Language Transformations

Chapter 1

Introduction

Modeling and model-driven approaches, methods, and frameworks play an important role in software development and evolution. With the introduction of modeling, the interest emerged to use these models not only as a way to discuss and plan the design of software, but to create programming code directly from these models. In Model-Driven Architecture (MDA) [OMG14], this is realized through multiple transformations from a platform independent model over a platform specific model to code artifacts. In other approaches, transformations are used in more complex chains to generate code (cf. Chapter 12). However, in all these approaches transformations and code generation play a central role.

In this thesis, we introduce our approach for generator construction and evolution. We motivate this approach in Section 1.1, summarize the approach and contributions in Section 1.2, and outline the document structure in Section 1.3.

1.1 Motivation

Modern software systems are complex and have the tendency to become more complex over time due to the introduction of new features, alterations of functionality, and changes in technology. Furthermore, changes to the requirements of a software systems become more and more common, resulting in more frequent modifications.

A solution to the issue of software complexity is modeling, which allows to abstract from technical details and focus on specific views and aspects [SV06]. Modeling allows to separate the different concerns of a software

system and addresses them separately. Furthermore, it allows to assess different qualities of the software before it is implemented [BKR09].

The division of software systems into different types of models has been addressed by different approaches, including Multi-View Modeling (MVM) [ABB+02; KAK09] and Aspect-Oriented Modeling (AOM) [KK11] approaches. MVM focuses on the separation of concerns based on views, like architecture, deployment, data, and behavior, which are also reflected in the Unified Modeling Language (UML) [UML15] by the various diagram types. AOM uses the notion of aspect and base model to separate cross-cutting concerns. It follows the same basic concepts as Aspect-Oriented Programming (AOP), but applies them to models. These models can use the same metamodel (cf. [KK11]) or separate aspect and base metamodels with specific abstract syntax and semantics [BCD10; JHS+14].

Models can be created and altered quickly by specialized tooling. However, they cannot be deployed and executed directly. Therefore, they must be transformed into code and other artifacts necessary to implement a software system. This transformation is realized with generators and subsequently by compilers. Therefore, generators play a central role in MDE [MJ13].

In AOM and MVM generators must process multiple source models, representing different aspects and views. Each aspect and view implies its own semantics which must be realized by these generators, which makes them complex artifacts, particularly when multiple metamodels are involved. Therefore, generators are hard to construct and evolve. Unfortunately, requirement changes of software systems affect not only functionality, but can also introduce new concepts into the modeling domain. This is usually addressed by modifying metamodels, which subsequently requires a modification of the related generators. Furthermore, the target platform of a software system may change. This can be triggered by the introduction of new platform versions, the end-of-life for the present platform, migration to a new system, and others. All these changes require a modification of the used generators. Repeated modification and extensions of a software application can lead to architecture degradation. A degrading architecture may lead to more errors in the implementation, requires more effort to modify the application in the future, and increases maintenance costs and time. As generators are themselves software applications, they are subject

to the same issues. In software engineering, the issue of architecture degradation is usually addressed by modularization, where the modules have high cohesion and the coupling between the modules is low [IS091; ISO11].

In this thesis we introduce a generator composition approach, called GECO, to support the construction, evolution and reuse of generators. This is achieved by three approaches. First, generators are divided along specific views and aspects of the application and technical domain, resulting in smaller generators, we call generator fragments. Second, these fragments are further modularized along metamodel features, like typing and expressions, and functionality. Finally, we use existing methods and technology based on AOM and AOP to combine the output of fragments.

Through this two level modularization, changes to metamodels, platform, and environment of the software system, only affect single fragments and within the fragments only those modules which are related to the metamodel.

1.2 Approach Overview and Contributions

In this thesis, we describe an approach for the construction and evolution of generators, called GECO. The approach relies on the understanding of metamodel semantics and provides two levels of modularization and composition for generators. We supplemented these approaches with a framework, tooling, and an aspect-oriented language for instrumentation.

1.2.1 Metamodel Semantics

Our generator construction and evolution approach relies on an understanding of semantics in metamodels, which is used to support their construction, the identification of views and aspects, and the partitioning of metamodels along boundaries based on semantics, like typing and expressions.

We therefore, provide a detailed categorization of syntactic and semantic properties of metamodels. Furthermore, we derived methods to identify views and aspects, and supplemented this with a method to find metamodel partitions.

1.2.2 Generator Composition Patterns

In GECO, generators are subdivided into generator fragments. The composition of generators from these fragments is expressed with megamodels [Fav04a], which allow to model the relationship of models, metamodels, and transformations. We concluded that only five megamodel patterns are necessary to construct generators out of fragments which support a minimal interface.

1.2.3 Generator Fragment Modularization

The megamodel patterns allow to construct generators from fragments, providing one level of modularization. As single fragments could still be complex components, we devised a modularization approach for fragments. This approach introduces two major dimensions of modularization for fragments based on functionality and source metamodel semantics.

1.2.4 Generator Composition Framework and Tooling

The generator composition approach and fragment modularization approach of GECO are technology independent. However, to illustrate the GECO and support evolution, we developed a framework and tooling for the Xtext language workbench [Bet13] based on GECO.

The framework provides interfaces for fragments, classes for common functional modules like trace models, and classes to integrate different transformation languages. The framework is supplemented by a Domain-Specific Language (DSL) and a generator, which allows to specify generator megamodels with a textual syntax and an automatically generated graphic view of the assembly utilizing KIELER [SSH13].

1.2.5 Instrumentation Aspect Language

Motivated by different software projects including iObserve [HHJ+13; HSJ+14], which rely on runtime application monitoring, we devised an Instrumentation Aspect Language (IAL). The IAL allows to specify monitoring sensors or probes and their integration in the software independent from

the underlying technology. The generators and tooling support the Kieker monitoring framework, which provides monitoring probes and events for different technologies [HWH12].

1.3 Document Structure

The remainder of this thesis is structured as follows:

Part I comprises the foundations

- ▷ Chapter 2 introduces a formalization for type-systems, which is used throughout the thesis.
- ▷ Chapter 3 comprises various modeling topics used in this thesis including models, metamodels, model transformations, domain-specific languages, model traceability, weaving, megamodels, and transformation languages.
- ▷ Chapter 4 summarizes an entropy based approach to measure size, complexity, coupling, and cohesion of software artifacts utilizing a graph and hypergraph abstraction.

Part II discusses the different parts of the GECO approach

- ▷ Chapter 5 provides an overview of the research including work packages and research questions.
- ▷ Chapter 6 discusses metamodel semantics and methods to partition metamodels.
- ▷ Chapter 7 introduces megamodel patterns dedicated to support the composition of generators from simpler generator fragments.
- ▷ Chapter 8 presents an approach for the modularization of generator fragments based on semantic and functional considerations.

Part III covers the evaluation and related work

- ▻ Chapter 9 provides an overview of the evaluation and contains the experiment design.
- ▻ Chapter 10 discusses prototypical implementations addressing different aspects of the GECO approach. The tools and frameworks were used during the evaluation.
- ▻ Chapter 11 contains the documentation and interpretation of the evaluation results including both case studies and the interviews.
- ▻ Chapter 12 discusses the related work of this thesis.

Part IV concludes this thesis

- ▻ Chapter 13 contains the conclusions.
- ▻ Chapter 14 provides an outlook on future work

Appendices contain supplementary material including interview results, instructions on how to find and use the tooling and the framework, and information on the replication packages and evaluation data.

Part I

Foundations

Chapter 2

Type Systems

Type systems are a central element of programming and specification languages. They comprise of rules which inscribe properties called types to language constructs, like variables, expressions, and functions. Types describe data structures and their static properties, while terms express behavior. Type systems together with operational semantics [Fer14] are a useful tool to define semantic constraints of programs and allow to provide feedback to programmers at compile and execution time. Pierce [Pie02, p. 1] classifies them as lightweight a formal method, which allow to implement automatic type checks in compilers and interpreters.

Type systems are used for various purposes including error detection, abstraction, and documentation. Static type-checking allows to detect errors, like, missing type conversions, undefined methods, and inappropriate method invocations at compile time. Type systems allow to define abstractions, e.g., interfaces, and provide type resolving for them. This supports the modularization, which is especially useful in large software systems.

In this thesis, type systems are used for metamodel partitioning and to specify type systems for DSLs. Metamodel partitions are identified based on metamodel semantics and the containment hierarchy. One specific kind of partitions are typing structures. Furthermore, we used type systems to specify the semantics for the GECO fragment composition language (Section 10.2) and other DSLs used in the evaluation.

The remainder of this chapter discusses foundational aspects of types and type systems (Section 2.1), simple types (Section 2.2), sub-typing and sub-classing (Section 2.4), type inference (Section 2.5), and concludes with a summary (Section 2.6).

2.1 Typing Overview

Type systems are build on base types, rules, and structures that allow the definition of more complex types. These rules result in typing constraints partially defining the semantics of a language. In this section, general concepts on types are discussed in Section 2.1.1, characteristics of type systems are explained in Section 2.1.2 and formalizing types with operational semantics are summarized in Section 2.1.3.

2.1.1 Types

Types and type theory [Jac99] are mathematical topics. In context of this thesis, types can be seen as sets of values. While types may share values, they are considered distinct, meaning that expressions of one type cannot be combined with expressions of another type without an implicit or explicit type cast or conversion. In this section we introduce three typing aspects to support the understanding of the formalization of type systems (see Section 2.1.3). These aspects are the foundation of types, the distinction of types, and the relationships of types.

As Cardelli and Wegner [CW85] states, an understanding of the semantic theory of types in detail is not necessary to work on type systems, a basic grasp of its foundation, however, is helpful. The main intuition of type theory is that there is a universe V containing all values that a particular set of types can incorporate. A type, in this context, is defined as a subset of V [CW85].

All subsets of V, which are types, form a lattice. The top of this lattice is the type `Top` containing all values of V. As values of subsets of V may overlap, a value can have different types. Furthermore, a subset A can be a subset of the subset B, which allows to express a sub-type relationship of the corresponding types.

Computer languages are not only formed to model values, they also describe behavior in form of terms also called expressions (cp. [Pie02, p. 24]). Like values, terms are typed. The type of a term is not always explicitly specified, but rather inferred from the context. In many cases only variables, properties, and function declarations are explicitly typed.

In this thesis, we categorize types in base types and composed types. Base types are atomic elements of the type system, their subset specification is relevant for the semantics of a computer languages. In generators base types of the source language must be mapped to the target language. Therefore, an understanding of base types and their semantics is important to the GECO approach. Composed types represent all other types which can be formed using typing rules and the base types. In contrast to base types, composed types can be defined by the language user.

2.1.2 Properties of Type Systems

Before providing a summary of the formal notation of types and type systems, some properties and general categories of type systems need to be explained.

The primary goal of type systems is to allow to check programs statically at compile time and in an editor. Languages providing such facilities are categorized as *statically checked* languages. However, some properties, for example the down-cast operator in Java which cannot be checked completely at compile time, require checks at run-time. Such language constructs are phrased as *dynamically checked* [Pie02, p. 6].

These two terms should not be confused with static and dynamic typing, even though they are very similar in meaning. *Static typing* describes type systems, where all checks can be performed at compile time, while *dynamic typed* languages are not able to perform all type checking at compile time and must, therefore, perform run-time checks [Pie02, p. 2].

Finally, we introduce language safety. A language is considered safe when it "protects its own abstraction" [Pie02, p. 6], meaning that any operation in the language does neither corrupt the code nor the data. Languages, like Java, which is mostly static typed languages, or Postscript are safe languages, as neither the data or the code can be corrupted by executing a valid operation. The programming language C, is statically typed, but it is an unsafe language, as it has no bounds checking for arrays, which allows a program to write beyond the boundaries of an array corrupting data of other structures. Furthermore, in C pointers can be manipulated and any given value can be assigned, allowing to modify

or even execute any byte in memory, resulting in unpredictable behavior. Hence, statically checked languages are not a priori safe nor are dynamically checked languages unsafe.

2.1.3 Formalizing Type Systems

Type systems and language semantics are interconnected [CDJ+97]. The specification of a type system can therefore not be made without the specification of other aspects of a language. Essentially, four aspects must be considered: First, an abstract syntax is required, which allows the typing rule to relate to syntactic structures. Second, scoping rules for the language must be defined to express the binding of identifiers to their declarations. This also defines the visibility of variables, types, and other named elements. Scoping can be realized formally by a set of free variables [CDJ+97]. Third, a static type environment is required to reason about types. Such environment collects all relationships between free variables and types. Fourth, typing rules to express the relation between terms and types must be defined [CDJ+97].

Typing Relations Before we can describe complex typing rules, we must introduce a set of relations. First, one must be able to specify that a term t has a specific type T. This is done with the *has-type* relation $t : T$ [CDJ+97]. Second, in languages and type checkers the equivalence and in some also a subtype relation are used to express relationships between types. The *subtype-of* relation expresses that if a term $s : S$ and a type S is subtype of type T then s is also of type T. However, a term $t : T$ is not necessarily also of type S. The subtype-of relation is written as $S <: T$, where S is subtype of T. It is considered to be reflexive and transitive [CDJ+97]. The *equivalence* relation $S = T$ expresses that S and T are equivalent, meaning if $a : T$ then $a : S$ and vise versa [CDJ+97]. The equivalence relation is used in contexts where types are resolved or reconstructed, such as in *duck typing* and other dynamic methods.

Static Typing Environment A *static typing environment* Γ is a set of free variables with an associated type [Pie02, p. 101]. They are used in processing

code fragments and are similar to symbol tables of a compiler [CDJ+97]. An empty environment is often represented by $\varnothing$.

The type of a term can only be defined in relation to a typing environment, because the same syntactical structure can be valid for different types. For example, in Java $a + b$ can be of any type. The type can be a numerical type, such as int and float, but also of type String. Therefore, the complete relation is written as $\Gamma \vdash t : T$, meaning term t has the type T under the static typing environment Γ. The domain of Γ ($dom(\Gamma)$) is the set of all free variables [CDJ+97].

The general form of such relation is called a *judgment*. It is normally written as $\Gamma \vdash \mathfrak{I}$ where $\mathfrak{I}$ is an assertion. The free variables used in $\mathfrak{I}$ must be in $dom(\Gamma)$. Judgments are used to express any number of properties of type systems. They can be used to define values or express logic consequences. For example, $\varnothing \vdash true : Boolean$ states that the value $true$ is of type Boolean. The term $\varnothing, x : Int \vdash x + 1 : Int$ extends the empty environment by one free variable x of type Int and then entails that $x + 1$ is also of type Int. To express that a typing environment has been composed properly, one writes $\Gamma \vdash \diamond$. The empty environment is by definition also properly composed and is expressed as $\varnothing \vdash \diamond$ [CDJ+97].

Typing Rules On the basis of such judgments, typing rules can be formulated. They are written as inference rules with a set of judgments above the bar, called premises, and one conclusion judgment below the bar.

$$\frac{\Gamma_1 \vdash \mathfrak{I}_1 \ldots \Gamma_n \vdash \mathfrak{I}_n}{\Gamma \vdash \mathfrak{I}} \qquad \text{(General rule form)}$$

The phrase 'the typing environment entails' ($\Gamma \vdash$) may appear repeatedly in typing rules. As this make the term more complex and harder to read, it is sometimes omitted if the typing environment is not modified (cf. [Pie02]).

The resolution of a term is expressed by repeatedly applying typing rules, which results in a tree structure. The leaves of such structure are called axioms. Cardelli, Donahue, et al. [CDJ+97] write axioms, like a normal typing rule with an empty premise. They also use type rules to declare types and their values, which could also be expressed as axioms. Their pattern for those declarations is:

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathfrak{J}} \qquad \text{(General declaration rule)}$$

However, Pierce [Pie02] describes such declaration rules as axioms and skips the horizontal bar in the notation, which results in the following compact notation, which we use in this thesis:

$$\Gamma \vdash \mathfrak{J} \qquad \text{(Compact axiom notation)}$$

To distinguish different kinds of rules, all rule names are prefixed, following the notation presented by Pierce [Pie02, p. 565]. Normal typing rules are prefixed with T and subtyping rules with S.

Type Derivation The type derivation starts with a term, which type soundness has to be proven. This term is the root of the derivation. The first step is to find a type rule which matches structurally the term to be checked for type soundness. Such type rule may have one or more premises, which hold the remaining terms. In the next step for each of these terms suitable type rules have to be found. This substitution stops when all remaining terms are axioms.

Expression Grammar

```
expr: value | if | not
not: '!' expr
value: 'true' | 'false'
if: 'if' expr 'then' expr
    'else' expr
```

(T-Type-Bool)
$\Gamma \vdash Bool$

(T-Value-True)
$\Gamma \vdash true : Bool$

(T-Value-False)
$\Gamma \vdash false : Bool$

(T-Inverse)
$$\frac{\Gamma \vdash t : T}{\Gamma \vdash !t : T}$$

(T-Condition)
$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{ if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T}$$

Figure 2.1. Minimal boolean typed expression language

In Figure 2.1, an example language for boolean values with some basic

expressions is presented. It comprises a type declaration for a boolean type based on axioms and two type rules, one for the inverse of a term and one to express a condition.

$$\dfrac{\dfrac{x : Bool}{!x : Bool} \qquad false : Bool \qquad true : Bool}{\texttt{if } !x \texttt{ then } false \texttt{ else } true}$$

Figure 2.3. Derivation tree for an expression of the minimal boolean language

For the expression $\texttt{if } !x \texttt{ then } false \texttt{ else } true$ and a $\Gamma = x : Bool$ the derivation tree in Figure 2.3 can be build. The same tree would fail if $\Gamma = \varnothing$, as x would not be bound to a type.

2.2 Simple Types

In the previous section basic terminology and formalisms for types and type systems have been introduced, which we rely on to discuss the structure of simple types in this section. There are numerous kinds of typing structures which are discussed in literature (cf. [Pie02; Pie04]). For this thesis we focus on typing structures which are common in many programming languages and DSLs, especially those specified in the evaluation of this thesis.

In Section 2.2.1, we start with base types as the axiomatic elements of a type systems. Section 2.2.2 introduces the type `Unit`, used in several subsequent type constructs, like enumerations and ascriptions. The latter are explained in Section 2.2.3. Most languages provide structured data types, such as records, introduced in Section 2.2.4, and variants described in Section 2.2.5. Enumerations are also an essential for many languages and can be modeled in different ways. A summary on a variant based approach can be found in Section 2.2.6. Let-bindings are used in some modern languages, such as Haskell [Hut07] or Scala [OSV08], and play a role in DSLs. A realization for let-bindings is introduced in Section 2.2.7. Finally, Section 2.2.8 presents the concept of references.

2.2.1 Base Types

Base types are mutually distinct elements, where each element represents one type. Base types have no further internal structure from the viewpoint of type systems [Pie02, p. 117]. This does not necessarily imply that their associated sets of values do not share values. For example, a 16 bit signed integer and a 32 bit unsigned integer have the interval $[0 : 32767]$ in common.

For the type system itself they are considered distinct entities. Therefore, the transformation of a value of one base type to another can only be provided by cast operators [Pie02, p. 193].

Classical base types are integer types, floating point types, and boolean types. In many languages they are accompanied by string as a base type. However, there are languages, like `C`, which use character sequences to implement stings on the basis of a reference to the integer type char.

2.2.2 Unit Type

The type `Unit` is also considered to be a base type. However, it plays a different role in type systems, as it is used to realize other typing constructs and plays a role in formulating statements as special kinds of expressions. The value set of `Unit` has only one value `unit`, written in lowercase. The type `void` in the programming language C is conceptually related to the type `Unit` and can be seen as a real world application of a `Unit` type.

Imperative languages, like `C`, work primarily on side effects of terms and the result of term evaluation is not important, which can be expressed by the type `Unit`. Such terms are called statements. In imperative languages, statements are used in sequences, which are in some cases separated by a semicolon. The *sequencing notation* $t_1 \ ; \ t_2$ used to describe a sequence of statements, implies that first t_1 is evaluated, its trivial result (`unit`) is ignored and t_2 is evaluated (cp. [Pie02, p. 119ff]). The typing rule for sequences [Pie02, p. 120] is defined as:

$$\frac{\Gamma \vdash t_1 : Unit \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \ ; t_2 : T_2} \qquad \text{(T-SEQUENCE)}$$

The typing rule states, that if we have a term $t_1 : Unit$ in the environment Γ and a term $t_2 : T_2$ of any type then the type of the sequence is also of that second type. The rule could recursively be applied to a sequence of statements. The type of the last statement will then define the type of the statements sequence.

In C or Java, the corresponding typing rule for sequences does not make use of the type Unit. Similar to Cardelli, Donahue, et al. [CDJ+97], the left statement can return a value of any type, which is then ignored. However, the use of Unit would force the use of a real statement as a statement and not misuse functions as statements. In safety context such distinction can be helpful.

2.2.3 Ascription

Ascription is a method to explicitly ascribe a type to a term. It can be used to document which type a term has and it can be used to validate an expression t whether it really is of type T. Furthermore, it can be used to abbreviate complex type expressions. For example, instead of writing $T \rightarrow T$ as a function type, we define $TT = T \rightarrow T$ and use TT as type throughout the program.

2.2.4 Record Types

Data structures help programmers to comprise related data in one element, often called record type. A record type comprises several labeled fields of any other type in one structure. This allows to model data types, like address or customer records. To describe the typing rules for record types, at least three rules and axioms are necessary (see Figure 2.4). In these rules the labels for record types are named l, general terms are represented by t and types by T.

The axiom T-Type-Record defines the general structure of a record type, which comprises n labels l_i with corresponding types T_i. The rule T-Value-Record expresses that for every label l_i of a record the type must comply to the type of the assigned term t_i. Finally, the rule T-Select-Record states

(T-TYPE-RECORD)
$$\Gamma \vdash \{l_i : T_i \; ^{i \in 1 \ldots n}\}$$

(T-VALUE-RECORD)
$$\frac{\forall i \in \{1 \ldots n\} \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\} : \{l_i : T_i\}}$$

(T-SELECT-RECORD)
$$\frac{\Gamma \vdash m : \{l_i : T_i \quad ^{i \in 1 \ldots n}\} \quad j \in \{1 \ldots n\}}{\Gamma \vdash m.l_j : T_j}$$

Figure 2.4. Typing rules for structured records [Pie02, p. 129; Car04, p. 18]

that the type of one label l_j from an element m can be concluded, if l_j is in the set of labels of the record type.

2.2.5 Variant Types

Variant types allow to model alternate structures within one type. For example, a binary-tree is composed of branches and leaves. Without subtyping, tree-nodes must be modeled with two references to the child nodes and the payload. However, the leaves do not need references pointing to child nodes as this wastes memory, and causes problems when handled inappropriately. A solution to this issue are variant types. The C programming language supports variant types, called unions.

The rule definitions in Figure 2.5 are derived from Cardelli [Car04] and Pierce [Pie02]. The axiom T-TYPE-VARIANT is used to describe variant types, which are defined as a sequence of labels l_i associated with a types T_i. In contrast to record types only one of the labels can be used on any instance of this type. The rule T-VALUE-VARIANT expresses this constraint.

The rules T-VALUE-IS-VARIANT and T-VALUE-AS-VARIANT model the semantics for two operators, which allow to identify and to access the label used in a specific variant instance, respectively. Finally, the T-VALUE-CASE-VARIANT rule is used to type the selection of the correct type.

(T-Type-Variant)

$$\Gamma \vdash < l_i : T_i \ ^{i \in 1 \ldots n} >$$

(T-Value-Variant)

$$\frac{\forall i \in \{1 \ldots n\} \ \Gamma \vdash T_i \quad \Gamma \vdash m_j : T_j \quad ^{j \in \{1 \ldots n\}}}{\Gamma \vdash l_j = m_j :< l_i : T_i \ ^{i \in 1 \ldots n} >}$$

(T-Value-Is-Variant)

$$\frac{\Gamma \vdash m :< l_i : T_i \ ^{i \in 1 \ldots n} > \quad ^{j \in \{1 \ldots n\}}}{\Gamma \vdash m \ \texttt{is} \ l_j : Bool}$$

(T-Value-As-Variant)

$$\frac{\Gamma \vdash m :< l_i : T_i \ ^{i \in 1 \ldots n} > \quad ^{j \in \{1 \ldots n\}}}{\Gamma \vdash m \ \texttt{as} \ l_j : T_j}$$

(T-Value-Case-Variant)

$$\frac{\forall i \in \{1 \ldots n\} \Gamma \vdash m < l_i : T_i > \quad \Gamma, x_i : T_i \vdash n_i : T}{\Gamma \vdash \ \texttt{case} \ m \ \texttt{of} \ < l_i = x_i > \ \texttt{then} \ t_i : T}$$

Figure 2.5. Typing rules for variant types [Pie02, p. 129; Car04, p. 18]

2.2.6 Enumeration Types

Enumerations are widely used in programming languages to define sets of mutually distinct values, which have, in most cases, a nominal character. In type systems, enumerations can be modeled as variant types, where each variant is of type *Unit* (see [Pie02, p. 138]). For example, a weekday enumeration type is coded as $Weekday =< monday : Unit, tuesday : Unit, wednesday : Unit, thursday : Unit, friday : Unit >$. Beside this definition, some languages allow to assign numerical values to the labels of the variant.

2.2.7 Let Bindings

When writing complex expressions, it might be advisable to break it up into several sub-expressions, especially when parts of the expression have to be repeated. In those cases an expression can be bound to a name. In the language ML [MTH90], the syntax for let is $\texttt{let}\ x = t_1 \ \texttt{in}\ t_2$. Where x is the name for the binding, t_1 is an expression which is evaluated and the result is bound to x, and the t_2 is the term which uses x. Similar let bindings are part of Lisp [McC62], F [FHM97], and Clojure [Hic08].

2.2.8 References and Storage-Allocation

Many programming languages use references to data objects. They are implemented as pointers or in modern languages as references to entities. References require three basic operations *allocation*, *dereferencing* and *assignment*.

(T-Reference-Type)
$$\frac{\Gamma \vdash T}{\Gamma \vdash \texttt{Ref}\ T}$$

(T-Reference)
$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ t_1 : \texttt{Ref}\ T_1}$$

(T-Dereference)
$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_1}{\Gamma \vdash \texttt{deref}\ t_1 : T_1}$$

(T-Assignment)
$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : Unit}$$

Figure 2.6. Typing rules for reference types [Pie02, p. 159; Car04, p. 19]

Similar to an object oriented `new`-operator, the allocation for a reference reserves new space in memory. In Pierce [Pie02, p. 154] this allocation is expressed with the `ref`-operator. The `ref`-operator allocates space for a value defined by the term following the operator and automatically stores the value in the allocated space. The term that includes this operator is of type `Ref` T, as the rule T-Reference in Figure 2.6 states. To access the value from the reference, it has to be dereferenced (see rule T-Dereference). The remaining rules are T-Reference-Type, which defines the reference type as such, and T-Assignment, which is used to model the assignment of a value to a reference.

2.2.9 Arrays

The previous type composition concepts only allow single entities of a type. For many software systems, such structures are not sufficient to express data in a concise way. A solution to this problem are recursive types, explained in Section 2.3, and arrays. An array is a sequence of elements of the same type called cells. Each cell holds a value of the same type. Arrays have a

defined fixed size, resulting in a lower and upper bound for entries. For this introduction, the lower bound is always 0.

On a theoretical level, an array consists of n elements for values or references to values of a given type. This could be written as $ArrayTn$ where T is the type for the cells and n is the number of elements.

(T-Type-Array)
$$\frac{\Gamma \vdash T}{\Gamma \vdash \ \texttt{Array}\ T}$$

(T-Val-Array)
$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash t : T}{\Gamma\ \texttt{array}\ (n, t) :\ \texttt{Array}\ T}$$

(T-Val-Array-Bound)
$$\frac{\Gamma \vdash t\ \texttt{Array}\ T}{\Gamma \vdash \ \texttt{bound}\ t : Nat}$$

(T-Array-Index)
$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash t :\ \texttt{Array}\ T}{\Gamma t[n] : T}$$

(T-Array-Update)
$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash t_1 :\ \texttt{Array}\ T \quad t_2 : T}{\Gamma \vdash t_1[n] := t_2 : Unit}$$

Figure 2.7. Typing rules for arrays [Car04, p. 20]

Figure 2.7 expresses the typing rules necessary for arrays. T-Type-Array allows the construction of an array type out of any other type. T-Val-Array describes that every array element belongs to the array. Whereas T-Val-Array-Bound realizes the typing for the bound expression, which is used to check the array's upper bound. The last two rules are used to describe the access and update of array elements.

2.3 Recursive Types

Most programming languages and DSLs allow to define types recursively, which is very useful for all kinds of types with arbitrary sizes, like lists, trees, or queues. A typical recursive type is a record type where one attribute of the record type is of the same record type.

Computer languages often introduce those concepts for type recursion implicitly. However, an explicit introduction and formalization is helpful to understand their characteristics. Therefore, an introduction is given in Section 2.3.1. An application for collections is subsequently specified in Section 2.3.2.

2.3.1 Concept

There are two different approaches to model recursive types with interference rules, called equi-recursive approach and iso-recursive approach. The equi-recursive approach is compatible with all proofs and theorems for type systems without recursive types. However, induction on type expressions can no longer be applied [Pie02, p. 276]. In the equi-recursive approach there is no distinction between the recursive type and its unfolding. Furthermore, type-checking algorithms cannot work directly with the infinite structures this approach implies. The iso-recursive approach differentiates between recursive types and their unfolding, but they are seen as isomorphic [Pie02, p. 276]. While this approach requires minor adjustments to proofs and theorems, it is, according to Pierce [Pie02], easier to realize in a type-checker. Therefore, most languages rely on this approach.

The basis for recursive types are the type variable X and the recursive type $\mu X.T$, where X stands for type variable and T for the recursive type, which uses X. For example, $\mu X. < nil : Unit, cons : \{Nat, X\} >$ resembles a list type.

In type systems with iso-recursive types, two functions are required to fold and unfold types. Unfold replaces each X in T with T, formally $\texttt{unfold}\ [\mu X.T] : \mu X.T \rightarrow [X \mapsto \mu X.T]T$. For the list type example, this would result in $< nil : Unit, cons : \{Nat, \mu X. < nil : Unit, cons : \{Nat, X\} > \} >$. The corresponding fold function is defined as $\texttt{fold}\ [\mu X.T] : [X \mapsto \mu X.T]T \rightarrow \mu X.T$.

(T-Environment-X)
$$\frac{\Gamma \vdash \diamond \quad X \notin dom(\Gamma)}{\Gamma, X \vdash \diamond}$$

(T-Recursive-Type)
$$\frac{\Gamma, X \vdash T}{\Gamma \vdash \mu X.T}$$

(T-Fold)
$$\frac{\Gamma \vdash t : [X \mapsto \mu X.T]T}{\Gamma \vdash \texttt{fold}\ [\mu X.T]\ t : \mu X.T}$$

(T-Unfold)
$$\frac{\Gamma \vdash t : \mu X.T}{\Gamma \vdash \texttt{unfold}\ [\mu X.T]\ t : [X \mapsto \mu X.T]T}$$

Figure 2.8. Iso-recursive types [Pie02, p. 276; Car04, p. 21]

Figure 2.8 defines four rules to define iso-recursive types in general. The rule T-Environment-X introduces the X type variable to the static typing environment. T-Recursive-Type defines the recursion function for types. And the last two rules define the typing for the fold and unfold functions.

2.3.2 Collections

Collections and list types can be defined either on the basis of recursive types, or similarly to the above array approach defining `get`, `put`, `insert`, and `next` operations. Utilizing the above rules, a generic list type can be defined as $List_T = \mu X. < nil : Unit, cons : \{T, X\} >$, where T can be any type already known to the type system.

2.4 Sub-Typing and Sub-Classing

In Section 2.1, the *subtype-of* relation was introduced. Based on this relation and on concepts of record types, typing rules for subtypes can be constructed.

In general, subtyping is a reflexive and transitive relation, which can be described with the axiom S-Reflexive and the rule S-Transitive.

$$S <: S \qquad \text{(S-Reflexive)}$$

and

$$\frac{S <: U \quad U <: T}{S <: T} \qquad \text{(S-Transitive)}$$

For record types, the subtype must provide the same fields as its supertype, but can add new fields. This is called *width subtyping* and can be expressed by the following axiom S-Record-Width (see [Pie02, p. 183]).

$$\{l_i : T_i^{i \in 1..n+k}\} <: \{l_i : T_i^{i \in 1..n}\} \qquad \text{(S-Record-Width)}$$

As expressed in this rule, the label and type relations in the subtype are the same as those in the supertype. The given rule, however, implies an order for the record labels, which is more restrictive than necessary. The

permutation rule from Pierce [Pie02, p. 184] solves that issue. However, for this brief introduction of type systems, we omit this particular rule.

As the name *width subtyping* suggests, there are other "directions" of subtyping. The second rule to introduce is the depth subtyping rule, named S-RECORD-DEPTH:

$$\frac{\forall i \quad S_i <: T_i}{\{l_i : S_i^{i \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}} \qquad \text{(S-RECORD-DEPTH)}$$

The previous rule only described the addition of new properties (label type relations) of a record, but did not allow to modify the type of the record properties. *Depth subtyping* describes the ability to change the type of properties which are specified in the supertype. However, not every type can be used in place of the original type used in the supertype, because the subtype must be valid in all places where its supertype is valid. Therefore, the type of a property can only be substituted by a subtype of that particular type (see [Pie02, p. 183]).

These two rules handle records which are also the basis for classes and play a role in variants. For functions $T_1 \rightarrow T_2$ an additional rule is required. Functions are structured types which involve at least two types, one for the argument and one for the result.

```
Customer <: Person
DeliverAddress <: Address

Customer r = { name=Charles Dickens }
Address i = f(r)

getAddress (Customer r) : Address
getDeliverAddress subtypes getAddress (Person r) : DeliverAddress
```

Listing 2.1. Function subtyping example

Listing 2.1 shows a short example of a function *getAddress* and its subtyped counterpart *getDeliverAddress*. To understand the subtyping rule for functions, it is important to understand the context where functions are used. Functions play a role in expressions, they take arguments and return values. The return value must conform the context of the expression. In the example the function *getAddress* is used in an assignment where the left side requires an `Address` type. Therefore, *getAddress* must return `Address` or

any subtype of Address. The function *getDeliverAddress* fulfills that criteria, as DeliverAddress is defined as a subtype of Address in the example.

The argument type for *getDeliverAddress* can be any supertype of Customer, which is the type Person in this example. Seen in the context of the expression, this modification still holds, as the Customer r is also a person and can, therefore, be used as parameter value. These two criteria for subtyping function types can be modeled in one rule S-Arrow [Pie02].

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \qquad \text{(S-Arrow)}$$

Languages supporting subtyping, such as Object-Oriented Programming (OOP) languages, provide some top most supertype, called Object. In Pierce [Pie02, p. 185] this type is named Top to indicate its top most position in the type system. All other types are descendants of this type. The rule S-Top expresses that property.

$$S <: Top \qquad \text{(S-Top)}$$

The counterpart of Top is the bottom type Bot. It is the subtype of every type. Bot does not define any values. However, it can be used where operations are not intended to return. This can be the case for exceptions and operations, such as continue and break in C-like languages.

The typing rules described in this section allow to formulate subtyping. However, they cannot be used directly to realize a typing algorithm (see [Pie02, p. 210]). For example, the typing rule S-Transitive uses the two metavariables S and T, which can match any subtyping statement. Therefore, a naive bottom-to-top algorithm might not be able to select the correct rule, if there is another rule with a subtyping relation in its conclusion. Furthermore, the metavariable U in S-Transitive can be any type. Since there can be an infinite number of Us, it is not guaranteed that a matching U is found [Pie02, p. 211]. As the implementation of type systems is not part of this thesis, we do not discuss the algorithmic realization of subtyping in the foundations. Instead, the introduction on type systems for programming languages by Pierce [Pie02, p. 209ff] should be used.

2.5 Type Inference

In terms, not all elements have a type directly and visually attached, as this would render expressions unreadable (compare $(a : T + : T\ b : T) * : T\ c : T$ with $a + b * c$). Operators, like $+$ and $*$, can often handle different types. Therefore, the resulting type depends on the type of the variables a, b, and c. Putting casts aside, the type of the operators is inferred by starting at the leaves of the Abstract Syntax Tree (AST) and applying the type of the leave to the operator.

There are more complex type inference algorithms used for casts, subtyping and -classing, or dynamic typing. However, complex type inference algorithms are not in the scope of this thesis. A detailed discussion on such algorithms can be found in [Pie02].

2.6 Summary

In this chapter, we introduced type systems and the basic building blocks used to construct them. The typing rules are expressed utilizing operational semantics. In GECO, type systems and operational semantics play a role in the definition of metamodel semantics and type mapping in generators. Furthermore, typing rules are used in the evaluation of GECO to support the understanding of DSLs.

Metamodel semantics comprise type systems and operational semantics, depending on the purpose of a particular metamodel Knowledge of the semantics is used as one basis for metamodel partitioning in Chapter 6. The fragment construction in Chapter 8 discusses, amongst others, the mapping of source to target language types. Therefore, an understanding of type systems is necessary. Finally, we use typing rules in the specification of the GECO fragment composition language (see Section 10.2), in a prototypical typing method (Section 10.3), and in DSLs used in the case studies (Section 10.4). The rules used in the evaluation of this thesis are intended to support the understanding of concepts exposed by these languages and help in the development of other DSLs.

Chapter 3

Modeling

Models are employed in a wide variety of tasks and activities in Model-Driven Engineering (MDE). They are used to define software systems and their requirements, steer the process of software development and maintenance, specify model structure and semantics, and determine the transformation of models and their serialization. In this thesis, the emphasis is on the development and evolution of code and model generators, and their division along the technical and semantical dimension. Therefore, a solid understanding of models and their syntactical and semantical properties is of essence for GECO. An introduction to models and metamodels is given in Section 3.1. This includes and introduction to aspect-oriented and view-based modeling, which provide the foundation for the modularization of metamodels and generators.

Generators are used to create models and code by transforming an input of source models into an output of target models. The latter may then be serialized as code. The basic terminology and different types of transformations are introduced in Section 3.2 to provide a common understanding throughout the thesis.

Metamodels provide an abstract foundation for models, but do not specify how models can be constructed by a developer. Domain-Specific Languages (DSLs) provide the means to specify models textually and graphically. Section 3.3 introduces DSLs, their construction, and their relationship to modeling and programming languages.

When a source model is transformed into a target model, elements from the former are processed to determine the creation of elements for the latter. Therefore, an implicit relationship exists between source and target model elements. These relationships can be stored in trace models which are used

to identify after the transformation which source and target model elements relate to each other. In GECO, these trace models are used to resolve aspect model references. An introduction to trace models and model traceability is given in Section 3.4.

While in aspect-oriented modeling, the cross-cutting concern is modeled in a model separate from the base concern and only a join point model relates both models, on a lower level, e.g., program code, the aspect must be integrated into the base model or code. This integration is called weaving and is discussed in Section 3.5.

The GECO approach focuses on the composition of generators from partial generators and the transformations they realize. To express the assembly of transformations and models, megamodels are utilized as language. The notion and concept of megamodels are introduced in Section 3.6 providing the basis for the definition of megamodel pattern occurring in such a model and to express the generator composition in both case studies.

While GECO itself is technology independent, it is important to understand different transformation languages and their underlying concepts to be able to use them correctly in the context of GECO. Therefore, we introduce a set of widely used languages in Section 3.7 providing information on the overall concept of each language and specific features relevant in context of GECO.

3.1 Models and Metamodels

The GECO approach addresses issues with the construction and evolution of generators for models. In this section, we introduce the foundation of model terminology, their formal definition and the notation used in this thesis. For a better understanding of the terminology, we provide an example metamodel of a software system for a supermarket.

This simple supermarket example comprises four classes for the inventory and the representation of a shopping cart. The Inventory has an association products referring to a collection of Product instances (see also Figure 3.1). A product can belong to a group of products, for example, pepper could belong to spices. An inventory alone is not sufficient to model a

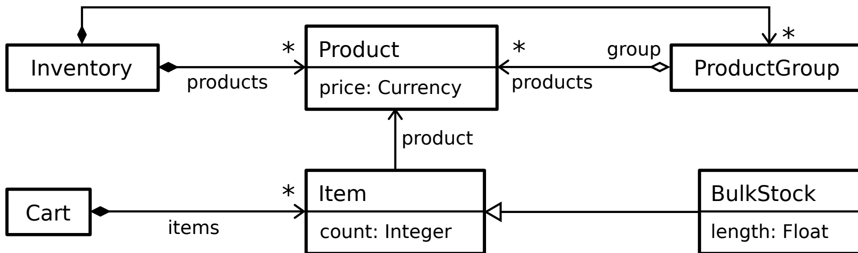


Figure 3.1. Example of a partial metamodel for a supermarket software system

supermarket. At least it must be possible to collect purchases in a shopping cart. Therefore, the example also has a class Cart with a collection of Item instances.

3.1.1 Introduction of Modeling Terminology

In MDE a *model* represents the structure, function and behavior of a software system [SV06, Sec. 2.4.1], as well as, supplemental information for configuration, test, and evaluation (cf. [BKR09]). A model comprises, in this context, instances which have attributes with primitive and enumeration values, and references pointing to other instances. The structure of instances of a model are defined by classes, their property declarations and data types (cf. [MOF15, p. 26ff]).

A *metamodel* is a model used to define how a model can be constructed [SV06, sec 6.1]. Meta Object Facility (MOF) defines a metamodel as a set of classes with typed properties where the type is either a data type or a class defined in the metamodel [MOF15, p. 26ff]. Guy et al. [GCD+12] interpret metamodels as the types of models. they argue that *MOF classes are closer to types than to object classes, thus a model type is closely related to metamodels [GCD+12].*

In this thesis, we use Eclipse Modeling Framework (EMF) [SBP+09], an implementation of Essential Meta Object Facility (EMOF), as modeling framework. In EMF, properties with a data type are called attributes, while class-typed properties are called *references*. Both kinds of properties can have

minimum and maximum cardinalities. In addition, a reference can represent an unidirectional association with a cardinality of one, an aggregation or a containment, and it can be declared the opposite of another reference [SBP+09, loc. 2373].

A *unidirectional association* (*source* → *target*) and *aggregation* (◇→) express relationships between two or more instances on the model level. However, each instance is independent from the other. In the supermarket example, a simple unidirectional association exists between the class Item and the class Product, expressing that the item refers to a product, but if the item is deleted the product must remain. An aggregation is defined between ProductGroup and Product, where each product group may consist of multiple products. In addition an *opposite* reference is also specified for the relationship between Product and ProductGroup, denoted as group in Figure 3.1.

In contrast, *containment* (◆→) defines that the referenced instance is part of the referring instance. In the above example, each item belongs to a shopping cart. If the cart is deleted so must be all items, as they are only part of the shopping cart.

Beside the semantic property of containment, the containment hierarchy of a model is used in model serialization to determine the sequence in which the model elements are serialized. Therefore, a model is considered rooted when each instance is reachable by only one path on the containment graph.

3.1.2 Formalizing EMF Models and Metamodels

The previous section introduced the terminology for models and metamodels on an informal level. For a deeper understanding of models and metamodels, we provide a summary of the formal foundation of EMF based on a graph formalism, starting with a basic graph notation [EEP+06; BET12] and a type graph formalism [BET12]. Based on these two, we introduce an EMF model graph formalism which is used in the remainder of this thesis [BET12].

In modeling classes and instances can be represented by nodes, and references between classes and instances are expressed by edges. Simple graph notations, such as $G = (N, E)$ where $E \subseteq N \times N$ is a set of tuples over

N, do not allow to define parallel edges, as they are only defined by their connecting nodes. Therefore, in [BET12] a graph definition is given which represents edges as first class elements and connects edges with nodes by two functions.

Definition 3.1 (Graph [BET12])
A graph $G = (N_G, E_G, s_G, t_G)$ consists of a set N_G of nodes, a set of E_G of edges, as well as source and target functions $s_G, t_G : E_G \to N_G$.

This definition does not explicitly forbid hypergraphs, meaning edges with more then one target node could be formulated. To limit the definition to graphs, we add the following constraint: $\forall e \in E_G$ there is only one $(e, n_s) \in s_G$ and $(e, n_t) \in t_G$.

Based on Definition 3.1, Biermann et al. [BET12] define a graph morphism. Graph morphisms are utilized in this thesis for the mapping of types to instances and in algebraic graph transformations which are discussed in Section 3.2.

Definition 3.2 (Graph morphism [BET12])
Given two graphs G and H, a pair of functions (f_N, f_E) with $f_N : N_G \to N_H$ and $f_E : E_G \to E_H$ forms a *graph morphism* $f : G \to H$ if it has the following properties:

1. $\forall e \in E_G : f_N(s_G(e)) = s_H(f_E(e))$ and
2. $\forall e \in E_G : f_N(t_G(e)) = t_H(f_E(e))$.

If $N_G \subseteq N_H$ and $E_G \subseteq E_H$, then f_N are f_E are inclusions and G is called a subgraph of H, denoted by $G \subseteq H$.

Type Graph Containment, inheritance and opposite references are important features of EMF which must be reflected in the type graph. Biermann et al. [BET12] provide the following type graph definition tailored for EMF.

Definition 3.3 (Type graph with inheritance and containment [BET12])
A type graph $TG = (T, I, A, C, OE)$ consists of

▷ a graph $T = (N_T, E_T, s_T, t_T)$,

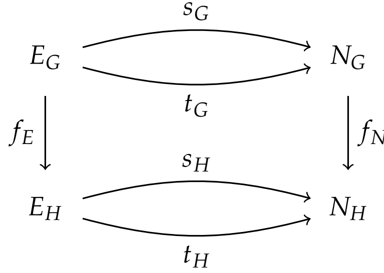


Figure 3.2. The functions of the graph morphism (f_E, f_N), depicted together with the edge and node sets E_G, E_H and N_G, N_H, and their source and target functions (s_G, t_G, s_H, t_H), of the two graphs G and H

- a relation $I \subseteq N_T \times N_T$, called *inheritance*,
- a set $A \subseteq N_T$ of *abstract nodes*,
- a set $C \subseteq E_T$ of *containment edges*, and
- a relation $OE \subseteq E_T \times E_T$ of opposite edges.

The graph T comprises all nodes and edges of the type graph, denoted as N_T and E_T, respectively. The functions s_T and t_T are the source and target functions, defined as $s_T, t_T : E_T \rightarrow N_T$, to express the relationship of nodes and edges.

Inheritance Based on the type graph TG, inheritance is defined as a binary partial relation I which is reflexive, anti-symmetric, and transitive [BET12], conforming to the subtyping rules from Section 2.4, i.e., for all l, m, and $n \in N_T$ the following rules apply:

- reflexivity: $(n, n) \in I$ as typing rule $n <: n$
- anti-symmetry: if (m, n) and $(n, m) \in I$ then $m = n$
- transitivity: if (l, m) and $(m, n) \in I$ then $(l, n) \in I$, as typing rule $\frac{l <: m \quad m <: n}{l <: n}$ (cf. subtyping in Section 2.4)

The graph $I = (N_T, E_I)$ represents the inheritance graph, conforming to these constraints. Its nodes N_T represent types and the edges E_I the inheritance relationship between these types, containing only the transitive reduction of I. E_I is defined by $E_{all} = \{m \rightarrow n | (m, n) \in I, m \neq n\}$ as a set of all edges in I without reflexive edges, then $E_I = E_{all} \backslash \{m \rightarrow n | \exists \text{ path } m \xrightarrow{+} l \xrightarrow{+} n\}$ [BET12].

Finally, the *inherits from* relation $m <: n$ from Section 2.4 is expressed in the context of type graphs by the *inheritance clan* as $m \in clan_I(n)$ with $clan_I(n) = \{m | (m, n) \in I\}$ covering all type nodes which can be reached from n following the inheritance edges. In the example from Figure 3.1, there is one inheritance relationship BulkStock $<:$ Item.

Containment The containment relation $contains_{TG} \subseteq N_T \times N_T$ is defined over the set of types (N_T) and the containment edges (E_T) of the type graph. A type $n \in N_T$ contains a type $m \in N_T$ when there exists a containment edge between them. As denoted in the following equation, this does not only apply if there exists a direct edge between n and m. This is also valid when n and m are subtypes of such a type.

In the example in Figure 3.1, the class BulkStock is a specialized Item which allows to specify the length of a bulk stock item with a float value. While the target function t_T will return Item as end of the containment relation, the BulkStock is also contained by this relation. This is expressed by the relation $<:$ instead of $=$.

The containment is also transitive. In the above equation this is expressed by the recursive definition when x contains y and y contains z then x contains also z [BET12].

$$\begin{aligned} contains_{TG} \quad = \quad & \{(n, m) | \exists c \in C : n <: s_T(c) \wedge m <: t_T(c)\} \cup \\ & \{(x, y) | \exists y \in N_T : (x \ contains_{TG} \ y \wedge y \ contains_{TG} \ z)\} \end{aligned}$$

For model serialization, all instances of types must be in a containment graph which has only one root source node. In EMF, references pointing to

another contained graph, are allowed as long as they are not part of the containment graph of the type graph. The example metamodel depicted in Figure 3.1, includes two separate containment graphs with Inventory and Cart as their respective root nodes. Both graphs are linked by the reference product between Item and Product.

Opposite The last feature of EMF to be covered, is the concept of opposite references. In Definition 3.3, OE is the relation of opposite edges of the type graph TG. A relation must obey the four axioms, opposite direction, anti-reflexivity, symmetry and functionality to realize this EMF feature [BET12]:

- ▻ Opposite direction: $\forall(e_1, e_2) \in OE : s_T(e_1) = t_T(e_2) \wedge s_T(e_2) = t_T(e_1)$
- ▻ Anti-reflexivity: An edge cannot be its own opposite edge $\forall e \in E_T : (e, e) \notin OE$
- ▻ Symmetry: For every pair there is an opposite pair $\forall(e_1, e_2) \in OE : (e_2, e_1) \in OE$
- ▻ Functionality: There can only be one opposite edge for each edge. For $(e_1, e_2), (e_1, e_3) \in OE : e_2 = e_3$

In the example from Figure 3.1, the aggregation products from ProductGroup towards Product has an opposite association called group.

EMF-model graph Based on the previous definition of graph and type-graph, Biermann et al. present an EMF-model graph formalism [BET12] comprising of a type graph, a graph representing the instances and a typing morphism to relate instances to types.

For better understanding, Figure 3.3 presents an small instance model based on the metamodel depicted in Figure 3.1.

The Figure 3.3 illustrates a model with two containment graphs. One graph covers a shopping cart and the items belonging to the cart. And the second covers an small inventory with one product group and two products. Beside containment, the model illustrates an aggregation (group) and opposite references pointing from each product to the product group.

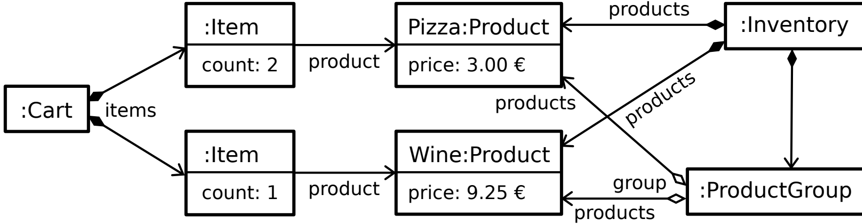


Figure 3.3. Example of an instance model for the exemplary metamodel from Figure 3.1

In the following these features of an EMF model are defined based on the introduced graph and type graph formalism.

Definition 3.4 (EMF-model graph [BET12])
Given a type graph $TG = (T, I, A, C, OE)$, an instance graph G and a morphism $type_G : G \rightarrow TG$. Then G is an EMF-model graph over TG if the conditions for correct typing, containment, opposite edges and the avoidance of parallel edges are fulfilled.

Definition 3.4.1 (Typing morphism)
The typing morphism to ensure correct typing $type_G : G \rightarrow TG$ consists of a pair of functions $type_{N_G} : N_G \rightarrow N_T$ and $type_{E_G} : E_G \rightarrow E_T$ with

▻ $type_{N_G} \circ s_G(e) <: s_T \circ type_{E_G}(e)$ and

▻ $type_{N_G} \circ t_G(e) <: t_T \circ type_{E_G}(e)$.

To complement typing and semantics of the model graph G, containment constrains must be fulfilled. In EMF each instance can only have one container, otherwise the containment hierarchy would be broken which would hinder serialization. Therefore, containment cycles are not allowed on model level in contrast to the metamodel level expressed in the type graph [BET12].

Definition 3.4.2 (Model level containment edges)
The set of containment edges C_G is determined by the type graph containment set C.

$$
\begin{aligned}
C_G &= \{\forall e \in E_G | type_G(e) \in C\} \\
contains_G &= \{(s_G(e), t_G(e)) | \forall e \in C_G\} \cup \\
&\quad \{(X, y) | \exists x \in N_G : (x\ contains_G\ y) \wedge (y\ contains_G\ z)\}
\end{aligned}
$$

A *rooted* model must not contain cyclic containment paths, i.e. $\forall x \in N_G : (x, x) \notin contains_G$. And there must be a *root node* $r \in N_G$ such that $\forall x \in N_G$ with $x \neq r : r\ contains_G\ x$.

In context of the example, this is only true if the example model is partitioned accordingly. Therefore, an alternate definition for a rooted model partition is: Every $r \in N_G : \forall x \in N_G(x, r) \notin contains_G$ is a root node.

For the type graph, opposite edges are defined as a pair of two type graph edges with opposite direction, i.e., $s_T(e_1) = t_T(e_2) \wedge t_T(e_1) = s_T(e_2)$. Based on that constraint, Biermann et al. [BET12] define opposite edges for the model graph, relating these edges to type graph edges.

Definition 3.4.3 (Opposite relation in model graph [BET12])
If $(e_1, e_2) \in OE$, then $\forall e_G \in E_G$ with $type_G(e_G) = e_1$ there is also a model graph edge $e'_G \in E_G$ with $type_G(e'_G) = e_2$ where the source of one edge is the target of the other: $s_G(e_G) = t_G(e'_G)$ and $s_G(e'_G) = t_G(e_G)$.

Finally, a model graph must not have parallel edges referring to the same edge in the type graph. This still allows for any cardinality of an type graph edge, as in that context the model graph's target function value is different.

Definition 3.4.4 (No parallel edges [BET12])
$\forall e_1, e_2 \in E_G$ with $s_G(e_1) = s_G(e_2)$, $t_G(e_1) = t_G(e_2)$, and $type_{E_G}(e_1) = type_{E_G}(e_2)$ we have $e_1 = e_2$.

Based on the previous definitions, arbitrary EMF models can be described and typed correctly. In contrast to an EMF metamodel TG itself is not an EMF model. However, for the purpose of this thesis the formalism of Biermann et al. [BET12] is sufficient to support formal grounds for modeling and

transformations.

3.2 Model Transformation

The central element of code generators are transformations. These transformations are used to collect information, process them and create and modify model elements. Based on two taxonomies on model transformations [MG06; Bie10], an introduction to basic concepts and characteristics of transformations is presented in this section.

3.2.1 Technical Characteristics

Transformation languages and technology have to respect a wide range of technical characteristics which affect the way how transformations are constructed and how their input and output is handled.

In context of the modeling framework, today most prominently are XML and XML Schema, as modeling environment in the context of web technologies, and EMOF-based Eclipse Modeling Framework (EMF) [MG06], which is widely used in modeling and code generation. For both technologies a wide range of query, and processing tools and languages have been created, e.g., Extensible Stylesheet Language Transformations (XSLT) for XML and ATLAS Transformation Language (ATL) for EMF. Depending on these technologies different ways to serialize and store information have been developed which can largely be categorized in textual representation which is often used for code generation, XML serialization, as a special text format [Bie10], and databases, e.g., CDO [Ste12] and Neo4EMF [BGS+14].

Mens et al. [MG06] categorize transformations by their complexity. They consider model refactoring as considerable small, while compilers and code generators can be very complex. In context of this thesis, code generators are considered amalgamations of transformations.

In general transformations can have multiple input and output models which conform to the same or different metamodels. In the special case of in-place transformation, the transformation requires only one input model. In code generation there is at least one input and one output model [MG06],

called source and target model respectively. The metamodels of source and target model, are therefore, called source metamodel and target metamodel.

In MDE models are changed and subsequently model transformations are triggered to update the target models. If the model update is realized by replacing parts of the old target model or when a target model is constructed by multiple transformations or transformation runs, then it is an incremental transformation. Otherwise when the complete target model must be reconstructed it is called non-incremental [MG06].

3.2.2 Syntactical and Semantical Transformations

Transformations can be distinguished in syntactical and semantical transformations. For example, the transformation from the Concrete Syntax Tree (CST) into an Abstract Syntax Tree (AST) is a simple syntactical transformation. Based on the AST a subsequent transformation can generate code, e.g., transform a declarative model into machine code [MG06].

3.2.3 Endogenous and Exogenous Transformations

The source and target model of a transformation may conform to the same metamodel, e.g., for optimization, refactoring, simplification and normalization. In that case the transformation is called endogenous [MG06]. Refactoring is often realized as an in-place transformation, since the source and target model are the same.

In contrast, exogenous transformation are based on different source and target metamodels. Exogenous transformations are used in code generation where the code is on a lower abstraction level, in reverse engineering, where the input is used to construct an abstract representation of the software, and in migration, where code in one language is translated into another one [MG06].

3.2.4 Horizontal versus Vertical Transformations

An orthogonal view on transformations is the distinction in horizontal and vertical transformations. In a horizontal transformation, the source and the

target model are on the same abstraction level. Refactoring and language migration are typical horizontal transformation [MG06].

A vertical transformation transforms a model into a model on a different level of abstraction. For example, code generation is usually a transformation form a higher level of abstraction to a lower more concrete level. For reverse engineering, the level of abstraction increases between source and target model [Bie10]. However, this change in abstraction does not necessarily involve a different metamodel. For example, in refinement, a coarse specification model can be refined with multiple steps into a complete implementation [MG06].

3.2.5 Transformation Language Paradigms

A wide range of model transformation languages have been developed, which have different types of paradigms. Imperative and operational languages allow to specify the transformation in a sequence of actions and functions. A developer has full control over the sequence of execution in these languages [Bie10]. Operational and imperative languages often originate from a more technical background, such as `Xtend` [Ite11].

Declarative, relational and graph transformation approaches originate from a research and mathematical based background. They describe transformations in rules and by relationships. Specifying what should be mapped by the transformation. Therefore, the developer has less control over the execution sequence. Declarative transformations are often considered more compact and concise compared to operational transformations [Bie10].

Graph transformation languages based on algebraic graph grammars describe transformations based on rules and sometimes use graphical notations, e.g., [ABJ+10]. So called triple graph grammars are composed of three graphs, a left-hand side graph, a right-hand side graph, and a correspondence graph. The left-hand side graph is a subgraph pattern to match the source graph, the right-hand side graph is a subgraph pattern of the target graph, and the correspondence graph describes the mapping between elements of both sides [Bie10].

Finally, template based languages are used to interpret models and generate source code. Such templates often represent a syntactical trans-

formation, as the abstract structure has been generated in advance by a model-to-model transformation [Bie10].

3.3 Domain-Specific Languages

The construction of models to specify the different parts of a software system are a central element of MDE. In Section 3.1, we introduced models and how they are defined based on metamodels. However, metamodels alone provide only an abstract structure for models. The construction and modification of models, requires a notation and tooling for developers. Furthermore, models must be persistable and transformable to realize software systems. Therefore, software is required to complement metamodels and provide developers with the necessary tools to work with models.

The notation, called Domain-Specific Languages (DSLs), can be graphical, textual, or otherwise structured. DSLs have a specific grammar and are designed with a particular domain in mind [MHS05; Bet13]. They correspond directly or indirectly with metamodels, which comprise of classes representing domain concepts. Textual DSLs are usually serialized in textual form [Bet13], whereas graphical and structured DSLs use some sort of model serialization, like XML and XML Metadata Interchange (XMI).

For this thesis, we like to provide a small overview on the topic of DSLs. We introduce a general approach to DSL development Section 3.3.1. As DSLs are computing languages we provide a short introduction into compiler construction in Section 3.3.2. Finally, we discuss the implementation of DSLs in Section 3.3.3.

3.3.1 DSL Development Approach

Domain-Specific Languages (DSLs) are languages specifically tailored to one domain, view, or aspect of a system, like web pages and workflows. In context of this thesis, DSLs are used to specify software systems.

The development of DSLs is a complex task which can be costly and time consuming. Mernik et al. [MHS05] splits the DSL development in five distinct phases:

Decision Before developing a DSL the expected cost and benefits must be considered, as the development of a DSL requires some effort. Furthermore, it must be considered which kind of DSL should be constructed. In general DSLs are categorized in internal and external DSLs. Internal DSLs are in fact libraries or frameworks which are implemented in the host language. For example, a drawing library for Java could be considered an internal DSL to express graphics with Java. In this thesis we use external DSLs. These DSLs define their own language primitives, syntax and semantics. These external DSLs can be textual languages which are defined based on grammars [KRV08; Bet13], graphical [SSH13], and arbitrary structures [VP12].

Analysis After deciding to create a DSL and the type of DSL, the language designer must collect domain knowledge to define the concepts of the language. DSLs for existing metamodels can use the domain knowledge collected for these metamodels.

Design In the design phase, domain concepts are mapped to language constructs. Textual DSLs are realized with context free grammars, like LL(*) and LR grammars (cf. [Bet13]). Depending on the DSL framework, DSLs can also inherit language features from other grammars and be embedded in other grammars [KRV08; SSH13].

Graphical and structural DSLs are specified based on graphical or structural representations [VP12; SSH13], which form the syntax of these DSLs. Like textual DSLs, graphical and structural DSLs may inherit features from other DSLs and are embedded in other DSLs.

Apart from the language syntax also the semantics of the language must be specified. Semantics are often expressed in semantic rules and may involve type checking (see Chapter 2).

Implementation The implementation of a DSL is based on the information and artifacts created in the design phase. The implementation includes in many cases an editor to create and edit models, model serialization and deserialization, and a generator to transform the models into code. Textual DSLs often use a character based serialization which allows to edit and process the artifacts like text files, e.g., use version control

systems to handle differences of files. However, graphical and structural DSLs often use structured serializations based on XML and XMI provided by the metamodel framework. These files are, in contrast to text files, hard to read with other tools and often cause problems when processed by version control systems.

Deployment Finally, the DSL is handed over to their destined users. These users, initially evaluate the DSL and may provide feedback to improve the language. their ideas would be used in a second analysis phase to extend the domain and finally result in a new version of the DSL.

The phases suggest that DSL development uses a waterfall development process. However, Mernik et al. [MHS05] does not suggest to use a specific process. Instead they discuss various dependencies between the phases and encourage to use the phases as a means of splitting up a DSL development effort and integrate it in any desired development process model.

3.3.2 General Purpose Languages

General Purpose Languages (GPLs) have different features and are used in various contexts. They implement different programming paradigms, e.g., imperative, functional, and object-oriented. Source code of GPLs is either compiled to target code, e.g., machine code, or interpreted. Like DSLs, there are graphical and textual GPLs. In the following we focus on textual languages, as they are the most widely used kinds of GPLs.

Usually the GPL compilation process is divided into six distinct phases. The lexical, syntactic, and semantic analysis are used to construct a valid abstract syntax graph (see Figure 3.4). Based on this graph the output is generated utilizing multiple transformations. As similar phases are used in DSLs, we shortly introduce them here for textual programming languages:

Lexical Analysis Source code is serialized in a character based text file. These files are read by a scanner which matches character sequences and returns a token stream (see Figure 3.4). These scanners are created based on token rules which are often defined with regular expressions. The scanner definitions are either part of the grammar language, like with

ANother Tool for Language Recognition (ANTLR) and Xtext [Bet13], or separate, like with lex and flex.

Syntactic Analysis The tokens are then processed based on rules. This analysis is often realized based on context-free grammars which are used to generate token stream processors by tools like ANTLR, yacc, and bison. Depending on the type of grammar and tool the syntactic analysis generates an Abstract Syntax Tree (AST) when the input is correct.

Semantic Analysis In the semantic analysis references are resolved and type checking is applied to the AST. While the syntactic analysis may accept `4 + "three"`, depending on the semantics of a language this is accepted as a string concatenation (`"4" + "three"`), a textual numerical which is converted to `4 + 3`, or rejected as an illegal symbol in the expression. The result of this phase is usually an AST with resolved references, which is then often an Abstract Syntax Graph (ASG).

Intermediate Code Based on the ASG intermediate code is generated. This code represents the source code in a form closer to the target language. A well known type of intermediate code is three address code [ASU86].

Optimization Depending on compiler settings, the intermediate code is further processed and optimized to ensure, for example, faster execution or less program size.

Code Generation Finally, the optimized intermediate code is transformed into target code, which is, in most compilers, machine code, also called object code, containing assembler commands and symbols for a linker.

As it would require lots of memory to store all the intermediate models in the compilation, the phases are entangled. Most prominently, the parser performing the syntactic analysis is not reading a token stream, but asks the scanner for the next token. Also most of the semantic analysis can be done continuously during the construction of the AST [ASU86].

3.3.3 DSL Construction

On a conceptual level DSLs are similar to General Purpose Languages (GPLs). They are both considered formal languages. Textual DSLs use grammars, an abstract representation for processed language artifacts, and use one or more stages to create target code. The two terms DSL and GPL are often seen two distinct categories, and any language falls in one of these two categories. However, such clear distinction is often not possible and the two terms rather define a continuum. For example, the Cascading Style Sheets (CSS) language is domain specific and allows to define the style of HTML elements. Therefore, it is a DSL. The C programming language is definitely a GPL, as it can be used to write operating systems and many different applications. However, the Model-Based Design Methods for a new Generation of electronic Railway Control Centers (MENGES) DSL also allows to specify a wide variety of programs, but is otherwise optimized for a specific domain. Therefore, it is mostly a DSL even though its purpose is less limited than CSS [MHS05].

Apart from the similar conceptual design, there are several larger and smaller differences between DSLs and GPLs. These differences concern the variability of the phases, serialization, language definition methods, type systems, and output generation.

Generator phases As Figure 3.4 shows, DSLs do not necessarily use all phases of a classical compiler. For example, some code generators produce the target model and code directly from the model. Through such shortcut, it is assumed that development of the DSL lightweight and can be accomplished with fewer resources.

Another major difference between GPLs and DSLs originates from the purpose of DSLs. They are designed for one domain and purpose. For example, there are DSLs for data types, graphical representations, configuration of software systems, and deployment descriptors. However, GPLs are designed to implement any kind of application regardless of their domain.

DSLs designed for a specific domain of a customer, are subject to short term alterations in syntax and semantics, as the domain changes over time [MHS05]. Therefore, evolution steps appear in shorter intervals and with

less planning and design time available. For example, the syntax and semantics of the MENGES DSL [GHH+12] changed on weekly and monthly basis. Also the IRL of the Kieker project has language updates to support new features regularly, and already provided two revisions since its initial introduction in 2013 [JHS13]. However, the C programming language released its initial version in 1973 [Rit93] and the first International Organization of Standardization (ISO) version of C was published in 1990 [Rit93], followed by the next two updates in 1999 [ISOC+99] and 2011 [ISO12].

Serialization Textual DSLs often use textual serialization for their artifacts. However, graphical and structured DSLs use alternative serializations for their models, like XML and XMI (see Figure 3.4). For example, graphical editors constructed with the KIELER framework [SSH13] can use the standard EMF serialization based on XMI. Similarly, Jetbrains/MPS, which supports the construction of structured artifacts, uses an XML serialization [VP12].

Language definition Textual DSLs are defined with context-free grammars. Xtext [Bet13] and MontiCore [KRV08] use Formal grammar which is parsed from left to right using the left most derivation (LL)(*) grammars, as they use ANTLR [Par07] as their internal parser generator. The Spoofax language framework [KV10] uses Syntax Definition Formalism (SDF) to specify grammars. In contrast, programming languages, like C, are often specified with Formal grammar which is parsed from left to right using the right most derivation (LR) grammars implemented with Lex [LS90] and Yacc [Joh79] or Flex [PEM12] and Bison [DS12] combination to generate code for lexical and syntactic analysis. LR grammars form a larger subclass of context-free grammars than LL grammars, and they are able to handle left-recursive grammars. However, parsers for LL grammars are simpler and recovery from errors can be achieved with less effort according to Xtext developers [Bet13]. Kats et al. [KV10] justify their choice of SDF with the greater flexibility this formalism has over LL and LR grammars.

The approach represented by Jetbrains/MPS [VP12] does not use grammars at all. MPS/Jetbrains defines the model always in an Abstract Syntax Tree (AST) and the presentation on screen is rendered automatically. This

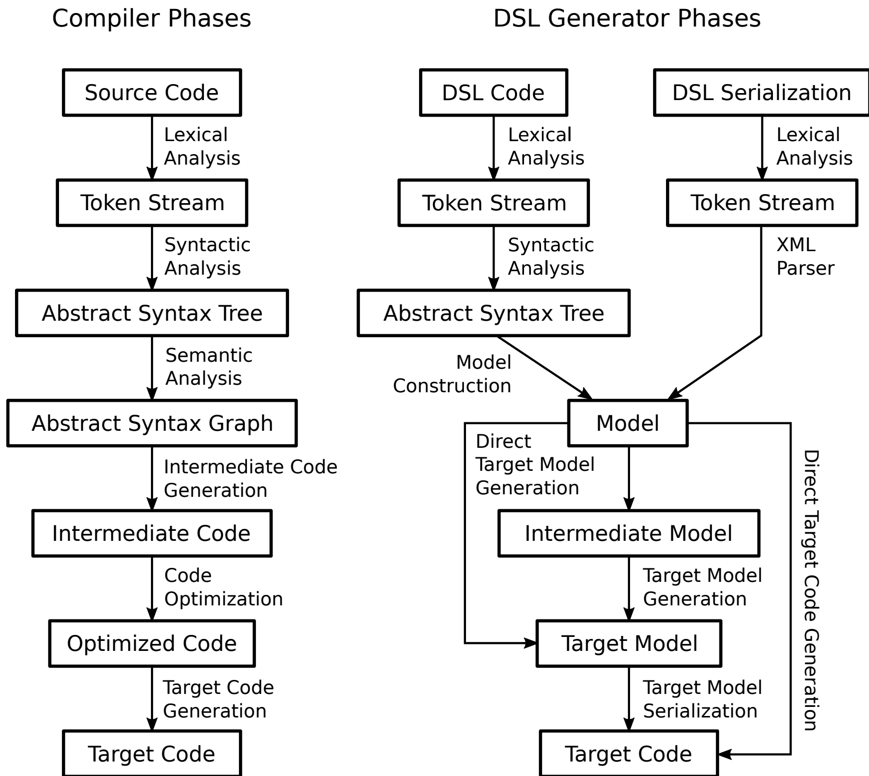


Figure 3.4. Comparison of compiler phases and DSL code processing (cf. [ASU86; MHS05])

has the advantage that new languages can be embedded in a main language without special markers where to switch grammar rules. Furthermore, this allows to mix textual, graphical, table and list oriented representations in one document. A downside of this approach appears when specifying textual languages. It originates from using the AST as the central model representation. In a grammar-based DSL, the artifact is a character sequence, which is parsed to compute syntax-highlighting and the model for the generator. During editing it can happen that the character sequence cannot

be parsed properly. However, this is not an issue, as long as a proper state is reached eventually. In Jetbrains/MPS these improper states cannot be represented in the AST. Therefore, the metamodel and construction rules for the DSL must be extended to support improper states [VP12].

Type systems One key argument for DSLs is that they are lightweight and do not need all the elements and formalisms of programming languages. As DSLs can be very simple, like configuration languages, they might not use a formalized type system. However, sometimes it is helpful to formally model or import tying rules. For Xtext, the Xbase framework [EEK+12] provides the Java type system for Xtext DSLs. Alternatively, Xsemantics [Bet11] can be used to define typing and semantic rules. In Spoofax, type systems can be expressed a Type Specification Language [Pro] which also uses a rule-based approach. Both approaches are largely founded on the formalisms described in Types and Programming Languages [Pie02] which are summarized in Chapter 2.

Code and model generation As depicted in Figure 3.4, there are different paths to generate target code for DSLs. This is also true to some extend for GPL compilers, as code optimization can be switched off in some compilers, whereas others do not have an optimization stage. However, DSL generators can be realized with simple model-to-text transformations, in which case the model is applied to code templates. Templating languages, like Xtend [Ite11], allow to insert variables into templates, which are computed based on the model. For more complex DSLs, such direct code generation is not an option. They use, therefore, multiple chained transformations [VAB06]. For example, such chaining can be used when the source metamodel paradigm is very different from the target metamodel, like mapping object-oriented concepts on an imperative language.

For target models, which are serialized by framework functionality, a generator only produces these models, and triggers the serialization by invoking framework functions. Transformation languages, like Query/View/-Transformation (QVT) [QVT05] and ATL [JK06], are designed to implement transformations which produce target models. Therefore, they use for the final step to target code a serializer, like the XMI serializer from the EMF.

3.4 Model Traceability

Model traceability plays an important role in different fields of MDE. For example, engineers would like to know how certain implementation artifacts are related to design models, and how design models are derived from use cases and other information gathered during requirements engineering. Given that not in every project these relationships are collected and adapted when necessary, approaches have been developed to recover and reverse engineer these relationships. While approaches have been developed for trace recovery [GK10], they suffer one problem: The relationship between source and target model nodes are based on heuristics or name equivalency and may be wrong. In context of requirements engineering such incorrect relationships can easily be identified and removed by hand. In GECO, the relationship must be correct, as they are used in transformations. Therefore, approaches providing a heuristic or name based reconstruction of traces are not viable for GECO. Thus, only traceability approaches designed to be used in transformations are discussed in the following sections.

Galvão et al. [GG07] compiled a survey on traceability approaches and provided a categorization of them. As GECO requires a model-based approach which provides unambiguous results, we selected two categories of trace modeling approaches, which can be used in a model transformation context. The first category are modeling approaches, which focus on how metamodels, models and conceptual frameworks are involved in the trace model generation. And the second category are transformation approaches which focus on the mechanism how traceability can be realized in the transformation process.

3.4.1 Modeling Approaches

The modeling approaches focus on how traces have to be modeled and expressed to cover the wide range of purposes traces are used for, and which actions must be performed when one node is created, updated and deleted. All approaches omit the read operation, as they do not affect the syntax and semantic of a model and, therefore, do not require an action in a derived model.

Aizenbud-Reshef, Paige, et al. [APR+05] describes traces as a set of source and target nodes with a relation between them implying no hierarchy on the mapping itself. As a modeling approach, it focuses on events which originate in the source model. The approach defines specific semantics on a triple comprising event, condition, and action. Events are create, update and delete operation on source model nodes. For example, when a source node is changed, the derived target must be adapted accordingly. However, the correct action might vary between different update events. Therefore, an additional condition can be specified to limit which action can be applied to an event. For example, an update to the return type of a method requires the update of all method invocations, while an update to the body does not require any changes.

The second approach [ES05], discusses an unifying scheme for traceability with a focus on the relationship among requirements, design artifacts, and code. Their traces are more coarse-grained, as they link complete artifacts and not nodes of a model. Their traces are not morphisms between two sets or even graphs. Instead each trace contains information on the type of a trace, e.g., code *is derived from* model, scenario *describes* requirements, the purpose for its existence, and which artifacts also relate to this trace. This approach even allows to group traces and defines sub-traces realizing a hierarchical organization of traces.

While the first approach defines the semantics of traces based on model modifications, and the second approach covers informal and formal artifacts, the third model approach focuses on formal models and transformations as users of trace models.

Vanhooff and Berbers [VB05] constructed an UML-profile which allows to describe trace information of model changes caused by incremental transformations. Their trace model distinguishes between different kinds of traces to cover the creation of new nodes, their replacement or modification, and the utilization of other nodes. The first kind stores which transformation created a node, while the other two refer to nodes in a previous model version. Due to the profile nature, the trace information is stored in the model itself and not kept separate, like in the other approaches.

In GECO, the source model is static during the generation process and the target model is newly created. Therefore, no actions must be defined to

keep the target model up to date. Furthermore, GECO works only on formal models and the trace models are only used internally, therefore, textual documentation of traces is not useful. However, a hierarchical relationship of target model nodes can be beneficial. From the modeling approaches, the last [VB05] is most interesting in the context of GECO, as it also describes the use of traces as input and output of transformations. However, the use of profiles implies access to the metamodel and its implementation, which would limit the ability for reuse and for independent development of different transformations.

3.4.2 Transformation Approaches

The modeling approach of Vanhooff and Berbers [VB05] already points in the direction of trace generation in and by a transformation. However, it does not provide a detailed method on how this integration can be achieved. In this section, the focus is on approaches where the transformation constructs the traces. We present two approaches which store the trace model in a separate model with its own metamodel.

The first approach augments ATL transformations [JK06] to produce a trace model relating source model nodes to target model nodes [Jou05]. The metamodel for the trace model comprises a simple trace class with an attribute *ruleName* referencing the rule which caused the relationship, and two sets of nodes. In the context of EMF the nodes are modeled as `EObject`.

The key idea in this approach is that traceability is a cross-cutting concern which must be introduced in every ATL rule. As this pollutes the transformation code, Jouault [Jou05] proposes a tool called *TraceAdder* which weaves traceability code in to the ATL transformation. Therefore, it is considered a higher order transformation [TJF+09].

As GECO on one hand, requires trace models to realize the decomposition of generators, and on the other hand fosters the modularization of transformations, this approach provides technically a good solution for ATL transformations to introduce traceability, and it provides the general idea on how to introduce traceability in transformations written in other languages.

A similar approach, including a traceability framework, has been proposed by Falleri et al. [FHN06]. Inspired by TraceAdder [Jou05], they define

a traceability framework for facilitating the trace of model transformations realized with the model-oriented language Kermeta [JBF11]. Their metamodel for trace models defines a trace as a bipartite graph on which nodes are source nodes and target nodes. Furthermore, they assume that transformations are considerable small and can be chained together. Therefore, they extend the metamodel Jouault [Jou05] by an additional class called Step which relates a trace to a single execution of a transformation. They also provide a visualization for their trace model. However, they do not provide an TraceAdder-like higher order transformation to extend Kermeta [JBF11] based transformation automatically.

In GECO trace models are only used inside of generators. Therefore, visualizations might only serve a purpose when debugging compositions. However, the concept of chaining trace models can be useful in generators, e.g., when two transformations generate the same type of intermediate model as result from different types of source models, and a third transformation generates a target model from these intermediate models, then simple traces will not work and chaining is required.

3.5 Model Weaving

Model weaving is defined by Fabro, Bézivin, Jouault, et al. [FBJ+05] as a correspondence between two models which can be used by a transformation to create a combined model, called *woven model*. Morin, Klein, et al. [MKB+08] define weaving as the action of weaving model elements of one model into another. The second definition originates from the AOM domain. In analogy to aspect-oriented programming, it defines a base and an aspect model, and a set of references (*join points*) to base model nodes identifying where an aspect can be inserted into the base model.

Join points can be defined explicitly or by pointcuts, whereas a pointcut is a query over the base model identifying nodes and a reference to an advice, which is a part of the aspect model able to be inserted into the base model. The result of the query are join points. In the first definition by Fabro, Bézivin, Jouault, et al. [FBJ+05], these join points form the correspondence between both models which they call weaving model.

In this thesis, we use the distinction of base and aspect model in conjunction with join points and pointcuts, as they are widely used in the research community and especially in the area of aspect-oriented and view-based modeling [KK07; MKB+08; MVL+08; KAK09; KK11; SAM+14], while the weaving model idea is mainly used in conjunction with the ATLAS model weaver [FBV06].

Model weavers are a special kind of model transformation which comprise of at least two kinds of source models and one kind of target model. The source models are the base model and the aspect model, whereas the aspect model can sometimes be split up into an advice model, representing the part to be inserted, and the join point or pointcut model. As a pointcut is a query over the source base model, its metamodel intersects with the metamodel of the source base model. Similarly, the advice model formulates a partial model which can be inserted into the base model, therefore, the metamodel of the advice intersects also with the base metamodel. Depending on the weaver realization, these metamodels are derived by hand or automatically.

Two prominent weaver realization, are the ATLAS Model Weaver (AMW) [FBV06] and the Generic Composer (GeKo) [MKB+08; Kra12] which are introduced in the following two sections.

3.5.1 ATLAS Model Weaver

The AMW is, despite its name, designed for different use cases of model manipulation and processing, including traceability, metamodel comparison and model matching, where model weaving of base and aspect model is only one use case.

In AMW, model weaving is seen as a special case of traceability, where the correspondence between advice and base model are seen as a collection of model traces. Fabro, Bézivin, and Valduriez [FBV06] call this collection a weaving model which is in fact a join point model. The AMW approach is metamodel agnostic because it is possible to apply it to any arbitrary base and aspect metamodel.

However, the rudimentary weaving metamodel (see Figure 3.5) comprises only abstract types providing the correspondence structures which

must be extended to be applicable for weaving real models. The extended weaving metamodel is realized with Kermeta [JBF11], an EMF compatible metamodel framework.

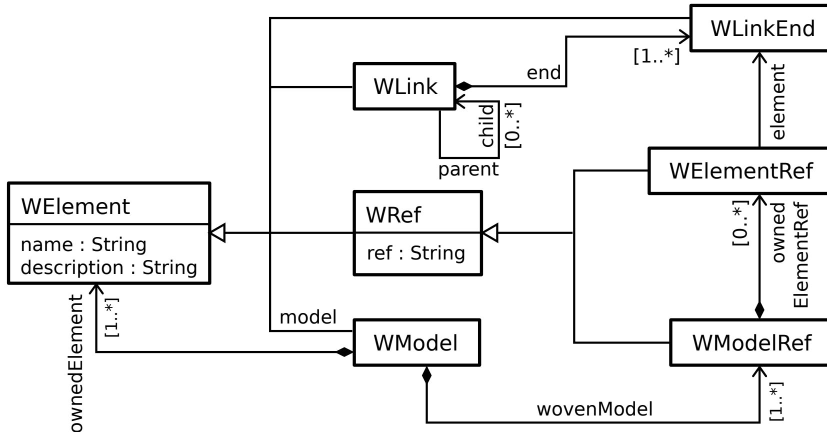


Figure 3.5. The basic AMW weaving metamodel [FBV06]

The weaving metamodel comprises one class derived from WModel, a set of WLink derivatives defining subsets over the end relationship, and specific WLinkEnd subtypes. The WModel class defines at least two subsets over wovenModel, e.g., baseModel and adviceModel, Finally, a weaving operation must be defined.

Based on the specialized weaving metamodel, a weaving model can be created. Fabro, Bézivin, and Valduriez [FBV06] discuss three different types of weaving model construction including heuristics, word matching approaches, and structural or subgraph based matching approaches. For GECO only the last are appropriate (cf. Section 3.4) and is realized in ATL. Based on the generated weaving model, the advice model is woven into the base model utilizing the previously defined weaving operation.

While AMW is a flexible model-processing tool based on Kermeta and the ATL, it requires human intervention to construct the weaving metamodel and programming effort to express pointcut equivalence used to realize a weaving model. Both tasks must be redone when a metamodel changes.

3.5.2 Generic Composer

Morin, Klein, et al. [MKB+08] first proposed the Generic Composer (GeKo) weaving approach based on Prolog. Kramer [Kra12] provided a new implementation utilizing the rule engine Drools:[1] and a detailed description. Like AMW, it is a metamodel agnostic approach. In contrast to AMW, it is able to create the pointcut and advice metamodel automatically.

The metamodels for pointcut and advice are in general less restrictive than the original metamodel of the base model, for example, multiplicity constraints are removed to allow incomplete partial models. Due to the derived nature of these pointcut and advice metamodels, models conforming to them are better understandable to the user. Furthermore, tooling used for the base model composition can be adapted to be used for pointcut and advice as well.

The overall process of weaving with GeKo comprises four phases which can be represented by different smaller transformations, as depicted in Figure 3.6. First, based on the pointcuts the base model is queried and matching join points are computed by the $T_{JoinPoint}$ transformation. Second, the $T_{Mapping}$ transformation generates a relation of pointcut to advice mappings. Third, based on the base model, the join point model, the mapping model, and advice model, the central $T_{Composition}$ transformation produces a woven model. And finally, the $T_{Cleanup}$ transformation removes all unwanted parts of the model [Kra12, p. 23].

The original implementation of GeKo [MKB+08] is based on Prolog. However, the current implementation [Kra12] is realized with EMF and Drools, two widely adopted technologies. Therefore, it can be integrated in many different modeling projects working together with other tooling. In addition, it is designed solely with aspect-oriented modeling in mind which allowed more automation compared to AMW. In the context of the herein presented generator composition approach, GeKo is better suited for the weaving task and the realization of the weaver, if complex weaving is required. In context of simple insert operation, hand written weavers are cheaper and faster to realize and therefore preferable.

[1] Business logic rule engine Drools `http://jboss.org/drools`

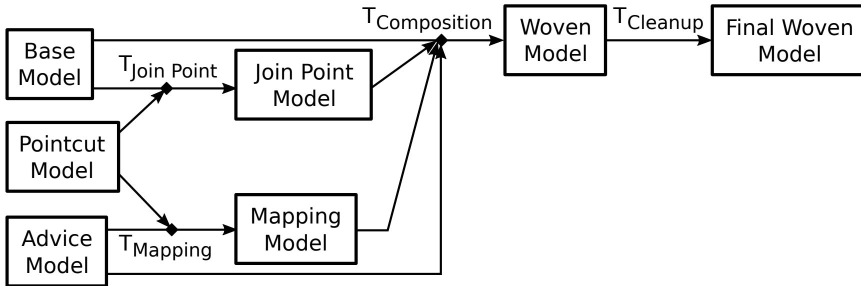


Figure 3.6. Generic Composer depicted as a megamodel of models and transformations (cf. [Kra12])

3.6 Megamodels

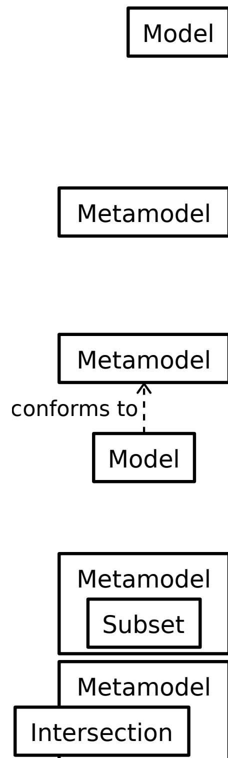
Model-driven development involves models, metamodels, and any number of transformations to produce models and code to implement a software system. While there exist notations for models and metamodels, e.g., UML [UML15] and MOF [MOF15], a common notation to describe the relationships between models, metamodels, and transformations is in its infancies. Favre [Fav04a] proposed the term *megamodel*, as a model which describes these relationships.

3.6.1 Basic Megamodel Notation

In [VJB+13], a megamodel is a terminal model, like transformation models, weaving models, or any other instance models, which conform to a metamodel. Depending on the purpose, the metamodel for megamodels defines different elements, which can be divided into relationships and models [VJB+13]. In the context of this thesis, the conceptual framework of different types of elements from Vignaga et al. [VJB+13] is sufficient with two minor additions. First, for some transformations only a portion of a metamodel is used on the source and target side of the transformation. And second, metamodels and models grouped to express that they have a close relationship. However, based on the introduction of modeling in Section 3.1,

a partial metamodel must fulfill the containment constraint and so must partial models.

While there are many different ways to express the elements and relationships of megamodels, e.g., Favre [Fav04a] used Z, Prolog and UML, there is no general accepted notation available. However, different textual[2] and graphical [Fav04b] notations have been proposed. In this thesis, the following terms and graphical representations are used for megamodels:



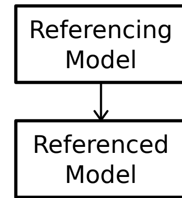
Model A model or partial model is a set of instances with various relationships between them. It is depicted as an rectangle with the models name in it. In a generic context the name is Model or M.

Metamodel Like any other model, metamodel is drawn as a rectangle with its name inside. The main difference between a model and a metamodel is the *conforms to* relationship. In most cases throughout this thesis, the name of metamodels is Metamodel or short MM.

Conforms to The *conforms to* relationship is a surjective relation where the models are the domain and the metamodel represent the range of the relation, e.g., (M, MM) which is normally written as $M \xrightarrow{\text{conforms to}} MM$.

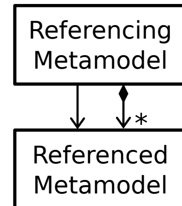
Metamodel Subsets Beside the notation of a partial metamodel which comprises a containment subgraph of the complete metamodel, a metamodel can also be an arbitrary subset of an complete metamodel or two metamodels can share common classes. For example, when constructing advice and pointcut metamodels, which include parts of a base metamodel. On the right the notation for a subset (upper) and an intersection (lower) is depicted.

[2] AtlanMod: http://www.emn.fr/z-info/atlanmod/index.php/Global_Model_Management



Referenced Model A model which is referenced by another model, is called *referenced model* and there exists a reference pointing from the model to the referenced model. It may have it own metamodel disjunct from the metamodel of the referencing model. It may use the same metamodel, or a subset thereof.

Referenced Metamodel Similarly to a model, a metamodel may contain references which are typed by classes of another metamodel. In that case the first metamodel is called *referencing metamodel* and the other *referenced metamodel*. The reference between both metamodels can be a reference (→) without mentioning of any cardinality, which implies there are references from one metamodel to the other without a cardinality constraint.



If the referenced and the referencing metamodel are part of the same metamodel, then the referenced metamodel may represent a subgraph of the containment graph. In that case the reference can be depicted as a containment relationship with a cardinality at the target end. On the illustration on the right, the referencing metamodel contains a class which has a containment reference which allows for an unlimited, but not infinite number of instances (cf. [CTB12]) on the referenced metamodel side.

Based on these primitive notational definitions, we can now express a complete transformation and weaving megamodel. Both in great detail and in a more condensed form, which will be used in the remainder of this thesis.

3.6.2 Transformation Megamodel

In a transformation, a source model is transformed into a target model. The models can conform to the same metamodel or have different metamodels. The target model can also be just a new modified version of the source

model or a new model with other semantics. All these details are explained in Section 3.2. In [FLV12] different generic megamodels describing transformations are depicted originating from various research domains. For this thesis, we use a variant based on the ATL documentation, as it perfectly matches the notation discussed in this section.

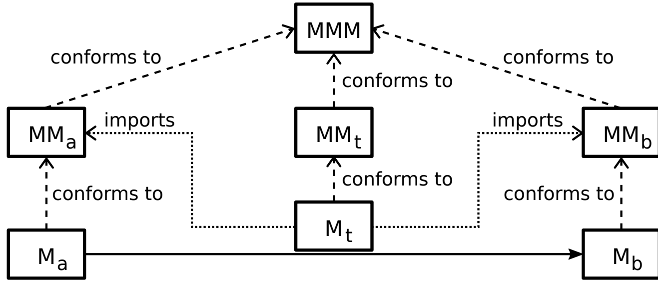


Figure 3.7. Generic transformation megamodel [FLV12]

In Figure 3.7, a generic megamodel of a transformation [FLV12] is presented which transforms a source model M_a into a target model M_b. Each model conforms to its respective metamodel MM_a and MM_b. In the megamodel, the transformation itself is a model, depicted as M_t to illustrate the model character of the transformation. The direction of the transformation is expressed by a solid line between source and target model with an arrow pointing towards the target model.

The transformation model (M_t) is defined over the metamodel of MM_a and MM_b which define the typing and, therefore, the structure of any model M_a and M_b, respectively. Of course the transformation also conforms to a metamodel MM_t.

While the comprehensive megamodel of a transformation in Figure 3.7 expresses all the relationships models and metamodels have in context of one transformation, this pattern is quite complex when used for larger chains of transformations or when multiple source and target models are used. Therefore, a transformation is often only depicted as an arrow with one box at each end. In Figure 3.8 different kind of simplified transformations are depicted which illustrate scenarios with multiple source and

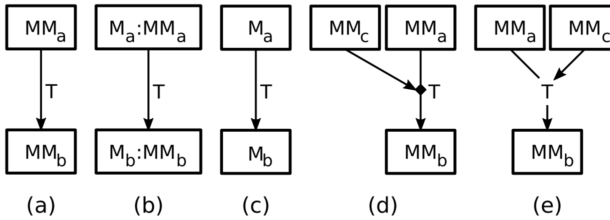


Figure 3.8. Different simplified transformation megamodel notations

target models. In (a) the transformation T is depicted with two metamodels MM_a and MM_b. This pattern is used to express the transformation on a metamodel and language level where the concrete model is not of importance. When a specific model instance is relevant, the transformation can be sketched either with M of type MM, in short $M : MM$ or only by its name M when the metamodel can be resolved from the context.

The last two notation examples represent a transformation T which uses an auxiliary input of type MM_c. While (d) is applicable for small numbers of auxiliary inputs where often MM_a and MM_b are of the same type, (e) is used when an arbitrary number of inputs is used. The lead input model can then be depicted without an arrow on the input side.

3.6.3 Weaving Megamodel

Model weaving, as explained in Section 3.5, is the action of integrating nodes of an aspect model, called advice, into a base model, based on a pointcut or join point model.

In Figure 3.9, in analogy to the generic transformation megamodel, we illustrate a weaving megamodel. The base model M_a is transformed by M_t into a model M_b which both conform the same base metamodel. The aspect AM_a which conforms to the aspect metamodel is represented as an auxiliary input to the weaving transformation. In the illustration, both, the aspect metamodel and model define a reference to their base counterparts. Such a comprehensive megamodel for a weaving transformation is too complex to be used in full detail in the context of multiple transformations.

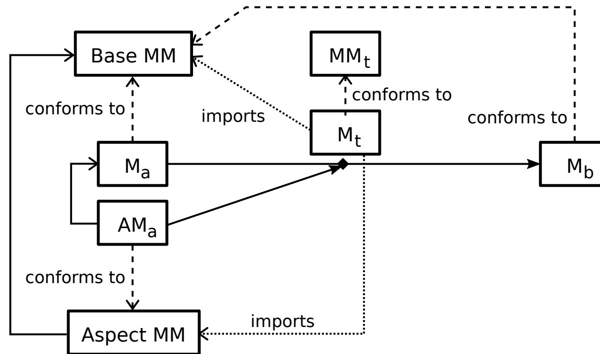


Figure 3.9. Weaving transformation megamodel

Therefore, it can be depicted as transformation with one auxiliary input, as illustrated in Figure 3.8 (d) and (e).

3.7 Transformation Languages

Transformations can be implemented in different ways with different programming paradigms and languages. As GECO is largely technology independent, it allows to use any number of transformation languages and frameworks. In research and industry a variety of transformation languages have been developed, based on declarative, e.g., graph rewriting and matching, functional, and imperative paradigms. Depending on the framework and tooling, the languages support multiple paradigms or provide separate languages for the different approaches. As this thesis utilizes EMF as framework to illustrate and evaluate the approach, we shortly introduce four languages compatible with EMF and their implementation illustrating different paradigms and approaches.

3.7.1 QVT

Query/View/Transformation (QVT) [QVT05] is a standard from the Object Management Group (OMG) comprising three languages, *Relations*, *Core*, and

Operational Mappings, which can be complemented with any operation defined in a MOF compatible language.

QVT-Relations is a declarative specification language which allows to define relationships between different MOF models utilizing complex object pattern matching. Transformations can be unidirectional or bidirectional, similar to triple graph grammars, and the language implicitly creates model traces to record what happened during a transformation execution [QVT05].

As a minimal transformation language, QVT-Core is also a declarative language. It is intended to be used to realize the Relations language. The pattern matching is based on a flat set of variables and the evaluation of conditions over those variables against a set of models [QVT05]. In contrast to Relations, Core requires to define and create the trace model explicitly.

The third language is called QVT-Operational and has an imperative paradigm. Transformations specified with it are always unidirectional. While being imperative, each operational mapping defined by that language provides a relation usable in the QVT-Relations. By this mechanism, relations can be implemented which are considered too complicated to be specified in a declarative way [QVT05].

As fourth option, QVT allows to specify transformations in any language which is MOF compliant. However, this allows to specify arbitrary model changes which violate the encapsulation of relations causing erroneous and broken models.

QVT is supported by numerous implementations. However, most of them are not feature complete. In this thesis, the basic modeling technology is EMF, an EMOF implementation. For this modeling framework, the Eclipse *Model to Model Transformation*[3] project develops an QVT implementation which is not yet feature complete.

3.7.2 ATLAS Transformation Language

The ATLAS Transformation Language (ATL) is a hybrid transformation language comprising declarative and imperative constructs [JK06]. It is developed by the AtlanMod Group. ATL transformations are primarily structured in modules. Each module comprises a header, imports, helper

[3] MMT `https://projects.eclipse.org/projects/modeling.mmt`

functions and transformation rules. In the header, first the name of the module is specified followed by the declaration of source and target models with at least one source and one target model.

To ease rule writing, ATL allows to specify helper routines which are conceptually derived from Object Constraint Language (OCL) attribute and operation expressions [OCL06]. These helpers may refer to OCL and source metamodel types and can be used to aggregate information of the source model before generation of the target model [JK06]. For example, such routing could collect all attributes of a class following the inheritance graph.

The central concept of ATL are rules which may either be declarative or imperative, called *matched rules* and *called rules* respectively. *Matched rules* are specified by a source and a target model pattern and a mapping between both. It follows the concept of graph transformations for EMF models [BET12]. Imperative rules can be implemented in native code, e.g., Java or in ATL. They are in fact procedures which can perform arbitrary source model matching and construct target model parts. For their implementation in ATL, the language provides *action blocks* which are statement sequences of model modifications.

ATL does not support trace models out of the box. However, Jouault [Jou05] provides a transformation, written in ATL to weave in traceability into ATL transformations.

For ATL a complete workbench comprising editor, compiler, and supplemental tooling are part of the Eclipse Modeling Project and supported by present Eclipse releases. In contrast to QVT, ATL is stable, feature complete, and has been used in numerous projects to implement transformations.

3.7.3 Henshin

The transformation language *Henshin* [ABJ+10] allows to define endogenous (in-place) and exogenous transformations on EMF models. The underlying concept of Henshin are graph transformations [EEP+06; BET12]. In contrast to ATL, Henshin has a graphical syntax with a graphical rule editor for the rule declaration.

A Henshin rule comprises of two graphs, representing a left hand side (LHS) and a right hand side (RHS) graph, where the LHS describes the

matching pattern and the RHS specifies the change on the input model, realizing an endogenous transformation. The EMF models are interpreted as typed graphs (cf. Section 3.1 and [BET12]).

For exogenous transformations, Henshin produces an additional output model. Trace models must be created explicitly. To support traces, Henshin comes with a generic trace model which can be used in transformation rules to collect trace information. However, the primary domain of Henshin are endogenous transformations. Therefore, exogenous transformations are not well covered by the approach.

3.7.4 Xtend

In contrast to the previous languages which have a strong background in graph transformations, *Xtend* is a object-functional language including a template mechanism [Ite11]. Due to its functional characteristics, model queries can be implemented similarly to ATL helpers, including closures and simple set operations.

Xtend is based on Xbase [EEK+12]. It is mapped to Java which allows to integrate functionality implemented in Java and other JVM languages. Model queries, aggregations, and model construction all must be described explicitly. To support rule dispatch, the Xtend supports two constructs. First, the switch statement allows to define complex expression to select cases, and second, methods can be declared as dispatch methods where the signatures must be compatible, i.e., they must share common super types for each parameter.

The one important feature of Xtend is its template mechanism which is especially helpful in model to text transformations. The templating algorithm supports pretty printing and shows in the editor which white spaces in the template will be written to the output and which are omitted, as they are interpreted as part of the indentation of the Xtend code.

Xtend is well integrated into the Eclipse modeling IDE and part of the Xtext language workbench [IE11]. In contrast to the other transformation languages, it does not have an interpreter, but compiles the Xtend code into Java code which is then subsequently compiled to Java byte code.

Chapter 4

Graph and Hypergraph-Based Metrics

Metrics are used in the evaluation of this thesis to measure properties of models and code and compare different revisions of software systems based on the measurements. The evaluation is essentially based on metrics for size, complexity, cohesion, and coupling. In literature [Koz11], many different metrics are introduced and compared. Based on the assessment in Koziolek [Koz11], we decided to use the graph and hypergraph based metrics developed by Allen [All02] and Allen et al. [AGG07], for two reasons. First, they focus on the information contained in a hypergraph instead of just counting elements, like lines, nodes and edges. this is of importance, as GECO is intended to support software development. Therefore, the information content which must be comprehended by a developer should be reflected by the measurement. Second, the graph and hypergraph abstraction allows to compare artifacts which are realized with different languages or coding styles. In contrast, Lines of Code (LOC) is affected by coding styles, such as placement of braces behind or below a statement and a declaration.

The metrics of Allen et al. measure the entropy of a graph [All02] and hypergraph [AGG07]. Higher values of entropy indicate a higher degree of interconnectedness of the graph and hypergraph, respectively. Allen et al. call this measure the size of the hypergraph [AGG07]. Based on this size metric, metrics for complexity, coupling, and cohesion have been defined. They allow to identify these properties of a hypergraph reflecting the effort of comprehension and processing humans and machines have [AGG07]. A detailed evaluation of the hypergraph and graph-based entropy measures can be found in Allen et al. [AGG07] and Allen [All02].

The remainder of this chapter introduces the use graph and hypergraph definitions used for the metrics in Section 4.1. The central size metric is introduced in Section 4.2. Based on the hypergraph size, the complexity metric is explained in Section 4.3 and the coupling metric in Section 4.4. Finally, the cohesion metric is based on a graph abstraction and discussed in Section 4.5.

4.1 Hypergraphs and Graphs

A hypergraph for this metric is defined as $H = (N, E)$, where N is a set of nodes $\{n_1, \ldots, n_k\}$ and E is a set of subsets over N. A modular hypergraph is defined as $MH = (M, N, E)$, where $M = \{m_1, \ldots, m_j\}$ is a set of modules, which are subsets over N which must be pairwise disjoint:

$$\varnothing = \bigcap_{i=1}^{j} m_i$$

For better understandability, we introduce the following symbols based on the approached laid-out by Allen et al. [AGG07]:

▻ **S** the hypergraph of a system

▻ $\mathbf{S}^{\#}$ the hypergraph of a system without nodes which are not connected by any hyperedge

▻ MS a modular hypergraph of a system

▻ MS^{*} a modular hypergraph with only inter-module hyperedges

▻ MS_g a modular graph of a system

▻ ${MS_g}^{\circ}$ a modular graph with only intra-module hyperedges

▻ ${MS_g}^{(n)}$ a complete modular graph sharing the same nodes as MS_g with edges between all nodes

4.2 Hypergraph Size

The metric computation emanates from the nodes and the hyperedges these nodes are connected to. For each node a pattern is computed containing ones and zeros to represent connected and not connected hyperedges, respectively. For example, the sequence 01101 of a node implies its connection to hyperedge 2, 3 and 5 (reading from left to right), but not to 1 and 4. This computation results in a pattern per node. The probability of each of these patterns is $1/k$ with k being the number of nodes. However, it is possible that multiple nodes have a pattern with the same sequence of zeros and ones. In this case, the patterns are aggregated and the probability is increased according to the number of occurrences of the pattern.

In the metric of Allen et al. [AGG07], the function $\hat{p}_l$ is used to return the probability of the l-th hyperedge pattern. The function $L(i)$ returns the index of the pattern of i-th node. Therefore, $\hat{p}_L(i)$ returns the pattern probability for node i.

$$Size(\mathbf{S}) = \sum_{i=1}^{n}(-\log_2\ \hat{p}_{L(i)}) \tag{4.1}$$

The size of a system is the amount of information in the system **S** and calculated as defined in Equation (4.1).

4.3 Hypergraph Complexity

The complexity of a system is defined over the sum of sizes of graphs $\mathbf{S_i^\#}$ containing only node i and all nodes which are connected to i via a hyperedge, less the contribution of the environment, which is achieved by subtracting the hyperedges only graph $\mathbf{S}^\#$ of the complete system (see Equation (4.2)).

$$Complexity(\mathbf{S}) = \left(\sum_{i=1}^{n} Size(\mathbf{S_i^\#})\right) - Size(\mathbf{S}^\#) \tag{4.2}$$

In context of the analysis of Java methods, the use of $\mathbf{S}^\#$ removes all

nodes representing methods which are not connected to the implementation, meaning they are neither called nor do they call any other method. They also do not access a common class variable.

4.4 Hypergraph Coupling

Coupling is defined for modular systems *MS* which represent the modularized variant of systems **S**. In our evaluation, a system is, for instance, a generator and the modules represent its implementing classes. The nodes represent the methods of the classes, and the hyperedges are either method calls or class local data accesses. The coupling for such modular systems is then defined by the complexity of a hypergraph MS^* only containing the inter-module hyperedges.

4.5 Graph Cohesion

To compute cohesion, Allen [All02] defines the modular graph metric, depicted in Equation (4.3). The metric is defined by the ratio of the complexity of the intra-module graph MS_g° versus the complete graph $MS_g^{(n)}$ of the modular graph MS_g. A complete graph is defined as a graph where all nodes are interconnected by edges.

$$Cohesion(MS_g) = \frac{Complexity(MS_g^{\circ})}{Complexity(MS_g^{(n)})} \tag{4.3}$$

While such a complete graph can be defined for any normal graph, it cannot be computed for a hypergraph in a meaningful way for the cohesion metric. The problem with hypergraphs in this context is that a complete hypergraph would not only contain hyperedges with two partnering nodes between all nodes, like a plain graph, but also all other possible hyperedges over a given node set. This would result in very low cohesion values for any realistic hypergraph. Therefore, no hypergraph based metric was defined by Allen et al. [AGG07].

However, we require a cohesion measurement for our evaluation of GECO. Therefore, we convert a given modular hypergraph MS into a graph MS_g. This is done in three steps:

1. All modules and nodes of the hypergraph $MS = (M, N, E)$ are transferred to the graph $MS_g = (M_g, N_g, E_g)$ with $M_g = M$ and $N_g = N$.
2. We transfer all hyperedges which are edges $E_g = \{\forall e_i \in E \mid |e_i| = 2\}$.
3. For all hyperedges with more than two partnering nodes $|e_i| > 2$, a node $n_j \in N_g$ is created, added to N_g, and edges are constructed to interconnect the original nodes and the node representing the hyperedge.

Based on this graph MS_g, we compute the complete graph $MS_g^{(n)}$, called fully interconnected graph [All02]. Subsequently, we derive the intra-module graph MS_g° from MS_g which only contains edges inside a module and no edge crossing module boundaries.

Part II

Generator Composition for Aspect-Oriented Domain-Specific Languages

Chapter 5

Contribution and Research Design

The GECO approach addresses the development, evolution, and reuse of transformations used in code generation to ease the utilization of models in software development and evolution. The approach itself comprises different methods, techniques, and tools to construct code generators and combine generator fragments and modules.

In this chapter, we give an overview of the research design and the contribution of this thesis. Section 5.1 defines the research scope. Section 5.2 introduces the key research questions addressed by the GECO approach. Section 5.3 concludes the chapter and provides the research plan, its work packages and results.

5.1 Research Scope

The scope of this research is to mitigate architecture degradation, support agile and distributed development, and foster reuse of code generators. The objective is to provide an approach to support the construction and evolution of code generators. The construction of generators can be supported by modularizing generators along functional boundaries focusing on high cohesion of each module and a low coupling of these modules. For the evolution, the effect of metamodel alterations and changes in semantics must be considered which can be orthogonal to the functional decomposition. Furthermore, metamodels can be partitioned into different view and aspect types. This partitioning provides a second, more coarse-grained level of generator composition.

Therefore, the envisioned approach addresses metamodeling and the modularization of generator on two levels. First, it must provide meth-

ods and procedures to partition metamodels and define view and aspect types. Second, it must support the composition of generators based on generator fragments, induced on view and aspect types, and provide an abstraction for this composition. Third, it must support the modularization for these fragments. Finally, the approach should be supported by tooling and techniques.

5.2 Research Questions

Derived from the scope and the envisioned approach in Section 5.1, we define the following research questions:

- RQ1: What composition and modularization strategies support generator construction and evolution?
- RQ2: How can an abstraction to the modularization of generators be provided to support the combination of fragments?
- RQ3: What are the criteria for the functional and semantic division of generator fragments?
- RQ4: Which semantical and technical properties must DSLs and metamodels fulfill to support generator evolution?
- RQ5: How can metamodel partitions be identified which comprise coherent semantics and reflect view and aspect types?

5.3 Research Plan and Summary of Results

The research presented in this thesis is structured into five work packages (WP1-WP5), which are described in the following sections:

- WP1: Generator Composition Patterns
- WP2: Generator Fragment Modularization
- WP3: Metamodel Partitioning

- ▻ WP4: Prototypical Implementation and Application
- ▻ WP5: Evaluation

For each work package we describe the goals, summarize its results, and refer to the specific part of the thesis for details. The thesis comprises the following four parts: foundations, approach, evaluation, and conclusion, supplemented by an appendix. The document structure is explained in Section 1.3.

5.3.1 WP1: Generator Composition Patterns

Work Package 1 is dedicated to the development of generator composition patterns. The patterns must be of a structure that allows any generator to be built with them, they must be limited in their number, and minimal in their complexity. This can be achieved by limiting the number of involved models, metamodels, and transformations.

This work package addresses the coarse-grained modularization and composition of generators and provides constraints for the fragment interface. Therefore, it addresses RQ2 and partly RQ1.

The central result of this work package are the deduction of five minimal composition patterns, their specification, and the discussion of their properties (Chapter 7). Furthermore, the work package resulted in a formal notation for megamodels used to specify the composition patterns.

5.3.2 WP2: Generator Fragment Modularization

Work Package 2 focused on the modularization of generator fragments, and therefore, completes RQ1 and addresses RQ3. Furthermore, it introduces the requirements for RQ3 and Work Package 3. Based on WP1, a set of constraints was defined which included the limitation to one source metamodel, one target metamodel, and access to trace models. The fragment modularization approach must adhere to these constraints, support modularization along different criteria, and specify reusable modules.

The fragment modularization approach introduced a method to break up fragments into modules based on their functionality, which followed

the classic layered decomposition of software components, and is based on the semantics of the source metamodel. Furthermore, it introduces a set of common modules, like trace models and name resolvers. The approach and the common modules are described in Chapter 8.

5.3.3 WP3: Metamodel Partitioning

Induced by requirements from WP1 and WP2, this work package discusses metamodel semantics and partitioning based on syntax and semantics. The work package addresses RQ4 and RQ5.

The results of this work package are discussed in Chapter 6. They comprise a discussion of typical use cases of metamodels, syntactical properties of EMOF based metamodels, the distinction of metamodel partitions in view and aspect types, and basic design of metamodels for type systems and expressions. The discussion is supplemented by a partition method based on metamodel structure.

5.3.4 WP4: Prototypical Implementation and Application

Work Package 4 is dedicated to the prototypical implementation of concepts and modules introduced in WP1 and WP2. It comprises the implementation of a framework for generator fragments including interface declarations, reusable modules, and abstract classes for the integration of different transformation languages (Section 10.1).

The framework is supplemented by a textual DSL and a generator to model generator composition in Section 10.2. The textual DSL is accompanied by a text editor and an automatic graphical view, which visualizes the generator composition.

To illustrate the ability of GECO to create reusable Aspect-Oriented Domain Specific Languages (AODSLs), we developed a base model independent AODSL for instrumentation. The DSL allows to specify instrumentation probes and the placement of these probes. The DSL and generators are discussed in Section 10.4.

5.3.5 WP5: Evaluation

The last work package comprises all activities related to the evaluation of GECO, specifically the generator composition and modularization approaches from WP1 and WP2.

The evaluation design is presented in Chapter 9. It is loosely based on the Goal Question Metric (GQM) method of Basili et al. [BCR94] and Solingen et al. [SB99] and defines three central goals for the evaluation. The evaluation itself comprises two case studies and interviews. The first case study resembles a feasibility test for the approach (Section 11.2). The second case study utilizes a past generator development project as basis for the evaluation experiment. This experiment re-executes the project limiting the knowledge for the developers in the experiment to knowledge of the original developers (Section 11.2). Finally, the evaluation is complemented by a set of expert interviews (Section 11.3). They are used to verify the premises used to define the challenge GECO is designed to solve.

The results of the evaluation are summarized in both Section 11.4 and in the conclusion (Chapter 13). They document the feasibility and practicability of GECO for the construction of generators. Furthermore, the interviews suggest that the construction and evolution of generators are important in industry and research, and that the interviewees consider the laid-out approaches and methods helpful in this area.

Chapter 6

Syntax and Semantics of Metamodels in GECO

Metamodels are used in Model-Driven Engineering (MDE) to define the syntax of models. In EMOF [MOF15] and its implementation in EMF [SBP+09], metamodels comprise classes and data types, where classes include attributes and references. Classes, attributes and references, can be constraint with OCL [OCL06].

Model-based tooling is realized based on these metamodels providing the means to define and transform models. The specification of models is often realized through textual and graphical Domain-Specific Languages (DSLs). The abstract syntax graph of DSLs, and thereof derived representations, e.g., in form of an AST, represent models conforming to appropriate metamodels [Fow10]. These metamodels are utilized by editors and visualizations to provide refined views on models and to validate DSL artifacts. Furthermore, transformations create and modify models based on internal rules and expressions. Transformations are used in a wide variety of tools and applications, e.g., editors, generators, interpreters, simulators, model checkers, and database schema mappings. All these tools affect how metamodels must be formed, due to the specific way they navigate and change models.

DSL grammars consist of syntactic rules which are accompanied by distinct informal or formal semantics, like typing, expressions, arbitrary structures, configuration and declaration of values. All these semantic aspects of DSLs have their counterparts in metamodels.

In context of the GECO approach, the partition of metamodels along semantic boundaries is used in the generator fragment and part decomposi-

tion [JHS+14] and play, therefore, an important role in the overall approach. Therefore, we discuss patterns and processes to support the partitioning of metamodels.

First syntactical properties of metamodels are discussed in Section 6.1 which are relevant to understand the remaining chapter. Section 6.2 illustrates five use cases of metamodels to motivate different semantical patterns used in metamodels, and Section 6.3 defines these semantical patterns based on the contexts where they appear. In Section 6.4 the different types of references used in metamodels and, especially, in the defined patterns are discussed. As metamodels comprise properties used for type systems and expressions, the modeling of type systems and expressions are explained in Section 6.5 and Section 6.6, respectively. The chapter is concluded by the introduction of a process to facilitate the separation of concerns in metamodels Section 6.7.

6.1 Syntactical Properties of Metamodels

A metamodel represents an abstract notation designed to describe concepts and knowledge of a specific domain, view, or aspect of a system, like architecture description, performance annotations, and business processes. In context of GECO, we utilize the notion of base and aspect metamodels, representing two different roles of metamodels: metamodels for trace models to represent node relations, and model query metamodels [JHS+14].

In EMF, an implementation of the essential subset of the Meta Object Facility (MOF) [MOF15], metamodels comprise data types, enumerations, classes, attributes, references, operations, and annotations. Classes contain attributes, references, and operation signatures which might be augmented by annotations. Attributes and references are typed with data types and classes, respectively. And operation signatures allow to declare, but not implement operations instances of a class.

References can be classified into three groups: simple references, aggregation references, and containment references (cf. Section 3.1, [CG11]). Furthermore, they can be derived from other references and be the opposite of another reference.

Containment references explicitly describe the containment hierarchy of the metamodel. They form a directed graph (see Section 3.1) where the classes are nodes and references are edges. A metamodel which allows to form self-contained models is one where all classes can be reached by a path over containment references from a root node. The aggregation of all containment paths are called containment graph [BET12]. Metamodels which are serializable, e.g., in XML or a DSL, must fulfill this property.

Opposite references point in the opposite direction of references (see Section 3.1.2). Their properties are limited by the reference they are opposite to, e.g., an opposite reference to a containment reference cannot be a containment reference too.

In the example metamodel in Figure 3.1 on page 31, the ProductGroup has an aggregation reference pointing to Product. Let us assume that we intend to generate a product catalog where we want to display additional information alongside a product depending on its ProductGroup. A generator for such catalog would iterate over all products in the inventory and produce catalog content for each product including additional ProductGroup related content. In that scenario, we need to identify the ProductGroup a Product belongs to. This can be achieved by searching all ProductGroups for the specific product, which is time consuming, because every element of the products aggregation of every ProductGroup must be checked for equality to the Product currently being processed. Alternatively, the opposite reference group can be used to navigate from the product directly to the ProductGroup. Therefore, opposite references can be helpful in model navigation. In EMF, the group reference is automatically set and changed when a Product is added to ProductGroup and moved between them, respectively.

6.2 Use Cases for Metamodels

On the primitives of metamodels, introduced in the previous sections, only a few properties, such as containment and aggregation have been defined [SBP+09; MOF15]. To achieve a deeper understanding of metamodels, they must be studied in their application context. Therefore, we identified and

analyzed four distinct use cases of metamodels, which are derived from the domain of MDE, comprising editors, generators, interpreters, and models at runtime [JHS+14]. Furthermore, in enterprise applications and embedded system software, metamodels can be used to describe the data models, rules, and transformations. However, we could not collect metamodels form industry of these two domains, as they are considered company secrets. Instead we have to rely on public standards, such as Java Persistence API (JPA) [Dem08] which includes an internal DSL for data modeling without a formal metamodel, and Java Server Faces (JSF) [Bur13] which relies on XML to express modeling constraints.

6.2.1 Editors and Views

Editors and views allow to read, inspect, create and modify models. They realize the presentation of models and communicate model information to the user. To fulfill this task, editors read and aggregate information from models, and allow the user to modify model elements. This can be realized through different interfaces, e.g., textual, graphical, and form-based interfaces.

In contrast, views provide special aggregated model information, like outlines and context information. For example, in Eclipse an editor may present Java code to a developer and support him or her with context sensitive views and an outline of the edited artifact on the side.

EMF-based editors can be constructed with various frameworks, like Xtext [IE11] and KLighD [SSH13], realizing textual and graphical input. And they might be realized with a tabular or tree structure input, like the auto-generated editors of EMF metamodels in Eclipse.

Metamodels must provide specific properties for editors to allow to implement the basic features of editors. In detail, editors must be able to traverse and navigate models to be able to find information for the user. They must be able to aggregate and process model content to construct a presentation of the information. For example, Xtext editors link text to an AST, which is then linked to a model derived from the AST. The syntax highlighting in Xtext is computed based on the AST, and contextual help utilizes the derived model. In addition an outline view provides an

aggregated view on the structure of the document in the editor which is also based on the derived model.

Auto-generated EMF editors use a direct approach and create their tree presentation from the structure of a loaded model. They relate tree elements directly to model elements. In addition, these tree editors aggregate information and provide it to the user to support configuring attributes and references.

Therefore, editors and views impose a set of requirements onto metamodels. First, they must be able to navigate and query models to find information. Second, they need to aggregate model information to support user operations, which is why, editors and views include aggregation models. Third, to handle the relationship between different models, e.g., AST, model, text, and presentation, they require trace models. And fourth, when considering complete editors, they also need the means to serialize and deserialize models.

6.2.2 Generators

Generators are programs which read models, process their content, and produce other models as output. They resemble, therefore, exogenous transformations (cf. Section 3.2). In most cases they are also vertical transformations, as they transform models into models of another level of abstraction. For example, the ProtoCom generator [GL13] produces Java code from Palladio Component Model (PCM) models. In context of reverse engineering, generators are used to process concrete models into abstract models [KDG+13] to extract architectural information from the code base.

Generators and especially the enclosing transformations must be able to aggregate information of models, and to query and navigate models. In context of GECO, they must be able to collect and store traceability models, and they need to derserialize and serialize models (cf. [Bie10]).

Navigation Navigation is required to traverse the model graph to specific elements. Navigation is also the basis of queries. For example, a generator for our Inventory example (see Figure 3.1) should produce documents for a label printer and product catalog for all products in the model. However, its

processing starts with the model root, an instance of the class Inventory. For the generation of the labels, the generator requires all products which can be reached by navigation. To produce specific product catalogs, the generator must be able to select a subset of products depending on a predefined criteria expressed by a model query or constraint.

Another example is the IRL [JHS13] used in our information system case study Common Component Modeling Example (CoCoME) (cf. Section 11.1). The IRL realizes multi-inheritance for record types. Its code generator collects different subsets of record attributes to produce serialization routines for records and the correct number of getters and setters.

In both generator examples, navigation and model query are relevant, and thus, metamodels must support both activities.

Aggregation Aggregation is a feature used in generators to combine information from different models and subgraphs of models [Bie10]. For example, the IRL generator aggregates query results to resolve multi-inheritance. For aggregation, generators require internal runtime storage, which can be expressed with a metamodel. Even in cases where this is realized directly in implementation language types, these types form a metamodel which has dependencies with the source metamodel.

Traceability To provide traceability for model nodes between source and target models, generators can produce trace models. They are often realized with tuple collections [ANR+06; GK10], but can also be structured based on different source and target types [HSJ+15]. Trace models use, therefore, types from source and target metamodel.

Serializability Depending on the context generators, they might have to deserialize a model themselves or can rely on the surrounding framework to provide them with a deserialized model. For example, in Xtext generators are invoked by passing them the root element of a model.

For code generation, code generators usually produce the code themselves and do not rely on the framework to serialize the target model. Therefore, the target model must be serializable. In text-producing genera-

tors serialization is often realized with a templating languages, like Xtend [Ite11] and Acceleo [MJL+06]. However, model persistence can also be realized with an XML generator serializing a Document Object Model (DOM) and, in case of EMF, with a serializer for XMI.

6.2.3 Interpreter and Evaluation

Model evaluation and interpretation is used in a wide range of tasks. For example, in Palladio, design-time models are assessed with an simulator, called SimuCom [BKR09], which interprets specified abstract behavior of software and hardware. An interpreter is also used in the project MENGES [GHH+12] to simulate expensive hardware devices during development with software.

From a modeling point of view, interpreter and simulators require terms for evaluation, data to be processed, and internal state, which are required by an algorithm implementing the semantics of the tool.

Terms are mathematical functions, expressions, operations, rules, and even sets of tuples, which allow to specify behavior to be executed by an interpreter or simulator. In simulators and interpreters, such terms allow to describe data modification and the transitions of an automaton.

The data processed by terms is kept in a data model, which usually is self contained and does not reference the interpreter state and the models representing the terms.

Distinct from data, the interpreter and simulator may have an internal state to control the execution and to provide faster access to data and terms. For example, the state is used to remember which term must be evaluated next. The state, therefore, differs from the data model, as it contains references to the data and behavior models, while the data model usually has no such knowledge.

In extendable simulators, the core simulator is not aware of additions to specific data and behavior model elements [JHS+14]. Instead it has to contact an extension module for every model element, to verify whether there is anything to be done by the extension. The extension must then iterate over all its extension elements to find whether there is something to be done for the given node. This is not very efficient. Therefore, a

utility model, mapping elements of the simulator core model to elements of the extension model, improves the lookup time for the extension, as initially the software determines all extension elements for core elements and provides this mapping during the simulation. These utility models are syntactically similar to trace models. However, they do not relate between source and target models, but between base and aspect models, used in the core simulator and the extension respectively.

6.2.4 Runtime Models

Runtime models provide abstract views on software systems and their context during runtime [GV13]. They are derived from design-time models and runtime monitoring data [EVT+13; JHS13]. While analysis tools exist, which operate on design-time models [BKR09], they might be too complex for runtime purposes. Therefore, approaches, e.g., from Vogel et al. [VNH+10], propose tailored views on runtime models covering specific aspects of the system to avoid complexity and ease runtime evaluation.

Runtime models are constantly changed and different revisions of them are fed into simulators, interpreters, and model checkers to analyze properties of the models and forecast future states of software systems.

To be able to relate runtime model elements, especially those resembling specific views of the system, to the design-time model and the present architecture of a software system, model traceability is required between specific and general models. Furthermore, to understand the changes made to runtime models, it is important to know how model elements relate to previous model elements.

6.3 Contextual Metamodel Patterns

Syntactic design patterns, as expressed in Cho et al. [CG11] and Section 6.1, are common knowledge and appear in technical and research literature, e.g., EMF [SBP+09] and EMOF [MOF15]. However, these syntactic design patterns provide only limited guidance, like realizing a consistent containment hierarchy, but they do not describe how to construct metamodels for specific

purposes. Therefore, we propose contextual metamodel patterns [JHS+14], based on the use cases defined in Section 6.2. We chose the term pattern over kind, because kinds are mutually exclusive entities and our patterns can occur together in a single metamodel.

Navigation The ability to query a metamodel is largely based on navigation and the interpretation of properties. Model navigation is usually realized by explicitly defined references in a metamodel and by inverse references implied by the containment hierarchy. In EMF [SBP+09], this is realized on the level of Java classes by the eContainer reference.

In metamodels where one class might be contained in different others. For example, a variable declaration in Java can appear directly in a Java class and in any statement sequence. Therefore, in a metamodel for Java, there exist different containment references for variable declaration.

As we previously introduced, it is possible to define opposite references to improve navigation. However, in this context the class representing the variable declaration would require two different opposite references which can be confusing and must be checked both to find the parent. Furthermore, when modifying metamodels opposite references must be adjusted separately which can result in errors. However, EMF provides an implicit opposite reference. Therefore, we encourage to omit explicit opposite references in context of containment.

However, in certain contexts, such as in our example Figure 3.1, where an opposite references eases navigation, they should be added to the metamodel to improve navigation abilities. Such opposite references occur preferably in data models such as in the example. They do not appear in any DSL we analyzed during our evaluation and are not allowed between base and aspect metamodels. Instead, when required the opposite references should be realized with utility models.

Traceability Traceability between model nodes is relevant in many different contexts. As expressed in the use cases above, trace models are used to relate source model elements to target model elements, extension elements to core elements, and base model elements to aspect model elements.

Usually in literature [GG07] only the first kind are called trace models, which may relay requirement documentation to design artifacts, ASTs to source models, and source models to code. However, also between different versions of a model, a trace model can contain the relationship of nodes between the new and the old model. The last two relationships are usually called join point models, as they describe where the aspect and the extension is linked to the base model.

Traceability can be expressed in a separate traceability model comprising in tuples of references to elements in the two involved models. These models can also be structured to massively reduce lookup time [HSJ+14]. Alternatively, they can be described as simple references, where one model has references to the other. In that case the traceability feature is integrated in the second model. When navigation over the traceability link is only required in one direction and the second model always depends on the first, this is a much more compact and fast method to realize traceability.

Derived Models Frequently, models are complex and not always suited directly for the analysis. Therefore, specialized models are derived from these models. These specialized models require metamodels specific to their task. For example, the PCM [BKR09] allows to define rich models covering different aspects of a system. For an analysis of a PCM model, a specialized model is derived expressing a specific view on the original model in form of, e.g., a Layered Queueing Networks (LQN) [KR08].

After the analysis, traceability references are used to relate the results back to the original model, otherwise the result is meaningless. These references are either part of the metamodel for the derived model, or alternatively are represented by a traceability metamodel realizing the mapping.

Data Data is modeled in metamodels for DSLs when they provide initialization of values, in parameterization of simulations, in data models for applications, like in our example (cf. Figure 3.1), and other areas where structures can be described by values.

Data models are self-contained, i.e., they do not have references to other models, except other data models. They may include additional opposite

navigation references to ease navigation during processing. They describe data, but no behavior. Therefore, functions, expressions, rules, and any other kind of behavior are not part of data models.

Aggregation Aggregation models usually include collections of elements of the same type which may fulfill additional common properties. They are used in editors and generators, for example in scoping and resolving attributes of record structures. In some cases, they are realized as maps, where one property of the contained element set is used as index value.

State In interpreters, simulators, and model checkers, during execution, they need to keep track of information which cannot be stored in a data model, like which rule to apply next, which rules required what kind of data, and register referring or holding information necessary for the next execution and evaluation step. Therefore, state metamodels (cf. [JHS+14]) comprise special state values, like counters, and references to data and execution metamodels.

Execution Execution metamodels provide the means to express behavior for the execution and interpretation of models. They can be processed by generators and interpreters, but have different dependencies in these areas.

For interpretation, they require data structures and values to operate on including values which are part of the state model, as they are defined to manipulate them [Ban98]. The latter are of the same type as in the data model, because types which are used for value storage are inherited from the data metamodel. For generators, execution metamodels refer to metamodels expressing structure (cf. [JSH13]).

The core of execution models, however, is similar in both cases and comprise notational features determined by underlying paradigms, used to specify behavior, like functions, rules, mapping tables, and automata.

6.4 Semantics of References

Based on the previous section, we can define a set of different reference semantics. This list is induced by the use cases and, therefore, inspected metamodels in our research [BKR09; GHH+12; JSH13; JHS+14].

Containment is extensively described in Section 3.1. In essence containment expresses that the referred elements are part of the containing element. For example, a book might be part of a library, but can be borrowed by a student. The latter can be modeled as an association (reference), even though some students might get confused and think they inherited it.

The containment relationship further implies two properties: First, when the containment is removed, all its parts must be removed, as they are part of the containment. And second, during serialization of a model, the contained elements can be serialized inside or directly after its containment without the need of additional reference information. This property is used in the serialization of EMF models and DOMs to XMI and XML files, respectively.

Reference and Aggregation The alternative to containment in EMF are simple references with a cardinality of one, or zero to one, and aggregations with a cardinality of zero and more, or one and more. Semantically, these kinds of references state that something relates to something else. Like in the student example, where a `Student` `borrowed` `Books`. From a language and logic point of view [BCM+10, p. 474ff], `Student` is the subject, `borrowed` the predicate, and `Book` the object.

An aggregation additionally implies that all aggregated elements belong together for some purpose, and aggregation may even impose an order on the aggregated elements forming an ordered set. Beyond these attribution there are no semantic properties associated with aggregations. However, based on our observation, we identified four distinguishable groups of meanings which are introduced in the following paragraphs.

Extension and Aspect Both reference and aggregation can be used to express an extension or aspect, where the referencing object expresses

the aspect and the destination of the reference is the base object which is enriched or extended by the aspect. Such references are used in context of AOM and also in our previous use case of interpreter and simulation to express extensions to a simulation model.

Specification In context of languages, one model might declare an operation signature. This signature is then referenced by its specification expressed in another model. In the programming language C [Ker88], such relationships can be found between function declarations and function implementations which are often separated in header and code files. Specification references are very similar to extension and aspect references, however, in their case, the referenced instance already defines is own properties, while with references from a specification to a declaration, the latter only provides an interface.

Description In expression and typing, references are used to formulate that an element is described in detail by the referred element. For example, a function call contains a reference to the actual function specification which must be executed when the call is executed. Also a variable declaration comprises of the name of the variable and a reference to a type declaration which describes the structure of the variable in more detail.

Derivation As discuss before, a model might be derived from another model. Therefore, the elements of the derived model are derived from elements of the original model. This can be, but is not limited to, a one to one relation. The semantics of a derivation reference is, that the origin of the reference is the derived element and the destination is the original element.

6.5 Typing in Metamodels

As aforementioned, metamodels may describe structure and expressions. Type systems, as discussed in Chapter 2, provide the formal means for structural declarations. They allow to check models for type safety, i.e., a

semantic check for data flow, and can be described with inference rules Section 2.1, e.g., with Xsemantics [Bet11].

In this section, we introduce how metamodels can be constructed with typing in mind. First, we introduce a hierarchy of classes modeling a type system. Second, we define base types in this hierarchy. Third, we introduce structured types such as records and classes. And finally, we describe the modeling of array, collection and map types. There exists a wide range of additional categories of types, however, they are outside the scope of this thesis. Still they can be added to a type hierarchy when required.

Type Hierarchy The type hierarchy is a tree of classes representing different categories of types [JSH13]. At its root, a class Type is declared, which has no further properties beside being the common denominator of types. Figure 6.1 depicts a basic type hierarchy with a root type Type.

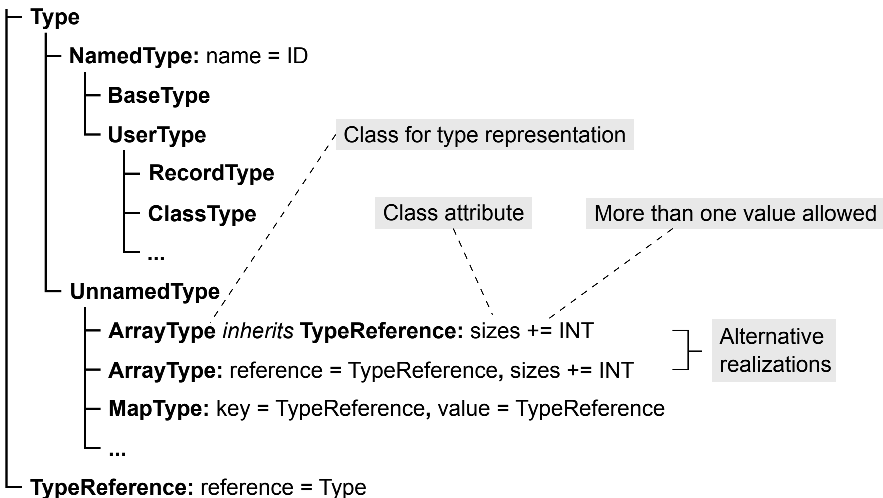


Figure 6.1. Minimal metamodel of a type system (cf. [JSH13])

Based on Type, two major categories are named type and unnamed type. Named and unnamed types are all types with and without a name, respectively. Named types play an important role in DSLs, as they can

be referred to by name, while unnamed types are constructed through syntactical concepts such as arrays [JSH13].

An important aspect of typing are references to types which are necessary to express, e.g., variable declaration and type casts. Depending on the intended type system, type references may only refer to named types, which would exclude array types to be referenced, or all types, which would require syntactical constraints to inhibit the instantiation and referencing of the class Type [JSH13].

Base Types are atomic types which have no internal structure, as described in Section 2.2.1. In a metamodel, they can be represented in different ways, starting with enumerations. Enumerations allow to define distinct entities, an essential requirement of base types. However, they are data types in EMF and EMOF [MOF15, p. 29] which prohibits that an enumeration is a subtype of NamedType. This would result in more complex modeling with types, because everywhere where a type must be referenced two alternatives must be present for the base type enumeration and the user types.

To solve this issue, a class BaseType could be designed with an attribute of such enumeration type to circumvent this issue. This BaseType could then be directly derived from Type, as it does not require a name attribute. For metamodels, where a textual representation of types is not necessary, this is an option. However, the base type would then not be a named type, as the names of other named types, defined by the user, require a string attribute. In DSLs, named types are usually identified by their name and it would require additional rules to handle types which are defined by an enumeration. Therefore, it is often better to define base types just as a subtype of NamedType and without further parameters. The ability to distinguish all base types must then be realized by the software itself. In Section 10.3, we provide a prototypical realization of type systems, which we also utilized in many languages, e.g., Instrumentation Record Language (IRL) [JHS13] and Data Type Language (DTL) [Jun13].

Structured Types Record and class types are very common types in programming languages and in DSLs, as they allow to combine multiple attributes together.

On a type system level, they are defined as sets or ordered sets of tuples (cf. Section 2.2.4). In metamodels this can be realized by a map type where the key is the name of the property and the value is the type. However, this does not integrate well with DSL frameworks and may not provide a stable sequence over the properties, because the order depends on the map implementation and may change by adding and removing elements. Therefore, it is more often realized by a simple list of property declaration which comprise a name and a TypeReference property.

In EMF [SBP+09], which is a metamodel to model types, attributes of base types are distinguished from references of structured types called EClasses. This is necessary to provide additional attributes to both kinds of properties, which are subtypes of EStructuralFeatures in EMF. For example, EAttributes can represent identities, therefore the class has a boolean attribute to indicate this feature. And EReferences may express containment references, references to the container, and opposite references, all features an attribute cannot have. Depending on the purpose of the type system modeled with a metamodel, it can be necessary to separate attributes and references in the same way. However, in all case studies and languages used in this thesis, this distinction was not necessary.

Array, Collection and Map Types Array types allow to model a sequence of instances with a fixed length (see Section 2.2.9). Collection types, like lists and sets, are similar to arrays. However, they do not posses an upper limit (see Section 2.3.2) and may not be ordered. A more general case of a collection, from the type system perspective, are map types. Instead of an integer typed index field, map types may use any other type as index type.

From a metamodel perspective, array types are defined based on a tuple comprising a type reference and a size parameter. Some languages allow to declare multi-dimensional arrays. In that case multiple size values are used, as depicted in Figure 6.1. In some domains, arrays come with a lower bound unequal 0 allowing to specify arbitrary ranges. For these domains the size must be replaced by a class providing a lower and upper bound.

Collections are realized in a similar way to arrays, however, they do not require a size property. Finally, map types require two type references one for the key and one for the value type of the map, as depicted in Figure 6.1.

6.6 Expressions in Metamodels

Another important part in many programming languages and sometimes in DSLs (cf. [EEK+12]) are expressions, statements, and other forms to describe behavior. According to Pierce [Pie02], statements can be seen as expressions with a unit as return type (see Section 2.2.2). Other means to describe behavior, for example workflows and automata, may use expressions, for example to model guards on transitions. However, their remaining metamodel structure is not discussed here, as they can vary largely depending on the behavior they want to describe. Therefore, we focus on expression modeling in this section.

Expressions are in general functions and operations which have parameters, literals and other atomic sources for values. The parameters of expressions are then other expressions. For the discussion of the construction of metamodels for expressions, we introduce five different groups of expressions: literals, properties, functional expressions, operations with two operands, and special features. Similar elements can be found in many metamodels, including the PCM [BKR09], Structured Metrics Metamodel (SMM) [SMM12], and all languages and metamodels used in the case studies for this dissertation (see Chapter 11).

Literals are atomic elements to represent values literally (cf. Figure 6.2), and they have distinct types. For example, `10` is a numeric and `"GECO"` is a string literal. Depending on the metamodel, there can be different kinds of literals including complex structures which allow to describe dates and complex numbers. In a metamodel, each kind of literal should be represented by one class with a sufficient number of attributes expressing the value of the literal, e.g., an integer `value` property for an `IntLiteral`. In addition, all literals should be subtypes of a general type `Literal` which can be used in other parts of the metamodel. This has the downside that syntactically someone could write an expression which combines two incompatible literals. However, it has multiple upsides which make it worthwhile:

1. Numeric literals may be used in different expressions where the other operand has a specific numeric subtype. If the type conformity is enforced

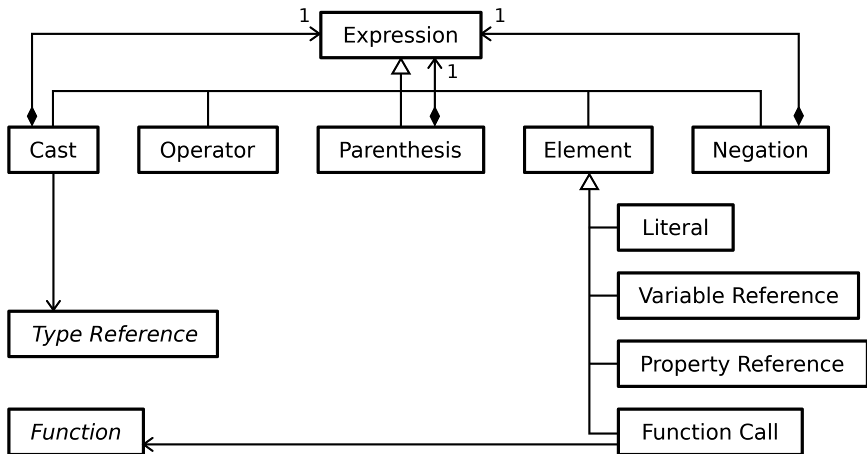


Figure 6.2. An illustrative example of a metamodel for expressions. Classes in italics belong to a corresponding types metamodel.

by the metamodel, this would either require different classes to support every relationship between the general numeric type and its subtypes, or an additional cast class as helper. Both make the metamodel more complex and solve semantics with syntax.

2. Especially in DSLs, automatic type coercion requires a more flexible setup.

3. The types as such are not represented by classes in the metamodel following Section 6.5, and therefore, it would be imbalanced to do so with literals.

4. Semantic conformity cannot completely be mapped to the class structure of the metamodel without making the metamodel incomprehensible, due to a large number of classes required to construct expressions with different types.

5. User defined types are represented on an instance level, as the number and shape of types is determined by the user. For example, an integer

value can be assigned to a variable typed with a user type Interval, which is a numeric type with an arbitrary range of values.

Therefore, it is preferable to use a common super type Literal for all literals which is also the preferred modeling approach [BSV+13] for the DSL framework and workbench Xtext and its expressions framework Xbase [EEK+12].

Properties and constants are single elements of an expression which were declared elsewhere and are now used in an expression. For example, variables and parameters can be declared outside of an expression. Inside an expression an element may then refer to a variable. In a metamodel, this ability can be expressed by a VariableReference and PropertyReference. Constants are only a special case of a variable, as they have a preassigned value and they cannot be modified, therefore, they are not allowed on the left side of an assignment. However, this is a semantic evaluation and should not be handled by a metamodel through structural constraints, as discussed in the previous paragraph.

Function and method calls occur in many expression languages. They comprise a reference to a function or method specification and a sequence of references to other expressions providing the input for parameters. Therefore, the parameter sequence is often modeled as a list of Expression elements. The same structure can also be used for mathematical operations, like addition, multiplication, disjunction, and conjunction, following a notion introduced by functional programming languages, such as Lisp [ABB+64].

Operators can be expressed as functions, and in some languages, such as Lisp and Scheme, they are. However, in many languages, like C [Ker88], Java [AG98], and Xtend [Ite11], operators are considered separate elements of the syntax. In the metamodel of SMM [SMM12], operations are expressed as binary functions with two distinct operands. In Xbase [EEK+12], an expression grammar and metamodel package for Xtext languages, operations are also defined separately from method calls. This can have advantages when processing the expression. For example, the evaluation of a conjunction

can be stopped at the first element which is false. Furthermore, operations, like equal, unequal, and greater as, have only two operands. Therefore, this could be reflected in the metamodel accordingly, as it is a syntactic property.

Special features depend highly on the application domain of the metamodel. Examples are single operand operations, like the *not* operation, often depicted as !, parentheses, and built-in actions, like send and wait. Each of them is usually represented by its own class, like NotOperation and Parenthesis, which have one property referring to Expression. In metamodels which are not used with a textual DSL, a Parenthesis class is not required, as it only links to another expression.

6.7 Separating Concerns and Views

The central notion in GECO is the decomposition and partitioning of models and metamodels along the pairs of roles: aspect and base concern, and dependent and independent view. In this section, we explain the different roles, and the kind of references used between them, and how they can be identified in larger metamodels.

In Section 6.7.1, we provide a simple example metamodel with all four roles. Based on this example, we discuss the different roles and which contextual metamodels are used to realize them in Section 6.7.2. Section 6.7.3 introduces a procedure to identify partitions in metamodels which may represent specific views and concerns. Finally, Section 6.7.4 discusses semantic criteria to select partitions which are preferable for a decomposition.

6.7.1 Example

A minimalistic metamodel resembling parts of the CoCoME case study comprises a Component class used to declare component types, an Execution Environment class to represent a containment which can execute a component, like a computer, a Sensor class describing the instrumentation of component types (cf. [JHS13]), and an Access Control class to define access

restrictions on components. Each concern and view covered by these classes is contained in its own root class, suffixed with Model (see Figure 6.3).

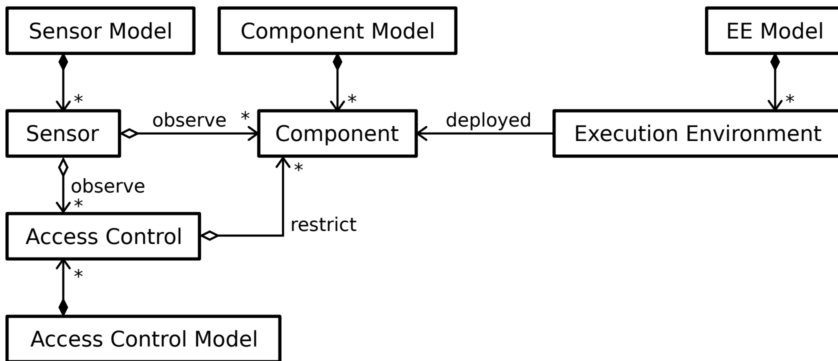


Figure 6.3. Example metamodel comprising the four different roles of metamodels

The example comprises four metamodel partitions covering four views and aspects of a software system. Between these partitions four references interconnect the aspects and concerns. The three references between Sensor, Access Control, and Component are aggregations. They allow to place sensors on multiple access control rules and component types, and express the application of access control rules on component types. The fourth reference is between Execution Environment and Component. It expresses that a specific component type is deployed on an execution environment.

6.7.2 Different Model Roles

In AOM, models are distinguished in base models, representing the core concern of a software system, and aspect models, representing a cross-cutting concern [KS08; KAK09]. A slightly different perspective provides MVM [ASB10] which distinguishes models in different views, like data model, deployment, component specification, and workflow. Kienzle et al. [KAK09] uses AOM to realize MVM, which suggests that both concepts share similarities. The main difference between them is that in AOM aspects refer to cross-cutting concerns which require either weaving at some point

during code generation or interleaving between base and aspect execution at runtime. Where models in MVM refer to separate parts of a software system which may not need weaving.

Aspect-Oriented Modeling (AOM) Like in AOP, an aspect comprises a pointcut and an advice. In some approaches the metamodel for the advice is a subset of the base metamodel (cf. [KK11]). However, in context of DSLs and the complexity of long-living systems, base and aspect models conform to different metamodels expressing different concerns. This allows to tailor metamodels for the concerns they describe instead of their technical realization. Furthermore, aspect metamodels used as an extension method are created after the base metamodel and introduce therefore new classes and terms not present in the base metamodel [JHS+14]. In the example in Figure 6.3, the Sensor class resembles an aspect metamodel which allows to describe one sensor. The pointcut is represented by the reference observe. In other aspect metamodels, a pointcut can be described with model queries and subgraph patterns. In the IAL (see Chapter 10) and AspectJ [Lad09], model queries are used to specify pointcuts.

Figure 6.3 depicts two aspects, sensors and access control which both represent cross-cutting concerns for component types. However, the metamodel of the aspect of access control is also a base metamodel, because sensors can be placed on access control instances. This suggests that the discrimination in base and aspect metamodel is not a final ascription for a metamodel, instead base and aspect metamodel are two roles metamodels can have in the relationship to other metamodels [JHS+14].

Multi-View Modeling (MVM) In Model-Driven Engineering (MDE) models represent different views on a software system [SV06; ASB10; UML15]. Each view can have its own DSL and metamodel. In MVM, models and metamodels can be distinguished in independent and dependent views, where an independent view has no references to other metamodels in contrast to a dependent view. In Figure 6.3, the Execution Environment class is a dependent view and the Component class is part of the independent view.

However, a view can depend on multiple other views, like in the CoCoME case study (see Section 9.4.1), where the behavior language refers to the PCM metamodel and the Data Type Language (DTL). And like the DTL, a view can be in the role of an independent view for multiple other metamodels. The PCM even switches roles. In the relationship with the behavior language, it is the independent view, where it is the dependent view in the relationship with the DTL language. Therefore, we interpret the property of dependent and independent view as roles, like the relationship between aspect and base model, explained above.

6.7.3 Syntactic Partitioning

EMF metamodels can comprise multiple packages which can contain classes, data types, and other packages. Therefore, packages introduce hierarchical partitioning for metamodels. However, the previously described roles of metamodels can also apply to parts of a metamodel and these parts may not coincide with the package borders. While we previously assumed that different concerns and views are stored in different models which have different metamodels, in existing metamodels, such as PCM [BKR09] and MENGES [GHH+12], a metamodel covers different concerns and views or may even cover only a part of a concern. Therefore, we provide a process based on syntactical properties to determine metamodel partitions which may coincide with concerns and views. Each class of a metamodel can only be member of one partition. However, there might be service classes, which do not belong to any specific view or concern. Partitions for these classes can be seen as library partitions.

The process assumes that the metamodel is consistent and does not import classes and data types from an external metamodel. However, if this is necessary, these metamodels must also be included in the process and all included classes are considered the input metamodel. The process is applied recursively over subdivisions of the metamodel under analysis. It may result in a partitioning which is too detailed or too crude, depending on its containment structure. Therefore, it might be helpful for an engineer to add or remove single containment references to improve the partitioning.

We use the type graph notion $TG = (T, I, A, C, OE)$ from Section 3.1.2,

where T is a graph consisting of a set of nodes N_T representing all classes in the metamodel and a set of edge E_T for the relationships between the classes. The remaining symbols are the inheritance relation I, the set of nodes for abstract classes A, the containment edges C, and the relation of opposite edges OE. For the process, we rely on the containment hierarchy which is expressed in C. The process includes the following six steps to achieve partitions.

1. Find all classes $R \subseteq N_T$ which are not contained in another class, i.e, the target function t_T is not defined for the particular node $n_T \in N_T$. They are the root classes R of a partition.

   $$R = \{\forall n_t \in N_T | \forall e_T \in E_T, ((e_t, n_T) \in s_T \wedge (e_T, n_T) \in t_T) \vee (e_T, n_T) \notin t_T\}$$

2. Find for each $r_i \in R$ all contained classes P_i:

   $$P_i = contains_{TG}(r_i) \cup \{r_i\}$$

   Each P_i represents one possible partition. However, partitions may overlap which would violate the criteria that metamodel partitions are disjoint.

3. Detect overlapping parts O_k of metamodels. The set of these overlapping parts is:

   $$O = \{P_i \cap P_j | \forall i, j \in [0 \dots n] \wedge i \neq j \wedge P_i \cap P_j \neq \emptyset\} \text{ where } n = |R|$$

4. Remove the overlapping section from all partitions P_i:

   $$\forall i \in [0 \dots n] \;\; P_i^{'} = P_i \cap (\bigcup_{j=0}^{m} O_k) \text{ with } m = |O|$$

5. Remove all identified partitions $P_i^{'}$ from the graph and repeat the process with the remaining graph until no new partition can be found.

6. Collect all remaining partitions.

As mentioned before, this process might result in a suboptimal set of partitions. Terefore, it is the task for the engineer to determine which partitions should be joined and which should remain separate. As a guideline, the following section introduces an approach to help to decide where to partition metamodels based on the semantics of metamodels.

6.7.4 Semantic Partitioning

The previous section provided a process to detect potential partitions based on syntactic properties of a metamodel. However, these partitions may be caused by a faulty metamodel design and they may not coincide with views and aspects of a software system. Furthermore, the process does not examine the relationships between partitions based on aggregations and plain references. It is possible to specify a test to evaluate whether the references between partitions are all pointing in the same direction or in both directions. However, such tests could not detect whether any of the existing references are design errors. Therefore, we propose an analysis involving the developer to decide where the borders between different partitions are, so that they coincide with concerns and views. To support the developer's decision, we supplement the syntactic partitioning with rules induced by metamodel semantics.

Primary reference direction As stated before, between aspect and base metamodel, the direction of references expresses the pointcuts and join points, which originate in an aspect class and end at a base class. In the same way a dependent view has references towards the independent view. However, in an existing metamodel or set of metamodels, this requirement might not be fulfilled precisely.

To support the detection of the primary reference direction, and, subsequently, the attribution of roles to metamodels, we evaluate any pair of partitions. Let P be the set of all partitions, $P_i, P_j \in P$ be two distinct partitions in this set, and $L = E_T \backslash (C \cup OE \cup I)$ the set of edges which are aggregations and simple references, without inheritance, containment, and opposite references. To determine the dominant reference direction between two partitions, we collect the references going from P_i to P_j and vice versa

with $e_{ab} = \{e \in L | \forall e \in L \wedge s_T(e) \in P_a \wedge t_T(e) \in P_b\}$ where e_{ij} refers to edges from P_i to P_j and e_{ji} refers to edges in the opposite direction.

Based on the size of the two relations e_{ij} and e_{ji}, we can conclude the primary reference direction, which indicates the partition is the dependent view or aspect. For example, $|e_{ij}| > |e_{ji}|$ suggest that P_i is a dependent view or aspect metamodel. However, this suggestion can be inaccurate when size difference of both sets is minimal. This may indicate two highly interlinked partitions which should be merged.

Semantic properties Developers should rely on domain knowledge to decide on partitioning metamodels, as the intended meaning of classes and relationships in metamodels can only be determined by humans. However, we introduced the semantics of references (Section 6.4), contextual metamodel patterns (Section 6.3), metamodel patterns for typing (Section 6.5) and expressions (Section 6.6) which can be used by developers to support their decision process.

Based on domain knowledge and the selection of inter-partition references, a developer can decide whether these references express an extension, aspect, or description relationship, or have another meaning. If references express an extension, aspect, or description relationship, they suggest that the partition where the references originate from is an aspect or dependent view and the references are indeed inter-partition references.

References expressing derivation can indicate an inter-partition relationship when the source and target class belong to a different level of abstraction or a different domain. However, they can also be intra-partition references. For example, the hypergraph metric used in this thesis includes a `derivedFrom` reference which can refer to a Java AST element or to a node and edge of another hypergraph.

Contextual pattern allow to classify parts of metamodels based on their use. They can also help to identify borders of partitions. The traceability pattern can occur between metamodels of different levels of abstraction. This is especially true, if the classes used for both sides of the relationship expressed in a traceability metamodel belong to different partitions. However, traceability is also used between different model revisions. In that case, the classes used to express the traces are the same on both sides.

Finally, typing and expression structures can be identified in metamodels. Both are common in metamodels designed for code generation. Typing classes of one domain usually belong together, as for example, it is unusual to have a class type in one metamodel and arrays in another. However, simpler types may be expressed in a separate metamodel. For example, in PCM [BKR09], class to model component types form a type system in the component repository of PCM, but they are also referenced in the allocation metamodel to construct deployable containers, which belong to a different view on the system. These deployable containers are also types which can be instantiated.

Similarly, classes being part of expression modeling are usually not defined in separate partitions, as they comprise executable semantics. For example, it is unlikely that compare operators are modeled in a separate metamodel from logical and mathematical operators. However, literals and function invocations can refer to functions defined in another metamodel.

While this discussion cannot provide hard criteria to decide in favor of subdividing a metamodel at a specific point, because such decision always include domain knowledge and design decisions, it supports developers by turning their attention to potential partition borders. Otherwise they would have to analyze the whole metamodel manually, which can be quite cumbersome, e.g., PCM comprises 147 classes in 20 packages in `pcm.ecore` (revision 29389) alone and all MENGES metamodels together have 1938 classes in 171 packages (cf. Revision 5 in Appendix C.1).

Chapter 7

Composition of Generators

The composition of generators is the central part of the GECO approach. It relies on the metamodel decomposition and partitioning discussed in the previous chapter to avoid cyclic dependencies between metamodel partitions and complete metamodels.

In this chapter we introduce the combination of smaller generators, called generator fragments into one large generator. Such composed generators can be seen as fragments in another context, thus, the composition can be nested, like puppets in a Matryoshka.

Due to the decomposition of generators in fragments, there arises the necessity to share information between fragments. Therefore, we explain how fragments can share information to work together for a joint result.

In Section 7.1, five composition patterns for generator fragments are discussed and motivated. Section 7.2 explains in detail the two central patterns, which preserve the aspect and base model relationship. Including a discussion of the application of these patterns in different scenarios. Section 7.3 describes how join point models can be computed for these pattern. Finally, Section 7.4 discusses how model traceability can be realized for GECO and which requirements it must fulfill.

7.1 Basic Generator Megamodel Patterns

Code generation for software systems utilizing Aspect-Oriented Modeling (AOM) and Multi-View Modeling (MVM) involves different base and aspect models, and different dependent and independent view models, respectively. These models are then transformed into target models and code. We established in Chapter 6 how metamodels can be partitioned to

provide metamodel parts which fulfill the necessary criteria for aspect and base metamodels, and dependent and independent view models.

In projects with multiple metamodels, like MENGES [GHH+12], Sprat [JH14], and CoCoME [HSJ+15], the generation involves multiple generators processing and combining information from different models conforming to different metamodels.

In GECO all these generators are parts of the combined generator used for the respective project. We call these parts *generator fragments* and the combined generator *generator*. Generators and fragments are technical terms in the process of code and model generation. We also use *transformation* to explain the task a generator and fragment performs. In many cases a fragment only realizes one transformation. However, in some generators and fragments, multiple transformations are chained together to realize the model and code generation.

We introduce candidate patterns for GECO in Section 7.1.1, and discuss their properties. Subsequently, we derive five patterns from these candidates and introduce their abstract structure in Section 7.1.2.

7.1.1 Candidate Patterns

The generator fragments and metamodels form a complex graph of relationships (cf. Figure 9.1 on page 176). These relationships are between metamodels, which can be in general uni- or bidirectional, and transformations represented by fragments, which are unidirectional. We investigated the relationships of fragments and metamodels based on pairs of metamodels as a minimal relationship between metamodels. Starting with these minimal relationships, we identified which fragments define transformations for these two source metamodels (see Section 3.2). Furthermore, we looked at the target metamodels (see Section 3.2) of these fragments. This investigation lead us to several different candidate patterns for generator megamodels [Fav04a] (see Section 3.6), which had one or two source metamodels, one or two target metamodels, and fragments representing transformations, which transform source models into target models.

However, our analysis covered only a few generators, e.g., Xtend [Ite11], Xbase [EEK+12], ProtoCom [GL13], and MENGES [GHH+12]. Therefore, we

deduced 48 different candidate patterns with three and four metamodels involved, as depicted in Figure 7.1 and Figure 7.2, respectively.

Between the four metamodels up to four transformations can be defined which can reside in different generator fragments. In real generators, it is possible to have multiple transformations between any two metamodels, however, such structures can be realized by applying the depicted pattern multiple times.

Patterns with three metamodels There are eight distinct candidate patterns with three metamodels respective models when we perceive it as a concrete execution of a transformation.

In contrast to the four metamodel patterns, the three metamodel patterns are simpler. However, they occur as parts in the four metamodel patterns. Therefore, we introduce three metamodel patterns first.

All eight possible permutations of three metamodels are depicted in Figure 7.1. However, even in this small set there are mirror patterns. The patterns 1B and 1C, and 2B and 2C express the same relationship. Furthermore, the patterns 1D and 2D do not satisfy the pre-condition that two metamodels may only have an unidirectional relationship. Therefore, the patterns 1D and 2D are invalid.

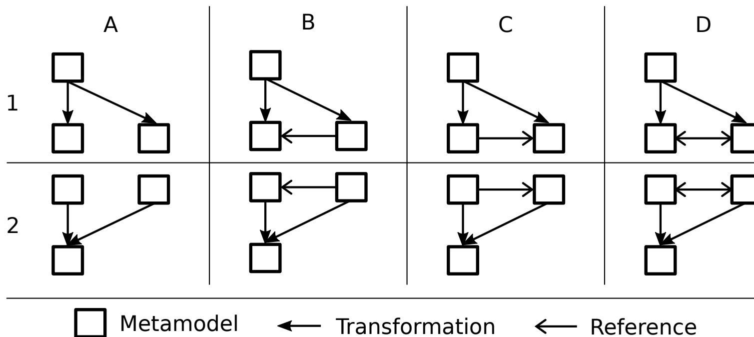


Figure 7.1. Matrix of eight basic candidate patterns involving three metamodels

1A represents two independent transformations. While they use the same

source model, their result is not linked. Therefore, these are simple transformations.

1B is a pattern with an unidirectional reference between the target metamodels, but only a single metamodel on source level. This pattern may occur when the source metamodel describes pointcut and advice together, whereas on target level they are formulated with different metamodels. In this particular case the pointcut references the advice.

2A indicates that two transformations write into the same metamodel instance. However, there are no references on the source metamodel level. Therefore, this is not a weaving scenario (cf. [MKB+08]). Instead it indicates that either a hidden link exists or that both transformations write subsequent and independent content into a model. In the former case it would be actually pattern 2B.

The 2A pattern does not require any explicit information sharing between the two transformations. However, it relies on a specific target model structure, which provides the necessary information for the second transformation. This requires that information is moved the model and metamodel domain into the transformations. This results in internal dependencies between the transformations, which should be avoided to ease adaptation and evolution.

2B resembles a typical weaving pattern where an aspect (right side) refers to a base model (left side) and a set of transformations is used to realize the weaving (cf. Section 3.5).

Based on the discussion of these patterns of three metamodels, we can identify three relevant patterns: simple transformation (1A), reference creation (1B), and weaving (2B).

Patterns with four metamodels The 40 patterns based on a four metamodel configuration are deduced from the four possible source metamodel relationships, four target metamodel relationships, and four transformation configurations, as depicted in Figure 7.2. The raw number of permutations is 64, however, several configurations are just mirrored configurations. For

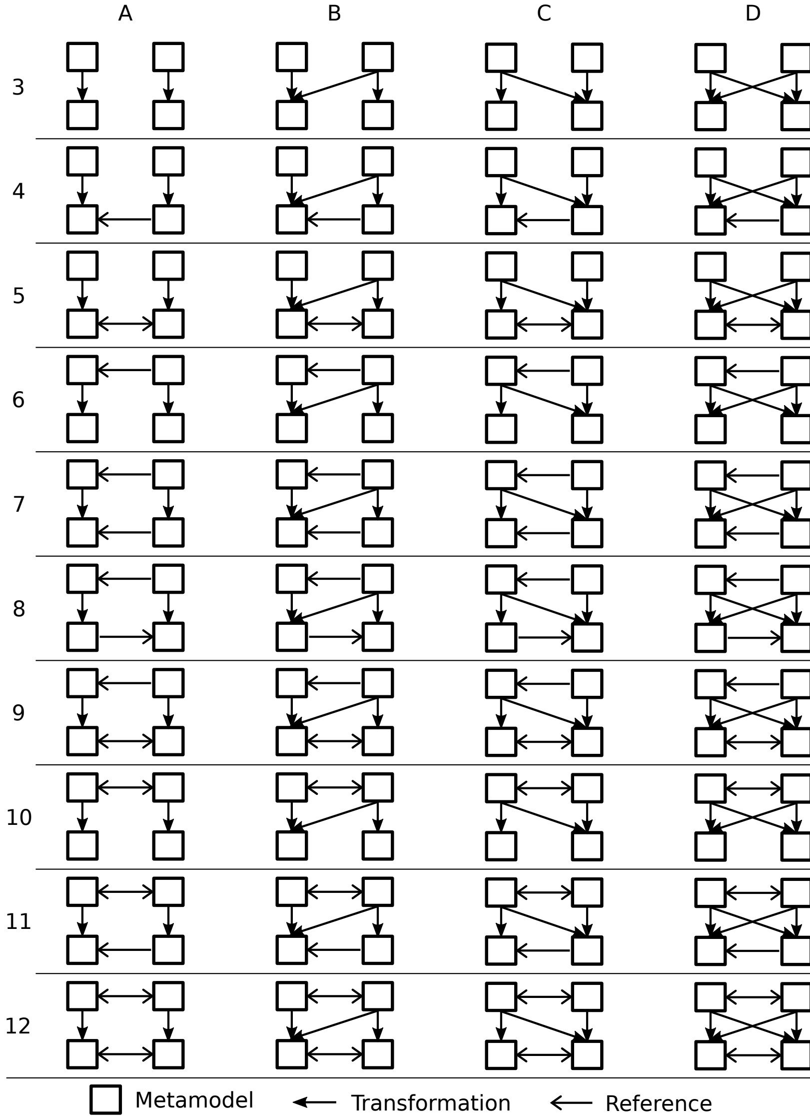


Figure 7.2. Matrix of basic candidate patterns involving four metamodels

example, the elements in row 4 cover references going from the left to right and in the opposite direction. The same applies to row 6, 7, 8, 9, and 11, resulting in 40 remaining patterns.

In this set there are still mirror patterns, 3B and 3C, 5B and 5C, 10B and 10C, and 12B and 12C. Furthermore, several configurations violate the constraint based on metamodel relationships in Section 6.7, which does not allow bidirectional references. Such configurations may, however, occur inside a generator fragment when a metamodel is partitioned and each partition has its own transformation associated with (cf. [EML+15]).

For the megamodel [Fav04a] patterns, we must exclude these variants to conform to this constraint. This results in the following 19 candidate patterns:

3A shows two transformations where there are no references between the metamodels on source and target level, and therefore, there are no dependencies between both transformations. This can be seen as a very simple candidate pattern with two independent transformations, which resembles actually the most simple pattern with two metamodels and one transformation.

3B includes one independent transformation on the right (3A) and two transformations which write to the same result model. This part of 3B resembles the pattern 2A. Like 2A, 3B is not a weaving pattern, as there is no information present between the source models, i.e. there are no references between the source models.

3D overlays 3B and 3C, where 3C is a mirror of 3B. As explained, 3B is a combination of 2A and the simple transformations from 3A.

4A shows references between the two target metamodels. However, there is no information on the source level which could be used to compute these references. This situation can appear when the source level contains reference information encoded in an attribute instead of a proper reference construct provided by the meta-metamodel. Alternatively, the information can be stored in a query model, which must be evaluated to compute real references, as with a pointcut. However, on the level

of abstraction used for these candidate patterns, pointcuts are also considered references, as they can be used to compute direct references. Therefore, the pattern 4A might be an obfuscated 7A or it relies on hidden knowledge in a transformation which should be avoided to support better maintainability.

4B can be an overlay of the 2A and 1B pattern under the premise that the reference between the target metamodels are related to the right source metamodel. Then the left side represents an additional contribution to the result which cannot be linked to the right side, comparable to the situation in 2A. However, it is possible that the reference on target level may originate from the situation explained in 4A with an additional contribution, like in 2A.

4C is similar to 4B, however, the additional contribution, like 2A, is on the right side.

4D is a pattern with two superimposed patterns of type 1B. The references' origin and destination are derived from the same source level model, i.e., the references are computed based on the left source model or the right source model, but cannot be mixed, as there is no information relating left and right source model elements to each other.

6A is a trivial case, as the reference information is not transformed to the target model level. Therefore, the transformations are independent, like in 1A and 3A.

6B is a combination of a weaving pattern, described in 2B realizing model weaving (see Section 3.5), and a separate independent transformation on the right.

6C is similar to 6B; the difference is the additional transformation is on the left side.

6D depicts two separate weaving transformations.

7A is a combination of metamodels and transformations where the references from the source level are retained on the target level. This is a

typical configuration for source level metamodels which relate to different DSLs and therefore are metamodels without a common subset of classes. In this case, the source models must be transformed into target models which conform to metamodels sharing classes before they can be woven (see Section 3.5).

7B depicts a situation where some elements of source models can be woven into the base model, like in the weaving scenario in 2B. And the remaining elements must be transformed to the target level separately, retaining the references, like in 7A.

7C comprises weaving of some elements and the remaining elements are transformed in a way to retain the references, similar to 7B. In distinction from 7B, the woven model parts are stored on the aspect side, while in 7B they are stored on the base model side. Therefore, it is a combination of weaving and configuration 7A.

7D is the combination of 7B and 7C.

8A to 8D are the counterparts of 7A to 7D with the difference that the reference direction is inverted.

9A to 12D use cyclic references between metamodels on source and target level, which we excluded for multiple reasons. First, in AOM references between metamodels originate from the aspect and end at the base metamodel. Similarly, in MVM references between view originate from the dependent view and point to the independent view. Second, we intent to have transformations, which depend only on the output of other transformations and can, therefore, be arranged based on these dependencies. Cyclic references imply information sharing between both transformations. Therefore, both transformations require trace model information from the other transformation which can only be available after execution. Unfortunately, parallel execution cannot mitigate this issue, as it is unclear when each transformation creates a particular entry in its trace model (cf. Section 7.2) required by the other transformations. This may lead to a dead lock.

Based on the considerations described above about patterns with three and four metamodels, we can identify five basic patterns: simple transformation (1A and 3A), normal aspect (5A), inverse reference direction (6A), weaving (2B), and reference creation (1B).

7.1.2 Basic Patterns

Based on the previous assessment and the investigation of existing generator implementations, we identified five different basic megamodel [Fav04a] patterns (see Figure 7.3) which allow to compose complex generator megamodels.

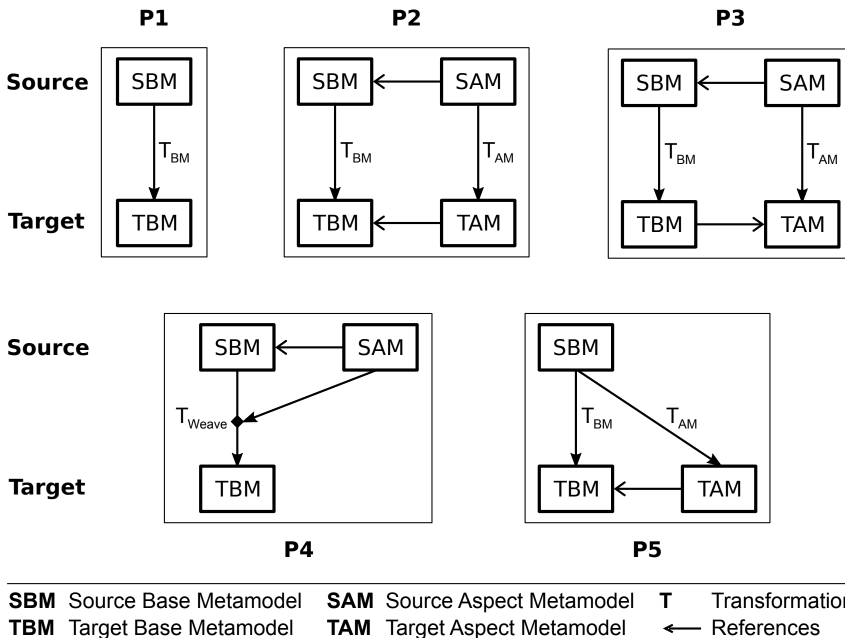


Figure 7.3. Five megamodel patterns for base and aspect metamodels with their respective transformations and target metamodels (trace models omitted)

P1 The simplest pattern comprises one source SBM and target metamodel TBM, and one transformation T_{BM}. The pattern has no additional external dependencies. Therefore, we call this pattern *simple transformation*.

P2 The *normal aspect* pattern addresses that source model references are mapped to target model references, i.e., the references and the reference direction between source aspect and source base metamodel is preserved on target metamodel level. To be able to realize this, trace information of the transformation T_{BM} must be passed to T_{AM}.

P3 The *inverse reference direction* pattern reflects the situation, where the direction of the references is inverted from source to target metamodels. This may happen in generation scenarios where there is no weaving technology present and aspect invocation is mapped to operation calls from the base model to the aspect model.

P4 The central element of Aspect-Oriented Modeling (AOM) and Aspect-Oriented Programming (AOP) is model and code weaving, respectively. The *weaving* pattern (P4), covers this integration of advices in models and code. It requires that the advice metamodel of the aspect is compatible with the base metamodel. The two metamodels SBM and TBM, shown in Figure 7.3, are therefore, identical or, in seldom cases, Source Base Metamodel (SBM) is a subset of Target Base Metamodel (TBM).

GECO allows to incorporate different weaver technologies such as the generic weaver of Kermeta (GeKo) [MKB+08] for models and AspectJ [Lad09] for Java code. In the CoCoME case study, depicted in Figure 9.1 on page 176, weaving is accomplished with T_{JW} which integrated Java snippets into code generated by $T_{ProtoCom}$ and AspectJ.

P5 The *reference creation* pattern is used in situations where two models parts are separated during code and model generation. This happens when a dependent view is separated from an independent view, an aspect model is separated from a base model, and when a pointcut is separated from an

advice. Similar to pattern P2, information must be transferred between T_{AM} and T_{BM}.

The CoCoME case study, depicted in Figure 9.1 on page 176, includes such reference creation. New references are created between aspect.xml and Sensors, where the Sensors are the advice and aspect.xml comprises the pointcuts referencing the advice. In this context, the pointcut metamodel is in the role of the Target Aspect Metamodel (TAM) and the advice metamodel is in the role of the TBM.

7.2 Generator Fragment Combination

The composition of generators requires for the megamodel patterns P2, P3, P4 and P5 to exchange trace information among generator fragments. Pattern P1 does not use any additional information and, therefore, does not require further explanation. P4 covers model and code weaving which is covered in literature [MKB+08; Kra12] and introduced in Section 3.5. Pattern P5 is a specialization of P2 or P3, where the source metamodel comprises two partitions with references between them. For example, in Figure 9.1 on page 176 uses this pattern for the generation of advice Sensors and pointcut in `aspect.xml`. Therefore, we focus our explanation on pattern P2 and P3 in Section 7.2.1 and Section 7.2.2, respectively. Figure 7.4 illustrates these two patterns and how the combination of fragments is realized with them. Furthermore, in Section 7.2.3, we discuss how a legacy generator can be used within the pattern P3. For pattern P2, the integration is simpler. Therefore, we mention the integration along with discussion of pattern P2.

7.2.1 Normal Aspect Pattern

In pattern P2, the generator fragment T_{BM} produces as main output a model conforming to a Target Base Metamodel (TBM). Similarly, T_{AM} produces an output conforming to a Target Aspect Metamodel (TAM). To be able to produce such output, T_{AM} must resolve the reference destinations to a target model based on the references expressed in an instance of the Source Aspect Metamodel (SAM). This task requires trace information for

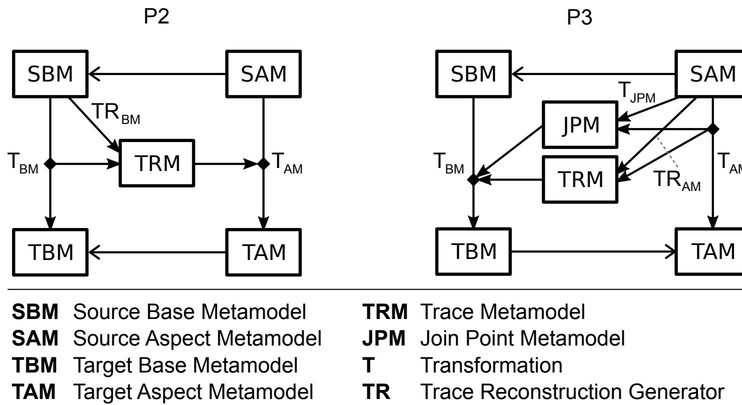


Figure 7.4. Illustration of generator fragment composition for pattern P2 and P3

the generated model nodes of the base model which are stored in a trace model, conforming to a Traceability Metamodel (TRM). A trace model can be generated by T_{BM} as a second output, or can be computed by a separate transformation TR_{BM}. The latter case is necessary when adding a second output is not feasible, e.g., the source code of T_{BM} is not available, or an addition is too complicated, or the code of T_{BM} is inaccessible. Depending on the transformation language, the generation of the trace model must be explicitly implemented or can automatically be added to a generator by a transformation (cf. [Jou05]).

7.2.2 Inverted Reference Direction Pattern

Pattern P3 requires a different approach than P2 to implement the composition. First, it has to invert the direction of the references. And second, the references must be mapped from source to target level. Therefore, T_{AM} or a supplement TR_{AM} produces a trace model for the advice nodes stored in the TRM. Additionally, the pointcuts formulated in the aspect model, conforming to SAM, are resolved to a set of joint points. This can either be done by T_{AM} or by an additional T_{JPM} transformation. The joint points are stored in a join point model, conforming to a Join Point Metamodel (JPM).

The latter is required to infer the inverse references, which are placed into the target base model. The trace model is used to compute target aspect model nodes corresponding to their source aspect model nodes. T_{BM} can then compute reference destinations. First, it applies the join point model on its source model nodes resulting in the corresponding nodes of the source aspect model. And second, based on the trace model, T_{BM} computes the reference destinations.

7.2.3 Integrating Legacy Generator Fragments

The GECO approach is designed to support evolution and one part of evolution is the integration of legacy software. Therefore, GECO needs to be able to handle legacy generators. The two patterns P2 and P3 rely on trace models which should usually be computed by the transformation and provided by the implementing fragment. Furthermore, in pattern P3 the base model generator T_{BM} must be able to integrate the necessary references pointing to the advice. Therefore, we describe solutions for these two challenges.

Trace Model Reconstruction Legacy generators might not come with implementing code, documentation might be incomplete, and it could be just too expensive to add a trace model output to a generator or fragment. In that case, a supplemental trace reconstruction generator is required to compute the required trace model.

The reconstruction rules for a trace reconstruction generator can be deduced by examining source and target artifacts of the fragment they supplement. To support this effort, existing deterministic and heuristic trace model recovering approaches might be helpful [FHN06; GG07; GK10; LAN+15]. However, the results of deterministic and heuristic approaches cannot be used without human intervention. They rely on matching names and structural similarities, which can result in erroneous and missing traces. Both is not acceptable in code and model generation. A developer may then check the proposed traces and derive a trace reconstruction transformation, depicted in Figure 7.4 as TR_{BM} and TR_{AM}.

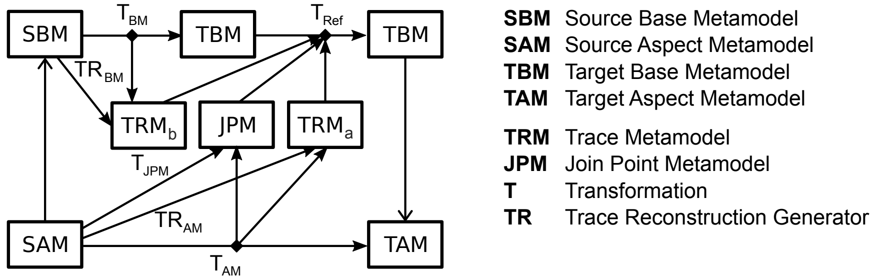


Figure 7.5. Illustration of generator fragment composition for P3 integrating a legacy base model generator T_{BM}, including a helper transformation to T_{Ref} to weave in references

Supplement Reference Introduction In the inverse reference direction pattern (P3), the base model generator must be able to process JPM and TRM information to compute inverse references. With legacy generators, an integration of this feature can be complicated and unfeasible, for the same reason trace model construction might not be feasible.

Figure 7.5 illustrates a scenario with a legacy base model generator T_{BM} supplemented by a reference introducing transformation T_{Ref}. This transformation computes reference destinations based on the join point and aspect trace model. Subsequently, it computes reference origins based on the trace model of the base model side and the source base model. Finally, it integrates these references based on a built-in weaving pattern, which adds the computed reference based on TBM semantics.

7.3 Computing Target Model Join Points

Target model join point references are important as they express the relationship of aspect and base model, and dependent and independent view, on the target level. They occur in the megamodel patterns P2, P3 and P5 and must be derived from source level references. In this section, we explain how the target model references can be computed based on source model references and a trace model. We focus in our explanation on pattern P2. However, the same process must be applied to the other two patterns. As

explained before, P5 is a specialization of P2 and can therefore be mapped to P2. In P3 the roles of aspect and base model generator are inverted. However, the task of determining the reference targets remains the same. Furthermore, we describe the single steps as discrete and independent operations. In real generators and fragments, all these mapping functions would be chained, and executed for each join point without storing large sets of nodes.

In Section 7.3.1, we discuss how nodes for the destination of a reference can be determined on source level, while Section 7.3.2 introduces the computation destination candidates on target level. Section 7.3.3 describes how to select proper destinations from the previously computed set of destinations. And in Section 7.3.4, we discuss how the determination of nodes can be achieved with model-to-text transformations where target nodes may not exist.

7.3.1 Identifying Reference Destinations

We identified three ways to formulate references between aspect and base models, and their view counterparts, which allow to express join points. These are direct references (see Section 6.4), model queries [JHS13], and subgraph patterns (see Section 6.7.2) [MKB+08].

These three kinds of references are realized in metamodels by the declaration of direct references [JHS+14], query expression metamodels, and subgraph pattern metamodels (cf. [MKB+08; Kra12]). The latter two represent pointcuts, as they define join points indirectly, i.e., when they are applied to a model they return a set of join points. These join points are tuples of two nodes representing the origin and destination of the join point reference. Such a join point model J_S on source level can be defined as:

$$J_S = \{(o,d) | o \in N_{aspect_S} \wedge d \in N_{base_S}\}$$

Where N_{aspect_S} and N_{base_S} are source level nodes of aspect and base model, respectively.

7.3.2 Generating Target Model References

As the join points in J_S refer to source base model nodes, they must be translated to retain the join points on the target model level. This can be achieved by replacing all destination nodes in the join point model by their target model level counterparts. As a generator might create multiple target model nodes for one source model node, the resulting target join point model might be significantly larger than the source join point model. In that case for each tuple in J_S multiple tuples must be created in intermediate join point models J_I, representing join points with target model destinations and source model origins.

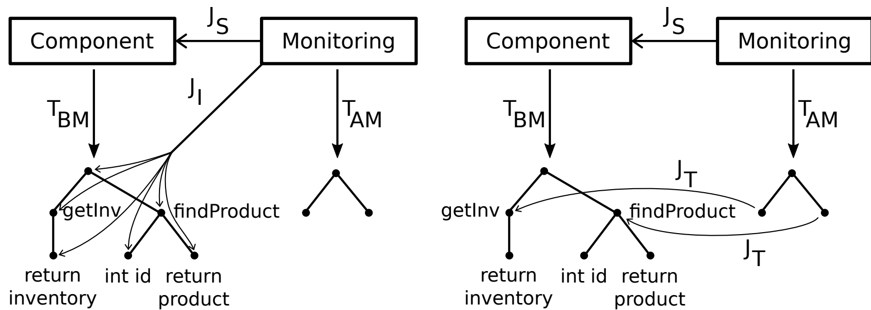


Figure 7.6. Illustration of the generation of target model references based on a component and a monitoring declaration

The example in Figure 7.6 describes a base model with one component and an aspect model with a monitoring declaration. The monitoring declaration references the component, which is the joint point on source level J_S in the example. The component is transformed into an AST-like model below, which depicts six nodes in the target model. The left side of the figure shows also the intermediate set of join points J_I, which describes references that originate at the monitoring declaration and refer to all six nodes of the target base model.

The origins in the join point model on target level are derived, in pattern P2, by the aspect generator during model generation. Based on the nodes of the target model (N_{aspect_T}) and their relationship to the source nodes

(N_{aspect_S}), the aspect generator can identify the origins for the join points on target level. To identify them, the aspect generator uses an internal traceability function, which can be realized with a trace model. Usually a discrete trace model can be omitted, because during generation the generator creates target model nodes while processing source model nodes. Therefore, the join point resolution can be performed in this context.

Similarly to the destination side of the join points, there could be more origins on the target model level than on the source model level. Therefore, the set of join points may increase when resolving origins for target level join point candidates J_{T*} from J_I. The set J_{T*} may include references that refer to nodes which are not suitable as join points. Therefore, we must select a proper subset of join points on target level, labeled J_T in Figure 7.6.

7.3.3 Selecting Proper Join Point References

Based on current approaches to determine and represent traces [GG07; GK10], trace models also contain traces to nodes which are semantically not suited for weaving. This is especially true when trace construction is automatically introduced by a higher-order transformation [Jou05]. In our example Figure 7.6, a join point representing an injection of a monitoring probe should reference an operation and not the parameter or return type of the operation. Therefore, only suitable nodes in the target model must be selected. In `GECO`, this is realized by a subgraph pattern match or a model query, which is applied to each potential destination in the set of join point candidates on target level J_{T*}. For each match of the subgraph pattern or the model query, the reference is valid and placed into the set of proper join points J_T. In Figure 7.6 this set J_T consists of two references, referencing `getInv` and `findProduct`. While all references that pointed to other target model nodes were ignored.

7.3.4 Considerations for Model-to-Text Transformations

The previous sections discussed the construction of join point references on the target model level. This is a suitable approach for code generation which first constructs a target model and then serializes it, based on a template

language. The second case study based on the MENGES project [GHH+12] uses such a generation scenario. The expression language framework Xbase [EEK+12] includes a target model inferrer, i.e., a model-to-model transformation for mapping source to target models. The serialization is then performed afterwards. However, in other cases, like in our first case study based on CoCoME (see Section 9.4 starting on page 174), generators are used which directly produce code and do not generate a target model that would have to be serialized. Therefore, the determination of the trace model and subsequently its use in an aspect generator are different from a scenario where target models are used.

Depending on the target metamodel, in this case a concrete syntax, nodes are expressed in different ways. For example, in Java a class can be addressed by its fully qualified name, which is a concatenation of the package and class name. And the package name also relates to the directory structure inside the project or archive. The programming language C [Ker88] does not have a package structure. Potential naming conventions used for functions may not necessarily reproduce the directory structure. Therefore, a general technique which can be used with any target language syntax cannot be specified. However, we can conclude that a trace model should contain the following values to refer to a node on target level:

Signature The signature of the node. That is, for example, the Java class name for Java classes, the method signature of a Java method, and the function signature of a C function. We have to use the method signature, as languages exist where the same name can be used multiple times as long as the signature is different.

Fully qualified identifier The fully qualified name of the named element. That is, for example, the fully qualified class name for a Java class, the fully qualified class name, the fully qualified class name together with the method signature for Java methods, and the function name of a C function.

Location A Uniform Resource Identifier (URI) identifying the file or resource containing the node. This includes directory structures stored in archives, file system paths, and online resources.

An aspect generator can then be based on this information, and construct a suitable representation of the reference destination in its target metamodel or language.

7.4 Achieving Model Traceability

Model traceability enables us to understand how software artifacts which were created in a software development projects are related [ANR+06]. In Model-Driven Software Development (MDSD), it is used to relate elements of derived artifacts to elements in the original artifacts. For example, requirements can be linked to features, which are linked to code and model artifacts. In code and model generation, model traceability is used to relate derived model nodes of a target model to nodes of a source model (cf. [Jou05]).

Various methods and approaches exist to obtain, store and use model traces [VVJ+07; GG07; GK10; LAN13]. They can be categorized in constructive, reconstructive, and recovery approaches. The latter use deterministic [ACC+02], probabilistic and heuristic algorithms to find matches [SHN+13]. They are primarily used to link code and models to design and requirement artifacts. And their main purpose is to automate the management of traceability between artifacts, as manual modification is time consuming [GK10].

In this thesis, we do not use recovery approaches, for the following reasons: Heuristic and probabilistic approaches produce non-predictable results. However, in a generator, like in any code and model translation which is intended to be executed, the results must be precise. Therefore, heuristics are not sufficient for generators. Deterministic recovery strategies use naming patterns which are considered identical in source and target models. However, this assumption cannot be guaranteed for all transformations, resulting in incomplete trace models. Therefore, only constructive and reconstructive approaches can determine traces in a quality suitable for generators.

Constructive approaches create traces during code and model generation, combining the information during construction of the target model. This

can be achieved either explicitly or automatically through a higher order transformation [Jou05] depending on the semantics of the transformation language.

In cases where traceability cannot be added to the generator or the modification of the generator is too costly, we propose to use a reconstructive approach. The reconstruction mimics the reference generation of the original generator and usually requires an analysis of the output of the original generator, which is then used to recover the reference construction process of that generator. Subsequently, this behavior is implemented in a supplemental transformation which can then reconstruct the necessary references (cf. Section 7.2.1) and provide a trace model for such black box scenarios.

Chapter 8

Generator Fragment Design

In GECO, complete generators are composed of generator fragments. These fragments depend on one source and one target metamodel, and may produce one or more target models from a single source model. In addition such fragments can read from trace models and create a trace model based on their own operation. This description of the interface of generator fragments allows to construct larger generators based on the patterns introduced in Chapter 7. Furthermore, it allows to provide generator fragments for metamodels and metamodel partitions that comply to the role-based distinction of metamodels.

In this chapter, we focus on the inner composition of fragments. Each fragment can be partitioned into various tasks and functions which it must implement to generate models and code. The modularization can be based on both, metamodel semantics and functionality required to implement a fragment. The functional dimension decomposes a fragment into model query, aggregation, state, target model construction, and control, and may be accompanied by serialization and deserialization [Bie10]. The semantical dimension is characterized by the decomposition of fragments along the semantical decomposition of metamodels (see Chapter 6).

In this chapter, we introduce a modularization approach for generator fragments in Section 8.1, which is founded on the two decomposition strategies based on functionality and semantics. While the functional dimension is discussed in literature, we focus in this thesis on the semantical dimension. First, we discuss in detail the mapping of type systems in Section 8.2. And second, we introduce an approach for expressions mapping in Section 8.3. Finally, we conclude the chapter with the realization of model traceability in Section 8.4, as it represents a common functional module which is used

in many fragments.

8.1 Essential Generator Modules

Generator fragments in GECO depend primarily on two metamodels, one for source and one for target models. Furthermore, they may use multiple trace models and produce a trace model. A generator fragment encapsulates at least one transformation, however, they may also chain multiple transformations [VVH+06] and use localized transformations [EML+15] which realize the generation process. These transformations can be uni- or bidirectional, depending on the used implementation paradigm [Bie10]. In GECO, we focus on transformations which are unidirectional, vertical and exogenous [MG06; Bie10], at a fragment level. Unidirectional transformations are the most common approach used in generators in the industry for code generation (see Section 11.3). And they are vertical and exogenous, as they transform models between different metamodels with different levels of abstraction [Bie10].

Each exogenous transformation reads and processes source models and creates new independent target models. Based on the transformation process, different generator functions can be defined and used for a decomposition, which we discuss in Section 8.1.1. Furthermore, we propose, in Section 8.1.2, the decomposition along the semantics of the source metamodel to foster better encapsulation.

8.1.1 Functional Dimension

Transformations can be written in various languages, which realize different paradigms, such as operational, declarative, graph transformation, and template-based. Despite the different paradigms, all transformations comprise five distinct functionalities (see Figure 8.1). These are source model query, aggregation, state, name resolving, and target model creation. They are supplemented by a central control module (cf. [MG06]). However, the control module could also be provided by the transformation engine [Bie10] and is, therefore, not always explicitly implemented. We call this

decomposition along functionalities the *functional dimension* of the fragment modularization approach.

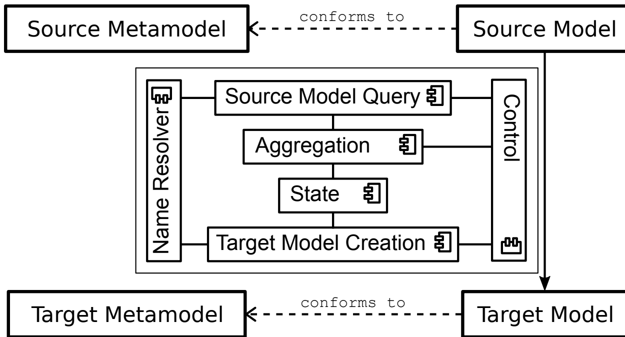


Figure 8.1. Functional decomposition of a transformation

Source Model Query Model query is realized differently in transformation languages. In an operational or imperative approach, the transformation follows the containment hierarchy of the model and checks nodes based on their type, their attributes and the relations to other nodes. For example, in the XSLT [Kay07] transformation languages XML Path Language (XPath) [W3C14] is used to query the model. Also, the language QVT operational [QVT05] allows to formulate queries in an operational paradigm.

Subgraph pattern matching is based on graph structures and the evaluation of attributes of nodes and edges [TFG+07]. For example, the Henshin [ABJ+10] transformation language provides subgraph pattern matching to find nodes in the source model.

Aggregation and Evaluation Query results may be extensive, and are, therefore, often aggregated and evaluated to reduce the amount of information to be stored. For example, the Instrumentation Record Language (IRL) [JHS13] has a multi-inheritance type system. Therefore, a model query may result in a list of attributes which contain some attributes multiple times,

depending on the inheritance graph. These duplicates are detected and removed during aggregation.

State Aggregated and evaluated information must be stored during the transformation process such that it is available for later code and model generation. Also parts of the target model or references to it are stored temporarily during execution. A state module must handle all these kinds of information (cf. Section 6.3). However, in some cases a transformation can directly use the aggregated information to produce its output. In this case the state only consists of a single value of a collection to hold the aggregation results.

Name Resolving Depending on the way in which elements are named in source and target models, a mapping for names is required. For example, the source model has a hierarchical naming scheme comparable to the package naming of Java However, the target model has a flat naming scheme, comparable to the C programming language. In that case proper target model names must be computed which are unique and retrievable based on a specified source model node. We suggest to encapsulate the naming functionality in a module when the mapping is complicated and when they are used in multiple modules. Such a module is particular helpful when the name of an element in the target model is also used to express a reference, e.g., in model-to-text transformation scenarios. In this case, name resolvers are very similar to trace model providers, as they provide a similar functionality.

Target Model and Code Composition A model or code transformation must create output. For a model-to-model transformation the output is constructed out of nodes conforming to target metamodel and subsequently serialized. The modeling framework usually provides serialization functionality, like DOM serializers for XML and model serializers for the Eclipse Modeling Framework (EMF).

In a model-to-text transformation, output construction and serialization can be directly part of the transformation process. In the latter case, the

serialization is part of the target model creation. Some transformation languages and frameworks provide specific templating constructs, like Xtend [Ite11], and separate languages, like Acceleo [MJL+06] which is the model-to-text templating language for ATL [JAB+06].

8.1.2 Semantic Dimension

In Chapter 6, we identified metamodels which imply different semantics, and we discussed the partitioning of metamodels along semantic boundaries. These partitions can exist on a logical level, allowing them to remain in the same metamodel artifact. Along the metamodel partitions, we can identify parts of a transformation and generator fragment which rely mostly on one specific metamodel partition and modularize the fragment based on that relationship (cf. [EML+15]). For example, a larger transformation for a metamodel covering typing and expression, can be subdivided into two transformations based on the metamodel partitioning separating typing and expressions. We call this modularization along metamodel semantics the *semantic dimension* of the modularization approach. The semantic dimension is not in opposition to the functional dimension. Instead both dimensions complement each other.

Metamodels can be designed with any kind of semantics in mind and depending on their use case. In Chapter 6, we discussed various metamodel semantics which can be used to partition metamodels semantically. For transformations we focus on the semantics, which occur in the metamodels we investigated and which are considered common in programming languages [Pie02]. We introduce in the following the four semantics: typing, declaration, expressions, and behavior.

Typing Typing is a common element in metamodels, even though it is often handled informally and perceived as structure (see Section 6.5). In an exogenous and vertical transformation, the source and target models are separate entities with potentially different metamodels and different semantics (see also Section 3.2.3). Therefore, the source model typing must be mapped to the target model typing in a similar way as compilers must map language types to hardware types. For base types, this is often a

one-to-one mapping, e.g., an integer is mapped to a 32 bit integer word, and a boolean is mapped to a byte which will only be used to store 1 and 0. For user types this can be more complex, and involves type structures and runtime support. For example, types, realizing object-oriented classes, require some sort of inheritance. If the target language does not support inheritance, then this inheritance must be realized with the type structures provided by the target language and often a set of runtime functions, like a polymorphic dispatcher.

Declaration Programming languages and DSLs allow to instantiate structures and initialize constants and variables. Structures can be any type including classes and components. Initialization is used to assign values to constants and variables. These values can be represented by literal values and expressions which may be evaluated at compile time.

In general, a declaration comprises an identifier, an explicit or implicit reference to a type, as well as literals and expressions to define the actual value of the declared element. A fragment module handling the mapping of declarations, requires, therefore, access to the typing module providing type references and also literal mapping. In case the expression is evaluated at compile time, the fragment module relies on its own compile time expression evaluation module.

Statements and Expressions Behavior can be described in many different ways, including statements and expressions. From a type system perspective (see Chapter 2), statements are expressions with the return type `Unit`, which is a type representing only one value. `Unit` is comparable to `void` in C [Ker88].

As discussed in Section 6.6, expressions comprise a diverse set of concepts and, therefore, metamodel classes. For example, they include operators, functions, and typing expressions, such as casts. Expressions use literals from the typing metamodel and may refer to declared variables and values. Depending on the language, a variable declaration can also be seen as a statement.

When mapping expressions to a target model, there are not always comparable operators available in the target metamodel. In these cases,

operators are usually represented by functions or the concept used to express functions in the specific target metamodel.

Behavior Beside statements and expressions, there are other means to define behavior, like clauses, rules, and transitions. Depending on the target metamodel and its semantics, additional effort is required to realize behavior which may even include additional runtime functionality.

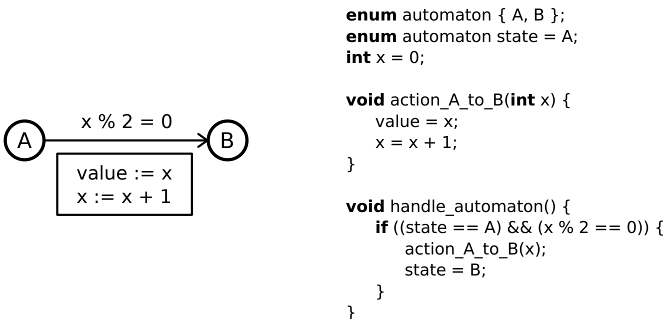


Figure 8.2. An example automaton (left) transformed into C code (right)

For example, a graphical DSL for automata is transformed into C code. The DSL allows to define states and transitions. Each transition has a guard expression and may have an action attached (see Figure 8.2). The generator fragment could then transform the guard expression, action, and automaton handler with its respective expression, action, and handler modules. A composition module could then combine all these parts together.

While transitions and actions can be handled, like expressions and operations with statement sequences, the semantics to handle state are much more complicated and must be defined carefully. For example, if a model contains multiple automata, the execution sequence of the automata must be determined for the resulting C code. Also it must be considered whether an automaton can switch states consecutively until it hits a guard which does not fire, or whether each automaton can only process one transition and then the next automaton must be processed. Such problems

are discussed in context of SyncCharts [And04] and SCCharts [HDM+14], and can be used to guide the mapping of transition and rule systems.

8.2 Type System Mapping

Typing is a central element of many DSLs, although it is often handled informally and not explicitly used as a source to define the capabilities of DSLs. For code and model generators typing is even more relevant, as both, the source and the target metamodel, have semantics which must be mapped to each other.

Typing is a complex field and includes a wide range of types (cf. [Pie02; Pie04]). Especially in DSLs, where typing is used to describe domain concepts, multiple structured types may be used. For example, in the MENGES case study, component, connector, and automaton types exist which comprise features of enumeration types and inheritance. While it is possible to construct any number of types and typing structures, they use characteristics of types discussed in Chapter 2. These common characteristics are base types, property declaration, enumeration sets, arrays and maps, and inheritance. Their realization in metamodels has been discussed in Section 6.5.

For the type mapping, the metamodel and its semantics must be defined. We refer to them both with the term language in analogy of grammar and semantics which define programming languages. This understanding is based on the similarities between both methods to express models and artifacts [PKP13]. In addition, present DSL tooling allows to automatically derive a DSL from a metamodel, and view versa (cf. [Bet13]).

In this section, we introduce principles to map types and typing constructs from source metamodel level to target metamodel level. We use therefore two different strategies. First, we map types based on their semantics directly to target level equivalent types. And second, if the target language does not provide a type with the suitable properties, then a combination of runtime functions and target language types is used to realize the source language semantics.

8.2.1 Base Types

Base types are the core of most type systems. In theory they are defined by their value sets (see Section 2.2.1) and often identified by a name, as described in Section 6.5. In transformations, base types of the source metamodel level must be mapped to the target metamodel level.

```
/** base type mapping. */
def static createPrimitiveTypeName(EClassifier classifier) {
    switch (classifier.name) {
        case 'int' : 'int'
        case 'long' : 'long'
        case 'short' : 'short'
        case 'double' : 'double'
        case 'float' : 'float'
        case 'char' : 'char'
        case 'byte' : 'byte'
        case 'string' : 'String'
        case 'boolean': 'boolean'
        default: classifier.name
    }
}

/** base type mapping with boxing types. */
def static createPrimitiveBoxTypeName(EClassifier classifier) {
    switch (classifier.name) {
        case 'int' : 'Integer'
        case 'long' : 'Long'
        case 'short' : 'Short'
        case 'double' : 'Double'
        case 'float' : 'Float'
        case 'char' : 'Character'
        case 'byte' : 'Byte'
        case 'string' : 'String'
        case 'boolean': 'Boolean'
        default: classifier.name
    }
}
```

Listing 8.1. Two base type mapping functions used in the IRL [JHS13] implemented in Xtend

This mapping depends on the capabilities of the type system on the target metamodel level. Source language types which have a corresponding type in the target language, can be mapped directly, like integer and boolean. A corresponding type must have the same value set and the same semantics

when reaching boundaries. In cases where this cannot be guaranteed, other types and runtime Application Programming Interface (API) must be used to realize the intended semantics.

Boundary semantics When the semantics of over- and underflows differ between type systems, runtime APIs must implement the intended specific boundary behavior of the source language, e.g., create an exception instead of wrapping. Such boundary checks, however, must be integrated into the target model or code where values are changed and set. Therefore, the boundary check is generated by a generator module which realizes the semantics to change and set values, as discussed in Section 8.3.

Differing value sets In cases where there is no direct equivalent type with the same value range, types can also be mapped to types with a larger value set, e.g., `boolean` can be mapped to `byte`. Similar mappings must be realized in compilers when types with smaller ranges are mapped to the CPU word size.

For example, a DSL defines an interval type, which has an arbitrary lower and upper bound. Such a type can be mapped to an integer type which has a comprehensive value set. The runtime API must then ensure that over- and underflows are handled with the intended semantics.

Surrogate types A source language may define types which do not exist as base types in the target language. For example, the types `string` and `date` of the DTL [Jun13] are base types. However, the target language `Java`, does not support these types as base type. Instead two classes of the `Java` runtime API, namely `java.lang.String` and `java.util.Date`, are used to represent the two base types of the DTL.

Base type mapping For the base type mapping, a fragment must provide a function or a mapping table to facilitate the type mapping, which relates named base types of the source language to their target level counterparts.

Some languages provide generics. Generics are used in generic programming to implement algorithms independent of a concrete type. For example

in Java [AG98], generics allow to define classes for lists and collections where the type of element is undetermined at the time of implementing the collection class. When that collection class is then used, an element type can be specified.

However, generics in Java only allow classes to be used in generic types but excludes Java base types. Therefore, the Java API defines so called boxing classes, like Integer, which can then be used with generics. To cope with such limitation, we propose to use two mapping functions supporting base types and their boxing variant.

For example, the IRL uses the proposed approach by implementing two type mapping functions (see Listing 8.1) [JHS13]. As the IRL uses a model-to-text transformation, the mapping function returns the name of the target language type as text instead of returning a model node.

8.2.2 Record and Variant Types

Record and variant types are used to describe structured information. Record types comprise multiple typed and often named attributes, which are used to express the different attributes of an instance (see Section 2.2.4). For example, a record type modeling a Person may have the attributes givenname and surname, both of type text.

Variant types allow to combine other types in a way that they can be used alternatively to each other. They can be realized as untagged and tagged unions. See Section 2.2.5 for a discussion of features of variant types. The C programming language calls variant types unions [Ker88].

For example, variant types allow to define a type Node which comprises two alternative types Branch and Leave, where Branch has two references to Node, and Leave has an attribute label, as depicted in Figure 8.3. This variant type allows to define binary trees with values attached to its leaves.

Record types Modern programming languages, such as C [Ker88] and Java [AG98], support record or class types. Therefore, the mapping of record types is straight forward, as types and attributes of types can be mapped one to one from source to target language.

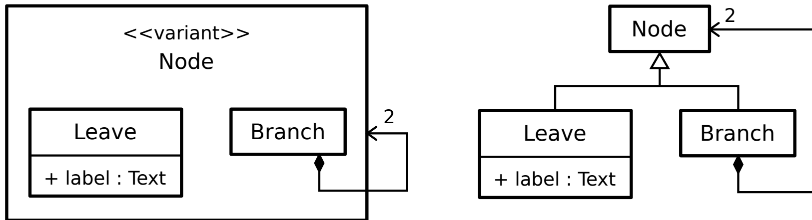


Figure 8.3. Example of a variant type (left) and its equivalent utilizing the composite pattern [GHJ+97, p. 163] (right)

The translation of type names may require more effort, depending on namespaces, character ranges, and naming conventions. First, when the source language includes some sort of namespace or packaging hierarchy while the target language does not provide such concepts, then unique type names must be constructed for the target language. Second, type names of the source language can use other character sets than the target language and may be case insensitive. This requires the translation of characters. And third, to support human readability of code in the target language, it is generally helpful to support the naming conventions of the target language. In GECO, we recommend to realize all three name mapping requirements with a name resolver module, as described in Section 8.1.

Variant types Variant types are realized as tagged or untagged unions and occur in some programming languages. They might also be used in DSLs, as they provide a simple alternative to inheritance. However, a target language might not have a variant type construct, which is the case with Java. Therefore, we propose a mapping based on inheritance to realize such variant types in OOP languages which do not support variant types. Our solution uses one class type for every contained type of the variant type and one common abstract class or interface for the variant type itself. For example, in Figure 8.3 a variant type Node is depicted on the left side. It allows to formulate tree structures where each leave can contain a text label. On the right side a solution based on class inheritance is depicted. The class Node has two children representing the inner types of the union. However,

if an inner type is used in multiple unions, then this would require multi-inheritance. In Java this can be achieved by realizing Node as an marker interface.

A realization of a variant type in a target language without inheritance cannot use this approach. One option would be to create a record structure where every inner type is present in one attribute. However, this increases memory consumption significantly. Another option is to merge types and create one type with all attributes of the variant type. However, this is only beneficial if different types use similar attributes.

For example, if one record type contains two string properties for givenname and surname, and another type has one string property and one integer property, named productname and amount, then the resulting record structure can use three properties to represent both types, namely givenname_productname, surname, and amount.

As this example illustrates, such a resolving strategy leads to complicated attribute names in types and increases the memory footprint. In addition, the readability suffers. Therefore, it depends on the context whether one of the two presented solutions is helpful. In any case, generator and DSL users must be made aware of the memory footprint of the chosen approach.

8.2.3 Enumeration Types

Enumerations can be seen as a special case of variant types in type systems (see Section 2.2.6). From a transformation point of view, however, they represent sets of nominal values, which can be defined by the programmer. As most modern programming languages support enumerations, source language enumerations can be mapped directly to the target language equivalent, if they only support value sets.

Enumeration typing can also support inheritance. For example, in MENGES enumeration types can inherit enumeration values from other enumeration types. Such inheritance requires the aggregation of the values over all inherited enumerations and the additional values of the new enumeration type. The aggregate is then used to construct an enumeration type in the target language. This solution constructs, therefore, one target type for every source type eliminating subtyping.

However, this approach has the limitation that the target level enumerations are not subtypes of each other. Therefore, the generated code cannot make use of subtyping features, such as passing a variable of an enumeration type to an operation with a parameter of a super type. To circumvent this limitation, enumerations can be realized through a set of constants, and the representation of an enumeration type as an integer type. Constants and values are the usual approach used by compilers to represent enumerations. The downside of this second approach is a reduced code readability and the lack of type checking in the code. While this is not a problem in compilers, it can be a problem for code generators, when developers are allowed to alter and extend generated code.

8.2.4 Arrays, Maps and Collections

Arrays allow to store multiple values of the same type in one variable. They have a fixed size and a defined lower and upper bound (see Section 2.2.9). Collection and map types, on the contrary, have a variable size. There are languages which offer a special array type with variable size, which they provide with the same syntax as fixed size arrays. However, the semantics of these arrays are still collections and maps, like in PHP [Cas99] where all arrays are maps.

Arrays In target languages, like C and Java, arrays have a lower bound of 0, whereas in Fortran [ANS78] the default lower bound is 1. However, in Fortran it is allowed to define any arbitrary number for the lower and upper bound, as long as the lower bound is smaller than the upper bound, i.e., $lbound < ubound$ with $lbound, ubound \in \mathbb{Z}$.

Source languages, like those in the MENGES case study (see Section 9.4), may also define arrays with arbitrary bounds or bounds which differ from the target language.

In case of matching bound properties of source and target array types, the mapping can be direct without any additional considerations. However, when the source language must be mapped to a target language with a fixed lower bound, the index must be mapped accordingly. For example,

in a target language with a lower bound of 0, the relationship of the array index is: $index_{target} = index_{source} - lbound_{source}$.

In this case, arrays can still use a direct mapping of types. However, the transformation must also calculate the size of the array, and the offset value. The offset value is used in the expression module to calculate the correct index values for every array field access.

Maps and collections In contrast to arrays, maps are expressed as key and value pairs, where the key and the value have other predefined types (cf. Section 2.3.2). A collection is then only a map with a numerical key which must be consecutive and have an order. The semantics of collections and maps differ from arrays:

- ▻ As elements can always be appended to a collection and map, they have no upper bound.
- ▻ In special cases, they provide insert and remove operations which cause automatic renumbering of entries of collections, while in maps, the key does not imply any order.

Programming languages, like C and `Java`, do not provide a collection and a map type. Instead they are realized by API functionality. In `Java`, collections and maps are implemented with user types implementing the `Collection<T>` and `Map<K,V>` interfaces.

Therefore, collections and maps of a DSL must be mapped to corresponding classes. For the type mapping, this is often possible with minimal effort, as key and value types can be specified with generics. In languages which do not provide classes, like C, collections require special runtime API depending on the implementation strategy. Possible implementations are linked lists with one or two pointers to realize the linking, a dynamically allocated array which can then be resized appropriately. For the typing module of a fragment, collections can then be mapped to pointer types of the collection value type. Maps can be mapped to two collections, one for the values and one for the keys, whereas each key has a reference to a value. While we provide this discussion to propose an implementation strategy, there are also other solutions possible. The correct solution depends highly

on the use case. Therefore, we do not discuss alternative realizations in this thesis.

8.2.5 Subtyping

Domain-Specific Languages (DSLs) may include subtyping as a feature for their user types. For example, subtyping can be used with enumerations, as explained in Section 8.2.3, in classes, and component types. In general, subtyping is used to specialize and extend types. In Section 2.4, some common subtyping rules have been introduced, which define property inheritance and allow to check the subtype relationship of types. While not specifically menioned, it does not restrict multiple inheritance.

In DSLs, such as the IRL [JHS13], multiple inheritance is used. However, programming languages, like Java, do not provide multiple inheritance for classes. Therefore, it cannot be mapped directly. The code generator for EMF solves this issue by realizing the inheritance via Java interfaces, which provide multiple inheritance in contrast to Java classes. However, this solution is limited to interfaces and does not allow to inherit functionality. Furthermore, imperative programming languages, like C, do not support subtyping at all. Therefore, subtyping must be provided through runtime functionality. In this thesis, we introduce a common approach to realize subtyping with imperative languages which have at least support for references and record types, which are based on the design from Objective-C [App15, p. 10ff].

Subtyping can then be implemented with record types, which comprise three kinds of properties (cf. Objective-C runtime):

1. A reference to the super type, or for multiple inheritance, an array of super types.
2. A dispatch table with operation keys and references to operations which realize the methods of the type.
3. The attributes and references implementing the properties of the type.

This record structure is complemented by generated runtime functions for each operation, which comprise at least two parameters, one for the in-

stance reference and one to identify the operation (cf. [App15, p. 10]). These functions are used at runtime to dynamically find the correct operation and execute it.

8.3 Mapping of Expression Semantics

We introduced the metamodeling of expressions in Section 6.6. Their mappings from a source metamodel to a target metamodel are introduced in this section, addressing syntactic and semantic properties.

The expression metamodel has references to the typing metamodel, as expressions make use of types and declarations containing types, e.g., variable declarations. Therefore, a transformation must be able to refer to these declarations on target level, which results in a similar relationship, like between aspect and base metamodel (see Section 7.2).

To be able to resolve references on target level, a fragment module requires a resolving component. For model-to-model transformation this can be realized through a trace model, while for model-to-text transformations, this is best realized through the name resolver.

Based on this general consideration, we describe the mapping of expressions in the remaining section. We first discuss literals, followed by an introduction of a mapping for properties, variables, constants, operations, and operators, which we introduced in Section 6.6.

8.3.1 Literals

In type theory [Pie02] literals can be seen as elements of the value sets which define base types. In some languages, literals can also be defined by the user in form of constants, and it is possible to construct comprised literals for structured types.

The mapping of literals depends largely on the kind of literal. Numbers are usually represented in the same way on source and target level. Nevertheless, there are often small differences. In some source metamodels and languages, the type of a value is determined by the context it is used in. Alternatively, the value has a general type, like `Number`, which is then

automatically casted to the correct variable or value type in the context they are used.

```
private def CharSequence createLiteral(Literal literal) {
    switch (literal) {
        IntLiteral: '''«literal.value»«if (literal.getRequiredType.name.equals('long')) 'L'»'''
        FloatLiteral: '''«literal.value»«if (literal.getRequiredType.name.equals('float')) 'f'»'''
        BooleanLiteral: '''«if (literal.value) 'true' else 'false'»'''
        ConstantLiteral: '''«literal.value.name»'''
        BuiltInValueLiteral case "KIEKER_VERSION".equals(literal.value):
            '''kieker.common.util.Version.getVERSION()'''
        StringLiteral case literal.getRequiredType.name.equals('string'): '''"«literal.value»"'''
        StringLiteral case literal.getRequiredType.name.equals('char'): '\'' + literal.value + '\''
        ArrayLiteral: '''{ «literal.literals.map[element | element.createLiteral].join(if (literal.
                literals.get(0) instanceof ArrayLiteral) ",\n" else ", ")» }'''
        default: 'ERROR ' + literal.class.name
    }
}
```

Listing 8.2. Construction of Java literals in the IRL generator

For example, in the IRL, the target language type is computed by the method `getRequiredType` of the type resolution module (see Listing 8.2). The method semantically checks the context of the literal and returns the type of the literal. In Listing 8.2, the type of the context is resolved to establish if an `L` or `f` postfix is necessary to indicate long and float values, instead of int and double values. Furthermore, the method provides an example for handling built-in named literals, which must be mapped to constants of the runtime API, in this case the `getVERSION()` method of Kieker [HWH12].

8.3.2 Properties, Variables, and Constants

Properties refer to both, declarations of fields and attributes in classes and records, as well as to parameters of methods and functions. Constants and variables are declarations, which can be defined in a global or local scope. Global scope refers to declarations which can be seen everywhere in a model. And local scope refers to the body of functions, methods, and bodies of loops and decision constructs (cf. [Pie02; EEK+12]).

All these declarations comprise usually a typing expression and a unique identifier. This uniqueness can either be in a global or local scope. In the

latter, the uniqueness is defined recursively over all parent scopes. In most languages and metamodels the identifier is realized as name, however, in other cases they are numbers or addresses.

Such declarations are used in expressions, for example to read from or write to a variable. In the metamodel, the use of declarations are expressed with a reference to the declaration. In a transformation these references must be mapped to target level counterparts. Therefore, a trace model can be used for model-to-model transformations or a name resolver for model-to-text transformations.

Depending on the type of the declaration and the representation of its type on target level, declaration must be mapped differently. For example, a variable declaration is typed on source level as `date` and the target language is `Java`, which does not have a base type `date`, then the transformation must rely on a structured type. In this case it is `java.sql.Timestamp`. While on source level it would be possible to write `var date v; v = v + 1`, on target level this must be `Timestamp v; v.setTime(v.getTime()+ 1);`. As this example shows, the declaration of variable `v` is referenced twice, one time as variable and one time as part of the expression calculating the assignment value. While this can be modeled both times as reference to the declaration, on target level there is a specific distinction between read and write operations. In essence, the variable accesses are mapped to function calls. Therefore, the syntactical context of a reference must be considered in the transformation. If the example used `int` as type, the target and source level expressions would look identical. Therefore, a transformation must be able to distinguish between different type realizations on target level.

8.3.3 Function and Method Calls

DSLs and metamodel expressions may include classes to specify the invocation of an operation. Depending on the paradigm and language, operations are named differently. Most modern languages call them function, method, or procedure. In the Programmable Logic Controller (PLC) language standard [IEC03], operations can be represented as function blocks.

A call to an operation comprises usually a reference to the declaration of the operation and a set of parameter assignments, which are also ex-

pressions. The mapping on target languages which support functions and methods, is straightforward, as their operation semantic is often identical. However, in case there are differences between the source and target language semantics, we introduce three techniques covering mapping scenarios of our evaluation scenarios. They are mappings of object-oriented methods to functions, arbitrary sequence of parameters, and operations with multiple input and output values.

Mapping methods to functions Mapping class methods to functions of an imperative language is a common challenge for programming languages, like in Objective-C [App15] and the MENGES project [GHH+12]. For example, in MENGES, the target language only provides constructs which resemble functions and procedures, while the source language includes subtyping. The mapping of methods for objects to functions is usually achieved by adding an additional first parameter to a function. This new first parameter is then used to pass a reference to the object data. Python [Ros95], despite being an object-oriented language, uses this pattern. In Objective-C [App15], the compiler uses the same strategy for methods.

Arbitrary parameter sequence In some source metamodels, like the Palladio Component Model (PCM), parameters are named and can be referred to by name, and they are used to describe input and output. The use of named parameter assignments allows to specify parameter values in an arbitrary sequence, which must be ordered to be usable on target level. This can be achieved by reordering the expressions on target level. However, if the source language allows to implement side effects, the reordering can change the semantics. To avoid this, the named parameter assignments can be mapped to assignment expressions or statements, and the call can then use the assigned values.

Multiple output value DSLs may allow to define multiple output values for an operation. For example, the PCM and the Common Object Request Broker Architecture (CORBA) standard support multiple input and output values. However, multiple output values are often not covered by function

and method semantics of programming languages, which only allow one return value. Therefore, transformation developers require a technique to realize such source level syntax and semantics on target level.

There are many different solutions to this challenge, which depend on the capabilities of the target language. For example, in C output parameters can be represented as pointer types of the basic data type used for the parameter. However, in Java this is not possible for base types, but could be emulated with boxing types. Another option is to define record types and classes with a set of attributes representing the different output parameters, which can then be used as return type. Some languages even support value set and tuple types to specify return types, which allow to construct a multi-value output.

8.3.4 Operators

Operators are a key feature in languages to formulate mathematical, logical and other expressions. They have one or more operands which they combine to a result. In essence they can be interpreted as functions with specific semantics for the combination of operands. The semantic is further refined by the types of the operands. For example, the + operator represents the addition of two or more values.

Some programming languages support operator overloading, which is a redefinition of the semantics of an operator depending on its context. In Java the + operator is overloaded for expressions with String type. For Strings, the + represents concatenation and not a numerical addition. Other languages, like C++, allow to introduce operator overloadings by the developer. This can be helpful if new data types, like a vector type, are introduced which should also be able to use the + operator.

To map operators from the source level to the target level, a semantically equivalent operator must be found. For scalar types and mathematical operators, like addition and multiplication, this is usually not complicated, as programming languages provide such operators. However, for other types of a DSL and metamodel, there might be no equivalent in the target language. In this case the operator must be provided by a runtime function.

Diverging value sets In case the source level type has a different range of values and overflow semantics than the target level type. The transformation must replace the operator with a proper function emulating the semantics.

For example, a DSL defines a byte type (8 bit unsigned integer). However, the target language Java does only support signed integers. The type is, therefore, mapped to short an 16 bit signed integer. An increment operator of a byte value of 255 would lead in the semantics of the DSL either to an overflow exception or 0 as the value wraps over. However, a direct mapping of the increment operator to the target language operator would lead to 256, as the used short type does not have a boundary at 255. 256 is an improper value for a byte typed value. Such illegal states must be prohibited. Therefore, a direct mapping of operators is not possible in this example and the increment must be realized with a function. Listing 8.3 shows an example method which realizes the proper semantics for an byte increment.

```
public short increment(short value) {
    value++;
    if (value > UPPER_BOUND) // UPPER_BOUND = 255
        throw new OverflowException();
    else
        return value;
}
```

Listing 8.3. Example increment method used to ensure correct type overflow semantics for byte

Runtime functions A DSL may define an operator for a type which does not have an equivalent in the target language. This can be operators for base types which are not present in the target language, like ∗∗ as power operator, and operators for source level types, which have no equivalent in the target language.

For example, a DSL supports vector additions with var vec x,y,z; z = x + y. In this language vec is a base type. However, the target language represents the base type vec with a record type vec. The addition operation is then translated to a runtime function invocation by the expression module.

In both cases, boundary checks and runtime functions for operators, it is important to define the semantics of operators precisely to be able to decide whether a direct mapping of operators is possible or if a runtime functionality is necessary.

8.4 Model Traceability

We introduced the term model traceability in Section 7.4, where it was defined as a relationship between nodes of two models where one is derived from the other [Jou05; ANR+06].

On a technical level, traceability is realized by a function which maps model nodes of the original model to nodes of the derived model. These relationships are called model traces. In this section, we explain different features of trace models and the design of trace models as required by GECO.

8.4.1 Features of Trace Models

Basically, a trace model is a relation of the node sets of a source and target model, i.e., $TRM \subseteq P(N_S) \times P(N_T)$ with the source and target model node sets N_S and N_T, respectively. This general definition allows to relate specific subgraphs of the source and target model to each other.

The key function of trace models is to provide the relationship of source and target nodes, which can be queried to return the associated source or target nodes for a given target or source node, respectively. As source and target models are typed structures (see Section 3.1), the nodes can be queried by node identifier, node reference, node attributes, and node type. To improve performance, the flat tuple base design of trace models can be divided by source and target types. Furthermore, in case of runtime models [HJS+15] other groupings and constraints might be associated with trace models.

In context of code generators, subgraph relations are not necessary for two reasons. First, the relationship expressed in the traces does not need to provide context information. And second, source and target model are self-

contained. Therefore, their respective containment graphs already allow to define subgraphs which can be identified by their root node. For example, a source model structure, such as a component declaration in a source model, is transformed into a target model structure, such as a class. The parts of the component are usually contained in the component which can directly be mapped to a class and its containment on target level (cf. [GL13]).

This containment property of models allows to define a trace model as a relation over source and target model node sets, i.e., $TRM \subseteq N_S \times N_T$. Without further restrictions, this allows to relate multiple target model nodes (N_T) to one source model node (N_S), and multiple source model nodes to one target model node. In GECO the relationship is usually one source model node to one or more target model nodes (see Section 7.3), i.e., $TRM \subseteq N_S \times P(N_T)$ where each $n_s \in N_S$ appears only once in the relation and all subsets over N_T are mutually distinct.

8.4.2 Realization of Trace Models

In GECO trace models are used to resolve reference destinations and origins on the target level based on source level nodes (see Chapter 7). Reverse lookup is not used in GECO, as the transformations have access to the source level nodes during generation. Based on these considerations we derive three requirements for trace models:

1. It must be possible to find all target model nodes related to a given source model node.

2. Such lookups are necessary for every join point in an aspect model and inside fragments, e.g., to resolve type declarations in expressions. Therefore, the operation should be fast to avoid long running code and model generation.

3. During model generation, new target model nodes are created iteratively. Therefore, it must be possible to add single tuples to the trace model.

Based on these requirements, we can define a minimal interface for trace models comprising an add and a lookup operation (see Listing 8.4). The first two methods, `add` and `lookup`, realize the minimal interface of a trace

```
interface ITraceModelProvider<S, T> {
    def void add(S source, T target)

    def Iterable<T> lookup(S source)

    def <V extends T> Iterable<V> lookup(S source, Class<V> clazz)
}
```

Listing 8.4. Minimal interface declaration for a trace model handler expressed in Xtend

model provider. The last method allows to add an additional constraint to the lookup query, limiting the result to those elements which are of type `V`. This allows to limit the result set to a specific type of target nodes. For example, a trace model may contain a tuple with a source node representing a record field, like in the IRL. For the target level, the tuple contains three nodes representing a class property, a setter, and getter for the property. For weaving purposes, we only want to refer to the methods. Then we can use the last method to return all nodes which conform to the method declaration type.

Part III

Evaluation

Chapter 9

Experiment Design

The GECO approach supports development, evolvability, and reuseability of generators by providing methods and procedures to create and change generators and fragments, and support their reuse by modularization.

The evaluation of GECO must investigate whether this claim can be sustained and whether such an approach is relevant to the industry. Therefore, we use a twofold evaluation approach combining a quantitative experiemtnal evaluation of GECO based on two case studies, and a complementary qualitative survey based on expert interviews to determine the relevance of the solution and to acquire insight into the potential adaptation of GECO in the industry.

In Section 9.1, we introduce the main goals of the qualitative and quantitative evaluation, and how they relate to questions and metrics. Section 9.2 discusses the quantitative evaluation based on the experiments. Section 9.3 explains the qualitative evaluation based on expert interviews. Finally, Section 9.4 introduces the two case studies used in the quantitative evaluation.

9.1 Assessment of Approach Qualities

Presently, there exist only few approaches addressing generator construction (cf. Chapter 12). They do not address evolution and reuse explicitly, focus on composing transformations, and are not embedded in aspect-oriented and view-based modeling [MJ13].

While Mehmood et al. [MJ13] state that research in the areas of generator construction, evolution and reuse is required, there exists no evidence how relevant these areas are in industry.

Therefore, we evaluate GECO with a twofold evaluation approach, where the relevance is investigated with expert interviews conducted with experts from the industry. And the feasibility and cost effectiveness are addressed by case study experiments. The interviews are, additionally, used to provide a qualitative view on the key goals of GECO and MDE use in the industry.

The complete evaluation is planned and executed based on the GQM approach [BCR94; SB99]. The goals of the evaluation, described below, reflect the three perspectives: construction and development, evolution, and reuse of generators.

1. **Construction and development** focuses on the path to initial generator development and its implementation. Covering the conception of DSLs and metamodels, and their semantics. While GECO assumes an iterative development approach, this might not be the case in the industry. Therefore, it is important to understand the development process of new metamodels, the challenges which may arise in the development of generators and metamodels, and evaluate how GECO may mitigate these issues.

2. **Evolution** overlaps with iterative development. However, in an iterative or agile development process, changes are considered small in comparison with alterations in evolution, e.g., changes in the target platform. For the broader scope, covered by the interviews, it is important to understand what the drivers behind requirement changes are, and how they affect metamodels and their semantics. From a quantitative point of view, we need to know how GECO affects and supports the evolution of generators.

3. **Reuse** of software is fostered by modularization. A property which software reuse shares with the previous two perspectives. In addition, reuseability is alleviated by generic modules which can be reused in other contexts with little or no modification. To determine the significance of reuseability for a generator composition approach, the interviews address reuseability from an industry perspective.

Based on these three perspectives, we define three GQM goals with three

typical stakeholders in the context of generator development: software developers, software architects, and project management.

Goal G1 *Determine* the effect of GECO on the *utility and program quality* from the viewpoint of *software architects, developers and project management*.

Goal G2 *Evaluate* the effect of GECO on the *evolvability* from the viewpoint of *software architects and developers*.

Goal G3 *Evaluate* the effect of GECO on the *reusability* from the viewpoint of *software architects and developers*.

Developers are relevant stakeholders, because they design and implement generators and are directly responsible for the inner structure of modules and fragments. Software architects have a more abstract view on generators and are responsible for their composition out of smaller generator fragments and modules. They are also involved in the development of the syntax and semantics of metamodels. The third group of stakeholders represent project management and product owners. They have direct contact to customers and are the source of new or altered requirements into a generator development and evolution project. From their point of view cost and resource utilization are of great importance.

9.2 Quantitative Evaluation

Beside the qualitative evaluation of GECO to determine the view of experts on the approach and its usefulness in industry, the key features of GECO must also be evaluated. This second part of the evaluation is based on two case studies originating from information system and control system domain. These case studies address the construction and evolution of code generators. The case studies are preceded by a prototypical realization of a framework and tooling supporting GECO, and an exemplary aspect language used in the case study evaluation.

The remainder of this section addresses the overall GQM plan comprising a mapping of goals to metrics in Section 9.2.1. And the mapping of artifacts to hypergraph and graphs later used to compute various measures, e.g.,

size, complexity, cohesion, and coupling. The mapping for Java code is discussed in Section 9.2.5 and for megamodels in Section 9.2.6.

9.2.1 Evaluation Plan

Section 9.1 describes three perspectives for the evaluation of GECO and defines three goals upon these perspectives. The first and the second goal address the development and evolution of generators and generator fragments, and are directly covered by the scenarios carried out in the case studies. However, the third goal – reuse – is not directly addressed by the case studies and only answered indirectly by the quantitative evaluation.

Therefore, this evaluation concentrates on two main goals addressing development and evolution based on the two case studies, showing to what extend GECO is feasible and increases efficiency.

Goal 1 addresses the utility of GECO for the development and the program quality of generators.

The utility of the approach comprises the effort put into the development and the ability to divide work in smaller portions which can be developed independently. Effort is defined as the development time in relation to implemented features, which results in the following question.

Question 1.1 What is the effort spent on the development of features?

The division of work by modularization allows to parallelize and distribute design and implementation. Furthermore, modularization allows to separate different concerns and encapsulate functionality [GHJ+97; ISO11]. Therefore, modularity is essential for the utility of GECO during the development of generators. The corresponding question is:

Question 1.2 How does the modularity differ between the different generator realizations based on the case studies?

The aspect of program quality is twofold and includes runtime properties, such as stability and execution time, and design time properties, like code quality and understandability [Lai96]. To evaluate execution time and runtime stability, a fast set of models and DSL artifacts would be required

to provide a solid test basis for the resulting generators. Such artifacts, however, are not available for both case studies. Furthermore, the runtime stability is largely affected by the code quality and reflects programmer skills as much as potential gains achieved by the approach.

Code quality also reflects the ability of the programmer. A modularization approach, such as GECO, results usually in moderate module sizes, which improves the ability to understand the code [Lai96]. This can result in better code quality, for example, because changes can be developed without harming intended functionality of the original code. Therefore, code quality metrics do not provide an untainted view on the program quality. However, the overall complexity of a software system also affects understandability [Lai96]. Therefore, we focus on this coarse-grained view on understandability in the following question:

Question 1.3 What are the differences in understandability between the different generator realizations based on the case studies?

Goal 2 addresses the evolution of generators and how the different approaches, GECO and the original one, affect the evolution of the generator.

Evolution of source and target metamodels and their semantics can cause changes to generators and fragments. Sources of generator evolution are API changes in frameworks used to implement metamodels and generators, the need for optimization of the generated code (cf. [RLL98]), and the realization of new semantics introduced by metamodel and DSL modifications. As in any software system, repeated modification of artifacts can cause their degradation, resulting in a higher entanglement of modules, which might have a negative impact on the evolvability of generators [Koz11].

Derived from Rowe et al. [RLL98] and Koziolek [Koz11], evolvability requires adaptability, scalability, extensibility, maintainability, and generality. However, this decomposition of evolvability contains subcategories which are not suitable for our case studies, as they do not apply to generators.

Adaptability [ISO11, p. 15] and scalability are runtime properties which play a prominent role in enterprise software systems. Adaptability addresses the ability to modifify the behavior of software at runtime which is undesirable in code generation.

Scalability describes the ability of a software system to handle changing workload intensity [ISO11, p. 15]. However, the artifacts used as input of code generators are limited in size, due to understandability constraints for the artifacts.

Generality describes the *ability of components to provide functionality beyond the actual required features* [RLL98]. In generators such anticipation of future requirements results in additional complexity and is usually avoided (cf. [MHS05]). Therefore, maintainability and extendability are the remaining subcategories to assess evolvability.

ISO 25010 [ISO11] provides a decomposition of maintainability into several subcategories. The first and foremost is modularity [ISO11, p. 14] which has received a lot of attention in research and industry (cf. [Koz11]). For example, Gamma et al. [GHJ+97] describes modularity as essential for modifiability and reuse of software components [Has02]. In ISO 25010 modularity is defined as the

> degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. [ISO11, p. 14]

In context of the evaluation of GECO, the fully assembled generator is the system, and its fragments and parts are the components of the system. Therefore, we ask about changes in the modularity during evolution.

Question 2.1 How does the modularity change during the evolution steps in the different scenarios?

The second subcategory, relevant for the evolution of GECO is analyzability, which is defined as the

> degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts [...]. [ISO11, p. 15]

The analyzability is mainly affected by the understandability of code and artifacts [Lai96], program [FFH+15] and systems comprehension [FKH15], and the overall complexity and coupling of the system. Therefore, we ask:

Question 2.2 How does the overall understandability change during the evolution steps in the different scenarios?

The third subcategory, used in this evaluation is modifiability, which is defined in the ISO 25010 standard as the

> degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality. [ISO11, p. 15]

It is affected by modularity and analyzability which are addressed above. Good modularization reduces the dependencies to other components which limits the potential negative effect of a modification. And the ability to analyze artifacts increases the understanding of the effect of a modification.

ISO 25010 states that modifiability is a combination of changeability and stability [ISO11, p. 15]. Where changeability is

> [t]he capability of the software product to enable a specified modification to be implemented. [IS091, p. 10]

And stability focuses on the unaffectedness of other parts which are not modified. ISO 9126 states

> The capability of the software product to avoid unexpected effects from modifications of the software. [IS091, p. 10f]

Changeability also covers extensibility, as it comprises any kind of change including adding, extending, altering and removing functionality. These two subcategories are reflected in the following two questions:

Question 2.3 How does changeability develop during evolution steps in the different scenarios?

Question 2.4 How does stability change during evolution steps in the different scenarios?

Goal 3 addresses reuse which cannot be evaluated directly based on the two case studies, as both do not provide a scenario that includes reuse of generator parts previously developed.

However, reusability is affected by qualities like modularity and separation of concern, which support the extraction of modules of one software system and the introduction into another system [ISO11]. In ISO 25010, reusability is described as the

> degree to which an asset can be used in more than one system, or in building other assets. [ISO11]

To be able to reuse a module, it might require modifications. Therefore, modifiability and modularity, which determine evolution, have also impact on the reusability. However, reusability is also affected by portability [ISO11, p. 15] and the generality of the module. In context of generators, this might be a functionality often used, like the mapping of expressions to Java code, which is covered by the grammar and generator fragment Xbase [EEK+12]. However, reuse is not covered by the two case studies in terms of a scenario where parts of a generator from one project are reused in another project. Therefore, only modifiability and modularity can be determined, which result from an indirect assessment of reusability. Due to these limitations, we evaluate reusability only in a qualitative manner based on the interviews.

9.2.2 Quality Categories

In the previous section, seven questions have been defined that rely on a set of quality categories which can be measured by the attributes effort, modularity, understandability, changeability, and stability.

In the following, we define these metrics based on a set of direct measures which can be directly measured or computed based on artifacts of the observed system.

Effort Effort is measured in time, especially hours or days of work. In this thesis, we are able to measure our own work effort in great detail, but effort required to implement and evolve the original system can only be determined in work days, based on commits to the Version Control System (VCS) and company logs. Therefore, effort is measured in both cases in *days of work*. To be able to relate the effort of the original system and the simulated reimplementation based on GECO, distinctive feature sets are

extracted from project documentation and the corresponding versions from the VCS. This results in days of work per feature values ordered by the date when the feature was implemented.

Modularity The modularity of a software system is determined by the basic metrics cohesion, complexity and coupling [All02; AGG07]. Good modularity of a system is indicated by high inner cohesion and low inter-module coupling. Furthermore, the overall complexity of the system represents the upper bound for inter-module coupling. A lower ratio of coupling and complexity indicates a better separation of concerns through modules. Modules correspond in GECO with generator fragments and parts.

The values of cohesion, complexity and coupling are affected by the overall size of the system and the complexity of the requirements realized in each evolution step. Therefore, it is impossible to determine a ranking of cohesion and coupling values over all potential software systems, classifying certain values as strong and weak. Instead, we measure cohesion and coupling for each given generator revision before and after alterations, and determine whether the alterations affected both values in any way. Furthermore, we compare the measurements between different generators implementing the same features over time.

Understandability Understandability of code is affected by many different factors which include writing and reading skills of the programmer, programming language, and naming patterns [Lai96]. However, these factors are not affected by GECO, because programmer skills and naming patterns are outside the scope of GECO, and GECO is a technology agnostic approach. Therefore, evaluating these factors does not provide the necessary insight in the feasibility and practicability of the approach. Understandability is also affected by the complexity of artifacts and the code implementing a generator. Laitinen [Lai96] expresses understandability as inverse proportional to complexity, as depicted in Equation (9.1).

$$Understandability(S) = \frac{1}{Complexity(S)} \tag{9.1}$$

Changeability is described in ISO 9126 as

> the capability of the software product to enable a specified modification to be implemented. [IS091, p. 10]

This is a rather vague statement, as *enabling* modification, can imply any number of structures, methods, and approaches. In Grover et al. [GKK08], changeability of aspect-oriented systems is induced by potential changes based on component and system level. They measure the impact of changes over the number of actual changes and the affected operations, e.g., methods. In their approach, changeability is good when a set of changes had a low impact. However, we require an architecture-level approach applicable to implemented revisions of generators, as not every change is recorded, especially when commits to the VCS are postponed by several days. Likewise, this change-based metric evaluates the impact of changes, but we need to know how the ability to change between generator revisions was affected.

In Tsui et al. [TK09, p. 219ff], cohesion and coupling are mentioned as metrics to characterize changeability. Cohesion is considered to be high, if it represents exactly one task of a problem domain and all its elements contribute to this single task [YC79]. This interpretation is further explained in Kabaili et al. [KKL01] who describe cohesion by the interconnection of methods through calls and common variable access. Furthermore, low coupling results in low dependencies to neighboring components and modules, and therefore, limit the impact of a change to other parts of a software system [AKK+99].

Stability is the last quality category which must be mapped to direct measures. Stability is the other side of the coin of modifiability, and follows from the same argumentation as changeability. Where changeability addresses the ability to change code, stability addresses the unaffectedness of other parts of the code by changes. In ISO 9126, stability is defined as

> The capability of the software product to avoid unexpected effects from modifications of the software. [IS091, p. 10]

This can be achieved by narrow interfaces of components and modules, which results in low coupling, following the argumentation of limited

impact of changes due to low coupling stated in [AKK+99].

9.2.3 Measuring Complexity, Coupling and Cohesion

The quality categories rely on the measurement of specific metrics, namely, size, complexity, coupling and cohesion. For the generator code we rely on the hypergraph abstraction of Java code from Section 9.2.5. Furthermore, we provide an complexity analysis of the megamodel (see Section 9.2.6) created for the new generators in both case studies. Based on the hypergraphs, we determine the measurements for system size, complexity, coupling, and cohesion.

9.2.4 Method Complexity

The metrics on the architecture level provide information of the size, complexity, coupling, and cohesion of a generator and megamodel, but they ignore the complexity of methods. This would allow to move complexity from the higher level of abstraction into methods to hide them. To be able to detect such complexity transfer and, therefore, avoid this thread of validity, we also measure the complexity of methods. While the metrics of Allen et al. [AGG07] address architecture and component level evaluations and are not designed to model method complexity, we utilize cyclomatic complexity metric [McC76] to target method complexity.

There have been concerns regarding the applicability of cyclomatic complexity of source code [She88]. We are aware of these limitations. However, we only apply the McCabe metric to methods to ensure that the complexity of the methods does not increase in favor to hide complexity from the entropy based metric used for the modular structure of the generator.

There is a general categorization available for values of cyclomatic complexity defining values below 5 as good, up to 7 acceptable and values higher than 10 as problematic [WM96]. Also in industry tools, like Checkstyle,[1] this categorization is used to identify good and too complex methods. To detect a complexity shift, we ignore this coarse-grained subdivision and

[1] Checkstyle `http://checkstyle.sourceforge.net/`

define groups for every complexity. For the analysis we can then compare the histograms of two generator revisions to identify a complexity transfer.

9.2.5 Java Mapping

The Java type system is very complex due to its multi-inheritance feature, enumerations with methods, and abstract classes, inner classes, anonymous classes, etc. In addition dependency injection frameworks, which resolve interfaces to class instances at runtime, cause additional complexity due to further indirection. Therefore, we present a concise description of a mapping of Java classes, interfaces and methods to modules, nodes, and hyperedges.

We distinguish classes in three groups representing the observed system, the infrastructure, and the data types. All classes to be handled as data types are listed in `data-type-pattern.cfg` file and the observed system classes are noted in `observed-system.cfg`. The remaining classes are considered framework classes. For user's convenience, both files allow to specify patterns instead of naming every single class.

Data Types Java defines several primitive types, like `int`, `short`, and `char`, supplemented by data classes, e.g., `String`, `Date`, and `List`. While these are classes, they serve – in context of a source code analysis – the purpose of data types. Therefore, they must be added to the list of data types. The same applies to all classes in the observed system which are used for the data model, e.g., Java persistence API entity beans must be added to the configuration file.

Classes and Interfaces All classes and interfaces belonging to the observed system are fully analyzed, meaning each class and interface is mapped to a module. All methods are represented by nodes, and data type properties, which are not constants, are mapped to hyperedges.

For classes and interfaces of frameworks and libraries, only those are added to the hypergraph mapping, which are used in the observed system, and only methods accessed by the observed system are mapped to nodes.

All other parts are hidden to the developer and do not contribute to the complexity of the observed system.

The relationships induced by subclassing must also be mapped to the hypergraph. The hypergraph is a flat model without inheritance. Therefore the inheritance must be resolved. This could be done by producing nodes for each inherited method resulting in node duplication. However, developers perceive super method calls like calls to other classes.

Our mapping distinguishes two kinds of hyperedges, with *call edges* for method calls, and *data edges* for access to data type properties. The latter are in fact hyperedges, as they may have more than two participants.

Special Class Type Features Abstract classes, inner classes, and anonymous classes are handled like normal classes. However, abstract methods are handled like method declarations of interfaces, where all derived method implementations are automatically connected to the declaring abstract method via a call edge. Inner and anonymous classes are in fact just classes. Therefore, they are represented as modules, like normal classes. However, they might contain references to the outer class in form of property access. Such property accesses could be represented via a data edge. However, this interpretation would hinder the application of the cohesion metric. Therefore, such direct data access is first mapped to a virtual setter or getter method in the called class, which can connect to the data edge, and the data access origin is the handled like a method call, resulting in a call edge.

9.2.6 Mapping Megamodels

Megamodels [Fav04a] comprise models, metamodels and transformations. In GECO [Jun14b], the notation comprises parts of metamodels identified by their root class, trace metamodels, and transformations. All entities of the megamodel are represented as nodes and the different relationships of metamodels and transformations are represented by edges. As these megamodels do not have a typical modular structure, the created hypergraph is a flat hypergraph without modules, limiting the computation to size and complexity metrics.

9.3 Expert Interviews

Expert interviews are a special case of qualitative interviews used to investigate the knowledge and insight of experts to a specific topic or domain [Smi02]. In case of the evaluation of GECO, the expert interviews are used to determine the views of experts of the DSL and generator development domain. The interviews address the three main perspectives: construction, evolution, and reuse of generators. Furthermore, the interviews are used to determine the initial assumption that generator construction, evolution, and reuse are relevant for the industry.

Before the interviews can be performed, the individuals who are considered experts in MDE and generator development must be determined. Furthermore, the setting for the interview must be defined, which includes the choice between group and individual interview, as well as, short verbal introduction versus a longer presentation.

9.3.1 Subject Selection

The selection of subjects for the interview must relate to the three groups of stakeholders from the GQM goals in Section 9.1. While the goals address the users of GECO, like developers and engineers of generators, additional potential target groups include researchers in the domain of metamodeling and code generation, and project managers of projects utilizing generators.

Developers and engineers of generators, DSLs, and development platforms for software systems are a relevant target group for interviews, as they are the designated users of GECO. They have an unique view on the development of code generators based on their daily experience. Therefore, they could compare the construction method of GECO to their practice and point out any shortcomings of and possible improvements by GECO.

While researchers may lack knowledge on code generation in industry projects, they have a deep understanding of surrounding research and can, therefore, provide insight on the interaction of GECO with metamodeling, DSL development, and other modeling technology. As GECO suggests several guidelines on metamodel creation and tailoring, their understanding of these fields may yield knowledge on benefits by and challenges for GECO.

Finally, project managers using MDE have experience with model-driven methods in their projects. They may have worked as developers which implies they have experience in using and creating generators. In addition they have experience in managing the development of DSLs and generators, which gives them an unique point of view for the approaches regarding the development of generators.

9.3.2 Interview Guide

Before an interview guide can be designed, it is necessary to establish the interview settings. First, we must decide whether we want to interview the subjects individually or in groups. Second, we have to determine the most suitable type of interview. And third, we must decide how to design the introduction.

Individual interviews have the advantage that every interviewee has the undivided attention of the interviewer and are not constricted to speak by other interviewees. This is especially an issue when personal opinions and feelings are involved. Group interviews have the advantage that the interviewees can discuss questions and gain an deeper understanding of the discussed matter which then can be analyzed by the interviewer. An additional advantage is that the interviewer must only introduce the topic of the interview once for each group instead of each individual.

For the evaluation of GECO, where we introduce a new approach to developers and engineers, it is favorable to foster the discussion of the newly introduced approach. Furthermore, the topic does not involve general taboos. Therefore, we favor group interviews.

In literature [Smi02; Kai14], interviews can be unstructured, semi-structured, and structured. An unstructured or narrative interview allows the interviewee talks to one or more topics, while the interviewer is only allowed to ask comprehension questions. A semi-structured interview imposes some structure, like introduction, warm-up, main topic, and closure. However, the sequence of questions can be changed depending on the actual course the interview takes. It is even allowed to skip questions and to ask questions not noted in the interview guide. The third, option are structured interviews. They comprise of a list of questions which must be asked in

the given sequence, as stated in the corresponding interview guide. Beside comprehension questions, the interviewer is not allowed to divert from the outlined course.

In context of the evaluation of GECO, it is necessary to guide the discussion close to the three perspectives of development, evolution, and reuse. However, it is also important to engage in a discussion which usually leads to topic changes during discussion. Therefore, the structured interview is inapplicable. To be able to provide some form of guidance during the interview, we prefer the semi-structured interview, as it provides enough freedom to follow interesting thoughts, but also allows to guide the interviewees back to the relevant topics of the evaluation.

The interview introduction usually comprise a verbal introduction of the interviewer, the topic of the interview, rules for the interview, like, that comprehension questions of the interviewee are allowed and the sequence of statements, and organizational aspects, such as recording, and information processing.

In case of GECO, we extend the introduction by a comprehensive but brief presentation of the approach. This is necessary to introduce the approach which is new and, therefore, unknown to the interviewees.

Based on this assessment, we use semi-structured group interviews with an introductory presentation of the approach. The necessary interview guide for the interviewer is introduced in the following the sections. The complete guide used in the interviews can be found in Appendix A.1.

Introduction The interview introduction starts with the introduction of the interviewer to the interviewees, followed by the motivation of its purpose and goal, and finally elaborates on the course of the interview. This includes informing all interviewees about the assurance of confidentiality on all collected information, and that the interviewee might ask questions for clarification at any time [Smi02; Kai14]. As mentioned before, the introduction is concluded by a presentation of the GECO approach. During the presentation, the interviewees are allowed to ask questions to improve their understanding of the approach.

Warm-up Questions According to Kaiser [Kai14], the warm-up phase is usually used to collect expertise on the interviewees, such as their occupation, experience in the field, and education. However, this is not always applicable in group interviews for two reasons: First, group interviews require the collection of this information from all attendees to understand their background and role in development and evolution. This may require too much time in an industry setting. And second, such questions are an interruption of the usual process of presentation and discussion. Therefore, we aim to gather this information in advance when organizing the appointment.

Additionally, during the warm-up part of the interview, we clarify any remaining misunderstandings according to the GECO approach before moving on to the remaining questions.

Main Questions The main part of the interview focuses on the three perspectives and key research questions selected for the interview. It is important to anticipate which question might be problematic in the organization, and therefore move them more to the end [Kai14]. However, in this merely technical context it is unlikely that certain questions result in the termination of the interview. It is more likely that certain areas are not answered due to company secrets. Therefore, it is primarily important that the questions are not dependent on each other [Smi02].

1. **Construction** For the perspective on construction of generators and the goal on utility of GECO (G1), it is important to understand the context and prerequisites of generator construction. For example, how DSLs are developed and how the process of generator construction look like. Subsequently, the interviewees should reflect on their process and evaluate whether GECO can be beneficial for it and allow to improve it.

2. **Evolution** Goal G2 focuses on evolution. Therefore, the interview must first discuss whether evolution is an issue at all and to what extend. Are generator adaptations a key problem in the evolution or do other issues have a greater impact? Subsequently, it is important to understand the drivers for evolution of generators.

3. **Reuse** Finally, goal G3 focusing on reuse is of great importance for the interviews, as the evaluation based on case studies does not cover reuse. Therefore, it is important to understand whether reuse is relevant for the industry and what the key obstacles are in this area. In addition, it is important to get the expert's impression of GECO as being supportive to mitigate these obstacles.

Cool-off Questions The cool-off part of the interview is intended to reduce the pressure and intensity of the interview. Therefore, the questions must be simple and straightforward [Kai14]. As the interview is merely on a technical subject which does not involve personal and social taboos, an explicit cool-off part of the interview is not required.

Closure The closure is used to conclude the interview. The interviewer thanks the interviewees for their time and insights, and summarizes his or her impression of the interview. Furthermore, the interviewer communicates the next steps, e.g., informing the interviewees about the analysis of the interview and the gained insights. In case of these interviews for GECO, the aggregated results of the interviews are sent to the participants for clearance.

9.4 Case Studies

In this thesis, two case studies, motivated by the domains information systems and embedded systems, are used to evaluate the GECO approach. In Section 9.4.1, the information system case study on basis of the Common Component Modeling Example (CoCoME) is introduced. And in Section 9.4.2, the railway control center DSL from the MENGES project [GHH+12] is used as representative of a DSL for embedded systems.

9.4.1 Information System Case Study

The Common Component Modeling Example (CoCoME) [HGH+15] is an information system resembling the IT of a supermarket chain. The archi-

tecture of CoCoME comprises multiple cash desks per store, multiple stores with a store server and a central enterprise server. It covers typical use cases of software systems and incorporates embedded and enterprise software components. We selected CoCoME, as it is defined as a benchmark for modeling approaches and techniques for component-based systems. It is also used in a German Research Foundation (DFG) priority program for software evolution (SPP 1593) [GRG+15]. CoCoME serves there as a test subject which is changed and extended according so a set of change scenarios. As they required a completely modeled system, we provide the necessary DSLs and generators.

In this case study, we model CoCoME with the Palladio Component Model (PCM) [BKR09], originating from the domain of performance prediction. The PCM allows to model component based software systems including their deployment. However, it lacks the ability to specify executable code. Therefore, we supplemented it with a behavior language. Furthermore, CoCoME requires to specify a data model for persistence which is realized by the Data Type Language (DTL) which allows to define a data model close to the CoCoME design documentation and at the same time supports the features of JPA.

To allow application monitoring, we added two DSLs, called IRL and IAL (see Section 10.4) to specify monitoring events and to express point cuts for sensor placement [JHS13].

For the evaluation, we defined the overall architecture of the generator for CoCoME based on the megamodel patterns depicted in Figure 9.1. We included the compilation of programming code to show how code generation and compilation can be described together with GECO.

The case study, therefore, integrates existing generators, such as ProtoCom [GL13] represented by $T_{ProtoCom}$ with newly written additions for behavior $T_{Behavior}$ and data structures T_{DTL}. They are complemented by generators for instrumentation records and sensors realized with the IRL and IAL [JHS13] following our integrated design-time and runtime modeling approach [HSJ+15] providing a record generator T_{record} and a sensor generator T_{aspect}.

Most of the generators produce Java code conforming to specific APIs and internal DSLs. The two main exceptions are T_{aspect} and $T_{Behavior}$. The

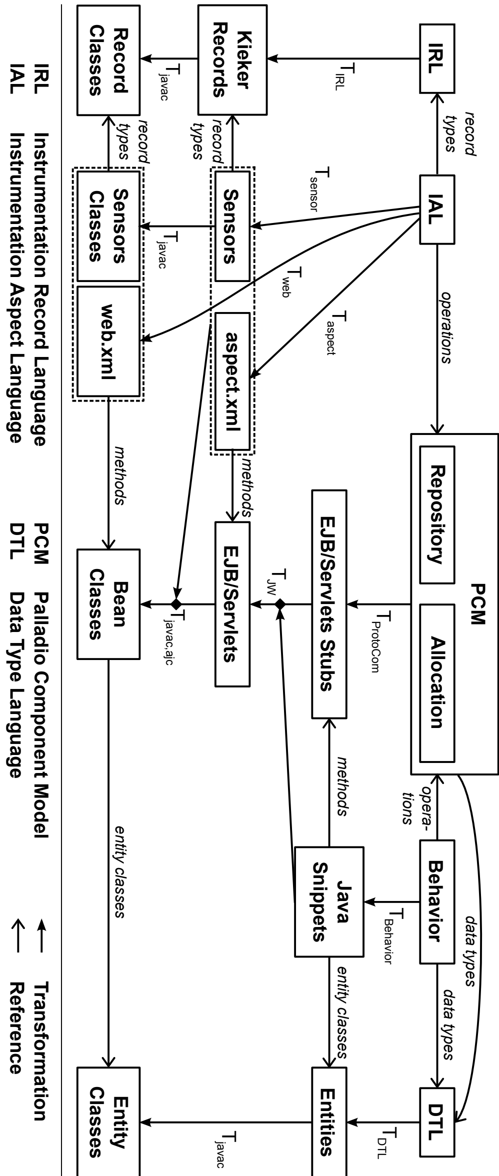


Figure 9.1. The generator megamodel excerpt including the main transformations of the generator. Trace models and their corresponding transformations are omitted.

first produces an XML serialization for interceptor configuration conforming to servlet and Enterprise Java Bean (EJB) APIs (web.xml) and AspectJ configuration (aspect.xml). And the second produces Java snippets which are woven in by T_{JW}. The remaining transformation is T_{javac} representing the Java compiler which represents the compilation of Java classes.

This case study is also an example where the different input and output artifacts are persistent in files and handed over to the next generator fragment with files. This is usually the case when legacy generators are integrated into the new code generators and when the generators are realized with model-to-text transformations.

9.4.2 Control System Case Study

The control system case study is founded on the MENGES project [GHH+12]. In MENGES, a set of DSLs and a corresponding generator have been developed for the domain of railway control centers. These control centers are realized on the basis of PLCs. The DSLs were developed with Xtext [IE11] and the generators with the Xtend transformation and templating language [Ite11]. The DSLs covered the implementation of behavior, simple structures, communication, architecture and configuration. The original generator supports the generation of the PLC language Structured Text (ST) (cf. [IEC03]) stored in an large XML file conforming to the PLCOpen XML specification.[2] Due to the size and shape of the megamodel, we do not include a graphical representation of it in this thesis. However, an interactive model can be viewed with the generator composition tooling (cf. Section 10.2). Furthermore, MENGES was developed in conjunction with industry partners, therefore, we cannot disclose detailed language and generator features. However, we provide a graph-based abstraction of the structure of the resulting generators on the project web page.[3]

The DSLs incorporate subtyping for component and connector types, whereas component types can additionally inherit properties from templates. Connectors allow to transmit values and messages which must adhere to a protocol. The architectural features of the DSLs are complemented

[2]PLCOpen Association http://www.plcopen.org

[3]GECO http://www.oiloftrop.de/geco-approach

by structured data types and enumerations. The behavior is expressed with automata and workflows which are realized as a separate aspect for components. The deployment of components on PLCs and the connection to external ports is covered by a deployment language and a complementing configuration language to define static values.

Each type requires its own transformation which produces model fragments which are integrated in one result model based on the PLCOpen XML Schema Definition (XSD). All structural types are mapped to structured data types represented by elements defined in the XSD. The behavior is realized with ST, and is embedded as text in the model. Therefore, the resulting generator combines model-to-text and model-to-model transformations.

Chapter 10

Prototype Implementations

The generator composition introduced in Chapter 7 and the fragment design approach presented in Chapter 8 are technology independent and can be used with any tooling and framework. However, for the evaluation of the approach it must be used with a concrete set of tools and frameworks. Furthermore, the advices and practices explained in the approach chapters must be applied to a prototype.

In this chapter, we first introduce a basic framework for generator fragments in Section 10.1. Based on this framework, we developed a DSL and generator which supports generator composition. The DSL and generator are documented in Section 10.2. As typing places an important role in any DSL and generator, we introduce a method and library to introduce an arbitrary type system into Xtext in Section 10.3. Finally, we use the typing approach and the generator framework to construct an instrumentation aspect language, which is discussed in Section 10.4.

10.1 Generator Composition API

In Chapter 7 a generator composition approach was introduced. Central element of this approach are generator fragments which have one source model or source model partition as input and one or more target models or partitions thereof as output.

The generator composition API, explained in this chapter, provides an API for generator fragments and weavers which can be used to assemble generators as described in Chapter 7. The API is designed for Java Virtual Machine (JVM) based languages and provides reusable modules for trace models.

In the present chapter, Section 10.1.1 introduces the overall design for the generator composition API. In Section 10.1.2, we describe the generic interface of generator fragments and weavers used in this GECO prototype, including the integration of different model-to-model transformation languages. In Section 10.1.3, we discuss the use of additional model input as a shortcut for supplemental information necessary for model and code generation in a fragment. In Section 10.1.4, we provide a set of wrapper classes which allow to use ATL and QVT transformations with GECO. And finally, we introduce a generic module for trace models and its use in fragments, which we describe in Section 10.1.5.

10.1.1 Framework Design

The framework design covers the requirements induced by the approach for generator fragment composition, as introduced in Chapter 7. Therefore, it focuses on generator fragments and weavers as main building blocks. In addition, we provide a module for trace models, which were discussed in Chapter 8.

Requirements Based on the definition of a fragment, the generator composition, and the inner fragment design, we can derive six requirements the framework must meet.

1. The definition of a fragment states that a fragment has one source model as input and one or more target models as output. All target models of a fragment conform to the same metamodel (Section 8.1).
2. Trace models are required to compute reference destinations, as explained in Section 7.3. They provide at least sets of tuples where one element represents a source model node and the other tuple element represents a target model node (Section 8.4).
3. A fragment may also require trace information of multiple trace models. For example, when types are declared in multiple other models and a fragment generates model or code using these models, then it requires the different trace models generated by the respective fragments. In

context of aspect-oriented modeling, a monitoring aspect may refer to component types and execution environment types, which are handled by different fragments. In this case, these fragments also produce different trace models, which must be used by the aspect fragment. Therefore, a fragment may read multiple trace models.

4. Fragments may only require a subgraph of a model and not the complete model. Therefore, they may use only a model partition as input or even parts of them. For example, a fragment is designed to produce a model for a method of a component type. Instead of using one fragment for the complete component type, it can be helpful to create a fragment to generate code for methods. This fragment would then be invoked for each method. Such a method may refer to other methods of the same component type. However, they might not be accessible via the model hierarchy. For example, in the MENGES case study, method implementations in form of state machines are defined in models separate from method declarations. In this case it is helpful to provide auxiliary models to look up method declarations. While this looks like a violation of the overall approach, it is only a convenient functionality to reduce fragment and megamodel complexity.

   Two alternatives to auxiliary inputs are separate aggregation fragments and aggregation inside the fragment generator. The first alternative aggregates and filters information to prepare all model data for the fragment, which introduces an additional fragment with minimal functionality. This would, subsequently result in a more complex megamodel and would require an additional target metamodel resulting in a general more complex implementation.

   The second alternative implements the aggregation and filtering inside the method making the fragment more complex. Therefore, it would move the complexity into the fragment itself. While this is a suitable option for a scenario where methods are used only in one place, it makes it harder to reuse the method generator fragment in other contexts, e.g., in other types which also implement methods, but are not component types.

   As both alternative solutions have significant downsides, we allow to

add auxiliary input to generator fragments. A developer has to keep in mind that they are only provided to eliminate such aggregation and filter fragments and should not be used to create large arbitrary generator fragments, as this defeats the purpose of GECO.

5. We use Xtend [Ite11] and Java for the generators in the evaluation and experiment. Therefore, the framework is based on in-memory models which are represented in Java objects at runtime. However, there are other EMF based transformation languages available, such as ATL [JK06] and QVT [QVT05]. Therefore, we require to provide adapters for our framework to be able to integrate generators written in these languages.

6. Beside fragments, the approach also mentions weavers. There are already large weaver frameworks and tooling available. For example, the Kermeta weaver [Kra12] and the AMW [FBV06] allows to create the necessary metamodels and construct weavers. However, in many cases weaving can be much simpler and the metamodels for base and aspect models are already present. Therefore, it is necessary to provide a generic interface for weaving, which can then be implemented with any technology, which is compatible with EMF.

Design considerations Based on these requirements, we can derive a basic design for the generator fragment framework. First, we need generic interfaces for generator fragments and weavers which allow to limit fragments and weavers to specific metamodels or metamodel partitions, identified by their root class. Second, we need abstract classes realizing all the boiler plate code for the integration of the Kermeta weaver and AMW. These classes are then specialized by specifying the resource referring to the transformation and declaring which source and target metamodel must be used. Third, we need to handle auxiliary input. This can be done by passing the auxiliary models either to the constructor of a fragment or to setters for each auxiliary model.

The constructor approach would force anyone who assembles fragments to make sure all necessary auxiliary models are passed to the fragment. However, it would also require to instantiate a new fragment every time the set of auxiliary models change. If we required to instantiate a fragment

for every run, we could delegate memory clean up of the generation to the garbage collector. The complete sequence to invoke a generator would then look like:

```
val methodGenerator =
    new GenerateMethod<Statemachine,Method>(componentType)
val resultMethod = methodGenerator.generate(statemachine)
```

The alternative is less restrictive, but requires one setter for each auxiliary model, which makes the configuration of a fragment more verbose. Furthermore, a missing auxiliary model cannot be detected at compile time. However, in case auxiliary models are optional, the constructor approach would either require multiple constructors which would result in losing the type constraint or the developer would require to use `null` as parameter value. Furthermore, when state handling is initialized in the generate call, then the generator can be instantiated once and used multiple times. Especially, when fragments are used to realize smaller parts, like methods, the performance of the overall generator would suffer from many instantiations.

For our framework we suggest to use the setter based approach. It is also the design, the GECO fragment composition language expects. In addition, the setter approach aligns better with an inversion of control design pattern, which can be used to invoke the correct fragment dynamically. Therefore, we assume that auxiliary models are passed by setters, even though we do not enforce this design paradigm in the API.

10.1.2 Common Fragment and Weaver Interfaces

At the core of the framework are two interface declarations defining the common interface of all generators and weavers.

Fragment Interface As explained in Section 8.1, a fragment processes one source model conforming to one source metamodel and outputs one or more target models which conform to a target metamodel. Furthermore, the source and target metamodel can be partitions of larger metamodels. Such partitions must be self-contained metamodels. As each metamodel or metamodel partition is self-contained, it can be identified by its root

class. Therefore, it is sufficient to create a generic interface with two typing variables S and T, as depicted in Listing 10.1.

```
/**
 * @param <S> type of the source metamodel
 * @param <T> type of the target metamodel
 */
public interface IGenerator<S,T> {
    def T generate(S input)
}
```

Listing 10.1. Generic generator fragment interface for model-to-model transformations

Type variable S represents the root class of the source metamodel, while type variable T is for the target metamodel root class. Instead of a root class of a complete metamodel, S and T can also be root classes of metamodel partitions. In model-to-text transformations the target metamodel root class can be replaced by CharSequence or String, as such transformations produce serialized information in form of text.

In case the fragment generates multiple target models, they must be returned in a Collection or Iterator. For example in MENGES, a state machine is transformed into a set of function blocks embedded in a PouType1. Therefore, the fragment for state machines produces a list of PouType1 instances, as depicted in Listing 10.2.

```
public class GenerateStatemachine implements IGenerator<Statemachine,List<PouType1>> {
    [...]
}
```

Listing 10.2. Example of a class declaration with multiple target models, derived from the MENGES case study

Weaver Interfaces A weaver integrates parts of an aspect model, called advices, in a base model based on a set of pointcuts. As introduced in Section 3.5, a weaver has two inputs, the base model and an aspect model, and one output, which is the woven model.

```
/**
 * @param <B> base metamodel
 * @param <A> aspect metamodel
 */
interface IWeaver<B,A> {
    def B weave(B baseModel, A aspectModel)
}

interface IWeaverSeparatePointcut<B,P,A> {
    def B weave(B baseModel, P pointcutModel, A adviceModel)
}
```

Listing 10.3. Generic weaver interfaces for model weaving, supporting an aspect model or alternatively two models for pointcut and advice

Depending on the context, the aspect can be represented in one model containing pointcut and advice, or pointcut and advice are stored in separate models. The first variant is more common with views, which define direct references to the independent view. And the second variant is more common in reusable aspect languages, which have more complex pointcut models. Therefore, we define two interfaces to support both modeling approaches, as depicted in Listing 10.3.

For the framework, we do assume that the input base model is not modified and the result of the weaving is a new model. However, this semantics is not enforced by the framework. In many cases it can be beneficial to just modify the input model resulting in better performance. Either way, the developer should explicitly mention the used implementation strategy.

10.1.3 Handling Auxiliary Input

Auxiliary input could be handled by additional parameters to the constructor of the generator or by a set of setters. As discussed before, we propose to use setters, which is also the approach we use in the generator composition language introduced in Section 10.2. This language allows to compose fragments to a larger generator. It scans, therefore, each generator fragment for auxiliary inputs, by searching for setters annotated with `@Auxiliary`. This annotation is necessary to distinguish auxiliary inputs from trace model related methods.

10.1.4 Support for Other Transformation Languages

We use Xtend [IE11; Bet13] as programming language in our evaluation. However, there are other transformation languages available which support EMF models. Most of them are designed to read serialized models, transform them, and write the resulting models to files. The transformation is then specified in a third file which is also read and then used by a transformation engine. This design violates the solution realized by this prototypical framework implementation, as we intend to keep the models in memory during all processing stages and only output resulting models.

However, ATL [JK06] and the QVT [QVT05] of Eclipse provide Java programming interfaces which allow to run the transformation engines directly from Java and Xtend. Furthermore, they allow to pass models which are already loaded. Therefore, we can implement abstract transformation classes for ATL and QVT which can then be specialized to support specific transformations.

```
abstract class AbstractATLTransformation<S extends EObject,T extends EObject>
    implements IGenerator<S,T> {
    val String transformationDir
    val String transformationModule

    new(String transformationDir, String transformationModule) {
        this.transformationDir = transformationDir
        this.transformationModule = transformationModule
    }

    override T generate(S input) {
        [...]
    }
}
```

Listing 10.4. Interface of the abstract class for the integration of ATL transformations in GECO

The two abstract classes AbstractATLTransformation and AbstractQVT-Transformation realize the respective fragments which must be subclassed and parameterized to be usable for the GECO fragment composition. They can be found in the GECO framework package [Jun16b] and on its website[1].

[1] GECO framework `http://https://github.com/rju/geco-composition-language`

As both classes provide a similar interface, we illustrate their implementation interface with the ATL fragment class AbstractATLTransformation, depicted in Listing 10.4.

The interface comprises two key elements. First, the abstract class inherits its interface from IGenerator. Therefore, it provides a generate method, which takes a model root object as input and outputs a root object as output. However, the abstract class limits the input and output mode classes to subclasses of EObject. Whereas the IGenerator can also be used for models which are modeled with plain Java classes.

Second, AbstractATLTransformation has a constructor with two parameters. The first parameter defines the path to the directory where the transformation is stored, and the second parameter defines which transformation must be executed. A concrete fragment class for an ATL transformation must subclass AbstractATLTransformation and define a constructor without parameters which internally calls the constructor of its super class passing the correct parameter values.

The setup for AbstractQVTTransformation is very similar. However, the transformation must be specified by URI, which includes the path to the directory of the transformation and the transformation file name.

10.1.5 Trace Model Integration

The GECO approach uses model trace information to realize the decomposition of generators in fragments (see Chapter 7). Further, we discussed the functionality and potential structures of trace models in depth (Section 8.4). Based on these considerations, we define a minimal interface for trace models (Listing 10.5). Based on this minimal interface, the framework provides a generic trace model implementation realized as TraceModelProvider.

Both, the interface and the default implementation TraceModelProvider, allow to limit source and target model nodes to a specific type. However, this can be too restrictive when different source and target metamodel types need to be stored in the trace model. For example, a component type with operation declarations is transformed into a Java class with a set of methods. Then the trace model should store a tuple relating the component type to the class, and tuples relating each operation to a method.

```
interface ITraceModelProvider<S, T> {
    def void add(S source, T target)

    def Iterable<T> lookup(S source)
}
```

Listing 10.5. Minimal interface declaration for a trace model provider expressed in Xtend

A simple solution would be to use a common super class for source and target metamodels. This could result in a trace model relating EObject to EObject when using EMF. While this would definitely work, the fragment code would have to check and cast the type of every returned element before it can be used. This unnecessarily complicates the code which could lead to faulty code.

To circumvent this issue, we could use multiple trace models, one for each pair of metamodel classes. However, this solution would result in as many trace models as there are pairs of metamodel classes.

For example, in a DSL for a pipe and filter framework, each filter has a name and an expression to specify the filter's behavior. Such filters are then transformed into Java classes, with one method implementing the expression, one property to handle the input, and one method to provide the output. In this example, the filter is related to two methods and one property, which would require one trace model for Filter-to-Method relationships and one for Filter-to-Property relationships. This can also complicate the implementation, as the developer must decide which trace model to use in a specific situation. Furthermore, if both relationships for a Filter are required, two queries are necessary.

To address both issues, we combine two design ideas. First, we allow one fragment to read and write multiple trace models. And second, we enhance the ITraceModelProvider interface with five additional methods:

1. A method to query the trace model for all target nodes of a specific type TV which correspond to a given source node. As the source node is already a concrete instance, it is not necessary to limit the search also to a specific source type.

```
def <TV extends T> Iterable<TV>
    lookup(S source, Class<TV> targetClass)
```

This method allows to limit the returned target nodes. In the filter example this would allow to return only the methods even if all relationships are stored in the same trace model. The resulting list is then already correctly typed and does not need any instance checks.

2. When handling join points, it is often necessary to get all target nodes for a distinct type of source nodes. The result is then later restricted and grouped. The GECO framework supports this lookup with the following method:

```
def <SV extends S, TV extends T> Iterable<TV>
    lookup(Class<SV> sourceClass, Class<TV> targetClass)
```

3. When using the previous interface method to retrieve a large set of target nodes, it is often necessary to find corresponding source node with a reverse lookup.

```
def Iterable<S> reverseLookup(T target)
```

4. To complement the reverse lookup, we support the retrieval of all source nodes. This method allows to iterate over all trace model elements which match a specific source model class:

```
def <SV extends S> Iterable<SV>
    allSources(Class<SV> sourceClass)
```

5. One key issue with a single and generic trace model is that in the code the results must be casted. While this can be avoided with special typed trace models, it is also complicated to create many different trace models. However, for some implementations it might be helpful to have such specialized trace models available. The following method allows to derive a specific trace model for defined node types from a common trace model.

```
def <SV extends S,TV extends T> ITraceModelProvider<SV,TV>
    subset(Class<SV> sourceClass, Class<TV> targetClass)
```

10.2 Fragment Composition Tooling

In Section 10.1, we introduced the GECO fragment design and composition framework. The framework provides a general interface for generator fragments and weavers. Furthermore, it provides a reusable module implementing a trace model provider, which was developed following the functional decomposition of fragments discussed in Section 8.1.

Until now, we introduced the fragment combination patterns in Chapter 7 and a framework supporting the key elements of these patterns. However, the assembly of fragments must be performed manually. For larger generators, this is a cumbersome task, as models and transformations must be instantiated and linked by hand. This may also result in faults in the combination of fragments, which can cause failures depending the on input. To reduce the necessary programming effort and to mitigate the risk of faults in the composed generator, we introduce a fragment composition language which abstracts from technical details and allows to combine fragments in a declarative way. The language is supplemented by a generator.

Furthermore, the language shows that the abstraction introduced by the patterns is sufficient to realize the combination of fragments.

The generator composition tooling is founded on a set of concepts, which are introduced in Section 10.2.1. In Section 10.2.2, we explain the grammar and the associated semantics of the language. Finally, in Section 10.2.3, we describe the generator design and composition.

10.2.1 Language Concepts

The composition language and generator are designed around four major concepts, which are metamodels and instances thereof, trace models, generator fragments, and weavers.

Metamodels and Models Generators are used to generate target models and code from source models. These models conform to different metamodels and are, therefore, instances of metamodels. In GECO, each generator is assembled of generator fragments. They have one source model as input, and one or more target models as output. The source model conforms to a source metamodel, and all target models produced by one fragment conform to one target metamodel.

As the fragment composition must be expressed at design time, it cannot refer to concrete model instances. Instead we use the root classes of metamodels or metamodel partitions in order to express a model or a set of models of the same type. The root class of a metamodel or metamodel partition is a sufficient identifier for metamodels in GECO, as explained in Section 6.7.

A metamodel is often structured in packages and the root class is not specifically marked. Therefore, we must be able to specify which class in an EMF metamodel is considered a root class.

As a class name is only unique in the context of the package where it is contained in, we have to address a class over its fully qualified name. The fully qualified name is constructed over the package hierarchy and the class name of the metamodel, e.g., the fully qualified name de.menges.types.metamodel.Model is composed of the fully qualified package name de.menges.types.metamodel, describing the package nesting, and the class name Model.

Self-contained metamodels have one single root class. However, this root class may not represent the key concept of the metamodel. In metamodels for DSLs and ASTs, the root class comprises package naming, imports, and other elements to establish the context of the artifact. The actual specification in such a model is modeled with instances which are contained in the root class. For example, the DTL root class has a containment reference to Entity which models an entity class. Therefore, the fragment must navigate the model to its actual content. This moves complexity into a fragment which is not always advisable. It may not be an issue in a model containing, for example, only one data type declaration and the fragment is designed to generate output for the complete model. However, if the model contains multiple data type declaration, like in the DTL, the fragment must generate

multiple output models. Here it is more advisable to move the iteration over data types out of the fragment into the generator architecture level. On that level, the fragment can be called for each contained type once and produce one output model for each invocation. This procedure is especially helpful when metamodel partitions are reused in different constructs. For example, in MENGES predicates can occur in different types. With moving the code generation for predicates into a separate fragment, it can be reused in the implementation. Therefore, the fragment must not depend on the context of the metamodel partition.

To achieve this, we require a way to navigate models along declared references. This can be realized with navigation and query expression constructs, which allow to navigate to a contained element. Furthermore, query expressions allow to further limit the set of models to be processed by a specific fragment.

For example, the IRL [JHS13] defines two different types, templates and entities, which are transformed into Java interfaces and classes. Therefore, the IRL uses two different fragments, one for templates and one for entities. As each IRL file and model may contain multiple template and entity declarations, the corresponding fragment must be invoked only on the correct types. In other generators, it might be helpful to constrain model sets not only by element type, but also by properties and structure.

Therefore, we consider that the GECO composition language must allow to specify and select root classes from metamodels, which must be complemented by a navigation and query expression grammar. The latter must allow to navigate to any element in a model and perform a selection of elements based on type and attribute values.

Trace Models Trace models have been extensively discussed in this thesis. In Section 8.4, we described its purpose and the principal functionality a trace model should provide. In Section 10.1.5, we introduced a trace model interface and implementation suited for the GECO composition framework. Especially, in the framework realization, we discussed limitations of the generics in Java, which do not allow to construct a trace model and a provider in a way which allows only certain combinations of source and target classes in trace model tuples, resulting in insufficient typing restrictions.

In the GECO fragment composition language, we can address this limitation in a way that the developer can define which classes can be used in one tuple. This means, it must be possible to define multiple pairs of classes, e.g., `ComponentType` to `Class` and `Operation` to `Method`. Furthermore, it would be convenient to compact the declaration, if multiple source types can have the same target type, and vise versa. Therefore, we propose a trace model declaration based on a set of pairs where each element of the pair is a set of types.

Generator Fragments In addition to models and metamodels, generator fragments are a key concept of the GECO approach. They have been defined as components which include at least one transformation and transform one source model into a one or more target models (see Chapter 8). They may offer a trace model to other fragments, and can use multiple trace models provided by other fragments. It is important to note that two fragments may not depend on each other's trace model, as this would imply a cyclic dependency of the two source metamodels used in the respective fragments (cf. Section 6.7). In addition, a fragment may use one or more auxiliary models for lookup purposes, as short hand for aggregation fragments (see also Section 10.1.3).

In case a fragment provides multiple target models, the target model must either be of a collection type or it must be a weaver. If the weaver only accepts single models, it is invoked for every element in the result. This allows the developer to concentrate on the metamodels and the transformational character of the fragments and does not have to bother if a fragment or weaver is executed multiple times.

Weaver The second kind of transformation component used in GECO are weavers. Weavers can be complex software components, as suggested by AMW [FBV06] and the Kermeta weaver [Kra12]. They have to resolve complex pointcuts, select nodes for replacement, insert new nodes, and reconnect edges. However, in the context of GECO, weavers can be very simple, especially when they compose partial models into a base model. In this case an advice model is added or referenced by an existing base model node. The weavers used in MENGES case study belong to this simple type.

In GECO, weavers follow the general scheme in which they weave advices of an aspect model to a base model, and return the modified base model after the weaving. As described in Section 10.1, we do not assume that the input base model remains unaffected and the output of a weaver is a copy of the input base model with modifications described in the aspect model. Therefore, the sequence of weaver invocation is relevant and must be considered when composing generators.

The GECO framework allows to specify two different weaver types, one of which expects one aspect model and one which expects pointcut and advice in separate models. Therefore, the GECO composition language must support both types. Furthermore, it would be helpful to omit explicit aspect models and replace them with generator fragment invocations, as this reduces the number of explicitly declared metamodels.

10.2.2 The GECO Composition Language

In the previous section, the concepts metamodel, trace model, generator fragment and weaver have been introduced. They are the foundation for the design and specification of the GECO composition language and its generator. The composition language realizes these concepts in a textual DSL implemented with Xtext [Bet13]. It is supplemented by an automatic visualization of the resulting megamodel realized with the KIELER Lightweight Diagrams (KLighD) [SSH13] layouting and diagram framework . In the remainder of this section we introduce the grammar of this DSL and discuss their semantics.

Basic Typing Rules The GECO language accesses Java classes directly utilizing the typing infrastructure of Xbase [EEK+12]. To model the typing rules for the GECO language, we need to define typing rules for Xbase. We define them based on the definition for Featherweight Java [Pie02, p. 245ff]. JvmType is used to hold any type in Xbase and is used in the grammar to refer to a class declaration. As it might be useful to refer to more than one type in the semantics rules, we use there C and D to refer to classes and interfaces. In addition we refer to the framework interfaces by their name, like *IGenerator* and *IWeaver*. To refer to metamodel root classes which must

be specified when using one of these interfaces, we write $IGenerator < S, T >$ where S and T refers to the source and target metamodel's root class.

$$C <: C \qquad \frac{C <: D \quad D <: E}{C <: E} \qquad \frac{CT(C) = \text{class } C \text{ extends } D\{\ldots\}}{C <: D}$$

Figure 10.1. Subtyping rules [Pie02, p. 255]

The term CT refers to the class table which maps class names C to class declarations CL. A class declaration in Xbase is represented by JvmDeclared-Type which comprises of JvmMember elements. These can be constructors, properties, and methods. In the GECO language, we need accessible properties which are technically identified by getters. Getters are public methods with no argument that return one typed value. The method's name corresponds to the propertiy name, but is prefixed by `get` or `is` according the Java coding style.

For our typing rules, we abstract from this technical detail and assume that *properties*(C) refers to the accessible properties. The definition of *properties*(C) is accomplished with two rules. In Figure 10.2, the first rule is the root clause, as every class definition refers in the end to Object, which has no properties ($\bullet$).

$$properties(\text{Object}) = \bullet$$

$$\frac{CT(C) = \text{class } C \text{ extends } D\{\overline{C\ f}\} \qquad properties(D) = \overline{D\ g}}{properties(C) = \overline{D\ g}, \overline{C\ f}}$$

Figure 10.2. Class property lookup rules (cf. [Pie02, p. 257])

The second rule expresses that if there is a class C which extends another class D then the properties of C are those from D in addition to those explicitly defined in C. The $\overline{C\ f}$ refers to a sequence of property declarations.

The dash above $\overline{C\ f}$ indicates that there can be multiple declarations. It is a shorthand for $C_1 f_1, \ldots C_n f_n$. Each C represents a reference to a type and each f represents the property. Similarly, $\overline{D\ g}$ is the equivalent for type D (cf. [Pie02, p. 257]).

Language Infrastructure Xtext grammars are LL(*) grammars which can be parsed without backtracking [IE11]. Even though Xtext allows to use the ANTLR [Par07] backtracking feature, this is strongly discouraged, as it causes parsing issues which are hard to detect [Bet13, loc. 3399]. The Xtext framework generates an EMF metamodel based on the grammar rules which can then be used by a generator. In short, the non terminal on the left side of the rule corresponds to a class in the metamodel, and the right side of the rule is interpreted as attributes and references.

The central part of every Xtext grammar is the first production rule. Its left side is the start symbol of the grammar. In Listing 10.6 the first production rule of the composition language is depicted. It comprises necessary declarations and its respective EMF metamodel class. We call this part of the grammar infrastructure, as it comprises general declarations, imports, and connects all parts of the grammar and metamodel.

The rule depicted in Listing 10.6 comprises a package declaration, imports, metamodel root class registrations, variables representing models, and the instantiation of generator and weaver fragments.

```
GecoModel:
    'package' name=QualifiedName
    imports+=Import*
    registeredRootClass+=RegisteredRootClass*
    models+=ModelSequence*
    fragments+=Fragment*
;

Import:
    'import' importedNamespace = [types::JvmType|QualifiedName]
;
```

Listing 10.6. Grammar infrastructure

The megamodel requires a package name to be able to properly integrate

the generated code into the Java package hierarchy. The imports allow to refer to other Java classes which will be used later in the megamodel. In Java, imports allow to import any Java class. In the GECO composition language, these imports are limited to generator fragment and weaver classes. These classes are identified by the interface they implement, which are IGenerator, IWeaver, and IWeaverSeparatePointcut.

Apart from fragments, the megamodel also uses metamodels and models. Therefore, a specialized import for metamodel root classes is necessary (RegisteredRootClass). For the GECO composition language, these classes must exist as Java classes generated from EMF metamodels. Subsequently, models and partitions of models must be identified, which are handled by the models property. And finally, the configuration of generator fragments and weavers is covered by a collection of Fragment instances.

Model Access Model access is handled by two rules. First, RegisteredRootClass (see Listing 10.7) allows to import root classes of metamodels. It also allows to support text containers identified by a string. The primary function of RegisteredRootClass is to import a Java class as root class of a metamodel and assign it to a shorthand which can be used in the textual DSL.

```
RegisteredRootClass:
    'register' name=ID (
        importedNamespace = [types::JvmType|QualifiedName] |
        (isText?='text' extension=STRING)
    )
;
```

Listing 10.7. Metamodel declaration

Semantically, the rule expresses the import of a Java type and the assignment of an alias name to this type. It also realizes access to all attributes and references of the imported class.

These registered root classes can then be used to declare variable-like identifiers which derive their type from a ModelType, as shown in Listing 10.8 A ModelType is a registered root class, the complete type of any of

its references, or any class reachable over references and the types of their references.

```
ModelSequence:
    'model' modifier=ModelModifier type=ModelType models+=Model (',' models+=Model)*
;

enum ModelModifier:
    INPUT = 'in' |
    OUTPUT = 'out' |
    INTERMEDIATE = 'io'
;

Model:
    name=ID
;
```

Listing 10.8. Metamodel declaration

The type derived from ModelType is either a root class of a metamodel or a collection type with a root class as element type. While name and type are sufficient to declare a model variable, they do not allow to infer which models should be exposed to the outside for serialization. Furthermore, it can be inconclusive which model variables must be initialized out of the set of input models. Therefore, we added a model modifier to indicate if the model is an input, output, or intermediate result model. The difference between these three types of models is that input models are initialized automatically from the set of input models handed to the generator. Output models are provided as output of the generator. And intermediate models are initialized by the output of fragments and weavers, and can be used as input for fragments and weavers.

Weaver and Generator Fragments Weaver and generator fragments are, together with models, the building blocks of megamodels in GECO. In the grammar the non-terminal Fragment, depicted in Listing 10.9, is used as common symbol for weaver and generator fragments. Usually we refer with the term 'fragment' to a generator fragment. In the GECO language grammar, fragment is the comprehensive term for both weaver fragment and generator fragment.

```
Fragment:
    Generator | Weaver
;
```

Listing 10.9. Weaver and Generator Fragment

Weaver In GECO a weaver has a base model and an aspect model as input, and a modified base model as output. Furthermore, the aspect model can be split up into a pointcut and an advice model. In Listing 10.10 the grammar rule to express the instantiation of a weaver is depicted. First, the grammar defines a reference to a JvmType. While this would syntactically allow to use any JvmType here, it is semantically constrained to types implementing IWeaver or IWeaverSeparatePointcut of the GECO framework.

```
Weaver:
    'weave' reference=[types::JvmType|ID]
    (sourceModel=SourceModelSelector|'link')
    aspectModel=AspectModel
    ('=>' targetModel=TargetModel)?
;
```

Listing 10.10. Weaver Fragment

The weaver rule does not assign a name to the instance, as direct access to the weaver instance is not necessary. Instead the grammar requires to refer to two models. First, a source model must be specified with a reference expressed with a SourceModelSelector. Second, a reference to an aspect model must be specified. However, a target model is optional. If omitted, the modified base model will be assigned to the source model reference.

The type of a SourceModelSelector can either be a single root class or a collection type with a root class as base type. In case of a collection type, the weaver will be executed for every single instance of the source base model. These multiple weaver executions also result in multiple target base models. When the target base model is omitted, the source model is used which already has the correct type. However, in case of a separate target model, it must have the same type.

In some generator composition scenarios, multiple different generator

fragments are used and all their results must be woven into a base model; The MENGES [GHH+12] case study, for instance, is such scenario. Such scenarios require a long sequence of weaver calls. But instead of having to specify a potential complex source model query over and over again, we allow the user to use the keyword `link`. It expresses that the source model query of the previous weaver instance must be used here too. In the model it is expressed by a null value assigned to `sourceModel`.

As the aspect model can include both pointcut and advice, or come in two separate parts, the language addresses this with the rules depicted in Listing 10.11. A `CombinedModel` is either specified by a target model reference which must identify a single element, or a generator fragment. The latter option is a shorthand, which allows to use the output of a generator fragment as aspect model. Therefore, it is not necessary to declare intermediate model variable for the generator output which is then used by the weaver.

```
AspectModel:
    CombinedModel |
    SeparateModels
;

SeparateModels:
    'pointcut' pointcut=TargetModel 'advice' advice=CombinedModel
;

CombinedModel:
    TargetModel | Generator
;
```

Listing 10.11. Aspect Model

In the second case with separate pointcut and advice models, the language provides the `SeparateModels` rule. This rule allows to specify the pointcut and advice separately. Whereas the advice uses the same rules as the aspect model rule, i.e. either a `TargetModel` or a generator instantiation, which implies a target model.

Generator A generator fragment processes one source model into one or more target models which then have all the same target metamodel. This

is realized by a collection with a specific base type for the target model. In the DSL (see Listing 10.12), the target model can be omitted, if the generator fragment instantiation is used in conjunction with a weaver. As stated in Chapter 7, a generator fragment may use one or more trace models and create one trace model. In the grammar they are called sourceTraceModels and targetTraceModel, respectively.

```
Generator:
    'generate' reference=[types::JvmType|ID]
    ('(' sourceAuxModels+=SourceModelSelector (',' sourceAuxModels+=SourceModelSelector)* ')')?
    'source' sourceModel=SourceModelSelector
    ('target' targetModel=TargetModel)?
    ('trace'
        ('out' targetTraceModel=TargetTraceModel)?
        ('in' sourceTraceModels+=TraceModelReference (',' sourceTraceModels+=TraceModelReference)
              *)?
    )?
;
```

Listing 10.12. Generator Fragment

Furthermore, a generator fragment can have multiple auxiliary models as input to reduce internal complexity and foster reuse. These auxiliary models are specified as SourceModelSelectors.

The grammar rule in Listing 10.12 allows to use any JvmType as generator fragment. However, semantically this is restricted to JvmTypes which implement the IGenerator interface, i.e., $JvmType <: IGenerator < S, T >$ where S and T are the root class of source and target model root elements, respectively.

Model Selectors Weaver and generator fragments need to access models or partitions thereof. It would be sufficient to directly refer to the declared models in a GECO megamodel represented as instances of Model. However, this approach would require to pass larger portions of models to a fragment, which would then require to find the necessary information in these models. This would also harm modularization when a fragment could be reused in different contexts. For example, a fragment generates code for methods and functions which can appear in a class implementation and module

respectively. While the specification of methods and functions are identical in this example, they are used in different metamodel partitions. Therefore, a reusable fragment would have to accept the root class for both model types and realize a model query for each model type. Alternatively, we could write two fragments for each model root class.

As this results either in unnecessary complex fragments or duplication of code, we allow to define model queries in the GECO composition language. They allow to find and collect parts of models, like for the method and function specification in the example above.

In Listing 10.13, the grammar rule SourceModelSelector defines how such model queries can be constructed. Each model query requires to refer to one Model declaration. Such Model declarations may refer to a single model or a collection thereof. In case of a collection of models, it can be necessary to pick only specific models from the collection and pass it to the fragment. Therefore, the DSL allows to apply a ConstraintExpression to the collection.

```
SourceModelSelector:
    reference=[Model|ID] ('[' constraint=ConstraintExpression ']')? ('/' property=NodeProperty)? |
    {SourceModelSelector} 'null'
;

NodeProperty:
    property=[types::JvmMember|ID] ('[' constraint=ConstraintExpression ']')? ('/' subProperty=
         NodeProperty)?
;

TargetModel: {TargetModel}
    (reference = [Model|ID])
;

ModelType:
    target=[RegisteredRootClass|ID] ('/' property=NodeProperty)? (collection?='[]')?
;
```

Listing 10.13. Selector notation for metamodels

In our example, we intend to pass a collection of method declarations to a fragment which is specified in a class model. To access these elements, the SourceModelSelector must be able to express property access. This is done with a NodeProperty. NodeProperty refers to a JvmMember of a Java class of

the type of the parent rule. The parent rule is either SourceModelSelector or NodeProperty. In the first case, the type is determined by the Model element type. And in the second, the type is the element type of the parent NodeProperty. Similarly to the SourceModelSelector, we allow to apply a constraint to the collection defined by NodeProperty.

One special feature of the SourceModelSelector rule is that it can be null instead of a reference to a Model. Such a selector is necessary for fragments which do not need any input. For example, in the MENGES case study, we use this feature to generate the basic structure of an PLCOpen model.

The rules SourceModelSelector, ModelType, and NodeProperty all express the ability to access nested properties, which are covered by the rule T-PROPERTY in Figure 10.3.

$$\text{T-PROPERTY}\quad \frac{\Gamma \vdash t_0 : C_0 \qquad \textit{properties}(C_0) = \overline{C\ f}}{\Gamma \vdash t_0 / f_i : C_i}$$

Figure 10.3. Type derivation of properties [Pie02, p. 259]

The term t_0 in Figure 10.3 represents a model reference, a registered root class, or a path, depending on the grammar rule. For example, for a path `model/property/subProperty` the term t_0 is `model/property` and f_i is `subProperty`.

The constraint expression usually has no effect on the type of a SourceModelSelector or ModelType. However, the InstanceOf rule (see Listing 10.15) results in an implicit type cast.

Constraint Expression The constraint expression is used to select specific models from a collection. It allows to formulate a criteria which must be met by the collection elements. The expression syntax allows to access property values, types, and provides logical and comparative operations.

The central grammar rule ContraintExpression resembles a logical expression. It utilizes the ability of Xtext grammars to automatically collapse the AST when no logical operator is specified. For example, an expression

`property == 0` would not result in a ContraintExpression and a CompareExpression tree node. Instead only one CompareExpression tree node would be created. The same feature is used in the CompareExpression rule.

The language presently only provides two logical operators, namely logical-and (&), and logical-or (|). To compare values, the grammar supports different operators. Except of the like ($\sim$) operator, all of them are applicable to numeric values. Whereas $\sim$ is only useful in context of strings with regular expressions. Furthermore, the operators equal (==) and unequal (!=) are applicable for all data types including strings and enumerations.

```
ConstraintExpression:
    CompareExpression (=>({ConstraintExpression.left=current} operator=LogicOperator) right=
          ConstraintExpression)?
;

enum LogicOperator:
    AND = '&' | OR = '|'
;

CompareExpression:
    BasicConstraint (=>({CompareExpression.left=current} comparator=Comparator) right=
          BasicConstraint)?
;

enum Comparator:
    EQ = '==' | NE = '!=' | GR = '>' | LW = '<' |
    GE = '>=' | LE = '<=' | LIKE = '~'
;
```

Listing 10.14. Comparison and logical operators

The semantics for these two expression rules are depicted in Figure 10.4. The first rule T-ConstraintExpression covers the typing required for the ConstraintExpression rule. It requires the left and right term to be of type Bool which is mapped to boolean in Java.

The remaining three rules define the semantics for the CompareExpression. As stated before, not all operators can be used with all types. We defined T-CmpExp which covers the operators equal and unequal, to use any type T with this rule. If it is used on objects, the respective equals method is used for comparison. T-CmpExpNumber covers all compare operations which can be performed on numerical types. As it would be too

cumbersome to add a rule for every numerical type, we use *NumberType* to represent the set of numerical types which are byte, int, long, short, float, and double.

T-ConstraintExpression

$$\frac{t_{left} : Bool \qquad t_{right} : Bool}{t_{left} \text{ LogicOp } t_{right} : Bool}$$

T-CmpExp

$$\frac{t_{left} : T \qquad t_{right} : T \quad \text{Comp} = \{==, !=\}}{t_{left} \text{ Comp } t_{right} : Bool}$$

T-CmpExpNumber

$$\frac{t_{left} : T \qquad t_{right} : T \qquad \text{Comp} = \{>, <, >=, <=\} \qquad T \in NumberType}{t_{left} \text{ Comp } t_{right} : Bool}$$

T-CmpExpString

$$\frac{t_{left} : String \qquad t_{right} : String \qquad \text{Comp} = \sim}{t_{left} \text{ Comp } t_{right} : Bool}$$

Figure 10.4. Typing rules for logical and compare expressions

And finally, the rule T-CmpExpString (see Figure 10.4) requires that the left and right term are of type String and the compare operation is the like operation (∼).

The constraint expression rules are complemented by basic expression elements depicted in Listing 10.15. The BasicConstraint rule collects these basic elements. First, Negation allows to express a negation of an expression. Therefore, the expression must be of type *Bool*, as shown in Figure 10.5 with rule T-Negation. The ParenthesisConstraint is necessary to group expressions and can be used with any type, as shown in T-ParenthesisConstraint.

Finally, the grammar rule Operand refers to elements which carry a value. Literal represents the different literals in the language, which are integer numbers, floating point numbers, true and false as boolean values, strings, and enumeration values. NopeProperty refers to a property of an element of the collection or any sub property thereof. And InstanceOf expresses a type conformance check which is applied to every element of the collection.

```
BasicConstraint:
    ParenthesisConstraint | Operand | Negation
;

Negation: '!' constraint = ConstraintExpression ;

ParenthesisConstraint: '(' constraint = ConstraintExpression ')' ;

Operand:
    Literal | NodeProperty | InstanceOf
;

InstanceOf: 'is' type = [types::JvmType|ID] ;
```

Listing 10.15. Basic selector constraint rules

The semantics of this grammar rule is described with T-InstanceOf, which defines that the operation returns a boolean value.

T-Negation
$$\frac{!t : Bool}{t : Bool}$$

T-ParenthesisConstraint
$$\frac{t : T}{(t) : T}$$

T-InstanceOf
$$\frac{\Gamma \vdash t : T}{\text{is } T : Bool}$$

Figure 10.5. Typing rules expressions

Trace Model Declaration The last part of this grammar we need to discuss, is the declaration of trace models, depicted in Listing 10.16. We already defined a trace model as a set of tuples relating source model elements to target model elements. While the Java type system has limitations to state which node types may be used for these tuples, the GECO language provides a more specific approach.

The central rule for trace models is TraceModel. It defines one trace model with a unique name and a set of NodeSetRelations. Each NodeSetRelation relates a set of source node types to target node types.

```
TraceModel:
    name=ID '<' nodeSetRelations+=NodeSetRelation+ '>'
;

NodeSetRelation:
    '('
        sourceNodes+=NodeType (',' sourceNodes+=NodeType)*
    ':'
        targetNodes+=NodeType (',' targetNodes+=NodeType)*
    ')'
;

NodeType: type = [types::JvmType|ID] ;
```

Listing 10.16. Declaration of trace models

10.2.3 Code Generator

Based on the megamodel specified with the GECO composition language, a generator produces an Xtend class implementing the functionality. We chose Xtend over Java, because Xtend with its functional syntax and lambda expressions provided language features, we could directly utilize. A Java mapping would have resulted in a more complex generator. Furthermore, we did not use Xbase for two reasons: First, the Xbase generator framework is not compatible with the GECO framework. Therefore, we could not use the Xbase generator framework and the GECO framework together. To be able to show the usability of the GECO generator composition framework, we choose our framework over potential benefits from the Xbase framework. Second, we intend to adapt this code generator to other platforms and build systems which may not use Java.

The declarative character of the language allows to declare generator and weaver executions in an arbitrary order. Furthermore, the language allows to define model queries which can be shared by fragment instantiations. We implemented the generator with two fragments, as shown in Figure 10.6. First, all fragments and weavers are arranged in the right order and stored in an intermediate model called BoxingModel. Second, Xtend output is generated based on this boxing model.

The grouping in the boxing model is based on the paths declared within ModelTypes and SourceModelSelectors and can, therefore, be nested. The

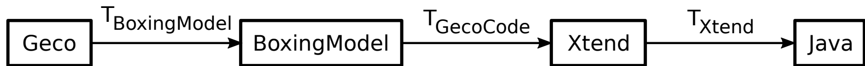


Figure 10.6. Graphical representation of the GECO composition language generator

single fragments are finally placed in an execution Unit which knows all dependencies of the fragment.

The module structure of the GenerateBoxingModel fragment is simple. First, it determines all models and fragments which are placed in model declaration and execution units, respectively. And second, based on the dependencies computed in the first step, groups are created and units are placed. Therefore, this functionality was realized in one class which only refers to the typing module for the GECO language.

The code generator fragment GenerateGecoCode comprises four modules: The typing module, a name resolver, a module for selector queries and constraint construction, and the main module containing the fundamental class template.

10.3 Realizing Typing in Xtext

We concluded in Chapter 6 that the semantics and specifically typing are important to understand and partition metamodels. Founded on this understanding, we defined our generator composition approach in Chapter 7 and the fragment design approach in Chapter 8. Both rely on the understanding of typing, and the fragment design approach provides type mapping of source to target level. All these chapters discuss typing in a technology independent way. In this chapter we evaluate the ideas and methods for the construction of typing and apply it to DSLs realized with Xtext.

Furthermore, we describe in detail how to model types, integrate them into the type resolving framework of Xtext, connect the typing with an Xtext grammar, and finally implement the type resolution and semantics.

10.3.1 Modeling Types

We discussed type modeling in great detail in Section 6.5. However, for readability reasons, we provide here a short overview on the same subject.

The central element of any type system with Xtext is a common class Type which allows to refer to any kind of type declaration. Depending on a concrete realization of types in a metamodel, the class can be named differently, like in Xbase where the common type class is called JvmType.

Based on the common class Type, a taxonomy of sub-classes for the different type structures can be build. The taxonomy shown in Figure 10.7 is our general pattern for the composition of type systems. It divides types in two larger groups, which are NamedTypes with an attribute name, and UnnamedTypes. However, in concrete DSLs certain aspects can be modeled differently. For example, the distinction of named and unnamed types, can be either modeled by sub-typing or with multiple inheritance where the naming feature is inherited from a general NamedElement class.

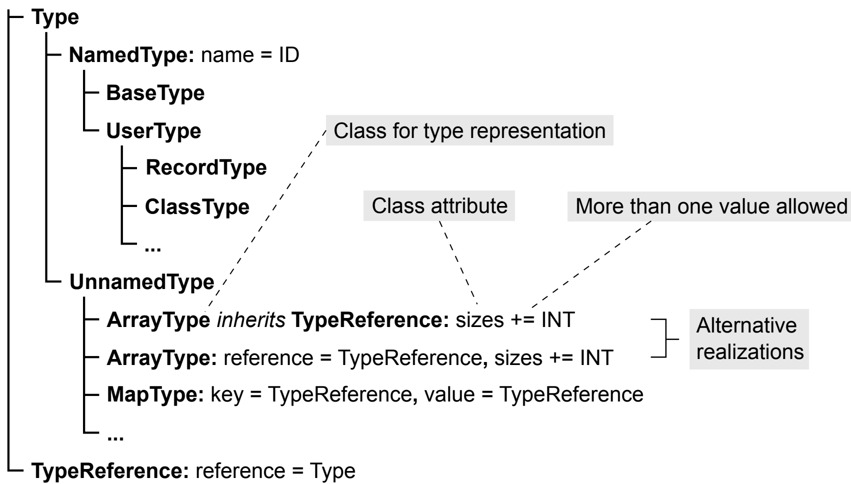


Figure 10.7. Taxonomy of types in a type system

As Xtext [Bet13] is a framework and tooling to create textual DSLs, base types are named types. Therefore, the BaseTypes are a subtype of Named-

Type. Base types are atomic types, as they have no internal structure. Some approaches suggest to use separate classes for each base type (cf. [Völ11]). However, we strongly suggest to use our approach, as it allows to handle base types and user types alike, which results in simpler grammars, compact type mapping functions in generators, and allows to handle all named type in the same way. This also increases code readability.

In DSLs, types are not only declared, but also referred to. This referencing can always be done in EMF adding an EReference to a class named reference and typed Type. Alternatively, this can be encapsulated in a TypeReference class. In MENGES [GHH+12] such a type reference class was used to decouple referencing and was beneficial for reusing type reference resolution code in the expression metamodel. Also in Xbase [EEK+12], the interface JvmTypeReference is used to cover all kinds of type references, like references with constraints for generics. In contrast, the Instrumentation Record Language (IRL) does not use a type reference class [JHS13]. The IRL is only used to model data types without any expression syntax which could make use of a distinct reference type.

10.3.2 Base Type Integration

Base types are the fundamental elements of a type system and are usually predefined types in DSLs. It is possible to construct DSLs so that the user can also declare base types, however, they are rather uncommon and require additional declarations to map source level base types to target level base types to realize generators. Therefore, we focus in this realization of typing on the common case where base types are part of the language.

There are different ways to realize base types in Xtext. One uses grammar rules of the Xtext grammar language to create the base types (see Listing 10.17). However, we advice against this approach for two reasons (see [JSH13]). First, it results in mixing syntax and typing in one artifact, which makes it less maintainable. The grammar should, therefore, only provide the syntax for the language and, in case of Xtext, hints for the construction of the AST. And second, defining base types with grammar rules results in additional instances of a base type every time it is used in a declaration. This requires a special type comparison feature for base types,

as they cannot be identified by reference, i.e., they are not really unique elements in the AST.

```
PrimitiveType returns typesPrimitives::PrimitiveType:
    TypeBoolean | TypeInteger | TypeString | TypeDuration;

TypeBoolean returns typesPrimitives::TypeBoolean: name = 'boolean';
TypeInteger returns typesPrimitives::TypeInteger: name = 'integer';
TypeString returns typesPrimitives::TypeString: name = 'string';
TypeDuration returns typesPrimitives::TypeDuration: name = 'duration';
```

Listing 10.17. Early realization of base types from the MENGES grammar (called primitive types)

Due to these considerations, we suggest to define base types in a module that provides the types to the tooling at runtime. The realization based on the Xtext API comprises three elements a type resource providing a model of all base types, an addition to the Xtext scoping API, and a type provider to connect resource and scoping.

Type Resource In Eclipse and Xtext, models are contained in resources. These resources are usually persistable into a file or database. However, for built-in base types persistence is not necessary. Therefore, our TypeResource class realizes an in-memory resource for base types compatible with the resource management of Eclipse, but without supporting persistence. This guarantees that the base types are integrated into the tooling and cannot be altered by the user. This design is similar to the type integration for Java types provided by the Xbase framework [EEK+12]. However, our implementation is simpler, as we only have to support base types and do not need to access and parse Java artifacts.

The collection of base types can be determined by means of an enumeration, as we have done in our use cases, or with an array of strings. The enumeration approach benefits from mutual disjointness of the enumeration literals (see Listing 10.18), while a string array would allow to add the same name twice. Therefore, we recommend our enumerated solution, even though it is slightly more code to type than a simple string array.

The example provided in Listing 10.18 depicts the base type declaration

of the Instrumentation Record Language (IRL) used in the Kieker project [HWH12] to define record structures for monitoring events. As the IRL allows to extend the data types of the language by importing metamodels defined in EMF models, it uses internally EClass and EDataType to represent user and base types.

```
public enum BaseTypes {
    LONG,
    INT,
    SHORT,
    BYTE,
    BOOLEAN,
    FLOAT,
    DOUBLE,
    CHAR,
    STRING;

    private EDataType etype;

    PrimitiveTypes(final String name) {
        this.etype = EcoreFactory.eINSTANCE.createEDataType();
        this.etype.setName(name.toLowerCase());
    }

    public EDataType getEType() {
        return this.etype;
    }
}
```

Listing 10.18. Base types of the IRL

The enumeration BaseTypes defines the different base types. This declaration is then used in the TypeResource to construct the base type model.

Handling Scope Xtext uses a scoping mechanism to resolve identifier names used by the editor and the generator. The scoping API requires at least two classes for the scope handling. First, a BaseTypeScope class provides access to types by name and reference. Second, a GlobalScopeProvider is required to add the base types to the global scope. The TypeGlobalScopeProvider is, therefore, derived from the DefaultGlobalScopeProvider which is extended with support for the BaseTypeScope. The TypeGlobalScopeProvider must then be registered with the editor. Therefore, the

RuntimeModule class must be extended. In Listing 10.19, we show the TypeGlobalScopeProvider of IRL.

```
public class RecordLangRuntimeModule extends de.cau.cs.se.instrumentation.rl.
        AbstractRecordLangRuntimeModule {

    @Override
    public Class<? extends org.eclipse.xtext.scoping.IGlobalScopeProvider> bindIGlobalScopeProvider() {

        return TypeGlobalScopeProvider.class;
    }
}
```

Listing 10.19. Excerpt of the IRL runtime module class depicting a method used to register the TypeGlobalScopeProvider

Type Provider Finally, the scope and the resource must be integrated. This is realized with the TypeProvider and supported by EcoreTypeURIHelper. The TypeProvider manages the creation and query of the TypeResource and serves as a factory for the TypeResource. This configuration was chosen to accommodate the Xtext API. The EcoreTypeURIHelper class handles the URI creation and processing used in the TypeProvider and incorporates URI processing features from the Xtext API.

We used the same integration pattern in other DSLs, such as the DTL used in the CoCoME case study (see Section 11.1) and later implementations of the MENGES DSLs.

10.3.3 Grammar Rule Patterns

DSLs often allow to define named and unnamed types which both combine other types into more complex structures. For example, record types allow to combine data in a structured way, like a Person record comprising name, title, and address in one entity. DSLs support the declaration of such types with specific syntactic rules (see Listing 10.20). As each type can have different properties to accommodate the specific domain, we cannot provide an Xtext with a fixed set of rules which can be imported in any Xtext grammar. Furthermore, Xtext can only import from one other grammar file,

which would render such general rule set useless when a developer has yet another grammar to import. Therefore, we only provide these rules as a template which can be modified by a developer.

The Xtext grammar rules follow loosely the Extended Backus–Naur Form (EBNF) with some extensions. First, names followed by a = are properties of the metamodel class associated with the rule. Second, expressions in square brackets are used to express references. For example, in TypeReference the expected token is an ID which is limited to names of type::NamedType of the metamodel. Third, metamodel classes in curly brackets are used to ensure the creation of an instance of the given type.

The first three lines in Listing 10.20 introduce the taxonomy of types (cf. Figure 6.1). The remaining rules define how types can be constructed and in which way attributes of types can be accessed.

```
Type returns type::Type: NamedType | ArrayType | MapType ;
NamedType: BaseType | UserType;
UserType returns type::UserType: ClassType | RecordType ;

ClassType: 'record' name = ID '{'
    properties+=PropertyDeclaration*
    methods+=MethodDeclaration*
'}'

TypeReference returns type::TypeReference:
    ChainedTypeReference |
    ArrayType | MapType
;

ChainedTypeReference: {type::TypeReference} reference = [types::NamedType| ID] ('.' remainder=
        ChainedTypeReference)? ;

ArrayType: {type::ArrayType} reference = TypeReference ( '[' sizes += INT ']' )+ ;
MapType: {type::MapType} 'map' '<' key = TypeReference ',' value = TypeReference '>' ;
```

Listing 10.20. Xtext grammar rules for type representation

We start with a ClassType which comprises properties and methods. The depicted rule requires to define the properties before methods can be declared. In Xtext, this can also be modeled in a way the properties and methods can be mixed. Details of the Xtext grammar DSL can be found in Bettini [Bet13]. To refer to a ClassType, a type reference is required. Previously in Section 6.5, we described array and map types as unnamed types. We follow this approach in this Xtext based realization. However,

in many languages array types are declared within type references. For example, in Java a return type can be declared as **public int**[] getValues(). Here a type reference to an array type is constructed together with the declaration of the array type. Therefore, ArrayType and MapType are UnnamedTypes and TypeReferences. Therefore, they are listed as alternatives in TypeReference.

The ChainedTypeReference rule allows to refer to named types. in Xtext grammars you cannot refer directly to elements which are unnamed. Therefore, type references are limited to named types. In some languages, types can be declared in other types. Java is such an example. Therefore, types cannot be referenced directly in all cases. In ChainedTypeReference, we addressed this issue with the optional remainder. We choose this representation, as it is better suited for the LL(*) nature of the Xtext grammar language. Alternatively, the grammar rule could have been Ref: parent=Ref type=[NamedType|ID]. Apart from ArrayType and MapType, any tuple type would also be added to TypeReference, as these can also appear in rules where type references can occur.

```
PropertyDeclaration returns type::PropertyDeclaration:
    modifiers += Modifier type = TypeReference name = ID ;

MethodDeclaration returns type::MethodDeclaration:
    modifiers += Modifier type = TypeReference name = ID
    '(' (parameters += ParameterDeclaration (',' parameters += ParameterDeclaration)*)? ')'
    body = Body ;
```

Listing 10.21. Xtext grammar rules for properties and functions

Listing 10.21 shows general patterns for property declarations as well as the definition of functions, methods, procedures or any other parametrized structure (cf. [JSH13]). In many languages, those elements can have modifiers such as **public** or **static**.

Further the property declaration requires a name for the property, which is usually an id and a reference to a type. This also explains why array type declarations are part of the type reference, as they cannot be referred to by name alone.

The second rule in Listing 10.21 shows a method declaration. It has a modifier, a name and a type. The type refers to the return type of the

method. Based on the considerations for tuple types above, the type reference can also refer to types which allow multiple return values. The rule ParameterDeclaration, not-shown in Listing 10.21, is similar to PropertyDeclaration. It only employs different modifiers which may even define if the parameter is for input or output. Such a construct is used in CORBA and the Palladio Component Model (PCM). Furthermore, properties are often initialized in its declaration. To realize initialization, the rule PropertyDeclaration must be extended by an optional call to an expression rule.

10.3.4 Implementing Type Resolution

Occurrences of types in declarations are represented by a TypeReference maintaining a non-containment reference to the actual type. Applying this delegation pattern is reasonable, as the reference resolution is limited to instances of TypeReference instead of providing one for each of the available kinds of declaration. In our Xtext-based setting, this resolution is realized in terms of a *scope provider* that is supported by the former mentioned TypeGlobalScopeProvider. The implementation follows the usual Xtext scope provider declaration scheme [Bet13].

In order to establish type checking of a DSL, like type compatibility of left and right hand side of assignments and operands of binary operations, different technologies can be used [Bet11; BSV+13]. One solution is Xsemantics [Bet11] which allows to declare typing rules with an inference rule notation. In this thesis we employ Xtend [Ite11] to implement the type resolving rules. While this can be less concise than Xsemantics, it often suffices, especially for smaller languages.

In Listing 10.22, we list seven cases for type resolving rules [JSH13]:

1. This is an example for values, here a BooleanValue. The type is resolved via a type provider call.

2. The TypeReference rule, as mentioned above, can be nested. Therefore, the rule checks whether the remainder is not null and the class resolveType in the remainder recursing into the structure. In case remainder is null, the type of the reference is returned.

```
/** case 1, axiom: determines the type of 'true' and 'false', similar to IntValue, Enumeration, etc. */
def dispatch Type resolveType(BooleanValue value) {
return typeProvider.findTypeByName("boolean")
}
/** case 2: determines the referenced type of a TypeReference by determining the type
 * of the last component of the (potentially compound) declared type, e.g. A.B[5] */
def dispatch Type resolveType(ChainedTypeReference ref) {
    return ref.remainder?.resolveType?:ref.reference
}
/** case 3: determines the type of a property, e.g. 'boolean x = 5;', by determining the declared type */
def dispatch Type resolveType(PropertyDeclaration decl) {
    return decl?.typeReference?.resolveType
}
/** case 4: determines the type of an identifier by determining the type of the identified element,
 * e.g. of the declared value 'x' */
def dispatch Type resolveType(ValueReference ref) {
    return ref?.valueRef?.resolveType
}
/** case 5: determines the type of a declared function by determining its referenced return type */
def dispatch Type resolveType(MethodDeclaration decl) {
    return decl.type.actualType
}
/** case 6: determines the type of a function result by determining the called function's return type */
def dispatch Type resolveType(MethodCall call) {
    return call.methodRef.resolveType
}
/** case 7a: determines the type of an assignment by determining the modified value's type */
def dispatch Type resolveType(Assignment assignment) {
    return assignment.target.resolveType
}
/** case 7b: determines the type of an assignment and checks it */
def dispatch Type resolveType(Assignment assignment) {
    val left = assignment.target.resolveType
    if (assignment.expression.resolveType.isSubTypeOf(left))
        return left
    else
        return null
}
```

Listing 10.22. Type resolution realized with Xtend

3. The type of a PropertyDeclaration is resolved by finding the type of the used type reference.

4. A ValueReference can be a variable or constant declaration. In our example, the type of the value references is the type of the value it refers to. Therefore, subsequently a method resembling our first case, is invoked to resolve the type.

5. The type of a MethodDeclaration is its return type. The return type is then resolved with the second resolver method.

6. A language usually not only allows to declare methods, but also allows to call methods. A MethodCall is usually part of the expression grammar and contains a reference to a MethodDeclaration. Following this reference the type of the call is resolved.

7. As an example of an expression, we provide also a rule for an Assignment. In this case the Assignment type is determined by the left side of the assignment and the right side is ignored. For correctly typed models, this is sufficient. However, to realize type checking, we must test if the type of the right side expression is a subtype of the type of left side. The downside of this solution is that the art of type mismatch is not communicated back to the user, as the result is just null. It also requires the generator to check if the result of the type resolving actually produced a type. Therefore, in context of generators the resolution should always return a type and it is part of the semantic checks to evaluate whether types match or mismatch.

10.4 Instrumentation Aspect Language

The Instrumentation Aspect Language (IAL) is an aspect language used to model instrumentation for monitoring purposes. It was designed for the Kieker framework [HWH12] which supports a wide range of programming languages and technologies. The framework includes monitoring sensors which can be integrated at compile, load, and runtime utilizing technologies, like AspectJ and Java Enterprise interceptors. While Kieker supports many technologies to introduce advices, each technology must be handled differently, as pointcuts and advices must be constructed in adherence to technology constraints. As this can result in a complicated setup, we designed the IAL to provide an abstraction from the technologies, so that a developer can concentrate on the monitoring task.

The IAL is designed to function independently from programming languages and metamodels for application and architecture modeling. Initially,

we designed the IAL to be configurable for various metamodels and programming languages (cf. [JHS13]). However, the mapping between various metamodels and the IAL required extensive rules which unnecessarily complicated the declaration of aspects. Therefore, we moved the mapping functionality to connector extensions which can be developed separately. For example, in the CoCoME case study based on PCM, we realized a PCM connector [HHJ+13; HSJ+14; HSJ+15].

In the following, we discuss the mapping of metamodels and DSLs, the syntax and semantics of the language, the overall generator API and three example generators which we used in the CoCoME case study.

10.4.1 Metamodel Mapping

The IAL is designed to be independent from concrete base metamodels for application and architecture modeling. However, it must be able to navigate the base models and interpret elements of the models correctly. Therefore, the IAL tooling allows to register model mappers. These mappers associate base metamodel concepts to more abstract representations accessible to the IAL.

Most metamodels for application and architecture models come with a certain type of package and namespace hierarchy. Inside these hierarchies structured types are defined which may have a nesting feature, inheritance, type references, base types, and attributes. The mapping must provide an abstraction for all these elements. In Figure 10.8, an excerpt of the mapping model is depicted representing the central concepts of the mapping model.

The root class MappingModel has two containment references for types and contents. The first reference comprises NamedTypes which are not inside a package hierarchy. This usually applies to base types, but can also be used for any other kind of type where the internal structure is not relevant for the monitoring aspect. The second reference contains Containers. They represent package and namespace hierarchies, as well as structured types, like component types and classes.

Containers can be nested to cover nested types and packages. A single Container can have Attributes with values. They must be typed with NamedTypes or CollectionTypes transitively with an element type of NamedType. In

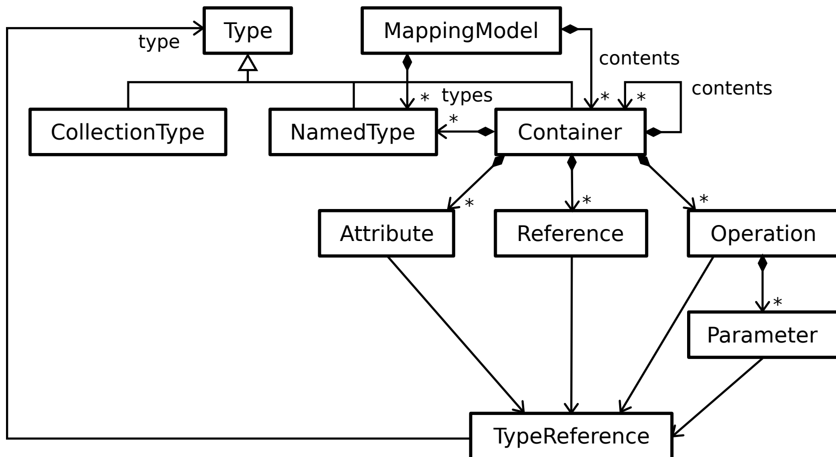


Figure 10.8. IAL mapping metamodel excerpt depicting the central mapping concepts

contrast References are used to refer to other Containers or CollectionTypes with a Container element type. Finally, Containers may realize operations. Each Operation has a return type, which can also be a Unit to express a void return type. Furthermore, an Operation can have multiple parameters. Each Parameter also has a type. All type references are expressed through TypeReference.

In addition to these concepts, the mapping model contains traceability links to the original elements in the base model, and supports modifiers for containers, operations, and parameters.

10.4.2 Syntax and Semantics

The IAL grammar can be divided into several parts reflecting different functions of an aspect model. The main division is between advice and pointcut. The advice partition of the grammar describes data collection at runtime and the pointcut is largely a way to formulate query over the source base model. Both parts are connected by an aspect composition rule. In the following explanation of the grammar we address header rules, rules

for aspect, advice, and pointcut, rules for model query, path navigation, values, and internal functions. We omit the basic terminals, as their specific declarations are not necessary to understand the syntax and semantics of the language.

Grammar Header The start rule AspectModel, depicted in Listing 10.23, integrates all elements of the grammar and allows to define the package name of the aspect. As the IAL is used to model the monitoring aspect, it must refer to monitoring events which are modeled with the IRL. These event record types are imported into the model with import statements which behaves like Java imports. The Import rule realizes this feature utilizing built-in functionality of Xtext [Bet13, loc. 4507]

```
AspectModel:
    'package' name = QualifiedName
    (imports += Import)*
    (sources += ApplicationModel)*
    (advices += Advice | pointcuts+=Pointcut | aspects += Aspect)*
;

Import:
    'import' importedNamespace = QualifiedNameWithWildcard
;

ApplicationModel:
    'use' handler=ID 'on' name=ID model=STRING
;
```

Listing 10.23. Start rule and facility rules for event type and application model import of the IAL

The rule ApplicationModel allows to refer to the models which are instrumented. The property model represents the path to the model resource, the name property defines a name to be able to refer to the model later, and the property handler specifies which model connector should be used to interpret the model, e.g., pcm for PCM models, java for Java projects. In the latter case the model path refers to the project root.

Aspect The central element of an aspect language is the declaration of aspects. This is realized with the Aspect rule (see Listing 10.24). It combines

a pointcut with a list of advices. In the present realization an advice can have parameters, which allows to reuse the same advice in different aspects. In addition this provides access to the model context defined by the pointcut.

```
Aspect:
    'aspect' pointcut=[Pointcut|QualifiedName] ':' advices+=UtilizeAdvice (',' advices+=UtilizeAdvice)*
;

UtilizeAdvice:
    advice=[Advice|QualifiedName] ('(' (parameterAssignments+=Value (',' parameterAssignments+=
            Value)*)? ')')?
;
```

Listing 10.24. Rules used to declare and configure aspects

From a typing point of view, the values used as parameter assignments must match the types specified in the advice declaration. An advice in the IAL is like an operation which is invoked with a sequence of values for its parameters.

Advice The advice specification (see Listing 10.25) is similar to the declaration of a procedure or function with a Unit return type (cf. Chapter 2). It can have multiple parameter declarations to configure the advice depending on the context defined by a pointcut associated by the aspect declaration. Each AdviceParameterDeclaration is typed via a TypeReference which allows to refer to NamedTypes and collection types with a named type as element type. The body of the advice comprises multiple collector actions. These collectors can be set to be invoked before and after the execution of the referenced element. The latter might be the regular termination of an operation (AFTER) or an exception (AFTER_EXCEPTION). The Collector rule covers this distinction with the property insertionPoint.

Each Event of a Collector can be seen as an instantiation of an event of a specific type and its storage into a monitoring log. The type is defined with IRL record types [JHS13] and initialized with a sequence of values which must conform to the properties declared by the type. These values may refer to parameter declarations, internal function, reflection values, and runtime data.

```
Advice:
    'advice' name=ID ('(' (parameterDeclarations+=AdviceParameterDeclaration (','
          parameterDeclarations+=AdviceParameterDeclaration)*)? ')')? '{'
        collectors+=Collector*
'}'
;

AdviceParameterDeclaration:
    type=TypeReference name=ID
;

Collector:
    insertionPoint=InsertionPoint events+=Event+
;

Event:
    type=[irl::RecordType|QualifiedName] '(' (initializations+=Value (',' initializations+=Value)*)? ')'
;

enum InsertionPoint:
    BEFORE = 'before' |
    AFTER = 'after' |
    AFTER_EXCEPTION = 'exception'
;
```

Listing 10.25. Syntactic rules for the specification of advices in the IAL

Value Expressions Monitoring data is defined by value expressions. They are used to assign values to an advice in an aspect declaration, and they are used inside an advice to specify the values for a monitoring event. As the rule Value, depicted in Listing 10.26 shows, the language supports four kinds of values.

First, Literals are fixed values, like a number or a string, which are specified by the developer. Second, ReferenceValues refer to the static model context and its structural information, like signatures of operations and classes. It can also be used to access the respective runtime value of a model element, if that represents data. This feature highly depends on the model handler and the therein supported features. For example, in some application models internal data in form of properties or variables is not present. In that case this feature cannot be used. Third, an InternalFunctionProperty allows to access runtime values from the monitoring framework, like a timestamp or trace id. Fourth, the advice parameters can be accessed. As

```
Value: Literal | ReferenceValue | InternalFunctionProperty | AdviceParameter ;

ReferenceValue:
    (query=LocationQuery)? property=Property
;

Property: RuntimeProperty | ReflectionProperty ;

InternalFunctionProperty:
    function=InternalFunction
;

enum InternalFunction:
    TIME = 'time' |
    TRACE_ID = 'traceId' |
    ORDER_INDEX = 'orderIndex'
;

ReflectionProperty: '$' function=ReflectionFunction ;

RuntimeProperty: {RuntimeProperty} '#' ;

enum ReflectionFunction:
    NAME = 'name' |
    SIGNATURE = 'signature' |
    CLASS ='classname' |
    RETURN_TYPE = 'return-type'
;

AdviceParameter:
    declaration=[AdviceParameterDeclaration|ID] ('[' collection=Value ']')?
;
```

Listing 10.26. Grammar rules for value expressions

these parameters can have collection types, the AdviceParameter rule allows to access individual values by specifying a value in brackets.

Pointcut A pointcut is a construct to find join points in a base model. They can be defined as complex insert, remove, and replace operations [MKB+08]. In the IAL, pointcuts are limited to model queries which return a list of join points. The join points are nodes in the base model and express which node is extended by the advice. The actual insert, remove, and replace operations must then be done by the weaver.

The pointcut rule, depicted in Listing 10.27, comprises a name, a query

over the hierarchical structure, and an optional pattern to match operation declaration. The name property allows to refer to the pointcut in the textual DSL. The query over the hierarchical structure of the base model is called LocationQuery and employs an XPath-like query syntax [W3C14]. A query can be specified utilizing the classes, attributes, and references of the meta-model of the base model. However, in the model handler certain classes can be hidden from the location query, as they are interpreted as classes used to model operations. The language provides a specific query syntax for operations, which is represented by the OperationQuery rule. The base model which is queried by the location and operation query, is specified through a reference to an ApplicationModel (see Listing 10.27).

```
Pointcut:
    (annotation=Annotation)? 'pointcut' name=ID
    'class' model=[ApplicationModel|ID] '.' location=LocationQuery
    ('operation' operation=OperationQuery)?
;
```

Listing 10.27. Syntactic rule for a pointcut declaration

A pointcut in the IAL can be seen as a function. Its return type is a collection of nodes of the base model. The semantic rules in Figure 10.9 express how the return type is derived for both, pointcuts with operation query and those with a location query only. T-Pointcut-Operation expresses that the type of the operation query is used as the return type, and T-Pointcut refers to the location query.

It is important be aware that both T_0 and T_1 are always collection types, which is ensured by the last two premises in T-Pointcut-Operation and the last premise in T-Pointcut following the notation from Pierce [Pie02, p. 147]. We omitted the reference to the ApplicationModel, as this is only a technical requirement to ensure unique names. It could, however, be expressed as the selection of a specific typing context Γ.

Model Query The central element of a pointcut is a model query which consists of operation and location queries. In the IAL grammar Location-Query, depicted in Figure 10.10 on page 227, defines that the query is a

T-Pointcut-Operation

$$\frac{\Gamma \vdash name : String \quad \Gamma \vdash location : T_0 \quad \Gamma \vdash operation : T_1 \quad T_0 = List\ T_a \quad T_1 = List\ T_b}{\text{pointcut } name \text{ class } location \text{ operation } operation : T_1}$$

T-Pointcut

$$\frac{\Gamma \vdash name : String \quad \Gamma \vdash location : T_0 \quad T_0 = List\ T_a}{\text{pointcut } name \text{ class } location : T_0}$$

Figure 10.9. Semantic rules for the pointcut syntax rule

sequence of nodes separated by a dot. The rule follows the idea of a chained type reference, illustrated in Listing 10.20 on page 214. Each node refers to a Container instance or any kind of wildcard, which is explained below.

```
LocationQuery:
    node=Node (('.' specialization=LocationQuery) | (composition=CompositionQuery))?
;

CompositionQuery: {CompositionQuery}
    (modifier=QueryModifier)? '{' (subQueries += LocationQuery)* '}'
;

enum QueryModifier:
    INCLUDE = 'include' |
    EXCLUDE = 'exclude'
;
```

Listing 10.28. Location query syntax

As it might be useful to include and exclude certain elements from the set of nodes returned by the LocationQuery, it is possible to create composite queries. Composite queries are sets of location queries which are either combined when include is used as modifier, or removed from the set when exclude is used. However, it is not possible to start with an exclude query, as nodes cannot be removed from an empty set. An example query is depicted in Listing 10.29. In this query, all inner nodes are collected, except for those inner nodes which are children of clazz. In addition spec nodes

are included. For this addition two different syntactical declarations are valid which yield the same results (see Listing 10.29). However, alternative one, first collects all spec nodes and then adds this collection to the parent collection. And alternative two, directly adds all spec nodes to the parent collection.

```
base {
    node.*.inner
    exclude { node.clazz.inner }
    include { node.clazz.inner.*.spec } // alt 1
    node.clazz.inner.*.spec // alt 2
}
```

Listing 10.29. Example composite location query

The typing rules of the LocationQuery are depicted in Figure 10.10 on page 227. The rule T-Node expresses that Γ entails that node has type T when node has type T in Γ. The general typing rule for LocationQuery defines that the type of the complete query is the type of the right part of the query expression. In essence this can be a Node, a LocationQuery, or a CompositionQuery. More interesting is the rule for the CompositionQuery. Its type is the common subtype of all types used in each sub-query. Alternatively, it would be possible to create a variant type with all types of the premise. However, that would require dynamic typing and runtime type checking which is hard to debug for language users.

T-Node

$$\frac{node : T \in \Gamma}{\Gamma \vdash node : T}$$

T-LocationQuery

$$\frac{\Gamma \vdash node : T_0 \qquad \Gamma \vdash query : T \qquad query : T \in \Gamma}{\Gamma \vdash node.query : T}$$

T-CompositionQuery

$$\frac{\{\Gamma \vdash subQuery_i : T_i\}^{i=1 \ldots n} \qquad \forall\ T_i | S <: T_i}{\Gamma \vdash subQuery_1 \ldots subQuery_n : S}$$

Figure 10.10. Semantic rules for location query related syntax rules

Path Navigation The elementary part of LocationQuery is called Node. A Node is one element of a path expression. The IAL supports four different node types which can all be constraint over the properties of a node (see Listing 10.30). The ContainerNode refers to a mapping model container identified by its type name or by the reference label. The common interface of Container and Reference is Feature. The scope of potential features are defined by the left part of the path expression. Instead of a specific container, it is also possible to specify a wildcard (∗) which expresses that all features are considered. Therefore, the type of the wildcard is the common super type of all contained elements.

```
Node:
    (SubPathNode | WildcardNode | ParentNode | ContainerNode) ('[' constraint=PropertyConstraint ']')?
;

ContainerNode:
    container=[mapping::Feature|ID]
;

WildcardNode: {WildcardNode} '*' ;

SubPathNode: {SubPathNode} '**' ;

ParentNode: {ParentNode} 'up' ;
```

Listing 10.30. Grammar rules for path navigation

While one asterisk (∗) represents a wildcard over the contents of one container, two asterisks (∗∗) are a wildcard for sub paths. This means, the collection of resulting nodes contains the contents of the present container and also all contents of each contained element, recursively. This recursion is only stopped when a node is found which matches the node right of the sub path element.

Finally, the DSL allows to navigate to the parent node. This is modeled with the symbol up. For example `node.subnode.up` is the same as `node`, and `root.node.subnode.up.up` is the same as `root`.

We omit the formal typing rules for sub path wildcard, as it is complicated and most likely subject to change in future versions of the DSL. However, we define a general rule for typing of ContainerNode. The rule

T-ContainerNode is depicted in Figure 10.11. It ensures that any expression formulated with the property constraint grammar must have a boolean type. Furthermore, the rule T-WildcardNode suggests that the type of the wildcard is the super type of all types of the fields of the container (called *parent* in Figure 10.11).

$$\frac{\Gamma \vdash node : T \qquad \Gamma \vdash t : Bool}{\Gamma \vdash node[t] : T}\ \text{(T-ContainerNode)} \qquad \frac{\Gamma \vdash parent : T}{\Gamma \vdash parent.\mathrm{up} : T}\ \text{(T-ParentNode)}$$

$$\frac{\Gamma \vdash parent : T \qquad fields(T) = \overline{C\ f} \qquad \forall C \in \overline{C} | C <: T_1}{\Gamma \vdash * : T_1}\ \text{(T-WildcardNode)}$$

Figure 10.11. Typing rules for node paths

Property Constraints The path navigation allows to follow the references in models and select nodes depending on their location in a graph. However, it might be necessary to exclude certain nodes from the result collection based on their context and the attributes of the node. Therefore, the language supports *property constraints*.

The syntactic rules depicted in Listing 10.31 allow to formulate boolean expressions with ConstraintElements which match elements in the collection provided by a path navigation. PropertyConstraint supports logical operators and PropertyConstraintCompare provides comparison operators.

The two syntax rules PropertyConstraint and PropertyContraintCompare use a special feature of Xtext which allows to collapse an AST. In PropertyConstraint the logical operator and right operand are optional. If they are left out the rule in fact does not produce a PropertyConstraint node, but a PropertyContraintCompare node. In the same way PropertyContraintCompare may fold into a ConstraintElement.

A ConstraintElement can be an attribute of a node, a literal value, and a test of the type of a node. To be able to access attributes of other nodes

```
PropertyConstraint:
    PropertyConstraintCompare ({PropertyConstraintExpression.left=current} logic=LogicOperator right=
            PropertyConstraintCompare)?
;

enum LogicOperator:
    AND = '&&' | OR = '||'
;

PropertyConstraintCompare:
    ConstraintElement ({PropertyConstraintExpression.left=current} operator=CompareOperator right=
            ConstraintElement)?
;

enum CompareOperator:
    EQ = '=' | LIKE = '~' | NE = '!=' |
    GR = '>' | LW = '<' | GE = '>=' | LE = '<='
;
```

Listing 10.31. Syntactic rules for base model object property constraints

reachable via the model graph the LocalQuery rule allows to prefix any attribute access or type test with a nested path navigation.

```
ConstraintElement: Literal | LocalQuery ;

LocalQuery:
    (locationQuery=LocationQuery)? (property=ModelProperty | typeof=Typeof)
;

ModelProperty: '#' reference=[structure::Feature|ID] ;

Typeof: 'istypeof' '(' reference=TypeReference ')' ;
```

Listing 10.32. Syntactic rules expressing access to base model attributes and nodes

The associated typing rules, depicted in Figure 10.12, must address these special cases with separate rules. Rule T-PropertyContraint defines that the left and right side of a logical operation must be of type boolean.

More interesting is PropertyConstraintCompare. The operator LIKE ($\sim$) is defined only for string data types. Therefore, we must define a separate typing rule for string comparison T-PropertyContraintCompare-String which also includes the $\sim$ as operator and is limited to string as type.

$$\frac{\textit{left} : \textit{Bool} \qquad \textit{right} : \textit{Bool}}{\textit{left}\ (\text{op} \in \{\&\&, ||\})\ \textit{right} : \textit{Bool}}\ \text{T-PropertyContraint}$$

$$\frac{\textit{left} : T \qquad \textit{right} : T \qquad T <: \textit{Number}}{\textit{left}\ (\text{op} \in \{=, \neq, >, <, >=, <=\})\ \textit{right} : \textit{Bool}}\ \text{T-PropertyContraintCompare-Numeric}$$

$$\frac{\textit{left} : \textit{String} \qquad \textit{right} : \textit{String}}{\textit{left}\ (\text{op} \in \{=, \neq, \sim, >, <, >=, <=\})\ \textit{right} : \textit{Bool}}\ \text{T-PropertyContraintCompare-String}$$

Figure 10.12. Typing rules for base model object property constraints

The elements in the constraint are ConstraintElements. These can be literal values, container attributes, and the test for type conformance. The literals have their specific types which are bound to them, e.g., a string literal is of type *String*. As a single ConstraintElement can also be a syntactic valid property constraint expression, the type of a PropertyConstraint can also be of any legal type. Thankfully, rule T-ContainerNode, depicted in Figure 10.11, prohibits that. It only applies if the constraint term is of type boolean. Therefore, no typing resolution can be found in cases where the type is different. This will be presented to the user as an error.

Apart from literals, there are two other kinds of constraint elements which allow to access attribute values and which allow to test an element for its type. First, the typing of the access to attribute values of containers is handled with T-ModelProperty depicted in Figure 10.13. The term 'this' in the rule represents the context expressed by the left side of the query, and the type T is the type of the context. The conclusion of T-ModelProperty shows that the type of the *reference* is the type of the corresponding field.

And second, the test for type conformance has actually two features. In the typing of the expression rule it is of type *Bool*. The rule T-Typeof checks if *reference* is a valid type. As any type would be sufficient, the premise could actually omitted. However, we wanted to express that *reference* is not

any term, but must be a type.

$$\frac{\Gamma \vdash \text{this} : T \qquad \mathit{fields}(T) = \overline{C\ f} \qquad \mathit{reference} = f_i}{\#\mathit{reference} : C_i} \ \textsc{T-ModelProperty}$$

$$\frac{\mathit{reference} \in T}{\text{istypeof}\ (\mathit{reference}) : \mathit{Bool}} \ \textsc{T-Typeof}$$

Figure 10.13. Typing rules for constraint elements

Operation Query The location query interprets objects of the base model as nodes of a graph without special semantics. In contrast, the operation query handles objects as if they represent operations. This mapping is realized by the model handler which maps classes representing operations and parameters to their mapping model representations. Through this feature, language users can refer to operations in their models in a more effective way. Therefore, the grammar aims to capture all facets of operations, like return types, parameter types, and modifiers for parameters and operations.

Typical modifiers for operations are `public` and `private`. However, the set of operation modifiers are derived from the mapping model. Similarly, parameter modifiers are also provided by the mapping model. In PCM [BKR09] the modifiers are `in`, `out`, and `inout` resembling those from CORBA [OMG06].

The typing for this part of the grammar is depicted in Figure 10.14. The resulting type of `OperationQuery` is the type of the class where `mapping::Operation` refers to. Furthermore, the types of the parameters, must be known types of the mapping model. As it is possible to specify an asterisk instead of a type, the type can also be the virtual type *Top* which matches any type.

```
OperationQuery: {OperationQuery}
    modifier=[mapping::OperationModifier|ID]?
    returnType=TypeReference?
    (
        (
            operationReference=[mapping::Operation|ID]
            ('(' parameterQueries+=ParameterQuery (',' parameterQueries+=ParameterQuery)* ')')?
        ) |
        '*'
    )
;

ParameterQuery: {ParameterQuery}
    modifier=[structure::ParameterModifier|ID]?
    (type=TypeReference|ID]|'*')
    parameter=[structure::Parameter|ID]?
;
```

Listing 10.33. Operation query syntax

$$\textsc{T-ParameterQuery}\ \frac{T \in Type \cup \{Top\}}{\Gamma \vdash parameter : T}$$

Figure 10.14. Typing rule for the parameter query. `Top` refers to any visible type and represents the asterisk ($*$) in the grammar rule.

Literals Finally, in Listing 10.34, we present the three literals of the IAL. All types support automatic type conversion. That means `INT` is applicable to all integer types, and `FLOAT` to all floating point types.

```
Literal: StringLiteral | IntLiteral | FloatLiteral ;

FloatLiteral: value=FLOAT ;

IntLiteral: value=INT ;

StringLiteral: value=STRING ;
```

Listing 10.34. The literals of the IAL

10.4.3 Generator API

The API for IAL generators utilizes the GECO framework to realize the fragments for the different technologies. However, it does not use the GECO generator composition language, as the IAL requires to choose the correct generator based on input model data and must be configured at runtime.

The API comprises a general dispatch and configuration class AspectLangGenerator with the aspect technology discovery routine, and generators for pointcut and advice generation.

The IAL supports different aspect injection frameworks. Through the model mapper, the generator API can discover which technology should be used for a specific application model. However, it is often not possible to infer the right target level technology, as different technologies are applicable and the base model generator does not provide such information in the trace model and via the model mapper. Therefore, the IAL allows to specify the intended technology with an annotation. Annotations and mapper information are handled automatically by the discoverAspectTechnology method of the central dispatch class AspectLangGenerator.

The AspectLangGenerator class implements a doGenerate method conforming to Xtext generator API. It aggregates pointcut and advice information and triggers the technology specific generators. The AspectLangGenerator implements, therefore, a specific aggregation method, e.g., createAspectJConfiguration, which collects the pointcut and advice data before invoking the corresponding generators. Many injection technologies, such as AspectJ [Lad09], JavaEE, separate aspects in a pointcut model including references to aspects, and an advice model.

The API is designed to support extensions to IAL and add new generators for additional technologies. For this case, we recommend to realize code and model generation for pointcut and advice in separate generators. Furthermore, we recommend to realize XML transformations as model-to-model transformation, i.e., the output model node should be org.w3c.dom.Document or a similar DOM class. This supports the reuse of the transformations in other contexts and allows to implement the model serialization in one single method. The API of class AspectLangGenerator provides, therefore, the method storeXMLModel.

10.4.4 Example Generators

The IAL allows to use any number of aspect injection technologies. In the current implementation generators for JavaEE [DK06], Servlets [Mor09], and AspectJ [Lad09] are provided. They all use the API discussed in the previous section. To further illustrate the construction of pointcut and advice generators, we discuss here the generators for AspectJ. The source code for the IAL can be found at Github[2] and in the replication package [Jun16a].

Pointcut Generator In AspectJ [Lad09], the pointcut configuration file `aop.xml` contains the pointcut expressions and the combination of advices and pointcuts, i.e., the aspect configuration. The advice implementation is kept in separate Java and AspectJ files. As the IAL allows to keep model aspects with different target languages and technologies in mind, the AspectLangGenerator class aggregates aspect information for target language and technology. Therefore, the pointcut generator is invoked with a collection of aspects which should be realized with AspectJ.

The `aop.xml` is an XML file. Therefore, we realized the fragment as a model-to-model transformation. The transformation returns an XML DOM which is then serialized with the standard XML serializer of the AspectLangGenerator class.

The pointcut queries of the IAL can be mapped by a direct query translation if the model mapper provides this functionality. For the Java model mapper such a direct query mapper is available. In case of the PCM the pointcut is evaluated by the mapper into join points and then translated with a PCM to Java trace model.

Advice Generator The advice generator is designed to generate AspectJ advices implemented in Java using annotations. As these advices can be constructed independently from the pointcut, they do not require additional information beside their own declaration. This allows to design the fragment to process one advice specification at a time. Each advice is realized with one Java abstract class and contains specific advice methods covering dynamic and static Java method invocations.

[2] IRL and IAL repository `https://github.com/kieker-monitoring/instrumentation-languages`

An advice class includes a set of static declarations to implement access to the Kieker [HWH12] monitoring infrastructure. Listing 10.35 depicts these declarations which instantiate the monitoring controller, obtain the time source, and optionally expose the trace registry to the aspect. This last part is only included in the generated classes when an event accesses the trace facility.

An aspect implementing class inherits standard functionality from AbstractAspectJProbe, which is used to control probe activation and may be used in logging.

```
@Aspect
public abstract class Abstract«input.name»Advice extends Abstract\name{AspectJ}Probe {
    private static final IMonitoringController CTRLINST = MonitoringController.getInstance();
    private static final ITimeSource TIME = CTRLINST.getTimeSource();
    «IF input.isTraceAPIUsed» private static final TraceRegistry TRACEREGISTRY = TraceRegistry.
          INSTANCE;«ENDIF»
```

Listing 10.35. Excerpt of the generator template containing the standard declarations of an instrumentation aspect class

The IAL does not support around advices for three reasons. First, they can be realized with one before and one after advice. Second, in AspectJ new features, like cflow cannot be used together with around advices [Lad09]. And third, if necessary a generator could easily map before and after advices in one around advice. Therefore, the IAL generator for AspectJ maps the declared events to before and after advices. Each advice is represented by separate methods. Furthermore, dynamic and static methods require different approaches to collect signatures. Therefore, multiple advice methods must be created. In case of calls where a caller and a callee are involved, this results in eight methods covering all cases. However, all these methods follow a similar layout as depicted in Listing 10.36. In detail the template for methods starts with the setup of the advice. It allows, therefore, to set one of the annotations Before, After, and AfterThrowing. Then it configures the advice tag with a pointcut specification. This seems to be confusing, as we generate the pointcuts into a separate configuration file. However, in AspectJ each aspect class must contain a pointcut. Therefore, they generate one pointcut called operation. Furthermore, the pointcut in AspectJ allows

to set properties of the advice method. As these differ between dynamic and static methods, the second part of the pointcut is covered with a templating variable.

```
@«annotation»("operation() && «pointcut»")
public void «methodName»(«parameters») {
    if (CTRLINST.isMonitoringEnabled()) {
        final String signatureString = this.signatureToLongString(«joinPointParameterName».getSignature
                ());
        if (CTRLINST.isProbeActivated(signatureString)) {

            // common fields
            «if (traceAPI) createTraceId»
            «collectors.createDataCollection(parameterAssignments)»

            // recording
            «collectors.map[it.events.map[it.createEvent].join('\n')].join»
        }
    }
}
```

Listing 10.36. Excerpt of the generator template realizing a monitoring method

In the method's body, the advice first checks if monitoring is enabled and subsequently if data recording should take place for the specific operation. If both conditions are true, the advice collects all data and records it with Kieker.

The IAL allows to define advices with parameters. In an aspect values can be assigned to these parameters which can also refer to the context of the base model. However, AspectJ does not provide such configuration of advices in the `aop.xml` file. Therefore, the generator produces a separate class for every utilized advice (see Section 10.4.2 on page 221). The data collection routine for the parameters is also handled by the createDataCollection template in the generator.

Like the pointcut generator fragment, the advice fragment also utilizes the trace model to resolve data types and property names used in the data collection. The complete implementation of the fragments is available at Github and via the replication package [Jun16a].

Chapter 11

Experimental Evaluation

The experimental evaluation comprises two case studies, namely the information system case study based on CoCoME and the embedded control system case study based on MENGES. They are supplemented by a number of qualitative interviews to address aspects of the approach which are not covered by the case studies, like reuse of generators, metamodels, and DSLs. Furthermore, they provide an external view on the approach from researchers and industry.

The execution of the case studies are described in Section 11.1 for CoCoME and Section 11.2 for MENGES. The results of the interviews are presented in Section 11.3, and a summary of the evaluation is given in Section 11.4.

11.1 Information System Case Study

The information system case study based on CoCoME [RRM+11] evaluates the GECO approach in a medium sized setup, incorporating existing metamodels and generators, e.g., ProtoCom [GL13], which are augmented by additional languages.

Two languages for modeling behavior and data types complement the PCM [BKR09] to provide the means to completely model CoCoME. For the purpose of monitoring, this is supplemented with two languages for event types and monitoring sensor application.

The driver for changes in the case study came from users of CoCoME, the iObserve research project [HSJ+15], which uses CoCoME in their design time and runtime observation and modeling approach.

In the following, we first introduce the involved languages and metamodels in Section 11.1.1 including their evolution. In Section 11.1.2 we

describe the execution of the case study, starting with the initial setup and the modifications performed during the case study. The results of the evaluation are then presented in Section 11.1.3.

11.1.1 Languages and Meta Models

In this case study, the PCM is used to model the overall component structure of CoCoME. However, the ability to describe data types in PCM is limited, especially the support of special features of the JPA. Therefore, we developed the DTL [Jun13] which allows to describe data types in a compact way. Secondly, the PCM does not support the specification of behavior in a way to write executable software, because the PCM addresses only an early state of software development and especially the quality analysis of software before the actual implementation. Therefore, we supplemented the PCM with a behavior language [Jun14a].

For the final addition of monitoring, we utilized the Instrumentation Record Language (IRL) and Instrumentation Aspect Language (IAL) [JHS13], whereas the IRL is part of the Kieker project [HWH12] and used to specify record types. Each record type represents one specific type of monitoring event, e.g., BeforeOperationEvent and AfterOperationEvent which are triggered when an operation is entered and left, respectively.

Data Type Language The data type language [Jun13] allows to model data types for Java entity beans providing an abstraction which is close to EMF and UML class diagrams.

The language organizes data types in packages, like Java. However, it is allowed to specify multiple data types in one file. The DTL supports the usual base data types (cf. Section 2.2), boolean, byte, short, int, long, float, double, char, string, date, id, and currency. The types follow the semantics of their Java counterparts, with the exception of string, date, id and currency. The latter types are modeled in Java with classes. The type id is mapped to long, but also marked as index property which will be automatically set once an object is stored for the first time.

Based on these base types, entities can be defined. Entities comprise of attributes and references to other entities, like record data types (cf.

Chapter 2). Each attribute and reference can be typed with a single base and entity type, respectively. In addition any named type can be used as value types of collections and maps, this includes base types, entity types, and enumeration types.

Entities can inherit attributes and references from one parent entity as its solely inheritance feature. Therefore, interface or template based inheritance are not present in the DTL. While such inheritance could be added to the language and would also provide a challenging evolutionary step for the generator, it was not necessary to model the data model of CoCoME, because CoCoME does not use multi-inheritance. Therefore, we did not implement it and are leaving this to the next user who requires that feature.

The language also allows to define enumeration types. These enumerations can also inherit enumeration values form one parent enumeration.

To support JPA features, the DTL supports special modifiers for references to be able to declare them to be opposite of another reference, mimicking the same feature of EMF and providing the necessary information to declare the correct JPA mapping to specify the feature in JPA semantics. Furthermore, both JPA and EMF support transient values. Therefore, the language allows to declare transient fields.

Behavior Language The behavior language [Jun14a] allows to specify expressions to implement operation bodies of component types specified with the PCM. We extended this language in four steps for the evaluation. The initial revision of the language allows to reference operation declarations of the PCM. These references are required by the weaver to integrate operation bodies with the stubs provided by the `ProtoCom` generator.

This first revision allows to specify additional variables and values for components and operations. For the operation body, three statement types are supported, namely decisions `if-then-else`, iterators, and assignments. Furthermore, the language supports expressions in all types and operation calls.

The second revision added support for stateful and stateless components. Until then all components where considered stateful. However, this limits the ability to parallelize and distribute components. As CoCoME is deployed in a cloud environment, distribution and automatic reconfiguration are

important features. Therefore, we added support to indicate stateless components, which was a necessary addition to improve the ability to distribute components.

In the third revision, we added support for the EJB life-cycle. In all previous language revisions it was not possible to define special initialization and destruction operations for a component. While this is not always necessary, it is provided by EJB containment to separate important startup tasks from normal operation.

Finally, we added support for JPA to the language, allowing to express simple operations inside the language instead of calling external Java helpers. We added, therefore, a new statement type for data access, which supports three types of access: store, update, and delete. For the present scenario of CoCoME this was sufficient, as a query language would have been a huge addition. Thus it was decided to realize queries for CoCoME through Java helpers. In the future, this could be extended by a database query feature or the integration of yet another language to express database and model queries.

11.1.2 Experiment Execution

The experiment execution can be divided into five steps starting with the initial setup with the generators $T_{ProtoCom}$, $T_{Behavior}$, and T_{DTL} depicted in Figure 11.1. This general setup stayed unchanged during the next three evolution steps of the behavior language. Only in the last step the generator was extended by adding the monitoring aspect (cf. Figure 9.1 on page 176).

As shown in Figure 11.1, two of the megamodel patterns were used. The *Normal Aspect (P2)* pattern appears four times. To avoid confusion only two examples are given in the figure itself, between Behavior and DTL, and Behavior and PCM. In addition, the P2 pattern also applies to the relationship between PCM and DTL, and between EJBs, Servlets and Entities.

The generators do not utilize the framework described in Section 10.1, as they are model-to-text transformations realized with Eclipse and Xtext, utilizing the Eclipse build infrastructure. Therefore, we integrated our additions with this underlying build technology.

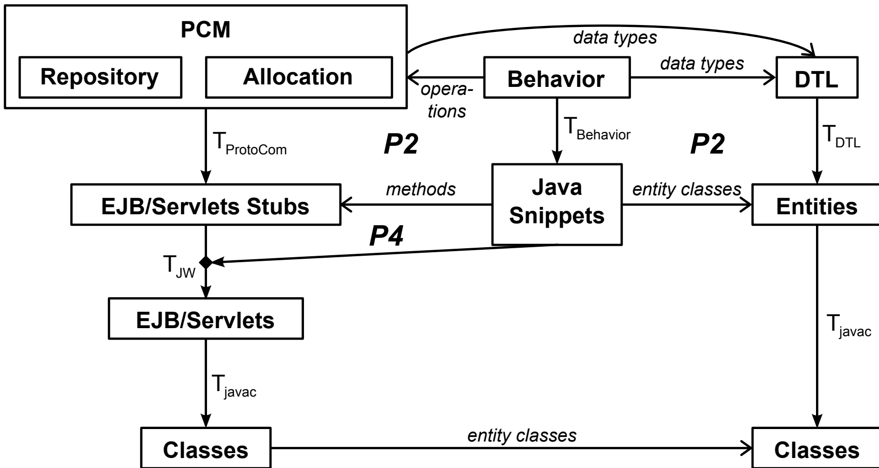


Figure 11.1. Initial version of the megamodel for the CoCoME generator, supplemented with labels to indicate the involved megamodel patterns *Normal Aspect (P2)* and *Weaving (P4)*.

Initial Realization Initially, we set up the generator depicted in Figure 11.1. We started by installing ProtoCom which is part of the Palladio distribution.[1] In the evaluation we initially used Palladio version 3.5 and later switched to version 4.0, due to bugs found in the initial ProtoCom version which have only been fixed in the new version.

ProtoCom was then extended to provide a trace model information. This extension was realized by the iObserve research project [HSJ+15] which also uses CoCoME as case study and provides its own trace model infrastructure as part of a correspondence model [HSJ+14].

After the initial setup, we specified the CoCoME data model from the original documentation [RRM+11] and the online information.[2] Subsequently, we executed the code generation with ProtoCom, based on a PCM model of CoCoME provided by the iObserve project. As a last step, we specified behavior for a small set of the components defined in CoCoME to illustrate the

[1] Palladio Installation `https://sdqweb.ipd.kit.edu/wiki/PCM_Installation`

[2] CoCoME Website `http://www.cocome.org`

general feasibility of the composed generator. The complete implementation of CoCoME functionality is not part of this dissertation, but is performed by the iObserve project to realize its evaluation case study.

Evolution of Behavior Meta Model Based on feedback from iObserve, we extended the behavior language, as described above in three steps, starting with the support of stateless components which allowed for better distributability.

It was then discovered that the execution time of beans significantly when information which will be used throughout the lifetime of an instance can be retrieved once. The EJB life-cycle [DK06] provides therefore the two operations post-contruct and pre-destroy which are called after instantiation and before the disposal of an EJB instance, respectively. Finally, database access had to be integrated. For our evaluation, this was the last extension made for the CoCoME case study. However, iObserve, will extend the language if necessary to accommodate additional features in the near future.

Extending the Generator Megamodel As the iObserve project requires monitoring data, it was necessary to model the monitoring aspect of CoCoME as well. Therefore, we used the two languages IRL and IAL. The IRL comes with a model-to-text generator producing record type classes conforming to the Kieker monitoring framework API. And the IAL comprises several generators, one providing weaving information for the AspectJ weaver ajc and to configure interceptors for Servlets and EJBs.

The IAL had to be adapted to support the trace model provided by the iObserve extension. However, we did not measure this change, as the IAL is designed to be adapted to different modeling contexts. Therefore, no change to the language and its generators were necessary. However, we measured the size and complexity of the generator megamodel to evaluate the effect of the extension on the megamodel.

11.1.3 Evaluation Results

For all revisions of the behavior generator, we measured complexity, cohesion and coupling as explained in Chapter 4. In addition we collected information on the number of source code classes used to implement the language, the number of modules involved in the measurement, and the number of nodes and edges in the hypergraph.

To provide insight into the complexity of the megamodels and the changes between versions, we also measured their size and complexity. As a point of reference and for documentation purposes, we also measured all other generators used as fragments of the composed generator, with the exception of ProtoCom, which was developed elsewhere and is seen as a black box.

Table 11.1 presents all values measured for this evaluation. Due to size constraints of the document, we aggregated the cyclomatic complexity of methods in Table 11.1b in five instead of eleven categories. As these complexity values are only used to illustrate that complexity is not obfuscated by hiding it inside methods, this aggregation still allows to make this assessment. Detailed values and measurements can be found in the data and replication package [Jun16a].

Megamodels The overall size and complexity of the megamodel for CoCoME increases by 120 % and 73.55 %, respectively, when the instrumentation aspect is added. This duplication in size is caused by the large addition by the instrumentation aspect which consists of two source metamodels and six transformations, which is in essence almost the same size as the original setup. More interesting is in this case that the complexity increase is significant lower than the size increase, which indicates that this can also be perceived as a modular addition. This observation is also supported by an examination of the complete megamodel, where the interface between both parts comprises the IAL operations references, the `aspect.xml` and `web.xml` methods references, and the trace model of ProtoCom.

IRL Table 11.1a shows that the IRL comprises the largest generator in number of classes. This originates from the fact that the IRL generator

Table 11.1. Measurement results of the generators and complete system of the CoCoME case study

(a) Revision information and counting measurements

	git Revision	# of Classes	LOC	Modules	Nodes	Edges
Megamodel v1		–	–	–	17	24
Megamodel v2		–	–	–	32	46
IRL	751b193c716	17	2052	28	208	593
IAL	751b193c716	1	641	8	28	47
DTL	4d2b80d903c	1	865	13	43	79
Behavior r1	be2dafbc53a	6	641	16	56	125
Behavior r2	83acc26830d	6	650	17	57	127
Behavior r3	0961df26eb7	6	723	17	58	134
Behavior r4	0c87a9e84c4	6	821	17	64	156

(b) Information theory based measurements

	Size	Complexity	Cohesion	Coupling	Cyclomatic Complexity				
					< 3	< 5	< 7	< 10	$\geqslant 10$
Megamodel v1	76.46	133.19	–	–	–	–	–	–	–
Megamodel v2	167.89	231.15	–	–	–	–	–	–	–
IRL	1452.28	4470.54	0.013395	3462.61	100	4	0	0	4
IAL	140.30	246.68	0.064206	156.50	20	2	1	0	0
DTL	221.99	475.40	0.046843	352.10	13	6	0	1	1
Behavior r1	314.25	802.70	0.043709	594.93	27	2	1	3	2
Behavior r2	321.54	813.99	0.043965	605.90	27	2	1	3	2
Behavior r3	328.86	873.37	0.043965	654.83	27	2	1	3	2
Behavior r4	373.23	1041.43	0.042075	781.88	27	2	1	3	2

supports multiple outputs for Java, C, and Perl. Furthermore, it comes with a generator for Java test and factory classes. The hypergraph size indicates that is the largest generator in this case study and also the most complex one.

IAL The IAL has a very compact generator which comprises the three transformations T_{sensor}, T_{web}, and T_{aspect}. Therefore, the whole hypergraph consists primarily of one observed system class and a small set of framework classes. Therefore, the cohesion is very similar to the module cohesion. Like, the IRL, the IAL was designed before we collected decomposition patterns and defined a generator decomposition approach. Therefore, they both do not follow the modularization scheme introduced by GECO.

DTL The DTL was implemented by the iObserve project [HSJ+15] without addressing internal decomposition, as proposed by GECO. This resulted also in a monolithic single module implementation. Also it is a very small language which limits the potential benefits of modularization based on classes.

Behavior The main interest in this evaluation are the changes to the behavior language and how this affects the generator based on four revisions of the generator.

First, the initial design of the generator, depicted in Figure 11.2 along the criteria in Chapter 8, seems to be stable, as the number of classes do not change over the four evolution step. The change in module count depends on minor changes in a model query which utilize lambda function in Xtend which are mapped to anonymous classes in Java.

Second, the introduction of one new property for stateless EJBs required one additional template method, resulting in one additional node and edge to realize the new feature. This minor change had also only minimal effects on coupling, complexity and size. Cohesion even increased, indicating that the intra-module complexity increased more than the complexity of the complete graph did, which indicates more interconnections in Revision 2 than in Revision 1. Also, this indicates that the addition was kept inside

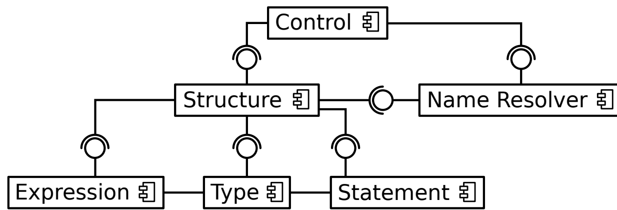


Figure 11.2. Architecture of the $T_{Behavior}$ generator

the modules and the limited increase in coupling indicates that the changes were local.

Third, the support of initialization and destruction method calls by the DSL caused larger metamodel changes and required more additions to the generator. However, the addition also only resulted in one new template for these functions, which resulted in one new method and, subsequently, in one new node. The increased number of edges indicates that the new templates make use of other templates and methods. The spike in complexity and coupling indicates that the calls are inter-module calls. A code review shows that indeed new calls originate in the structure module of the generator and use templates from the statement module (cf. Figure 11.2).

And finally, we added database access statements and expressions to the language supporting JPA. This resulted in a large addition to the statement and expression module resulting in an increased overall size and complexity. The massive increase in complexity and coupling suggests that the addition of the database features inside the existing modules of the generator might be a bad choice and that the generator should be refactored by creating a separate class for the database functionality.

11.2 Control System Case Study

The control system case study is based on the joint research and industry project MENGES [GHH+12], which developed a set of DSLs and a generator over a period of three years. During the project run time, the developers added and removed language features and modified the associated genera-

tor (see Section 9.4.2). The feature changes were driven by DSL user feedback and domain analysis.

Therefore, the case study provides typical change scenarios present in agile development and evolution, as single features where added over time and evaluated in use case scenarios. Subsequently, based on evaluation outcome, larger changes were performed, where language features were replaced by different constructs, e.g., replacement of commands by message protocols (see Appendix C.1).

From the original project, we also possess coarse-grained time measurements of the effort invested in the generator. Therefore, we can evaluate the practicality and to some extent the cost-benefit of GECO in this scenario.

The experiment was performed as a re-execution of the development, where we extracted features from documentation and the code base of the original project and reimplemented them in a new generator based on GECO. The extraction and reimplementation were performed for each selected revision of the original project, simulating the progress of the project. The implementation of the new generator was performed by one researcher and a student volunteer.

To provide a better understanding of the MENGES DSL, we introduce in Section 11.2.1 the underlying metamodels which correspond with the different views present in the case study. Section 11.2.2 explains the process of recovering the features and changes of the original project. Section 11.2.3 discusses the simulation of the evolution based on the recovered features and changes. Section 11.2.4 summarizes the code evolution of the old generator. Finally, Section 11.2.5 discusses the results of the evaluation.

11.2.1 Languages and Metamodels

The MENGES project developed multiple DSLs to cover different views and concerns of the specification of electronic railway control centers based on PLCs [GHH+12].

At the beginning of the development of the old generator, these languages used one large metamodel for all types, structures, and declarations, e.g., of operation signatures, and one metamodel to specify behavior. These were supplemented by small metamodels for hardware description. As the

large *types* metamodel seemed less maintainable to the developers, they split the metamodel up into eleven smaller metamodels in the beginning of generator development.

The partitioning of views and concerns among the languages was stable during the time frame of the original project used in the experiment. Therefore, the languages provide a better insight into the structure of the project than the metamodels. Even though the metamodels reflect the language partitions, they are not split along these partitions. In the following, we briefly introduce the different language constructs provided by the MENGES project [GHH+12].

Interlocking Elements are component types which comprise a set of methods implementing internal functionality and realize interfaces. The actual method implementation are partly realized in conditionals, state machines, and processes which are specified separately.

Communication Descriptions are connector types which may define properties for data exchange and message protocols.

Enumerations and State Sets allow to specify general enumerations and special sets used for state machines.

Conditionals are like decision trees. In each leave of that tree an action can be triggered which will then be executed.

Actions are sequences of statements which allow to implement data manipulations.

State Machines are defined over sets of states and transitions between these states. Actions are assigned to transitions of the state machines to specify their side effects.

Processes allow to model workflows similar to UML activity diagrams.

Deployment allows to specify hardware instances and the distribution of components and connectors on the hardware instances.

Instantiation is a DSL partition which allows to configure all component instances.

These language constructs were extended and changed during the course of the project. For example, commands and answers were used to model inter-component communication in the communication descriptions. However, commands and answers only allowed to define pairs, one command was related to one answer. However, the MENGES developers wanted to describe the complete communication protocol between two components. Therefore, they replaced commands and answers by messages and a protocol specification based on a regular expression, which provided a more expressive way to describe legal message sequences.

11.2.2 Recovery of Changes

In MENGES all documentation and program code were stored in a central VCS repository realized with Subversion. We analyzed this original documentation and code repository for feature descriptions and their implementation to reverse engineer the functionality of the original generator.

The general intent was to simulate the development of a generator in distinct steps. Therefore, we analyzed only the development from one to the next revision and used this information to implement and extended the new generator based on GECO principles. This provided us with the same information and knowledge the developers of the original generator had, and prevented to develop the new generator in knowledge of future changes. Unfortunately, most documents stored in the repository referred to tickets of the ticket system of the MENGES project which was not available anymore. In addition, many of the Subversion commit messages are rather brief and not descriptive of the actual changes to the generators. Therefore, we relied mainly on code analysis. For the code analysis, we used a code difference tool marking deviation between the two source code revisions compared in every analysis step. Based on this comparison, we collected the additions and deletions of the code and how they affect the functionality.

11.2.3 Simulation of the Evolution Steps

The simulation of the MENGES project was performed based on distinct revisions of the document and code revisions stored in the VCS. To be able

to identify and tag these revisions, and be able to provide us with a running example of the original code generators, we had to compile and execute these revisions.

However, the original tool chain could not be used, as the old tool setup was not archived and could not be rebuild properly. Therefore, we had to adapt each selected revision to present tool and framework implementations. While these changes were minimal, it would have resulted in many subversion branches occupying a lot of disc space. Therefore, the original repository was migrated to a `git` repository which was then used to tag and branch revision for the experiment.

Based on an up-to-date setup of tools and frameworks, we performed the same procedure for each revision: We identified the first revision of the original generator (G_{old}), created a branch for that revision and adjusted the code artifacts to the current tooling and framework to fix any incompatibilities. Subsequently, we introduced a new project for the new generator (G_{new}) and implemented the same initial features as in G_{old}.

After this initial setup, we performed the following activities for each revision:

1. We identified a distinct revision of the generator and the DSLs and tagged them in the `git` repository. The tagging was based on the documentation and code analysis, as the examination of the migrated commit messages were often insufficient. Each tagged revision comprises multiple commits.

2. Based on the identified documentation and source code from the repository, we determined which tasks the developers really performed in each revision, like feature introduction and maintenance operations.

3. A branch was created for the tagged revision, which we corrected to compile with our up-to-date development environment.

4. To be able to find the corrected revision later for measurement, we tagged it, e.g., `revision-2`.

5. For the revisions one to four we introduced the corrections of the previous DSL and generator branch by merging them into the present revision. We considered this an effective way to transfer the changes. However,

this was more time consuming than reapplying the changes by hand and an automation script. Therefore, starting with Revision 5 we used an update script to perform the necessary adjustments.

6. We identified the differences between the present and previous revision of G_{old}, and documented the feature difference accordingly. For example, the file `differences rev-01 to rev-02.txt` describes the changes in the implementation and features between Revision 1 and Revision 2.

7. We implemented the identified features in the new generator and compared the output of both generators. Due to differences in code format and identifier naming, this was performed by hand. As not all revisions of the original generator were able to produce compilable output, we could not check the equivalence of the correctness of both outputs by compiling the output.

8. When both generators provided a comparable output, we analyzed both with the analysis metrics (see Chapter 4) for which we implemented an analysis tool.[3]

9. Finally, we committed and tagged this revision.

In total, we identified 14 revisions of the generator together with the corresponding languages and metamodels from the original repository covering the time from the first revision of the generator implemented in Xtend supporting Structured Text (ST) to the last revision which only supported ST. In later revisions, the generator was massively refactored to support a second output language of the IEC 61131-3 [IEC03] called Function Block Language (FBL). FBL serializes the complete functionality in XML instead of plain text section used for ST. However, the developers did not use an XML DOM to store the result and instead produced XML code with text templates. Therefore, they needed an additional internal model to store and retrieve references for elements in the XML output. Especially, the connection of components required to compute reference ids before the actual elements were generated.

[3] Analysis Tool `https://github.com/rju/architecture-evaluation-tool`

The refactoring and the design decision to use text templates to generate XML resulted in code which is hard to read, and it did not produce any usable output. Therefore, we could not use these revisions of G_{old} and compare them to G_{new} revisions. Thus, we decided that the last revision only supporting ST should be the last revision used in this evaluation.

For the evaluation, we had to decide whether to implement the intended feature based on our interpretation of the original code or duplicate the implemented behavior which might deviate from our interpretation. We decided to replicate the original behavior as close as possible without violating GECO construction principles for two reasons: First, the extraction of the intended behavior from the analyzed code may lead to a misconception of the original programmers indent. Therefore, an implementation based on such an interpretation might deviate too much from the original implementation rendering a comparison meaningless. Second, the algorithm of the intended feature may have a different complexity than the original behavior. This would hinder a comparison of the measurements. Therefore, we followed the original behavior as close as possible, except where they violate the design principles of GECO.

Particularly, the original generator uses many caches to handle context information inside one module of the generator and between different parts of the generator. In GECO such caches cannot be used for three reasons:

1. In GECO the execution of fragments can be of any order (even parallel) as long as data dependencies are fulfilled. Therefore, caches might not be completed.

2. GECO requires that information is shared between fragments either via input and output of models or via trace models. These caches do not fall in any of these categories.

3. The caches used in the original generator realize global state. GECO implies that global state is limited to output and trace models.

Caches are used in the original generator to store trace information and target level names. In addition, they provide and handle index values of elements. In the new generator, we realized the same functionality using trace models, name resolvers, and model queries.

11.2.4 Code Evolution

Before we discuss the measurement results, we provide this summary on code and feature alterations. Detailed information on these alterations can be found in Appendix C.1. For a better understanding, we divide the development in six phases which express the focus of the developers for specific ranges of revisions.

Structure and typing The first two revisions of the original generator (G_{old}) established basic infrastructure, like constant declaration and name resolving, and the modularization of the generator. In this stage, they focused only on the modularization of typing structures. In Revision 3 the developers added source level type resolving which is necessary for composed types, and to resolve properties and operations of types in a subtyping context. They also added a global state handler including a cache for source to target model element mapping and a context-based set of counters. To ease the development of generators they defined a set of common code templates. In Revision 4 the last missing partitions of the structural part of the MENGES DSLs were realized, namely instantiation and deployment. This addition also provided the first implementation of the translation of references between source and target model. Furthermore, the developers implemented a first version of statements for dependency-tree operations called conditionals.

Expressions and statements Starting with Revision 5, they added support for expressions and statements to the generator. As statements had been introduced first in Revision 4, they were now moved to the expression part of the generator and massively refactored. Expressions are generated in different modules to handle complex expressions which cannot be mapped directly to target model equivalents. For example, boolean operations, like for all (∀) and exists (∃), which operate on value collections, are realized with loops in the target model. Therefore, such functions must first be computed and subsequently their result is used in the remaining expression. The used implementation does not support nested complex expressions which could have been avoided by realizing complex expressions in separate functions.

Refactoring and communications After the large addition of expressions and statements, Revision 6 mainly shows additional refactoring and code cleanup. The refactoring was used to improve modularization and reuse of code inside the generators. The only added feature were communication between components. In Revision 7, the communication constructs were extended and the construction of operations realize, which decision trees were refactored.

Polymorphism In Revision 8 the polymorphism mechanism was extended and corrected. This was necessary due to multiple shortcomings of the old mechanism. First, different parts in the original generator computed indexes differently. Second, the reference construction module did not support polymorphism which resulted in references pointing to the operation implementation of the base type.

Timers and template improvements Revision 9 introduced additional statements and expression features which were left out in Revision 5 and 6. In Revision 10 timers were added and static text strings in templates were replaced with constants. The latter required minor refactoring, but hand minimal effect on coupling.

In Revision 11 the naming scheme was changed. The MENGES DSL uses packages with a `Java` like structure. However, the target language does not support package names. Therefore, all global names for functions and variables must be fully qualified. This can result in long names which are hard to read. However, target code readability was an important requirement, as the target code must be verified by human technicians. Therefore, functionality was added to remove the common prefix of packages from all names which are identical in all names. For example, `de.menges.common` and `de.menges.device` share the common prefix `de.menges`. Apart from naming, the developers modified the indexing functionality again and introduced the timer reset statement.

Maintenance In the remaining revisions, from Revision 12 to 14, the developers added the duration data type as additional integer type for

timers, extended text templates, altered timer evaluation to make it more robust and to support the duration data types, and they introduced several changes to the state machine generator.

11.2.5 Measurement Results

The measurements for all metrics show a significant increase between Revision 4 and Revision 5 coinciding with the introduction of expression support (cf. Figure 11.3 and Figure 11.4). Other changes in the code base have a more subtle effect and only show in specific metrics. We now discuss all these changes along the six phases described above.

Structure and typing For the original generator (G_{old}), the developers gradually added support for each type. Therefore, the module count and lines of code also increased gradually (see Figure 11.3). The same can be perceived to some degree with other metrics, like node and edge count. It also affects coupling and complexity (see Figure 11.4).

The new generator (G_{new}) was developed differently. The GECO approach requires to create one generator fragment for each type present in the metamodels. Therefore, G_{new} comprised of eight classes for the fragments, their modules, and the main class controlling the execution of all fragments. In Revision 2 the class count increased to 13 classes. In contrast, the class count of G_{old} increased from two to 25 classes (see Figure 11.3).

In order to test whether code alterations affect complexity intra-modular or cause an increase in inter-module complexity, we can compute the ratio between complexity and coupling (see Figure 11.5). In the structure and typing phase, the complexity-coupling-ratio for G_{new} is 1.04 in Revision 1, as complexity is low (78.22 bits) due to generator fragment stubs with minimal functionality. The ratio raises to 2.82, as the different type generators in G_{new} are implemented which share only minimal code at this stage. This decreases in Revision 4 to 2.07 when the first conditional statement generator code is added and type resolving code results in more commonly used code.

For G_{old}, the complexity-coupling-ratio decreases quickly and stays low (see Figure 11.5) which indicates that coupling and complexity are close

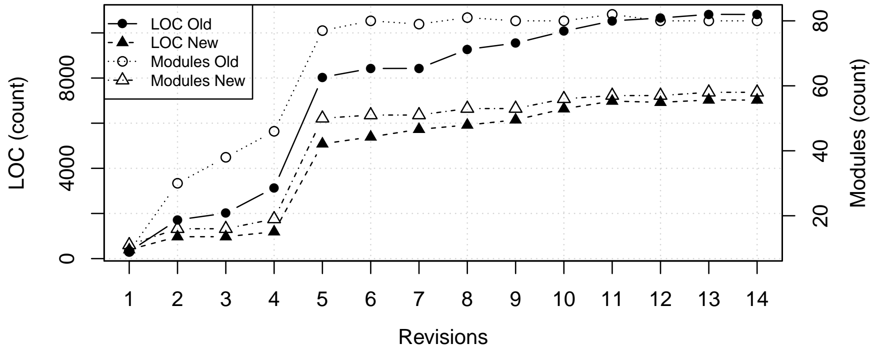


Figure 11.3. Development of module count and lines of code for G_{old} and G_{new} in the 14 evaluation steps

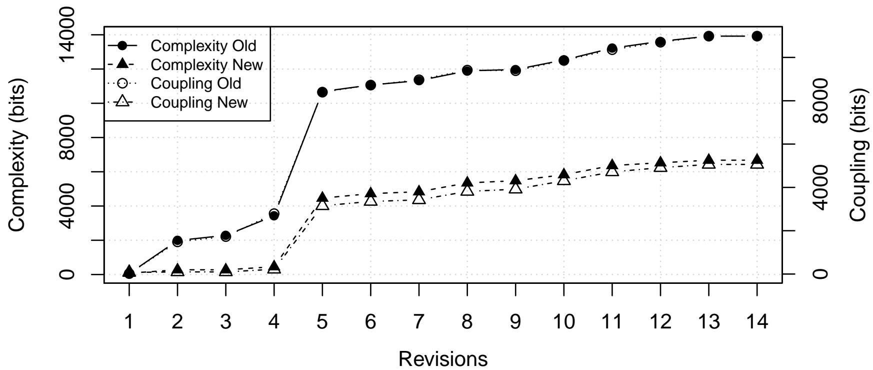


Figure 11.4. Development of complexity and coupling for G_{old} and G_{new} in the 14 evaluation steps

together.

Coupling is in fact the complexity of the inter-module hypergraph (see Section 4.4). Therefore, a lower ratio of complexity and coupling indicates less encapsulation of information inside the modules, which implies low cohesion. As high cohesion and low coupling are good for modularity, the

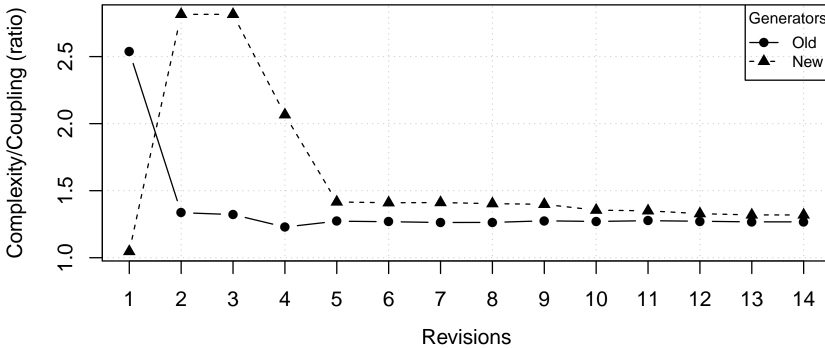


Figure 11.5. Development of complexity coupling ratio for G_{old} and G_{new}

lower the ratio, the worse the modularity.

Expressions and statements In Revision 5 the generator was extended to support expressions and statements. This resulted in the biggest increase in measurements for both generators (see Figure 11.3 and Figure 11.4). While node count increased similarly for both generators (384 nodes in G_{old} vs. 314 nodes in G_{new}) the difference in edges is much larger with 1091 and 708 for G_{old} and G_{new}, respectively (see Figure 11.6). The additional edges result in an increase in complexity and coupling in both generators. However, the increase in complexity and coupling is higher in G_{old} than in G_{new}. The complexity increased by 7258.63 bits for G_{old} and 4006.72 bits for G_{new}, and coupling by 5607.51 bits (G_{old}) and 2930.95 bits (G_{new}), resulting in 1.81 times higher complexity and 1.91 times higher coupling for G_{old}. This indicates that the application of the GECO approach results a better complexity-coupling-ratio which favors modularity. Furthermore, the lower complexity and coupling values themselves, also favors modularity.

However, the complexity and coupling might be affected by the programmers' coding style. Depending on the coding style the number of methods may be different resulting in more or less nodes. In addition, more methods can result in more method invocations which leads to more edges.

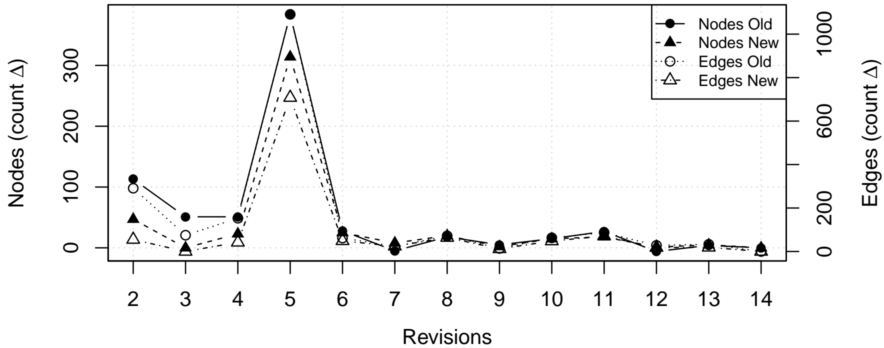


Figure 11.6. Node and edge count Δ for both generators

To mitigate the effect of personal coding styles on our measurements, we reused code fragments from G_{old} in G_{new} (see also Section 11.2.3). Furthermore, we base our assessment of the modularity not only on the measured values of complexity and coupling, but also on the ratio of both values. The ratio compares the complexity of the complete node and edge hypergraph with a portion of the hypergraph which describes the inter-module connections (coupling). Therefore, implementation details are factored out.

Figure 11.5 shows that the addition of expressions and statements to G_{new} (Revision 5) affected coupling more than complexity (ratio dropped from 2.07 to 1.42). This is caused by different fragments and the modules accessing the same expression processing modules. Furthermore, we reused the method bodies of the expression and statement generation from G_{old} in G_{new}, which also introduced the modularization strategy of G_{old} for this part of G_{new}. Nevertheless, the ratio of 1.42 (G_{new}) is better than 1.27 (G_{old}) by 11.26%.

Refactoring and communications After the large addition of expressions and statements in Revision 5, the developers started cleaning up the code of G_{old} to eliminate duplicate code and reorganize methods and templates. They also introduced inter-component communication. Therefore,

the changes in measurements are caused by both activities.

From Revision 6 to Revision 7, the refactoring caused the module count to shrink in G_{old} (see Figure 11.3). However, complexity and coupling still increased (see Figure 11.4). During the refactoring, the developers moved methods and templates and removed duplicate code, which should have resulted in a decreased count of nodes for G_{old}. Therefore, the increase in measurements is most likely the effect of the addition of inter-component communication.

In contrast, G_{new} shows one new module in Revision 6 which is part of the inter-component communication fragment. As G_{new} did not carry the same amount of duplicate code, refactoring and cleanups were not necessary. Thus, the addition of the new feature resulted in 33 new nodes in G_{new} compared to 23 new nodes in G_{old}.

However, complexity and coupling grew faster in G_{old} than in G_{new} (Figure 11.7). Complexity and coupling increased for G_{old} by 637.67 bits and 571.27 bits, respectively, while G_{new} saw an increase in only 372.25 bits in complexity and 273.68 bits in coupling. This indicates that GECO helped to limit complexity and coupling growth in this phase.

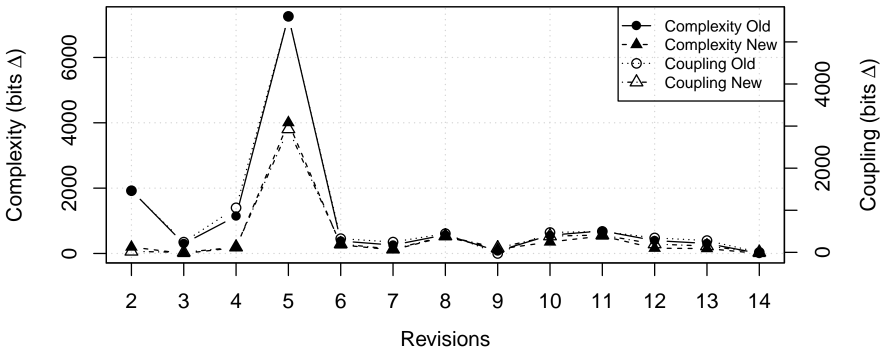


Figure 11.7. Complexity and coupling Δ for both generators

Polymorphism The reworking of the polymorphism mechanism created significantly more lines of code (843) in G_{old} (see Figure 11.3). This increase is caused by code duplication and the use of `Xtend` text templates in favor of `Xtend` expressions in the reference computing module. G_{new} has only one routine to compute the references and uses a more compact construction of them, as it does not distinguish references for the left and right side of an assignment. The context handling is realized by the contexts themselves, e.g., the assignment operation handles the output appropriately. Therefore, G_{new} shows only a small increase of 189 lines of code (see Figure 11.3). However, the other counting measures show a less drastic effect. G_{old} adds 18 nodes and 72 edges, while G_{new} adds 20 nodes and 64 edges. Still, the higher number of edges cause a greater increase in complexity 570.85 bits vs. 516.07 bits and coupling 446.41 bits vs. 387.57 bits for G_{old} compared to G_{new}. Here the differences between G_{old} and G_{new} are 54.78 bits (9.60 %) in complexity and 58.84 bits (13.18 %) in coupling, which indicates a stronger increase in inter-module connections for G_{old}.

Timers and template improvements This phase comprises minor alterations of templates, the introduction of timer statements, and instance configuration support. These changes had varying effects on both generators. In Revision 9 G_{old} had a decrease in coupling (-26.71 bits) and an increase in complexity by 76.12 bits, while G_{new} increased coupling and complexity by 104.79 bits and 127.66 bits, respectively. The decrease in G_{old} is the result of the replacement of a complex mechanism to handle context information in the expression generator by an approach which passes context information as parameter. In G_{new} a similar approach was used from the beginning. Therefore, the benefits of this change only affected G_{old} resulting in an improvement of its measurements.

However, in Revision 10 coupling increased by 472.59 bits and complexity by 558.23 bits for G_{old}. The measurement for G_{new} show only an increase by 386.20 bits for coupling and 352.43 bits for complexity. This shows that the introduction of timer functionality had a bigger impact on G_{old} than on G_{new}.

In Revision 11 both generators' name resolver functions were modified to support the removal of the naming prefix used for fully qualified names

and a new indexing model was introduced. In G_{old} this resulted in a large increase in nodes and edges (27 and 87) compared to 18 and 71 in G_{new}. More pronounced is the effect in complexity and coupling: (690.72 bits and 498.19 bits in G_{old} to 533.65 bits to 410.71 bits in G_{new}) which can also be seen in Figure 11.4.

Unfortunately, the developers of the original generator realized the new indexing process together with the naming changes. Therefore, the results do not show more prominently the higher locality of the changes in G_{new} than in G_{old}.

Maintenance The remaining revisions focused on bug fixing on template and method level and the introduction of a new integer data type. This resulted in minor alterations to the number of nodes and edges. In G_{old} the complete number of nodes was even reduced by two. However, the few added edges resulted in an increase in complexity and coupling. Even though for G_{new}, an increase in complexity and coupling can be seen, the measurements are much lower indicating a higher locality of the changes.

Summary Summarizing the measurement results, in all phases the complexity and coupling increased more significantly in the original generator (G_{old}) than in the new one (G_{new}). This is primarily caused by a different modularization strategy to foster reuse. The original developers did not consider semantic boundaries of the metamodels as an indicator to create separate generator fragments and modules. They used one global state handling module which, therefore, was used in every generator and alterations to this module affected all fragments. The state handler also provided naming services and indexing, which are also provided by other modules resulting in some overlap in functionality.

Due to the more ad hoc nature of modularization, the developers had to refactor their generator in every revision to make the additions of new features possible. The original generator has modules dedicated to name resolving, like the new generator, but this separation of concern was not used consequently for all fragments.

Typing and type mapping is done in different modules over and over again making the original generator more complex than necessary. Internal

variables in the generated code were typed using the source level type which was then mapped to the corresponding target level type. This also contributed to the increased complexity and coupling.

Finally, the inheritance features and polymorphism offered by the source language resulted in many special routines to generate code which can map the polymorphism onto the target language. These routines also considered other complex mappings, like expressions, which contained special operators, e.g., for all (∀) and exists (∃). This reciprocal influence could have been factored out using intermediate models realizing the polymorphism mapping separate from the expression mapping. In addition, the complex operators of the expression grammar could have been realized with functions on target level. However, G_{old} automatically inlined these functions. All these design decisions increased the complexity of the code used to realize the mapping between source and target level.

11.3 Semi-Structured Expert Interviews

The two case studies evaluated whether GECO supports the development and the evolution of code generators. The second case study was also used to evaluate the whether GECO provides an advantage for the construction and the evolution of code generators over a classic development process. However, both case studies rely on the assumption that complex generators and generator assemblies are widely used, and that these generators are subject to evolution and reuse. Therefore, we conducted interviews with developers and engineers in industry and research.

In this section, we provide a summary on the interviewees, their, focus, domain, and experience. Subsequently, we discuss their insight into the three topics construction, evolution, and reuse. The interviews were performed in German. The corresponding translated interview guide can be found in Appendix A.1 on page 307. Considerations for its design are given in Section 9.3 on page 170 . The paraphrased and translated interviews can be found in Appendix A.2 on page 310.

11.3.1 Interviewees

All but one interview were performed as group interviews. Three interviews were held with engineers from industry and two interviews were performed with researchers. The interviewees from industry have a background in DSL, metamodel, and generator development as consulting party, software vendor, contract work, and in-house development. They provided their services for embedded and information systems.

The interviewees with a research background focus on metamodeling as a central element, but also develop graphical and textual DSLs. They use transformations and generators not only to derive models and code, but also to relate knowledge between models of different abstraction.

The experience of the industry interviewees ranged from first year developers to over 20 years experience modeling. Furthermore, they gathered knowledge in different domains and industries. While the research interviewees had experience mostly in research projects ans doctoral students and post-doctoral researchers.

The scope of duties varies greatly between the different groups in industry and research. However, they all construct new DSLs and metamodels. They need to maintain a certain subset of their DSLs and metamodels. Some also develop and maintain tooling and frameworks for the development and evolution of DSLs and metamodels. Some researchers and industry developers even use multi-level modeling and deep modeling.

11.3.2 Construction and Development

In industry new metamodels, DSLs, and generators are developed for in-house use to develop frameworks and tooling, to integrate different modeling processes used internally, and based on customer needs. In research the development is related to evaluation of modeling approaches and techniques, to support knowledge-driven approaches, and to provide an abstraction to a specific technical domain.

In both areas, metamodel and generator development is used to provide abstractions which are then used to generate code. Furthermore, modeling is used to generate documentation, and to integrate model and code.

The development process for DSLs and metamodels is iterative and often agile in industry and research. Both kinds of interviewees use different approaches for the development, which includes the development based on a domain model or ontology, an iterative construction of the metamodel, a grammar-centric construction of DSL rules based on requirements and informal domain knowledge, and the construction of DSLs based on prototype artifacts.

In customer related project the domain knowledge and sometimes the metamodel is constructed in cooperation with the customer, which often require some informal, semi-formal, or formal intermediary representation of domain knowledge. Research and in-house products, domain models are rarely used instead domain concepts are directly expressed as metamodel classes.

In research and some industry development groups, they require DSLs and metamodels to be small and concise. The decomposition of DSLs and metamodels along technical and semantical boundaries, as GECO proposes, was considered a good advice and one group declared this common sense.

Regarding generator construction, the situation is much more diverse. The first group of interviewees requires that generators must small, based on a single domain, stateless, and fast. Stateless means in this case that the generator does not produce a persistent state beside the target model. By fulfilling these criteria, a generator can be invoked on every model change or save operation. Their generator criteria are similar to GECO counterparts. GECO proposes that fragments shall relate to one aspect or view of a software system, like data, behavior, and user interface, where the interviewees see these aspects as separate domains. However, they avoid explicit trace models.

Other groups in industry and research, however, also create more complex code generation chains. This approach is used in large modeling frameworks, like in the performance prediction and software modeling framework Palladio [BKR09], and where generator chains are used to implement a modeling process.

Especially, groups using complex model and transformations chains, were interested in the metamodel partitioning approach of GECO. They follow similar thoughts, however, an explicit specification of criteria, were

considered helpful (see Chapter 6 on page 81). Similarly the composition patterns (see Chapter 7 on page 109) introduced by GECO have appeared in some form in their projects. Therefore, the precise definition of the patterns in GECO was considered helpful.

11.3.3 Evolution and Maintenance

In the research context, generators tend to become complex. This complexity is influenced by metamodels and the intended context of generators. Furthermore, they might not have a specific structural design, as this was at first not expected to be required. Evolution is triggered by target platform changes, like the adaption to a new versions of a technology. Also metamodels and subsequently generators must be adapted to integrate new technologies.

In industry settings, evolution occurs through changes in the target platform and on source level mostly through the customer. These changes are mostly additions. In case of more fundamental changes in syntax and semantic, a new DSL and generator is constructed. The transfer is then realized by a generator which transforms artifacts of the old DSL into the DSL.

In research and industry new features are added iteratively. Projects for customers are usually feature-driven and the customers are integrated into the development process. Frameworks and in-house artifacts are evolved based on internal requirements which may originate from customer related projects or are initiated by the company itself.

For industry and research, the concepts of GECO for generator composition and fragment modularization are considered helpful to mitigate issues in generator maintenance and evolution. The modularization of metamodels were also considered helpful. Furthermore, the understandability of existing generators are limited which hinders evolution. Subsequently this results in the development of new independent generators addressing specific target artifacts even if the variations to the original generator is limited. The interviewees assumed that a better modularization approach, like GECO, might be helpful to mitigate these issues and support evolution.

11.3.4 Reuse and Variants

Reuse is not practiced by any interviewee in industry and research on the level of implemented artifacts with a few minor exceptions. Some generic metamodels are reused which is then specialized for different domains and levels of abstractions. Also the generic expression framework Xbase can be seen as reuse of DSL, metamodel, and generator parts. Outside of these two exceptions, reuse appears only in domain knowledge of the developers which are supported by tooling designed around knowledge on the technical domain. For example, the Xtext [Bet11] tool chain is such supportive tool and the typing approach for Xtext, presented in Section 10.3, can be categorized as domain knowledge on DSL design.

Generators from other developers are hardly reused, as they are considered unpredictable. This assumption is based on experience and the fact that old generators are often undocumented or the documentation is erroneous and incomplete. Furthermore, the source code is not available, which further hinders to analyze the old generator. And in cases where code is available, it is often more feasible to rebuild the generator.

Another argument against reuse was that every project and every language includes small differences, this is even true for data modeling languages. Therefore, developers tend to build DSLs and generators from ground up for every project.

One group of interviewees pointed out that they try to reuse some model queries. However, they have not much experience whether this is cost effective. However, this relates to an approaches from Sánchez Cuadrado et al. [SG08], Wimmer et al. [WKR+11], and Rose et al. [RGL+13] based on model types [SJ07] which realize the reuse of model transformation rules based on model types as generic models.

Based on this discussion we can conclude that reuse is not relevant on an artifact level for the interviewed group of developers. However, reuse as such is still an issue to them. Therefore, they try to reuse developer knowledge and support this with specialized tools.

Their answers somehow contradict past research which focused on reuse of rules, transformations, and metamodels. This might originate from the inadequacy of the approaches, the tool integration, and the lack of

knowledge of these approaches. However, it could also be that research focused on reuse of artifacts and not on knowledge which could be used to extend tooling.

11.4 Evaluation Summary

The evaluation of GECO comprises two case studies and a set of supplemental interviews. In the previous sections, we documented the evaluation measurements and described the comments and answers of interviewees. Based on these findings we provide in this section a concise summary, and relate them to the goals from Section 9.1.

The CoCoME case study demonstrates that the GECO fragment composition approach can be used to specify the construction of a large generator. Furthermore, generator modularization approach has been applied to the construction and evolution of the Behavior generator fragment. Especially, between the Revisions three and four, database support was added. Still the relationship between complexity and coupling decreases only minimal (0.15%) indicating, therefore, a minor degradation of the overall architecture.

In the second case study, based on MENGES, we redeveloped a generator and simulated its evolution. Key findings are that the GECO based generator is smaller, as it avoid code duplication, uses a different modularization resulting in lower measurements and lower increases in complexity and coupling. Furthermore, during the development of the original generator, refactorings appeared in multiple evolution steps. In contrast, the GECO based generator did not require refactoring, as the modularization in fragments and fragment modules was chosen in a more suitable way. For example, in the first revision, the GECO fragment composition approach indicated to build separate fragments for different metamodel partitions. In the original generator this was not the case from the beginning. Instead the different partitions were handled by separate functions, some of which were later transferred in separate modules.

We supplemented the case study based evaluation with interviews of experts from industry and research. The interviews addressed the three topics development, evolution and reuse of DSLs, metamodels, and generators.

Furthermore, the interviewees were asked whether they assume that the approaches of GECO would support these topics, especially for generators.

The results varied between different interview groups, but modularization of code generation was seen as important. Some realize this through small independent generators other try to modularize generators. For the second group GECO was considered a supportive approach for the construction and evolution of code generators. They also stated that evolution appears regularly and is triggered by customer needs, changing target platforms, and, in case of software vendors, also based on internal requirement changes.

In contrast, reuse of generators and in many cases also DSL and metamodel was called not relevant. This is surprising considering the large amount of effort in fostering research for model and generator reuse, which is partly documented in Chapter 12. Reuse is hindered by the fact that actual metamodels and generators vary in syntax and semantics between customers, which would require many different adaptations. Therefore, developers consider it more time and cost efficient to use DSL and generator development tooling to create new languages, than modifying existing artifacts. However, most interviewee groups consider reuse as potential helpful in product-line scenarios.

Based on these results of our evaluation and interviews, we can assess the three central goals of our evaluation plan (see Section 9.1). In the following we will discuss the results in respect to the three goals of the evaluation plan.

Goal G1 *Determine* the effect of *GECO* on the *utility and program quality* from the viewpoint of *software architects, developers and project management*. The CoCoME case study showed that GECO generator composition approach allows to model large generator assemblies with the patterns provided by GECO. In addition, the generator fragment modularization approach was used for various DSLs and especially for Behavior language. The results indicate that iterative development approaches are supported by the GECO modularization approach.

The MENGES case study, utilizing the GECO composition language, revealed that the complex generator of MENGES can be modeled with the

GECO composition patterns.

Finally, through the interviews we learned that the composition patterns appear in industry settings and, therefore, the specific discussion and definition of this pattern is considered helpful. Furthermore, the modularization along semantic and functional boundaries was considered supportive of generator construction.

Goal G2 *Evaluate* the effect of *GECO* on the *evolvability* from the viewpoint of *software architects and developers*. The interviews revealed that iterative development and evolution is relevant in industry and research. This affects metamodels and generators. The latter are even more affected, as target model and platform changes only affect generators, while requirement changes affect source metamodels and generators alike. This fosters our assumption that evolution is relevant and that generator evolution is affected from a wider range of changes than metamodels.

In the MENGES case study, we evaluated how the GECO composition and modularization approaches affected modularity, changeability, and extensibility based on size, complexity and coupling values. Therefore, we simulated the development and evolution of a generator for the MENGES DSL (see Section 11.2) which relied on a step by step process only providing information for one step to the developers. This was done to simulate an environment for the developers of the original project. The results show that the new generator provides better results for all measurements and in the increase of the measurements. This allows us to conclude that GECO helped to limit complexity and coupling throughout the evaluation. Therefore, GECO supports modularity, changeability, and extensibility and subsequently fosters evolvability.

Goal G3 *Evaluate* the effect of *GECO* on the *reusability* from the viewpoint of *software architects and developers*. The third goal addresses reuse. Reuse, like evolution, requires modularity which is supported by GECO. Unfortunately, we could not acquire case studies suitable to evaluate reuse. However, the interviews revealed that reuse is not an issue in the industry and the interviewed researchers. In case of product-lines, they are not directly in

focus of GECO. However, they may require a more fine grained reuse, like a transformation rule based reuse, which is addressed in other approaches, such as the approach of Kapova et al. [KGH+10].

Chapter 12

Related Work

The GECO approach addresses generator composition in the domain of aspect-oriented and view-based modeling. Chapter 6 introduces metamodel semantics and structural properties of metamodels which are used in the generator composition and fragment design. The generator composition approach subdivides a generator into smaller fragments representing the components of a generator (see Chapter 7). The fragment design, as introduced in Chapter 8, discusses the construction and decomposition of fragments based on the functional and semantic dimension. In all these areas related work exists which either address similar challenges or provides supplemental methods and approaches.

Following the approach introduction, we initially discuss aspect-oriented and view-based modeling approaches in Section 12.1. Subsequently, we introduce aspect-oriented code generation in Section 12.2 and relate these approaches to GECO. In Section 12.3, we compare different generator composition approaches with GECO and discuss their differences. As GECO aims to support modularization, evolution and reuse, we look into other approaches that address reuse in Section 12.4 and modularization in Section 12.5.

12.1 Aspect-Oriented and View-Based Modeling

As mentioned in Chapter 6, aspect-oriented and view-based modeling share similar characteristics. In both cases one model refers to another model. In aspect-oriented modeling, the reference expresses the extension or enrichment of the referenced element, and in view-based modeling, the reference can have different meanings depending on the view. For example, an architecture model may refer to a component type model for its elements,

and a dialog model may refer to tasks in a workflow to describe where the dialog should be used.

In the following two sections, we discuss the aspect-oriented modeling approach from Kienzle et al. [KAK09] and the multi-view modeling with orthogonal views from Atkinson, Stoll, et al. [ASB10].

12.1.1 Aspect-Oriented Modeling

In Klein et al. [KK07] and Kienzle et al. [KAK09] an aspect-oriented modeling approach is presented which aims to support reuse of aspect models. Their Reusable Aspect Models (RAM) approach is based on the UML, specifically on class, state, and sequence diagrams. The different diagram types are called views in their approach. In RAM, aspects comprise one or more of these views. For each view there is a pointcut and advice part in the aspect model. The advice uses the normal UML notation, while the pointcut extends UML with a wildcard mechanism. For example, to match any call of an UML sequence diagram, the pointcut allows to specify a call with an asterisk ($*$) as name. Such wildcards can be used for any UML notational element.

The RAM approach is designed for UML without profiles. It allows to reuse aspect models modeled in the supported UML diagram types. While the approach is intended to be extended for other diagram types and potentially for custom DSLs, it does not provide a procedure or approach to realize it.

RAM focuses on the modeling of weavable aspects. Therefore, they require that advice and base model share a common subset of the classes and interfaces of a metamodel. Furthermore, their pointcut metamodel extends this subset by wildcards for attributes, instances, and references. In GECO, we follow this abstraction for the weavable models used in the weaving pattern (see composition pattern P4 in Section 7.1.2 on page 118). However, GECO allows to model aspects in metamodels which do not share a common subset. In contrast to RAM, GECO does not provide a specific metamodel and notation for aspects. Instead it supports the reuse of existing metamodels.

12.1.2 Orthographic Modeling

Orthographic Software Modeling (OSM) [ASB10] is based on the three principles, dynamic view generation, dimension-based view navigation, and view-oriented methods. It is founded on the approach Komponentenbasierte Anwendungsentwicklung (engl. component-based application development) (KobrA) [ABB+02] which defines five dimensions for models of software systems [ASB10]:

Composition covers the decomposition of a system into components and subcomponents

Abstraction describes the different levels of abstraction, such as the platform independent model, the platform specific model, and the implementation

Encapsulation provides either the external, black box view or the internal white box view of a system

Projection allows to look at different aspects of a model entity, like structural, operational, behavioral, and variational.

Variant covers different variants of the system.

OSM provides a separate view for any configuration along these five dimensions. Each of these views is generated on-the-fly out of a Single Underlying Model (SUM) which contains all information on the software system. For example, there exist two views for a component type, one as Java class and one as UML component model, then these views can be realized as projection of the SUM. Lets assume the name of the Java class is changed. This will eventually change the component name in the UML representation as well. Therefore, the SUM must be able to store all kinds of information. This may result in a complex metamodel, as it must cover all concepts used in all views. Alternatively, a very simple SUM can be created, able to handle graph information. In that case the semantics of the SUM would be defined by the various projection transformations which create the views. This results in complex projection transformations.

The Vitruvius approach [Bur14] addresses this issue by replacing the SUM with a set of metamodels, which are able to model certain aspects and elements of a software system separately. These metamodels and specialized transformations between them form a virtual SUM. Around the virtual SUM, a set of transformations is used to provide specific views.

In contrast to the SUM-based approaches, GECO uses vertical and exogenous transformations which transform a source model or even a set of source models into target models. GECO is unidirectional and it assumes that the source models are on a higher level of abstraction and these must be transformed into models which are less abstract.

However, the metamodel partitioning method of GECO and the fragment modularization approach are also useful in SUM-based approaches. The metamodel partitioning allows to identify parts of metamodels, which can be transformed by separate transformations, and the modularization supports the construction of transformations. The single megamodel patterns of GECO also appear in SUM-based approaches, especially, when unidirectional transformations are used. Also revised versions of the patterns could be used to describe bidirectional transformations.

12.2 Aspect-Oriented Code Generation

Most Aspect-Oriented Modeling (AOM) approaches include code generation for base and aspect models. According to a survey [MJ13] these approaches are limited in their code generation abilities. Either they do not address the evolution and reuse of code and model generation, or they support evolution and reuse only insufficiently. Mehmood et al. [MJ13] even conclude that the approaches do not fulfill the common requirements for reuse and evolution of transformations and code generators. However, they are AOM approaches and address at least aspect-oriented modeling. Furthermore, they may benefit from a generator construction approach such as GECO.

We categorize these approaches based on the used modeling technology in UML-based and DSL-based approaches. For both categories, we compare the ability of the approaches to support changes in metamodels or metamodel semantics, and evaluate how they can be applied to larger software

projects.

12.2.1 UML-based Approaches

The UML approaches use UML models to define base and aspect model. They aim to support the reuse models. Apart from the model weaving approach [KAK09], all approaches map AOM to AOP, namely Java and AspectJ, and use the AspectJ weaver to realize aspect weaving.

In the following we introduce the code generation approaches for Reusable Aspect Models (RAM) [KK07], Theme/UML [CB05] and Formal Design Analysis Framework (FDAF) [Dai05].

RAM The Reusable Aspect Models (RAM) approach addresses the definition of reusable aspect models which are woven into existing models [KK07]. The base model may use UML class, sequence, and activity diagrams. The corresponding aspect models comprise an advice model and pointcut model. The advice uses an extended version of one of the UML types which allows to specify wildcards. The pointcut is also based on UML diagram types and allows to query UML model structures. In RAM the advice can add, replace, and remove model elements.

Kienzle et al. [KAK09] use the Kermeta weaver [MKB+08] to weave RAM aspects into models. This solution corresponds to the weaving pattern of GECO (pattern P4, see Section 7.1.2 on page 118) where a base model is woven with an aspect model. Alternatively, Kramer and Kienzle [KK11] transform base and aspect model into Java and AspectJ code which can then be woven with AOP technology. This approach could be modeled with the megamodel pattern P2 (see Section 7.1.2 on page 118) and P4. First, the UML base model and RAM-based aspect model are both transformed by separate transformations to their respective target languages Java and AspectJ. Second, the AspectJ weaver weaves aspects into Java classes.

However, the existing generator is realized as one software component. The mapping of references from source to target model level is achieved without a trace model, as the complete model is present for both transformations.

The solution by Kramer and Kienzle [KK11] does not address the inner structure of the generator and the transformations. They mainly discuss the mapping difficulties of UML multi-inheritance [UML15, p. 97] onto Java and AspectJ, which do not support multi-inheritance for class attributes and method implementations. The issue of handling a large number of different aspects, is circumvented by two properties of the approach. First, RAM only supports UML class, sequence and activity diagrams, and second, it does not support UML profiles, which would affect the semantic of the UML entities. Therefore, combination of multiple aspects can be handled by the same generator and weaver.

Apart from their own generators for Java and AspectJ, they argue that they can use any model-to-code generator for UML which makes their approach target model independent. This is true when utilizing the Kermeta weaver [MKB+08]. However, UML code generators are often limited to generate class and method stubs (cf. [BCD10]). In this case, stubs must be implemented by human developers. In case of model evolution, the reintegration of code produced by humans into stubs can be costly and error prone.

In contrast, GECO provides the means to construct generators for RAM and similar approaches, which use likewise principles, but different metamodels. It also allows to integrate DSLs and UML profiles with the RAM approach, as GECO addresses the mapping of references between source and target model level where both levels use different metamodels.

Theme/UML Theme is an aspect-oriented analysis and design approach. It covers requirement engineering with Theme/Doc and software design with Theme/UML [CB05]. The central element in the Theme approach are themes, which are a collection of structures and behaviors that represent one feature. Each theme is either a base theme or a cross-cutting theme. A complete system is then composed of multiple themes. Each theme is present in Theme/Doc and Theme/UML. Like RAM, Theme/UML uses an extended UML notation to express advices and pointcuts in cross-cutting themes.

Clarke et al. [CB05] do not explicitly introduce a generator and weaving mechanism for Theme/UML. However, Hecht et al. [HPP+05] discuss gener-

ator construction for Theme/UML and provide an example generator implemented in XSLT which transforms the XML representation of Theme/UML themes into Java and AspectJ code. The primary design is comparable to the RAM generator approach from Kramer and Kienzle [KK11], where aspect and base model are transformed separately and the weaving is realized with AspectJ.

While Theme addresses the complete design process of model-driven software development, it has multiple limitations. First, it does not support UML profiles in modeling and code generation. Therefore, the modeling cannot be adapted to specific domains. Second, the presented generator [HPP+05] produces Java and AspectJ code. However, the generator lacks support for multi-inheritance.

In contrast to GECO, the Theme/UML generator construction only discusses a subset of common issues for code generators used in an aspect-oriented modeling context. Apart from the mapping issue of multi-inheritance in Java, they do not discuss how to transfer pointcut information from source to target model level. In their example, they realize it by directly mapping UML packages to Java packages, and UML class names to Java class names. This assumption allows to reconstruct the pointcut reference destinations in the aspect. However, this makes the aspect generator directly dependent on the base generator. In case the naming scheme changes in the base generator, the aspect must be adapted accordingly, or both must share their name provider component. In the latter case, the aspect generator must have access to the base model and its metamodel. In GECO, the two generators only interact via a trace model which does not change in its metamodel when the base generator changes its naming scheme. Therefore, the interface for the aspect generator does not change.

FDAF Bennett et al. [BCD10] present a model-driven code generation approach based on graph transformations, which generate code stubs in AspectJ. Their FDAF approach is based on an extended UML metamodel which supports the modeling of aspects. The approach provides tooling for normal and extended UML models. Their code generation uses an intermediate model which is realized with an XML metamodel, i.e., an XML schema. Therefore, they have a three step transformation process where

first a UML model is transformed into their intermediate XML model and then in AspectJ code for the aspect and Java for the base model. Finally, both AspectJ and Jave code are woven with the AspectJ weaver. Therefore, this approach uses pattern P2 of the GECO approach (see Section 7.1) for the two transformation steps, and pattern P4 for the final integration. As they only support class diagrams, FDAF has only one source metamodel which allows to express structure, in contrast to approaches, like RAM, which support multiple diagram types. Furthermore, the authors do not discuss how pointcuts are mapped from the UML level to the XML intermediate model level, and subsequently to the code level. As they do not pass the information between the base model and aspect model transformations, it must be computed based on a shared algorithm. Unfortunately, they do not provide any source code and other artifacts relating the approach. Therefore, we could not investigate the implementation of the tooling and evaluate how the transformations realize the information transfer.

12.2.2 Aspect-oriented DSLs

In the recent survey on the emerging area of AODSL [FDN+15] 22 different AODSLs and their generators were analyzed. One key problem determined in the analyzed languages is the integration of the AODSL generator in the base language generator. Most existing solutions extend the base language generator in an ad-hoc manner which has a negative impact on reuse of the AODSL generator fragment and maintainability of both generators [FDN+15]. Two AODSL frameworks comprise an extensible base language generator to allow additions for AODSL generators [NCM03; EH07]. However, both frameworks have multiple shortcomings. First, they do not address the integration of multiple AODSL and cascading multi-step configurations, like in GECO. Second, they limit the integration of AODSLs to their own specific base language. Third, they do not address the construction of extensions (generator fragments in GECO), and the separation of aspect generation and aspect weaving

12.3 Generator Construction Approaches

The previously mentioned approaches comprise a modeling and a code generation part in their approaches. In this section, we focus on approaches dedicated to create code generators.

Higher Order Transformation Higher-Order Transformations (HOTs) are transformations used to transform the models which specify transformations [TJF+09]. Usually HOTs are exogenous transformations, as they transform a abstract model into an artifact containing a transformation.

Kapova et al. [KGH+10] use HOTs in their approach to compose transformations for different platforms of a product line based on domain-specific templates. Their main goal is to reduce cost through increased reusability and customizability of transformation rules. The approach combines a transformation composition method with a process to construct the necessary transformation fragments. Their transformation fragments should not be confused with the generator fragments of GECO. They are only a small set of rules and not complete transformations.

Their process assumes that a metamodel exists which is used to specify parts of the software of the product line. Based on this metamodel and domain knowledge a general structure for the transformation is derived, which they call *frame*. Furthermore, templates are created based on domain knowledge. These templates are associated with features of the target platform of the product line. The specific template for a feature is called *custom rule*. All features together are represented by a feature model expressing the potential configurations of the transformation.

Based on a configuration of the feature model, i.e., a selection of features, custom rules are selected. They are then combined with the frame to a *refined transformation*. Such refined transformations can then be used for a specific target platform. This mechanism allows to change the transformation without changing the source model of a software artifact. For example, if one target platform comes with a numerical processor and another lacks that support, the respective transformations would realize the implementation for numeric operations.

The approach of Kapova et al. [KGH+10] also addresses reuse and modu-

larity in transformations. They focus on single templates and transformation rules. In contrast, GECO modularizes generators in generator fragments and subsequently in modules. Each module in GECO may still contain multiple templates and also other code. Therefore, GECO addresses a broader scope of use cases than the approach of Kapova et al. [KGH+10]. However, in the context of product lines their approach could be used to support the development of single generator fragments.

Their approach is based on the transformation language QVT-Relations for the HOT and for the resulting refined transformation. Although not specifically described, their approach could also be realized with another rule based language as long as it provides a metamodel accessible for the language used for the HOTs. The resulting transformation could then be integrated into a GECO generator using the provided QVT and ATL-generator fragment class from the GECO framework.

Finally, their domain-specific templating approach addresses the construction of templates and their reuse for variations of the target platform, but do not address evolution. Furthermore, they compose their custom rules based on the overall feature model describing all potential target platforms regardless of the kind of semantics these features comprise. In contrast, GECO encourages to divide fragments along a semantic and functional dimension (see Chapter 8). Therefore, the approach of Kapova et al. [KGH+10] might combine rules in a template which have more relationships to another template of custom rules than to the rules in the same template. This may result in higher coupling between the templates, which can cause code degradation, hindering code evolution.

Genesys Approach The Genesys approach [Jör13] focuses on correctness and reuse of code generators. It is built upon the jABC framework [NLS+12] which provides the infrastructure and basic library for Genesys.

The jABC framework is designed with Extreme Model-Driven Development (XMDD) in mind. XMDD assumes that all parts of a software are modeled and subsequently code is generated from these models. The resulting code is not touched or modified and no code modification is propagated back to the source models.

The framework introduces two major concepts, called Service Logic

Graphs (SLGs) and Service Independent Building Blocks (SIBs). A SLG is a directed graph connecting different SIBs. In this graph the SIBs are nodes representing tasks, and the edges indicate the control flow. As a SLG does not express data flow and models are not entities in a SLG, it is not a megamodel (see Section 3.6). In contrast, the GECO composition language does not define the sequence of execution, but the data dependencies of generator fragments. Each SIB is a reusable and configurable component which performs one operation. After executing the operation, it provides information which outgoing edge should be followed, resembling a decision node in a workflow.

In Genesys, a SIB can be a generator for a class header or a function body. The usual modularization discussed in Genesys is based on templates which produce a part of the resulting code, e.g., a class header. Therefore, the modularization is based on target metamodel and language structure and not on the source metamodel structure, as in GECO (cf. Chapter 6). Furthermore, Genesys does not discuss how the target model and code might be derived from source level artifacts. This is resolved with SIBs provided by the jABC framework.

Genesys also uses the facilities of the jABC framework to verify its generator and uses the jABC Tracer to compile SLGs into native Java programs. In GECO, the generator for its composition language performs a similar task. However, it has also to compute the execution sequence of the fragments, as the language only defines data dependencies. Furthermore, fragments in GECO do not provide information on which path can be taken next, like SIBs, which results in a simpler interface of GECO fragments, but does not allow for alternative generations in a GECO generator.

Apart from tooling differences, the GECO megamodel patterns may help to develop generators with jABC and Genesys by allowing to model and understand data dependencies between SIBs, a feature the present realization lacks.

12.4 Reuse of Transformations

Writing transformations is a complex task. However, specific transformation rules occur in many different DSLs and metamodels. For example, the collection of attributes of structured types in DSLs with a type system supporting sub-typing, is a reoccurring task. Especially in product lines, the target platform and the respective target metamodel may vary between different products, but most of the metamodel and the platform are identical. For such scenarios it is helpful to be able to generalize transformation rules and reuse them for specific products and transformations.

As an example, we discuss two approaches fostering reuse of transformation rules. The first approach is based on defining so-called concepts which are intermediaries for transformation rules and metamodel elements. Such a concept identifies classes, references, and attributes as a representation of a concept which is then provided to transformation rules. The second approach uses model typing which allows to derive specific metamodels from general metamodels.

Genericity for Model Management Operations Wimmer et al. [WKR+11] and Rose et al. [RGL+13] address reuse in metamodeling and model transformation. The central part of their approach is to apply ideas from generic programming to modeling. Generic programming allows, for example, to write an operation once and reuse it with different data types. In `Java` the `List` class is implemented using generics which allows to use the same list class for different element types. In generic programming the `List` class represents a concept which can be bound to a type resulting in a specialized class, e.g., `List<Item>` with `Item` being the specific element type.

UML supports genericity through templates [RGL+13]. However, it is limited to classes and ignores model transformations which are not part of the UML. The approach of Wimmer et al. [WKR+11] and Rose et al. [RGL+13] introduces the notion of metamodel concepts. These concepts comprise meta-classes, meta-associations (references) and attributes. These meta-classes can then be bound to classes of a metamodel. It is important to note that this binding does not introduce new attributes and references to the metamodel. The binding only relates attributes, meta-associations,

and meta-classes of a concept to the respective elements of a metamodel.

The key idea of this approach is to define operations on the elements of a concept. These operations are in fact transformation rules which are used for endogenous and exogenous transformations (see Section 3.2). This way, the operations are independent of a specific metamodel and can be reused for different metamodels in case the same transformation rules are required. To apply these operations to a concrete metamodel, first a concept must be bound to the metamodel, and second, HOTs are used to transform the concept-based rules into concrete rules for the bound metamodel.

For example, the DSLs IRL [JHS13] and DTL (see Section 9.4.1) define inheritance and their respective transformations require an operation to collect all properties of a data type. In the present generators of these DSLs, the same operation was implemented for each generator separately, which could have been avoided utilizing this concept-based approach. Therefore, concepts can be a helpful way to reuse functionality of code generators.

In conjunction with a semantics based partitioning of metamodels, the concepts of their approach can further reduce the need for reimplementation of similar functionality between different languages. It can also help to limit architecture degradation, as the operations are not affected by metamodel evolution. Only the binding must be updated. These bindings are relatively small in comparison to all the operations used in generators. As GECO already encourages separation of generator transformation rules in modules based on semantics, the approach of Rose et al. [RGL+13] can directly be used with GECO. In addition, their approach might benefit from the idea of semantic separation to group meta-classes in concepts.

Factorization and Composition of Transformation Reusability is an important software quality which may help to reduce cost and development effort. Sánchez Cuadrado et al. [SG08] introduce an approach to generalize and reuse transformations. They specifically target product lines, but their approach can also be supportive in other cases where metamodels or types of metamodels are reused.

The approach is founded on model typing [SJ07] which addresses the issue of metamodel compatibility for transformations, like the concepts of Wimmer et al. [WKR+11] and Rose et al. [RGL+13]. The main difference

of model typing to concepts is that concepts do not change and affect a metamodel, they only define which part of a metamodel refers to a specific part of a concept. Model typing, however, allows to derive metamodels from each other [SJ07].

The approach of Sánchez Cuadrado et al. [SG08] provides two tasks to generalize transformation rules, which they call factorization, and compose transformations. Factorization is the process of finding and identifying rules of transformations which provide a reoccurring functionality. This is accomplished by identifying common model types for the involved source and target metamodels. Based on the identified model types, transformation rules which only use the commonalities comprised by the model types, can be factored out and moved to a base transformation. Such base transformations can then be imported by specific transformations.

To be able to reuse base transformations, they must be combined with specific transformations. The composition is performed in so-called phases. Each phase represents a set of transformation rules which addresses one specific task, like collecting all attributes of a type. The phases are executed sequentially by applying their respective rules. The rules can be base and specific transformation rules, and they can be normal rules and refined rules. Normal rules define how new target nodes are created based on the source model. However, refined rules cannot create nodes. They can only modify existing nodes.

For example, a normal rule collects all attributes for a type and creates a sequence of attribute nodes in the target model, the refined rule may add to the attribute initialization values. To facilitate this behavior, refined rules do not rely on source and target model queries. Instead they use model traces to identify source and target model nodes. This trace model is created during the previous phases by the corresponding normal rules.

The approach of Sánchez Cuadrado et al. [SG08] originates from the scope of product lines. Like other approaches from this domain, they intend to reuse transformations which are very similar in syntax and semantics. They focus on reusing small sets of rules which are common for a product line. In context of GECO this approach could be used to create single generator fragments. They intend to structure transformations by transformation tasks. Even though they do not defines how this specific tasks can be found,

the idea resonates with our semantic dimension of fragment modularization. In a product line scenario, their approach can be beneficial to further reduce the amount of rules necessary for the code generator. For example, the MENGES case study could be extended to support other languages of the IEC 61131-3 [IEC03] which share common type structure and function signatures. For such scenarios, the rules concerning the generation of function signatures and type structures could be reused by this phasing mechanism.

12.5 Modularization of Transformations

Transformations used in MDE are complex software artifacts, which require an approach to make this complexity manageable. In software development, one central approach to handle complexity is modularization. Therefore, researchers have developed modularization approaches. First, we introduce transformation chaining which intends to reduce complexity by executing different transformations sequentially, each providing only one step towards the resulting model. Second, we summarize an approach which introduces localized transformations, which have conceptual similarities with the semantic modularization in GECO (see Section 8.1).

Chaining Transformations Software projects utilizing Model-Driven Engineering (MDE) comprise a multitude of models and metamodels, which must be transformed into artifacts for a specific target platform.

The approach of Vanhooff, Van Baelen, Hovsepyan, et al. [VVH+06] provides a metamodel and language to specify the coupling of transformations [VAB06] combined with a basic schema for a chain development process. They separate requirements and assign them to four distinct concerns, which are not cross-cutting. In detail, these are functional, non-functional, technical, and implementation concerns. To ensure reuse of the constructed transformation chains and support variability, they use model types to define the interface of the metamodels [SJ07].

They describe requirements for transformations and a transformation chain language which must be met to realize transformation chains and the reusability of transformations. First, transformations must be mutu-

ally exclusive, which means different transformations should not provide the same functionality. Duplication of functionality hinders evolution and maintainability, as changes have to be applied to different transformations, duplicating the effort. Second, each transformation can be assigned to one concern. Third, the transformations must be loosely coupled, which means they shall not depend on the internal implementation and technology of other transformations. The exchange of information is solely based on models. Finally, a transformation chain specification must be independent of the used technologies. This allows to combine transformations and generators with different technological background.

The process schema they propose has at least four stages, which they call elaborations. Initially, the engineers must gather initial models and concerns for the model-driven software project. They must also collect all functional and non-functional requirements which correlate with the first two concerns of this approach. Second, the engineers identify transformations and model types on a conceptual and informal level. Third, they refine the conceptual model by formalizing the input and output model types of all transformations. Furthermore, the abstract platform for the software project is specified. Fourth, the engineers further refine the transformations and include now the technical and implementational concerns. Vanhooff, Van Baelen, Hovsepyan, et al. [VVH+06] state that in a real project there can be more elaborations, as the different refinements require multiple iterations.

This approach describes a similar context for transformation evolution and reuse, as the GECO approach does. Specifically, they address concerns and loosely coupling of transformations. However, the GECO approach defines generator fragments as building blocks, which imply more constraints on the elements of the transformation chain. Additionally, it distinguishes between trace models and target models of a transformation and fragment, while Vanhooff, Van Baelen, Hovsepyan, et al. [VVH+06] do not address this issue explicitly. While they mention concerns and aspects, they do not elaborate on the specific implications of aspects and views. In contrast, GECO defines specifically how to handle the information exchange. Furthermore, they do not discuss metamodel semantics and metamodel partitioning as a way to identify potential borders for separation in transformations

and transformation rules. However, they incorporate the idea of model types into their approach, which might also be beneficial for fragments and modules developed with GECO.

Localized Transformations Model-Driven Engineering (MDE) relies on models, metamodels and model transformations. As metamodels can be large and complicated, like the PCM [BKR09] and MENGES metamodel [GHH+12], the associated transformations tend to be complex in order to handle models defined with these metamodels. Therefore, it is important to modularize transformations. This can be done with transformation chaining, as expressed above. However, this has the disadvantage that the transformation must be able to understand the complete metamodel even if it is only providing a minor transformational step. While such decomposition is necessary in the MDA [OMG14], it is not sufficient for large transformations.

Etien et al. [EML+15] suggest modularizing transformations based on locality. They define locality by model and metamodel parts which represent the source and target of a transformation. They formulate three basic criteria which must be fulfilled for localized transformations. First, source and target metamodel of a localized transformation must overlap. This is necessary as they only replace a portion of source model nodes by target model nodes and use the remaining model nodes to connect to results of other transformations. Second, each transformation only applies to a restricted part of the source metamodel. Third, the transformation is limited to one concept and intention.

These criteria imply that for large transformations with one source and one target metamodel, there exist several metamodels in between, which incorporate metamodel features of the source and target metamodels in various degrees. This could also be facilitated with model types [SJ07]. However, they do not use this notion in their approach. Between these different metamodels, they define local transformation.

Local transformation must be combined to create a complete transformation. In the approach of Etien et al. [EML+15] this is performed by transformation chaining. They assume two local transformations T_a and T_b with different source and target metamodels. In case the target metamodel

of T_a conforms to the source metamodel of T_b then the execution sequence of both is T_a, T_b. In case the metamodels do not match, they belong to different transformation chains or do not directly follow each other.

Compared to GECO, they address the same challenge of transformation modularization. The three criteria for localized transformations correspond partially with the GECO idea to partition metamodels along semantic properties and locality. However, GECO does not require or suggest to use metamodels which are cross-overs of source and target metamodels. Instead GECO assumes that the result of one generator fragment has a disjunct metamodel from the source metamodel. However, the GECO approach does not forbid such intermediary metamodels. Therefore, both approaches can be used together to realize code generators.

Furthermore, GECO addresses aspects and views, and the necessity to exchange trace information between transformations of views, aspects and base models. Etien et al. [EML+15] do not address views explicitly. Instead they use shared metamodel classes and model nodes to transport similar information. Therefore, they do not separate the concern of target model production and trace models, which might lead to a stronger entanglement of both, thus hindering reuse.

Finally, GECO provides two different levels of decomposition while the localized transformation approach uses one concept. In GECO, the first level addresses view, aspect and base models, and second level, targets fragment modularization based on semantic properties of metamodels and transformation functionality.

For example, a metamodel comprises typing structures and expressions to specify component types, and component instances to model software architecture. In GECO, this would be handled by two generator fragments representing one level of decomposition. The component types would be generated by one fragment and the architecture by another. Inside the component type generator, there would be at least three modules providing transformation rules for typing, expressions, and trace information which is required by the architecture model. The localized transformation approach, however, would realize this with three transformations and two intermediate metamodels. In the context of metamodel evolution, these intermediate metamodels require additional effort, as they have to be updated every time

the source metamodel changes, making this specific part of their approach less efficient than GECO.

Part IV

Conclusion and Future Work

Chapter 13

Conclusion

The development and evolution of generators is essential for the application of Model-Driven Engineering (MDE) in real world projects. Surveys, like Mehmood et al. [MJ13], suggested that an approach for complex generators is required to advance MDE. However, generator evolution and construction were only addressed on the basis of single transformations and transformation rules or limited to assemblies with one aspect language (see Chapter 12). Our GECO approach, presented in this thesis, addresses this issue of generator construction and evolution. We further evaluated the approach with two case studies and a supplemental set of expert interviews.

In the following sections, we summarize the contributions of this thesis in Section 13.1. In Section 13.2, we provide an overview of the evaluation results. Finally, in Section 13.3, we discuss prototypes, which support different aspects of the GECO approach.

13.1 Contributions

The core of the GECO approach are the megamodel patterns for generator composition in Chapter 7 and the considerations on the design of generator fragments in Chapter 8. However, the megamodel patterns imply requirements regarding the metamodel structure and the relationship between metamodel partitions. Therefore, the core of GECO is preceded by a discussion of metamodel semantics and partitioning in Chapter 6.

In the following we summarize these three parts of the GECO approach as major contribution, supplemented by tooling we developed to assess generator qualities.

Metamodel Partitioning In Chapter 6, we discussed syntactical properties of metamodels and use cases for metamodels to support the understanding of their properties based on their context in Section 6.2. Thereof we deduced seven contextual metamodel patterns (see Section 6.3) which reoccur throughout metamodels. Based on these patterns, we discussed semantics of references in Section 6.4 and especially of containment references, which play a central role in understanding metamodel semantics. Exemplary, we introduced two reoccurring partition types based on semantics, namely one for typing (Section 6.5) and one for expressions (Section 6.6). Finally, we provided a method and consideration supporting the partitioning of metamodels based on syntax and semantics (Section 6.7).

Megamodel Patterns The megamodel patterns are used for the composition of generators based on specific generator components, called fragments. In Chapter 7, we derived five distinct megamodel patterns (Section 7.1) from 57 potential combinations of minimal fragment and metamodel setups. These patterns allow to construct any complex generator and support the modularization of generators.

Subsequently, we discussed the internal relationships and dependencies of these patterns (Section 7.2), technical aspects of approach (Section 7.4 and Section 7.3), and how legacy generators can be integrated as fragments in GECO (Section 7.2.3).

Fragment Construction The second level of modularization addresses the construction of single generator fragments which we discussed in Chapter 8. We explained in this chapter the functional and semantic dimension of fragment modularization, and discussed how they affect the evolution in Section 8.1. Based on these considerations we described two kinds of modules based on semantic dimension in Section 8.2 and Section 8.3. A reoccurring module, originating from the functional dimension, is the trace model handler, which we introduced in Section 8.4.

Evaluation Tooling The evaluation of GECO required tooling which was able to assess quality attributes such as size, complexity, and coupling,

based on the entropy of the analyzed generators. We choose a graph and hypergraph-based approach (Chapter 4) to calculate the necessary measurements, which has the advantage over simple counting metrics that it reflects the dependencies in software artifacts and the complexity which is induced by the mesh of dependencies. Unfortunately, there was no tooling available realizing this measurement approach. Therefore, we implemented the metrics of this approach and supplemented it with transformations for Java code (Section 9.2 and Appendix B.3).

13.2 Experimental Findings

Our evaluation was based on two case studies and five interviews. The case studies were constructed for the information system and embedded control system domain. The information system case study utilized CoCoME as a model of a software system and PCM as a metamodel. In the case study, we implemented generators for the CoCoME models (Section 11.1). The embedded control system case study replayed the MENGES DSL and generator development project (Section 11.2). Furthermore, we conducted five interviews with engineers and developers from industry and research. They provided an external view on the proposed GECO approach. Their answers were used to evaluate our premise that generator construction and evolution is a relevant topic.

In Section 9.1, we defined three goals we intended to achieve with GECO. In the following we summarize the results in respect to these three goals of the evaluation plan.

Goal G1 *Determine* the effect of *GECO* on the *utility and program quality* from the viewpoint of *software architects, developers and project management*.

The CoCoME case study shows that the GECO generator composition approach allows to model large generators with the megamodel patterns defined by GECO (see Chapter 7). The complete CoCoME generator comprises five fragments including the ProtoCom generator as a legacy fragment.

The fragment modularization approach of GECO was evaluated during the iterative development of the Behavior language for the CoCoME case

study. The results, documented in Section 11.1 and Section 11.4 indicate that iterative development approaches are supported by the GECO modularization approach, as the complexity to coupling ratio did not change significantly.

The MENGES case study (Section 11.2), utilized the GECO composition language (Section 10.2) to model the megamodel of the new generator. This case study revealed that the complex generator of MENGES can be modeled with the GECO composition patterns. The overall size, complexity, and coupling measurements of the new generator were lower than those from the original generator, and even the increase in measurements were lower. Therefore, GECO leads to better modularization and subsequently to better program quality.

Finally, through the interviews we learned that the composition patterns appear in industry settings and, therefore, the specific discussion and definition of these patterns is considered helpful. Furthermore, the modularization along semantic and functional boundaries was considered supportive of generator construction.

Goal G2 *Evaluate* the effect of *GECO* on the *evolvability* from the viewpoint of *software architects and developers*.

The interviews revealed that iterative development and evolution is relevant in industry and research. This affects metamodels and generators. The latter are even more affected, as target model and platform changes only affect generators, while requirement changes affect source metamodels and generators alike. This supports our assumption that evolution is relevant and that generator evolution is affected from a wider range of changes than metamodels.

In the MENGES case study, we evaluated how the GECO composition and modularization approaches affected modularity, changeability, and extensibility based on size, complexity and coupling values. Therefore, we simulated the development and evolution of a generator for the MENGES DSL (see Section 11.2) which relied on a step by step process only providing information for one step to the developers. We choose this process to be able to simulate a development context for developers close to the context of the original project. The results show that the new generator

provides better results for size, complexity, and coupling. Furthermore, the increase of values for these three measurements are also lower for the new generator compared to the old generator. This allows us to conclude that GECO helped to limit complexity and coupling throughout the evaluation. Therefore, GECO supports modularity, changeability, and extensibility and subsequently supports evolvability.

Goal G3 *Evaluate* the effect of *GECO* on the *reusability* from the viewpoint of *software architects and developers*.

The third goal addressed reuse. Reuse, like evolution, requires modularity which is supported by GECO. Unfortunately, we could not acquire suitable case studies to evaluate reuse. Therefore, we could not evaluate whether GECO supports generality and portability, as required by our experiment setup in Section 9.1.

However, the interviews revealed that reuse is not an issue in industry and research from a developer's perspective. This is slightly different for product-lines where interviewees suggested that reuse of parts might be beneficial. Still product-lines of tooling where not used by any party.

Concluding, we showed that GECO supports construction and evolution of generators. We were able to see especially in our second case study substantial improvements in modularization and evolvability compared to classic development.

13.3 Prototypical Application of GECO

During this thesis, we developed multiple DSLs and generators utilizing concepts of the GECO approach. These prototypes and their design are documented in Chapter 10, and online sources are listed in Appendix B.

We created a target platform independent AODSL (Section 10.4) for instrumentation, which allows to model sensors without the need to explicitly consider the different technologies used to realize aspect integration. In Section 10.2, we documented the development and design of a DSL to support

fragment composition. We supplemented this language with a generator, which maps DSL artifacts into Xtend code.

To support the construction of fragments, we implemented a framework which provides interfaces to ensure a specific fragment design (Section 10.1). These interfaces were supplemented by a generic trace model handler and classes to integrate QVT and ATL transformations into GECO. The tooling, developed for this thesis utilizes this framework. Furthermore, the fragment composition DSL (Section 10.2) requires fragments to be implemented with this framework in order to access their properties and create an assembly class for the generator.

Finally, we designed a prototypical type system combined with exemplary implementation artifacts (Section 10.3) usable with the Xtext DSL tooling and framework. This implementation is intended to be customized and adapted to fulfill the requirements of specific DSL. We utilized this design within all DSLs implemented during this thesis.

Chapter 14

Future Work

GECO introduces an approach for generator composition and fragment design. Both are founded on the understanding of semantics and structure of metamodels. While the focus of this thesis is on generator construction and evolution, it relates to other areas of model-driven engineering and reverse engineering. Furthermore, for its adaptation in the industry, it must be integrated in tooling and processes.

Therefore, we identified four possible research topics for future work. In detail they are: the evaluation and refinement of the metamodel partitioning (Section 14.1), the application of GECO for the modernization of legacy generators (Section 14.2), and finally, the application of the megamodel patterns in different technical and technological contexts.

14.1 Evaluation of Metamodel Partitioning

In this thesis we discussed the structure of metamodels, their semantics, and how both can be used to modularize and partition metamodels. While we used these considerations and methods in our case studies to identify metamodel partitions, we did not provide a separate evaluation of the metamodel partitioning and the underlying considerations and ideas. Therefore, we propose to perform such evaluation in future. Specifically, we intend to integrate our considerations and methods with other ideas on metamodel construction and evolution, for example the approach of Strittmatter et al. [SRH+15]. Subsequently, we will evaluate this approach by analyzing a wide range of metamodels from different domains, such as PCM [BKR09] and SMM [SMM12]. The evaluation must investigate the applicability of our methods and assess whether the resulting partitions support metamodel

construction and evolution better than the original metamodel. In case of a working approach, we could also assess the quality of existing metamodels based on the deviations from the desired design patterns introduced by our approach. Therefore, the approach can be used for constructive and analytic purposes.

14.2 Generator Modernization with GECO

GECO addresses the construction and evolution of generators. However, it does not explicitly address modernization of existing generators. In the interviews we conducted during the evaluation, interviewees revealed that they have to integrate existing generators, which is often seen as being too costly. Therefore, they reverse engineer these generators and implement new generators with present day tooling. Based on our own experiences during the evaluation of GECO, we consider that GECO's megamodel patterns and fragment modularization can be used to support the recovery of the generator structure and drive its modularization which would support modernization. Therefore, we propose to combine GECO with model-driven reverse engineering approaches to aid generator modernization.

14.3 Technical Aspects of Megamodel Patterns

The megamodel patterns discussed in this thesis are abstract and technology agnostic. However, for their application in software projects they have to be realized with specific technology. We already discussed some aspects of integrating legacy generators in Section 7.2.3, but only on a generic level. In real world scenarios, the specific technologies and frameworks used for a generator play an important role in the application of GECO. During our evaluation, we already used GECO with different technological setups. Therefore, we suggest to investigate technological derivatives of the megamodel patterns and the adaptation of our composition language prototype.

Architecture styles can be realized with various kinds of technology, which introduce additional constraints on the application of the styles

[Gie08]. Similarly, we expect that technologies, like modeling frameworks and build systems have an impact on the realization of the megamodel patterns and their adaption in the field. Therefore, we suggest to investigate the integration of GECO with build systems, such as Maven and Eclipse.

This investigation also includes the fragment composition DSL which presently produces one class containing a generator assembly for a set of fragments. While this was sufficient for the MENGES case study, it was not applicable to the CoCoME case study. Therefore, it would be interesting to use the DSL in conjunction with build systems.

This DSL also requires all fragments to work with in-memory models, which is, however, not applicable to every context. For example, in the CoCoME case study, we could not use the composition DSL. Therefore, the DSL must be refined to support file based passing of models and dependency based build systems.

Based on the interviews during our evaluation, we learned that passing of trace information is not always done explicitly with trace models. In some cases this is realized with components similar to name resolvers, as described in Section 8.1.1.

All these proposed investigations and developments would support the applicability of the approach in the industry and research, as it would further mitigate technical hurdles and support development with proper tooling. We hope to achieve this in the coming years.

Part V

Appendix

Appendix A

Interview Development and Evaluation

This chapter documents the artifacts used for interviews including the interview guide in Appendix A.1 and the list of paraphrased results of the interviews in Appendix A.2. The interviews were conducted in German, paraphrased, and then the paraphrased results were translated for this thesis.

A.1 Interview Guide

GECO is an approach for the development of code generators and fragments of these, which is intended to support the creation, evolution and reuse of generators. In the course of the evaluation of GECO, multiple expert interviews with expert groups were performed to determine the relevance of such an approach for the industry, and to discuss the applicability of the approach.

The interview is divided into five parts described in the following sections. The interview started with the introduction of the interviewer, followed by an introductory presentation of the approach. Subsequently, the participants described their expertise and experience. After the introduction the three main topics were addressed. The interview concluded with a closing summary.

A.1.1 Introduction

▻ Introduce yourself and the purpose of the interview

- For later analysis, the interview should be recorded, however, the recording will be kept confidential. Only the aggregated and paraphrased result will be published. The recording will be available to the interviewees, but not to third parties. If a recording is not possible, the interview can be executed based on notes only. However, this would limit the later evaluation.
- All interviewees may ask questions during the presentation and the interview to clarify questions and the approach.
- The interview comprises three topics, induced by the claims of the GECO approach.
- The presentation will require 15 minutes, for the interview we expect 90 minutes.

A.1.2 Warm-up Questions

The warm-up questions where asked prior to the interviews, as they unnecessarily delay the core interview in larger groups.

- Academic background of experts
- Their experience in the industry
- Scope of duties

A.1.3 Topic: Construction and Development

- Please describe the creation process of DSLs and metamodels.
- Which challenges arise during the development of generators?
- Which influential factors affect generator development?
- Are you developing new generators for existing DSLs?
- If so, what are the usual triggers for the development of new generators for existing DSLs?

- ▻ Do you consider the GECO approach useful for your development process?

A.1.4 Topic: Evolution and Maintenance

- ▻ How do you determine new requirements of DSLs and generators? What are the sources of requirement changes?
- ▻ How do DSLs evolve and which influences does this have on generators?
- ▻ Which challenges arise during the evolution of generators?
- ▻ How do you organize and exercise the evolution of generators in your organization?
  - ▻ Are generators developed with agile methods or rather with conventional methods?
  - ▻ How is the process for generator maintenance laid out?
- ▻ Do you consider the GECO approach useful for your evolution process?

A.1.5 Topic: Reuse and Variants

- ▻ Does reuse of DSLs and generators play a role in our organization?
- ▻ How do you realize the reuse of generators of DSLs?
- ▻ What are the influential factors to be considered when reusing generators? For example, source and target languages, relationships to other languages.
- ▻ Do you consider the GECO approach useful for reusability?

A.1.6 Closure

- ▻ Sum up the results of the interview.
- ▻ Ask for further question and remarks.
- ▻ Thank the interviewees for the participation.

A.2 Analysis of the Interviews

The analysis of the interviews must be performed in an anonymous way. Therefore, only the paraphrased results which corresponded to a set of questions of each interview are provided to the reader. All interviews have been performed as group interviews with varying numbers of interviewees. Please note, all interviews were conducted in German. The paraphrased results were translated and anonymized for the thesis.

A.2.1 Interview 1

Setting

Interviewees 6

Domain Industry, consulting and DSL development

Experience Multiple years in DSL development for different domains in industry

Scope of duties DSL construction, maintenance, tool development for the development of DSLs

Topic: Construction and Development

▻ DSLs must be small and concise

▻ Generators should be small

▻ Decomposition along technical and semantical boundaries is considered common sense

▻ Xbase uses internally highly interconnected transformations to map DSLs to Java

▻ Possible simple generators suited for one single domain

▻ Tracing, validation, type-checking are cross-cutting concern

- ▻ Integration of black-box generator
- ▻ AOP in development results in additional problems when the original generator changes
- ▻ Requires a lot of testing and test cases which is not economically feasible
- ▻ Minimal modeling, no OMG-like attribute and user data structures
- ▻ Generators are stateless
- ▻ Use intermediate models in generators
- ▻ Generators must be fast

Topic: Evolution and Maintenance

- ▻ Evolution and maintenance is relevant
- ▻ Re-engineering of generators
- ▻ Features are added iteratively
- ▻ Syntax and metamodel are loosely coupled, changes can be made to both gradually
- ▻ Sometimes evolution is performed by transforming old DSLs into new DSLs
- ▻ Evolution of DSLs must be attended carefully, due to legacy issues

Topic: Reuse and Variants

- ▻ Reuse is not very practicable in industry cases
- ▻ General domain solutions do not occur in reality. There are always variations
- ▻ Reuse does not play a central role. DSLs are mostly constructed from the ground up every time. One exception are `Xbase`-based languages which reuse `Xbase` grammar and expression mapping

- ▻ Do not want to import generators from other developers, at it is unclear what they do and if they work properly
- ▻ Product-line engineering and variants are relevant
- ▻ Instead of reuse of DSLs, use tools which allow to create new generators and DSLs quickly

A.2.2 Interview 2

Setting

Interviewees 5

Domain Research, application of modeling to software development and evolution

Experience PhD candidates and postdoc researchers

Scope of duties model-driven development and performance prediction, model evolution

Topic: Construction and Development

- ▻ Identify concepts of a DSL, then classes, then syntax
- ▻ Depends on new or existing metamodel
- ▻ For behavior or declarative languages, they start with a textual syntax
- ▻ Depends on the use case of the metamodel:
  - ▻ For humans: we start with a textual DSL
  - ▻ For technical purposes, e.g., intermediate models: we start with the metamodel
- ▻ Do not use ontologies to model
- ▻ Start mostly with one root class for a new metamodel and DSL

- ▻ Starting with: who is the user of the DSL and what does the user do with it
- ▻ Implementation of a concrete and specific example of the resulting code for a narrow use case
- ▻ Approach could be helpful in the design and development of own generator projects

Topic: Evolution and Maintenance

- ▻ Problem with large generators when adding new functionality. Especially, as they did not have a specific structural design
- ▻ Extensibility is an issue
- ▻ Complexity also induced by the context of a transformation
- ▻ Understandability of existing generators in Palladio is limited. They are used in their current form and are not modified
- ▻ Evolution mainly triggered by technology change and seldom by changes to the (source) metamodel

Topic: Reuse and Variants

- ▻ Generator reuse and adaptation is not used. Instead a new generator is created for the specific purpose

General Discussion

- ▻ GECO could be used when different target models must be generated and the generators share some functionality.

A.2.3 Interview 3

Setting

Interviewees 1

Domain Research, model-driven co-evolution

Experience PhD-candidate researcher (6 years)

Scope of duties model-driven development, reverse engineering, model and code evolution

Topic: Construction and Development

▻ Create metamodels, but no textual DSLs of architecture and behavior

▻ Do not have a specific metamodel development process

▻ In one project only, which developed an aggregated metamodel the following process was used initially:

1. Collection of information from one other metamodel
2. Incorporating of additional features based on a second metamodel, etc.

▻ Later, the metamodel has been modularized and new concepts from other metamodels are incorporated by adding modules

▻ Generator can go both ways

▻ Generators are multi-threaded

▻ Uses two-pass compilation

▻ Use of GECO might be beneficial for the generator

▻ Uses trace models to connect generated nodes with the original nodes

Topic: Evolution and Maintenance

▻ No long time evolution, but the construction of the metamodels used an iterative process during the research project (see above)

▻ Adaptation to new EJB versions

Topic: Reuse and Variants

▻ Reuse is not a direct issue for them

A.2.4 Interview 4

Setting

Interviewees 8

Domain Industry, Model-Driven Engineering (MDE), software vendor and consultant, development of model-based/model-driven platforms

Experience Different backgrounds, ranging from over 20 years of experience in modeling and architecture, to first year developers

Scope of duties Maintenance and evolution of a software platform and tooling, application of MDE

Topic: Construction and Development

▻ Two types of projects:

a) Development of an inhouse platform and framework used in customer projects

b) Development of specific DSLs, metamodels, and tooling in cooperation with customers

▻ Agile development

▻ Use of domain models

Topic: Evolution and Maintenance/Reuse and Variants

a) Framework

- ▻ New features based on customer need and internal requirements
- ▻ No product lines, but different versions

b) Custom development

- ▻ No reuse
- ▻ Integrated development with customer
- ▻ Development is iterative and feature-driven

General Discussion

- ▻ Practice metamodel partitioning, but not necessarily at aspect and view borders
- ▻ Transformation composition could be realized with GECO, similar approach already in use, but not consequently applied

A.2.5 Interview 5

Setting

Interviewees 3

Domain Industry, application of Model-Driven Engineering (MDE) in embedded systems and project planning

Experience Different backgrounds, software developers, Postdoc-level designers and developers

Scope of duties Application of MDE techniques for various purposes in planning, design, and code generation, multi-level modeling/deep modeling

Topic: Construction and Development

- ▻ Fragment modularization of GECO appears in own generators
- ▻ Use of graphical DSLs
- ▻ Construction of meta-metamodels in cooperation with customers
- ▻ Meta-modeling induced by customer
- ▻ Multiple outputs from single input models
- ▻ Multiple stages in modeling related to the (development) process in projects
- ▻ Motivation for modeling: Integration of partial processes
- ▻ Iterative development approach of models and generators (use agile methods)
- ▻ A key task is document generation and not code generation

Topic: Evolution and Maintenance

- ▻ Evolution trigger based on target model changes
- ▻ Evolution appears in long running projects and is triggered by customers

Topic: Reuse and Variants

- ▻ Variability is important
- ▻ Use generic metamodels and use inheritance to specialize metamodels
- ▻ Variability in domain differentiation and in abstraction level
- ▻ Try to reuse model queries in form of library functions
- ▻ Reuse of generators or generator variants did not occur in context of the presently used metamodels. It may occur in the future

Appendix B

Tooling

B.1 Fragment Framework

GECO provides a framework supporting the development of generator fragments (Section 10.1). The sources of the framework can be found on Github and Zenodo, as well as part of our Eclipse update site for GECO. In the Github repository, the framework is located in `de.cau.cs.se.geco.architecture.framework`.

▻ Github `https://github.com/rju/geco-composition-language.git`

▻ Zenodo `http://dx.doi.org/10.5281/zenodo.47129`

▻ Update site `https://build.se.informatik.uni-kiel.de/eus/geco/snapshot/`

The Eclipse update site can directly be added in Eclipse and used for installation of the GECO framework and tooling.

B.2 Fragment Composition Tooling

The GECO fragment composition DSL and generator tooling is also available on Github, Zenodo and our Eclipse update site. The first two locations provide the source code, and the latter installable and executable artifacts.

▻ Github `https://github.com/rju/geco-composition-language.git`

▻ Zenodo `http://dx.doi.org/10.5281/zenodo.47129`

▻ Update site `https://build.se.informatik.uni-kiel.de/eus/geco/snapshot/`

The source code repository of the GECO fragment composition language comprises eight sub-projects including framework, editor, generator, visualization, and release management. The single sub-projects are:

- ▷ `de.cau.cs.se.geco.architecture.framework` comprises the GECO generator composition framework classes and interfaces
- ▷ `de.cau.cs.se.geco.architecture.graph` provides the automatic visualization component based on KIELER
- ▷ `de.cau.cs.se.geco.architecture.releng` contains Maven configuration and shell scripts to build the tooling
- ▷ `de.cau.cs.se.geco.architecture.sdk` contains the declaration of the associated Eclipse feature
- ▷ `de.cau.cs.se.geco.architecture.tests` includes tests for components of the DSL, editor, and generator
- ▷ `de.cau.cs.se.geco.architecture.ui` is the project of the DSL editor part of the GECO composition language
- ▷ `de.cau.cs.se.geco.architecture.updatesite` defines the necessary parts for the update site of the Eclipse plug-ins
- ▷ `de.cau.cs.se.geco.architecture` comprises the DSL grammar, metamodel, semantic checks, and scope handlers in an Eclipse plug-in

B.3 Entropy Analysis of Models and Code

In Chapter 11, we used metrics founded on information theory to measure the entropy of software artifacts, like models and, in case of generators, Java code. The metrics utilizes hypergraph and graph abstractions of these software artifacts and allow to calculate the size, complexity, coupling and cohesion of the abstractions [All02; AGG07]. These entropy metrics are supplemented by counting metrics which relate to the hypergraph abstraction, a cyclomatic complexity metric [McC76], and a lines of code metric, which are presented in a common result view.

The cyclomatic complexity metric is used to calculate the inner complexity of Java methods which are subsequently aggregated in buckets usable for histograms and violin plots. While the cyclomatic complexity metric and lines of code metric can only be applied to Java code, the hypergraph and graph-based metrics are supplemented by a set of mapping transformations for Java code, EMF metamodels, GECO megamodels, and PCM deployment graphs. The analysis tool can be added to an Eclipse installation using the following update site `http://build.se.informatik.uni-kiel.de/eus/se/snaptshot/`. After installation, two new option appear in the context menu for projects (right clicking on a project entry in the Package and Project Explorer in Eclipse).

B.3.1 Java Code Analysis

The Java code analysis is triggered by choosing the Java Analysis option from the context menu for projects in the Package or Project Explorer. To be able to execute the analysis, the analysis algorithm requires information on the scope of the software, i.e., the set of classes and interfaces implementing the software. Furthermore, the classes representing the data model must be identified before the analysis. Both sets of classes and interfaces must be specified in two files in the projects's root directory. The classes to be considered part of the software must be listed in a file called `observed-system.cfg`, and the data model classes must be listed in `data-type-pattern.cfg`. In both files, wildcards can be used to specify patterns, like `de.cau.cs.se.software.evaluation.*` referring to all classes and interfaces with this prefix in their fully qualified name.

B.3.2 Model Analysis

The model analysis is activated by selecting a supported model in the Package or Project Explorer. Supported models are EMF metamodels, GECO megamodels, and PCM deployment models.

B.3.3 Views

The tooling comprises three views to present measurement results and visualize the abstract hypergraphs and modular hypergraphs used in the analysis.

Analysis Result View This view presents the numerical results of the executed code and model analysis. It provides three buttons in the upper right corner of the view which allow to save the hypergraph or modular hypergraph of the last analysis, save the numerical values with labels in a CSV-file, and clear the results, which is helpful to reset the view before a second analysis run.

Code and Model Analysis This view is triggered when hypergraph models are opened. Depending on the content, the hypergraph or modular hypergraph view is used to present the model. The modular hypergraph view allows to view the complete hypergraph or an aggregation based on the modules of the hypergraph. These visualizations provide the developer and analyst with visual feedback regarding the analysis. For example, this allows to identify that classes have been included which should have been excluded and vise versa. Furthermore, do these views provide an intuitive accessible visualization to assess the complexity of the software.

B.3.4 Repositories

The tooling can be found in the following repositories:

▷ Github `https://github.com/rju/architecture-evaluation-tool.git`

▷ Zenodo `http://dx.doi.org/10.5281/zenodo.47129`

▷ Eclipse update site `http://build.se.informatik.uni-kiel.de/eus/se/snapshot/`

Appendix C

Generator Evolution

C.1 Identified Revisions of MENGES DSL

Revision 1 – initial-2011-11-22

▻ Initial generator, iterates over the complete model

Revision 2 – case-2012-01-02

▻ Support for types struct, state variables, enumeration

▻ Split interlocking elements metamodel in parts

▻ Support from duration value

Revision 3 – case-2012-01-11

▻ Added helper for type expressions

▻ Added name provider

▻ Refactoring and more generator functionality

▻ Type expansion to map object-oriented structures to records

Revision 4 – case-2012-01-25

▻ Metamodel restructuring, replaced commands by message protocols

▻ Minor grammar fixes

▻ New reference model in grammars

▻ Refactoring and more generator functionality

Revision 5 – case-2012-01-26

▻ Grammar modifications after evaluation

▻ Changes to the reference implementation of DSL and generators

▻ Implemented decision trees

▻ Implemented architecture support

Revision 5 – case-2012-02-02

▻ Metamodel and logic language changes

▻ Behavior expressions metamodel changed

▻ Refactorings

Revision 5 – case-2012-02-06

▻ Support of temporary variables in output

▻ Bug fixes

▻ Improvements to the polymorphic dispatch

▻ Reference resolving implemented

Revision 5 – case-2012-02-07

▻ Fixed target model variable naming

▻ Support for boolean expressions

▻ Variable and constant realization fixed

▻ Function call handling improved

Revision 5 – case-2012-02-08

- ▻ Improved reference handling
- ▻ Improved state machines
- ▻ Metamodel changes

Revision 5 – case-2012-02-09

- ▻ Metamodel minor modification

Revision 5 – case-2012-02-10

- ▻ Fixes in generating deployment
- ▻ Fixes to configuration value generation
- ▻ Fixes to generated XML structure

Revision 5 – case-2012-02-13

- ▻ Constants for array
- ▻ Name resolver improved
- ▻ Changed generation of collections of record types
- ▻ Fixes to deployment and configuration

Revision 5 – case-2012-02-15

- ▻ Support for runtime environment improved
- ▻ Changed reference resolving for enumerations
- ▻ Improved boolean expressions
- ▻ Changed naming scheme

Revision 6 – case-2012-02-16-communication-properties

▻ Target model improvements for property names

▻ Generator support of connectors

Revision 7 – case-2012-02-17-numbering-of-user-defined-types

▻ Generator fixes in type mapping

▻ Bug fixing references

Revision 8 – case-2012-02-20-update-conditionals

▻ Decision tree generation improved

▻ Task support

▻ Bug fixing references

Revision 9 – case-2012-02-21-added-initialized-flag-to-generated-code

▻ First implementation of value initialization

▻ Many small bug fixes

Revision 10 – case-2012-02-21-property-propagation

▻ Connector generation improved

Revision 10 – case-2012-02-27-time-integration

▻ Bug fixes

▻ Timer integration

Note Large gap in original subversion commits, caused by vacations, documentation and release preparations, and conference visit.

Revision 11 – case-2012-03-26-property-name-generation

▻ Bug fixes

▻ Type mapping improvements

▻ Update of loops

▻ Generator component structure update

Revision 11 – case-2012-03-26-state-machine-generation-improved

▻ Improved state machine generator

▻ Name resolver improved

Revision 12 – case-2012-04-02-name-provider

▻ Name provider fixes

▻ Bug fixes

▻ Properties of array instances fixed

▻ Duration value and expression changes

Note Vacations and miscellaneous developer absence. Subversion and documentation updates.

Revision 13 – case-2012-04-16-timer-start

▻ Timer start supported

Revision 13 – case-2012-04-17-timer-call

▻ Timer state access

Revision 13 – case-2012-04-19-instance-name

▻ No generator changes; only extended test case and runtime library update

Revision 13 – case-2012-04-19-timer-restart

▻ No generator changes; only extended test case and runtime library update

Revision 14 – case-2012-04-23-ticket-459

▻ State machine generator improved

▻ Grammar changes

C.2 MENGES and CoCoME Replication Package

The MENGES and CoCoME replication package [Jun16a] contains all information and sources used in the evaluation of the GECO research project. For software based on git repositories, we provide the revision ID (Git hash value) of the used software, as other revisions might produce other results inside the archive. In general the tooling for the analysis are placed in a software package [Jun16b].

The package contains a README.txt containing information regarding the setup and content of the package, and two directories for the MENGES and CoCoME case studies. Unfortunately, the MENGES directory does not include the source code of the MENGES artifacts, due to legal and copyright consideration of the involved companies. However, you may contact b+m informatik AG (Thomas Stahl thomas.stahl@bmiag.de) to arrange access to the source code for research purposes.

Index

AbstractATLTransformation
  generate, 187
ITraceModelProvider
  add, 153
  lookup, 153

Aggregation, 32, 91, 92
Array, 96
Array Types, 22
Ascription, 19
Aspects, 92
Attribute, 31

Base Types, 18, 95

Collection, 96
Containment, 32, 83, 92
Contextual Patterns
  Aggregation, 91
  Data, 90
  Derived Models, 90
  Execution, 91
  Navigation, 89
  State, 91
  Traceability, 89

Data, 90
Depth Subtyping, 26
Derivation, 93
Derived Models, 90
Description, 93

Enumeration Types, 20
Execution, 91
Extension, 92

Generator, 110
  Fragment, 110, 129
Generator Fragment, 110, 129

Judgment, 15

Let Binding, 21

Map Types, 96
Megamodel, 57
Metamodel, 31
  Contextual Patterns
    Aggregation, 91
    Data, 90
    Derived Models, 90
    Execution, 91
    Navigation, 89
    State, 91
    Traceability, 89
  Use Case
    Aggregation, 86
    Editors, 84
    Evaluation, 87
    Generators, 85

    Interpreter, 87
    Navigation, 85
    Runtime Models, 88
    Serializability, 86
    Traceability, 86
    Views, 84
Model, 31
  Traceability, 86, 89
Model Traceability, 86, 89
Modeling
  Aggregation, 32, 91, 92
  Aspects, 92
  Attribute, 31
  Containment, 32, 83, 92
  Data, 90
  Derivation, 93
  Derived Models, 90
  Description, 93
  Execution, 91
  Extension, 92
  Megamodel, 57
  Metamodel, 31
  Model, 31
  Navigation, 89
  Opposite, 83
  Reference, 31
  References, 92
  Root Node, 38
  Rooted Model, 38
  Source Metamodel, 39
  Source Model, 39
  Specification, 93
  State, 91
  Target Metamodel, 39
  Target Model, 39
  Traceability, 89
  Unidirectional Association, 32

Opposite, 83

Record Types, 19
Recursive Types, 24
Reference, 31
Reference Types, 21
References, 92
Relation
  equivalence, 14
  has-type, 14
  subtype-of, 14
Root Node, 38
Rooted Model, 38

Source Metamodel, 39
Source Model, 39
Specification, 93
State, 91
Static Typing Environment, 14
Structured Types, 95
Subtyping
  Depth Subtyping, 26
  Width Subtyping, 25

Target Metamodel, 39
Target Model, 39
Transformation
  Incremental, 40
  Non-Incremental, 40
Type Ascription, 19
Type Checking
  Dynamically Checked, 13
  Statically Checked, 13

Type Hierarchy, 94
Typing
    Array, 96
    Array Types, 22
    Axioms, 15
    Base Types, 18, 95
    Collection, 96
    Constants, 99
    Dynamic Typing, 13
    Enumeration Types, 20
    Function Calls, 99
    Literals, 97
    Map Types, 96
    Method Calls, 99
    Operators, 99
    Properties, 99
    Record Types, 19
    Recursive Types, 24
    Reference Types, 21
    Rules, 15
    Special Features, 100
    Static Typing, 13
    Static Typing Environment, 14
    Structured Types, 95
    Type Hierarchy, 94
    Unit Type, 18
    Variant Types, 20
Typing Rule
    S-Arrow, 26
    S-Record-Depth, 25
    S-Record-Width, 25
    S-Reflexive, 25
    S-Sequence, 18
    S-Top, 27
    S-Transitive, 25, 27
    T-Assignment, 22
    T-CmpExpNumber, 204
    T-CmpExpString, 205
    T-CmpExp, 204
    T-ConstraintExpression, 204
    T-ContainerNode, 229, 231
    T-Dereference, 22
    T-Environment-X, 24
    T-InstanceOf, 206
    T-ModelProperty, 231
    T-Negation, 205
    T-Node, 227
    T-ParenthesisConstraint, 205
    T-Pointcut-Operation, 225
    T-Pointcut, 225
    T-PropertyContraintCompare-String, 231
    T-PropertyContraint, 230
    T-Property, 203
    T-Recursive-Type, 24
    T-Reference-Type, 22
    T-Reference, 22
    T-Select-Record, 19
    T-Type-Array, 23
    T-Type-Record, 19
    T-Type-Variant, 20
    T-Typeof, 232
    T-Val-Array-Bound, 23
    T-Val-Array, 23
    T-Value-As-Variant, 20
    T-Value-Case-Variant, 20
    T-Value-Is-Variant, 20
    T-Value-Record, 19
    T-Value-Variant, 20
    T-WildcardNode, 229

Unidirectional Association, 32
Unit Type, 18
Use Case
    Aggregation, 86
    Editors, 84
    Evaluation, 87
    Generators, 85
    Interpreter, 87
    Navigation, 85
    Runtime Models, 88
    Serializability, 86
    Traceability, 86
    Views, 84

Variant Types, 20

Width Subtyping, 25

Bibliography

[ABB+02] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based product line engineering with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0-201-73791-4.

[ABB+64] Paul W. Abrahams et al. *The programming language LISP; its operation and applications*. Ed. by Edmund C. Berkeley and Daniel G. Bobrow. First Edition. Cambridge, Massachusetts: M.I.T. Press, 1964.

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. "Henshin: advanced concepts and tools for in-place EMF model transformations". In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*. MODELS'10. Oslo, Norway: Springer, 2010, pp. 121–135. DOI: 10.1007/978-3-642-16145-2_9.

[ACC+02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. "Recovering traceability links between code and documentation". In: *IEEE Transactions on Software Engineering* 28.10 (Oct. 2002), pp. 970–983. DOI: 10.1109/TSE.2002.1041053.

[AG98] Ken Arnold and James Gosling. *The Java programming language (2nd ed.)* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1998. ISBN: 0-201-31006-6.

[AGG07] Edward B. Allen, Sampath Gottipati, and Rajiv Govindarajan. "Measuring size, complexity, and coupling of hypergraph abstractions of software: an information-theory approach". English. In: *Software Quality Journal* 15.2 (2007), pp. 179–212. DOI: 10.1007/s11219-006-9010-3.

[AKK+99] M. Ajrnal Chaumun, H. Kabaili, R.K. Keller, and F. Lustman. "A change impact model for changeability assessment in object-oriented software systems". In: *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*. 1999, pp. 130–138. DOI: 10.1109/CSMR.1999.756690.

[All02] Edward B. Allen. "Measuring graph abstractions of software: an information-theory approach". In: *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*. 2002, pp. 182–193. DOI: 10.1109/METRIC.2002.1011337.

[And04] Charles André. "Computing synccharts reactions". In: *Electronic Notes in Theoretical Computer Science* Volume 88 (Oct. 2004), pp. 3–19. DOI: 10.1016/j.entcs.2003.05.007.

[ANR+06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. "Model traceability". In: *IBM Systems Journal* 45.3 (July 2006), pp. 515–526. DOI: 10.1147/sj.453.0515.

[ANS78] ANSI – Committee X3. *American national standard programming language fortran: approved april 3, 1978*. Standard. American National Standards Institute, Inc., 1978. URL: https://books.google.de/books?id=y9uxoQEACAAJ.

[App15] Apple Incorporated. *Objective-C Runtime Programming Guide*. 2015. URL: https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/ObjCRuntimeGuide.pdf.

[APR+05] Neta Aizenbud-Reshef, R. F. Paige, J. Rubin, Y. Shaham-Gafni, and D. S. Kolovos. "Operational semantics for traceability". In: *ECMDA Traceability Workshop*. Nuremberg, Germany, Nov. 2005, pp. 7–14.

[ASB10] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 'Orthographic software modeling: a practical approach to view-based development'. In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski. Vol. 69. Communications in Computer and Information Science. Springer, 2010, pp. 206–219. DOI: 10.1007/978-3-642-14819-4_15.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing, 1986. ISBN: 0-201-10088-6.

[Ban98] Jerry Banks, ed. *Handbook of simulation: principles, methodology, advances, applications, and practice*. Wiley-Interscience, 1998.

[BCD10] Jeannette Bennett, Kendra Cooper, and Lirong Dai. "Aspect-oriented model-driven skeleton code generation: a graph-based transformation approach". In: *Science of Computer Programming* 75.8 (2010). Designing high quality system/software architectures, pp. 689–725. DOI: 10.1016/j.scico.2009.05.005.

[BCM+10] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, eds. *The description logic handbook: theory, implementation, and applications*. 2nd edition. New York, NY, USA: Cambridge University Press, 2010. ISBN: 0-521-78176-0.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 'The goal question metric approach'. In: *Encyclopedia of Software Engineering*. Wiley, 1994.

[Bet11] Lorenzo Bettini. "A DSL for writing type systems for Xtext languages". In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Kongens Lyngby, Denmark: ACM, 2011, pp. 31–40. DOI: 10.1145/2093157.2093163.

[BET12] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. "Formal foundation of consistent EMF model transformations by algebraic graph transformation". In: *Software & Systems Modeling* 11.2 (2012), pp. 227–250. DOI: 10.1007/s10270-011-0199-7.

[Bet13] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing, Limited, 2013. ISBN: 978-1-78-2160-311.

[BGS+14] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. "Neo4EMF, a scalable persistence layer for EMF models". In: *European Conference on Modeling Foundations and Applications*. Ed. by Jordi Cabot and Julia Rubin. Vol. 8569. University of York. York, UK, United Kingdom: Springer, Apr. 2014, pp. 230–241. DOI: 10.1007/978-3-319-09195-2_15.

[Bie10] Matthias Biehl. *Literature study on model transformations*. Technical Report ISRN/KTH/MMK/R-10/07-SE. Stockholm, Sweden: Royal Institute of Technology, July 2010.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (Jan. 2009), pp. 3–22. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.03.066.

[BSV+13] Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo. 'Approaches and tools for implementing type systems in Xtext'. In: *Software Language Engineering*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, 2013, pp. 392–412. DOI: 10.1007/978-3-642-36089-3_22.

[Bur13] Edward Burns. *JSR 344: Java Server Faces, Version 2.2*. Java Specification Request. Oracle, 2013.

[Bur14] Erik Burger. "Flexible views for view-based model-driven development". PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology, July 2014. ISBN: 978-3-7315-0276-0.

[Car04] Luca Cardelli. 'Type systems'. In: *The Computer Science and Engineering Handbook*. Ed. by Allen B. Tucker. CRC Press, 2004. Chap. 97.

[Cas99] Jesus Castagnetto. *Professional PHP programming*. Professional Series. Wrox Press, 1999. ISBN: 9781861002969.

[CB05] Siobhàn Clarke and Elisa Baniassad. *Aspect-oriented analysis and design*. Addison-Wesley Professional, 2005. ISBN: 0321246748.

[CDJ+97] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. “Type systems”. In: *The Computer Science and Engineering Handbook*. CRC Press, 1997, pp. 2208–2236.

[CG11] Hyun Cho and Jeff Gray. “Design patterns for metamodels”. In: *Proceedings of the Compilation of the Co-located Workshops on DSM’11, TMC’11, AGERE!’11, AOOPES’11, NEAT’11, & VMIL’11*. SPLASH ’11 Workshops. Portland, Oregon, USA: ACM, 2011, pp. 25–32. DOI: 10.1145/2095050.2095056.

[CTB12] Benoit Combemale, Xavier Thirioux, and Benoit Baudry. ’Formally defining and iterating infinite models‘. In: *Model Driven Engineering Languages and Systems*. Ed. by Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson. Vol. 7590. Lecture Notes in Computer Science. Springer, 2012, pp. 119–133. DOI: 10.1007/978-3-642-33666-9_9.

[CW85] Luca Cardelli and Peter Wegner. “On understanding types, data abstraction, and polymorphism”. In: *ACM Computer Survey* 17.4 (Dec. 1985), pp. 471–523. DOI: 10.1145/6041.6042.

[Dai05] Lirong Dai. “Formal design analysis framework: an aspect-oriented architectural framework”. PhD thesis. The University of Texas at Dallas, 2005.

[Dem08] Linda Demichiel. *JSR 317: Java Persistence API, Version 2.0*. Java Specification Request. Sun Microsystems, Java Persistence 2.0 Expert Group, 2008.

[DK06] Linda Demichiel and Mike Keith. *JSR 220: Enterprise JavaBeans 3.0*. Java Specification Request. Oracle, Java Persistence 2.0 Expert Group, 2006.

[DS12] Charles Donnelly and Richard Stallman. *Bison*. Version 2.6.4. GNU Software Foundation. Nov. 2012. URL: http://www.gnu.org/software/bison/bison.html.

[EEK+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. “Xbase: implementing domain-specific languages for Java”. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE ’12. Dresden, Germany: ACM, 2012, pp. 112–121. ISBN: 978-1-4503-1129-8. DOI: 10.1145/2371401.2371419.

[EEP+06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. DOI: 10.1007/3-540-31188-2.

[EH07] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, pp. 1–18. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297029.

[EML+15] Anne Etien, Alexis Muller, Thomas Legrand, and Richard F. Paige. “Localized model transformations for building large-scale transformations”. English. In: *Software & Systems Modeling* 14.3 (2015), pp. 1189–1213. DOI: 10.1007/s10270-013-0379-8.

[ES05] Angelina Espinoza and Juan Garbajosa Sopeña. “The need for a unifying traceability scheme”. In: *ECMDA-TW 2005*. SINTEF ICT Norway, Nov. 2005, pp. 175–184. ISBN: 82-14-03813-8.

[EVT+13] Kerstin I. Eder, Norha M. Villegas, Frank Trollmann, Patrizio Pelliccione, Hausi A. Müller, Daniel Schneider, Lars Grunske, Bernhard Rumpe, Marin Litoiu, and Anna Perini10. ’Assurance using models at runtime for self-adaptive software systems‘. In: *State-of-the-Art Survey on Models at Runtime*. Lecture Notes in Computer Science. Springer, 2013.

[Fav04a] Jean-Marie Favre. “Foundations of model (driven) (reverse) engineering – episode i: story of the fidus papyrus and the solarus”. In: *Post-Proceedings of Dagstuhl seminar on model driven reverse engineering*. 2004.

[Fav04b] Jean-Marie Favre. "Towards a basic theory to model model driven engineering". In: *In Proc. of the UML2004 Int. Workshop on Software Model Engineering*. 2004.

[FBJ+05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. "Applying generic model management to data mapping". In: *21èmes Journées Bases de Données Avancées, BDA 2005, Saint Malo, 17-20 octobre 2005, Actes (Informal Proceedings)*. Ed. by Véronique Benzaken. 2005.

[FBV06] Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. "Weaving models with the Eclipse AMW plugin". In: *In Eclipse Modeling Symposium, Eclipse Summit Europe*. 2006.

[FDN+15] Johan Fabry, Tom Dinkelaker, Jacques Noyé, and Éric Tanter. "A taxonomy of domain-specific aspect languages". In: *ACM Computer Survey* 47.3 (Feb. 2015), 40:1–40:44. DOI: 10.1145/2685028.

[Fer14] Maribel Fernández. *Programming languages and operational semantics*. Springer, 2014. DOI: 10.1007/978-1-4471-6368-8.

[FFH+15] Florian Fittkau, Santje Finke, Wilhelm Hasselbring, and Jan Waller. "Comparing trace visualizations for program comprehension through controlled experiments". In: *23rd IEEE International Conference on Program Comprehension*. ICPC '15. Florence, Italy: IEEE Press, 2015, pp. 266–276.

[FHM97] Ralph Frisbie, Richard Hendrickson, and Michael Metcalf. "The F programming language". In: *SIGPLAN Notices* 32.6 (June 1997), pp. 69–74. ISSN: 0362-1340. DOI: 10.1145/261353.261363.

[FHN06] Jeanremy Falleri, Marianne Huchard, and Clémentine Nebut. "Towards a traceability framework for model transformations in Kermeta". In: *In Proceedings of the European Conference on Model Driven Architecture – Workshop on Traceability*. 2006.

[FKH15] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. "Hierarchical software landscape visualization for system comprehension: a controlled experiment". In: *3rd IEEE Working Conference on Software Visualization*. IEEE, Sept. 2015.

[FLV12] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. 'Model driven engineering languages and systems: 15th international conference, models 2012, innsbruck, austria, september 30–october 5, 2012. proceedings'. In: ed. by Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson. Berlin, Heidelberg: Springer, 2012. Chap. Modeling the Linguistic Architecture of Software Products, pp. 151–167. ISBN: 978-3-642-33666-9. DOI: 10.1007/978-3-642-33666-9_11.

[Fow10] Martin Fowler. *Domain specific languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 9780321712943.

[GCD+12] Clément Guy, Benoît Combemale, Steven Derrien, JimR.H. Steel, and Jean-Marc Jézéquel. 'On model subtyping'. In: *Modelling Foundations and Applications*. Ed. by Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos. Vol. 7349. Lecture Notes in Computer Science. Springer, 2012, pp. 400–415. ISBN: 978-3-642-31490-2. DOI: 10.1007/978-3-642-31491-9_30.

[GG07] I. Galvão and A. Goknil. "Survey of traceability approaches in model-driven engineering". In: *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. 2007, pp. 313–313. DOI: 10.1109/EDOC.2007.42.

[GHH+12] Wolfgang Goerigk et al. "Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke". In: *Software Engineering*. Ed. by Stefan Jähnichen, Axel Küpper, and Sahin Albayrak. Vol. 198. LNI. GI, 2012, pp. 119–130. ISBN: 978-3-88579-292-5.

[GHJ+97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns – elements of reusable object-oriented software*. 37th. Amsterdam: Addison-Wesley Longman, 1997. ISBN: 0-201-63361-2.

[Gie08] Simon Giesecke. "Architectural styles for early goal-driven middleware platform selection". PhD thesis. Carl von Ossietzky University of Oldenburg, 2008. ISBN: 978-3-940019-54-7.

[GK10] Birgit Grammel and Stefan Kastenholz. “A generic traceability framework for facet-based traceability data extraction in model-driven software development”. In: *Proceedings of the 6th ECMFA Traceability Workshop*. ECMFA-TW ’10. Paris, France: ACM, 2010, pp. 7–14. DOI: 10.1145/1814392.1814394.

[GKK08] P. S. Grover, Rajesh Kumar, and Avadhesh Kumar. “Measuring changeability for generic aspect-oriented systems”. In: *SIGSOFT Software Engineering Notes* 33.6 (Oct. 2008), pp. 1–5. DOI: 10.1145/1449603.1449610.

[GL13] Daria Giacinto and Sebastian Lehrig. “Towards integrating Java EE into ProtoCom”. In: *KPDAYS*. 2013, pp. 69–78.

[GRG+15] Ursula Goltz, RalfH. Reussner, Michael Goedicke, Wilhelm Hasselbring, Lukas Märtin, and Birgit Vogel-Heuser. “Design for future: managed software evolution”. English. In: *Computer Science - Research and Development* 30.3-4 (2015), pp. 321–331. DOI: 10.1007/s00450-014-0273-9.

[GV13] Holger Giese and Thomas Vogel. *Model-driven engineering of adaptation engines for self-adaptive software*. Technical Report 66. Hasso Plattner Institute, 2013. (Visited on 11/12/2013).

[Has02] Wilhelm Hasselbring. ’Component-based software engineering‘. In: *Handbook of Software Engineering and Knowledge Engineering*. Singapore: World Scientific Publishing, 2002, pp. 289–305.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. “SCCharts: sequentially constructive statecharts for safety-critical applications: hw/sw-synthesis for a conservative extension of synchronous statecharts”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 372–383. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594310.

[HGH+15] Robert Heinrich, Stefan Gärtner, Tom-Michael Hesse, Thomas Ruhroth, Ralf H. Reussner, Kurt Schneider, Barbara Paech, and Jan Jürjens. "A platform for empirical research on information system evolution". In: *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2015, pp. 415–420. DOI: 10.18293/SEKE2015-66.

[HHJ+13] Wilhelm Hasselbring, Robert Heinrich, Reiner Jung, Andreas Metzger, Klaus Pohl, Ralf Reussner, and Eric Schmieders. *iObserve: integrated observation and modeling techniques to support adaptation and evolution of software systems*. Technical Report. Kiel, Germany: Kiel University, Oct. 2013. URL: http://eprints.uni-kiel.de/22077/.

[Hic08] Rich Hickey. "The Clojure programming language". In: *Proceedings of the 2008 symposium on Dynamic languages*. DLS '08. Paphos, Cyprus: ACM, 2008, 1:1–1:1. DOI: 10.1145/1408681.1408682.

[HJS+15] Robert Heinrich, Reiner Jung, Eric Schmieders, Andreas Metzger, Wilhelm Hasselbring, Ralf Reussner, and Klaus Pohl. "Architectural run-time models for operator-in-the-loop adaptation of cloud applications". In: *IEEE 9th Symposium on the Maintenance and Evolution of Service-Oriented Systems and Cloud-Based Environments*. IEEE, 2015.

[HPP+05] Marcelo Victora Hecht, Eduardo Kessler Piveta, Marcelo Soares Pimenta, and R. Tom Price. "Aspect-oriented code generation". In: *Simpósio Brasileiro de Engenharia de Software*. 2005.

[HSJ+14] Robert Heinrich, Eric Schmieders, Reiner Jung, Kiana Rostami, Andreas Metzger, Wilhelm Hasselbring, Ralf Reussner, and Klaus Pohl. "Integrating run-time observations and design component models for cloud system analysis". In: *Proceedings of the 9th Workshop on Models@run.time*. Vol. 1270. Workshop Proceedings. CEUR, Sept. 2014, pp. 41–46.

[HSJ+15] Robert Heinrich, Eric Schmieders, Reiner Jung, Wilhelm Hasselbring, Andreas Metzger, Klaus Pohl, and Ralf Reussner. *Run-time architecture models for dynamic adaptation and evolution of cloud applications*. Research Report. Kiel, Germany: Kiel University, Apr. 2015. URL: http://eprints.uni-kiel.de/28566/.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007, p. 184. ISBN: 0521692695.

[HWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Kieker: a framework for application performance monitoring and dynamic software analysis". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22-25, 2012: ACM, Apr. 2012, pp. 247–248. ISBN: 978-1-4503-1202-8.

[IE11] Itemis AG and Eclipse Foundation. *XText - dsl devlopment framework*. 2011. URL: http://www.eclipse.org/Xtext/.

[IEC03] *IEC EN 61131-3*. Standard. 2003.

[IS091] *International standard ISO/IEC 9126. information technology: software product evaluation: quality characteristics and guidelines for their use*. Standard. International Standards Organisation, 1991.

[ISO11] *ISO/IEC 25010 - systems and software engineering - systems and software quality requirements and evaluation (SQuaRE) - system and software quality models*. Standard. International Standards Organisation, 2011.

[ISO12] *ISO/IEC 14882:2011 information technology — programming languages — C++*. Standard. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.)

[ISOC+99] JTC 1/SC 22/WG 14. *ISO/IEC 9899:1999: programming languages – C*. Standard. International Organization for Standards, 1999.

[Ite11] Itemis AG. *Xtend 2*. 2011. URL: http://www.eclipse.org/Xtext/documentation/2_0_0/01-Xtend_Introduction.php.

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. "ATL: a QVT-like transformation language". In: *OOPSLA Companion*. 2006, pp. 719–720.

[Jac99] B. Jacobs. *Categorical logic and type theory*. Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland, 1999.

[JBF11] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. "Model-driven language engineering with Kermeta". In: *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*. GTTSE'09. Braga, Portugal: Springer, 2011, pp. 201–221. DOI: 10.1007/978-3-642-18023-1_5.

[JH14] Arne N. Johanson and Wilhelm Hasselbring. "Hierarchical combination of internal and external domain-specific languages for scientific computing". In: *Proceedings of the 2014 European Conference on Software Architecture Workshops*. Vienna, Austria: ACM, 2014, 17:1–17:8. DOI: 10.1145/2642803.2642820.

[JHS+14] Reiner Jung, Robert Heinrich, Eric Schmieders, Misha Strittmatter, and Wilhelm Hasselbring. "A method for aspect-oriented meta-model evolution". In: *Proceedings of the 2Nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO '14. York, UK: ACM, 2014, 19:19–19:22. DOI: 10.1145/2631675.2631681.

[JHS13] Reiner Jung, Robert Heinrich, and Eric Schmieders. "Model-driven instrumentation with Kieker and Palladio to forecast dynamic applications". In: *KPDAYS*. Vol. 1083. CEUR Workshop Proceedings. 2013, pp. 99–108.

[JK06] Frédéric Jouault and Ivan Kurtev. "Transforming models with ATL". In: *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*. MoDELS'05. Montego Bay, Jamaica: Springer, 2006, pp. 128–138. DOI: 10.1007/11663430_14.

[Joh79] Stephen C. Johnson. *Yacc: yet another compiler-compiler*. Technical Report. 1979.

[Jör13] Sven Jörges. *Construction and evolution of code generators - a model-driven and service-oriented approach*. Vol. 7747. Lecture Notes in Computer Science. Springer, 2013, pp. 3–221. ISBN: 978-3-642-36126-5.

[Jou05] Frédéric Jouault. "Loosely coupled traceability for ATL". In: *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*. 2005, pp. 29–37.

[JSH13] Reiner Jung, Christian Schneider, and Wilhelm Hasselbring. "Type systems for domain-specific languages". In: *Software Engineering 2013 Workshopband*. Ed. by Stefan Wagner and Horst Lichter. Vol. 215. Lecture Notes in Informatics. Gesellschaft für Informatik e.V., 2013, pp. 139–154.

[Jun13] Reiner Jung. *Data type language*. Nov. 2013. URL: https://github.com/research-iobserve/pcm-data-type-language.

[Jun14a] Reiner Jung. *Behavior language*. Jan. 2014. URL: https://github.com/research-iobserve/pcm-behavior-language.

[Jun14b] Reiner Jung. "GECO: generator composition for aspect-oriented generators". In: *Doctoral Symposium – Models 2014*. Sept. 2014.

[Jun16a] Reiner Jung. *Replication package of the generator composition approach (GECO)*. Oct. 2016. URL: http://dx.doi.org/10.5281/zenodo.46552.

[Jun16b] Reiner Jung. *Software for the GECO replication package*. Mar. 7, 2016. URL: http://dx.doi.org/10.5281/zenodo.47129.

[Kai14] Robert Kaiser. *Qualitative experteninterviews*. VS Verlag für Sozialwissenschaften, 2014.

[KAK09] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. "Aspect-oriented multi-view modeling". In: *AOSD*. Ed. by Kevin J. Sullivan, Ana Moreira, Christa Schwanninger, and Jeff Gray. 2009, pp. 87–98. ISBN: 978-1-60558-442-3.

[Kay07] Michael Kay. *XSL transformations (XSLT) version 2.0*. W3C Recommendation. Word Wide Web Consortium, Jan. 2007. URL: http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[KDG+13] Marco Konersmann, Zoya Durdik, Michael Goedicke, and Ralf H. Reussner. "Towards architecture-centric evolution of long-living systems (the advert approach)". In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA '13. Vancouver, British Columbia, Canada: ACM, 2013, pp. 163–168. DOI: 10.1145/2465478.2465496.

[Ker88] Brian W. Kernighan. *The C programming language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[KGH+10] Lucia Kapova, Thomas Goldschmidt, Jens Happe, and Ralf H. Reussner. "Domain-specific templates for refinement transformations". In: *MDI '10: First International Workshop on Model-Drive Interoperability*. Oslo, Norway: ACM, 2010, pp. 69–78. ISBN: 978-1-4503-0292-0. DOI: 10.1145/1866272.1866282.

[KK07] Jacques Klein and Jorg Kienzle. "Reusable aspect models". In: *11th Workshop on Aspect-Oriented Modeling, AOM at Models'07*, 2007.

[KK11] Max E. Kramer and Jörg Kienzle. "Mapping aspect-oriented models to aspect-oriented code". In: *Proceedings of the 2010 international conference on Models in software engineering*. MODELS'10. Oslo, Norway: Springer, 2011, pp. 125–139. DOI: 10.1007/978-3-642-21210-9_12.

[KKL01] H. Kabaili, R.K. Keller, and F. Lustman. "Cohesion as changeability indicator in object-oriented systems". In: *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. 2001, pp. 39–46. DOI: 10.1109/.2001.914966.

[Koz11] Heiko Koziolek. "Sustainability evaluation of software architectures: a systematic review". In: *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium –*

ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS. QoSA-ISARCS '11. Boulder, Colorado, USA: ACM, 2011, pp. 3–12. DOI: 10.1145/2000259.2000263.

[KR08] Heiko Koziolek and Ralf Reussner. 'A model transformation from the Palladio component model to layered queueing networks'. In: *Performance Evaluation: Metrics, Models and Benchmarks*. Ed. by Samuel Kounev, Ian Gorton, and Kai Sachs. Vol. 5119. Lecture Notes in Computer Science. Springer, 2008, pp. 58–78. DOI: 10.1007/978-3-540-69814-2_6.

[Kra12] Max E. Kramer. "Generic and extensible model weaving and its application to building models". Master Thesis. Karlsruhe Institute of Technology, 2012.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 'MontiCore: modular development of textual domain specific languages. Objects, components, models and patterns: 46th international conference, tools europe 2008, zurich, switzerland, june 30 - july 4, 2008. proceedings'. In: *Objects, Components, Models and Patterns*. Ed. by Richard F. Paige and Bertrand Meyer. Berlin, Heidelberg: Springer, 2008, pp. 297–315. ISBN: 978-3-540-69824-1.

[KS08] Ekkart Kindler and David Schmelter. "Aspect-oriented modelling from a different angle: modelling domains with aspects". In: *AOM '08: Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*. Brussels, Belgium: ACM, 2008, pp. 7–12. DOI: 10.1145/1404920.1404922.

[KV10] Lennart C.L. Kats and Eelco Visser. "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497.

[Lad09] Ramnivas Laddad. *AspectJ in action, second edition*. 2009.

[Lai96] Kari Laitinen. "Estimating understandability of software documents". In: *SIGSOFT Software Engineering Notes* 21.4 (July 1996), pp. 81–92. ISSN: 0163-5948. DOI: 10.1145/232069.232092.

[LAN+15] Youness Laghouaouta, Adil Anwar, Mahmoud Nassar, and Jean-Michel Bruel. 'A generic traceability framework for model composition operation'. In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by Khaled Gaaloul, Rainer Schmidt, Selmin Nurcan, Sérgio Guerreiro, and Qin Ma. Lecture Notes in Business Information Processing. Springer, 2015, pp. 461–475. DOI: 10.1007/978-3-319-19237-6_29.

[LAN13] Youness Laghouaouta, Adil Anwar, and Mahmoud Nassar. "A traceability approach for model composition". In: *Computer Systems and Applications (AICCSA), 2013 ACS International Conference on*. May 2013, pp. 1–4. DOI: 10.1109/AICCSA.2013.6616448.

[LS90] M. E. Lesk and E. Schmidt. 'Unix vol. ii'. In: ed. by A. G. Hume and M. D. McIlroy. Philadelphia, PA, USA: W. B. Saunders Company, 1990. Chap. Lex – Lexical Analyzer Generator, pp. 375–387. ISBN: 0-03-047529-5.

[McC62] John McCarthy. *LISP 1.5 programmer's manual*. The MIT Press, 1962. ISBN: 0262130114.

[McC76] Thomas J. McCabe. "A complexity measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[MG06] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation". In: *Proc. of the Int'l WS on Graph and Model Transformation*. Vol. 152. Elsevier, 2006, pp. 125–142.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Computer Survey* 37.4 (Dec. 2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[MJ13] Abid Mehmood and Dayang N.A. Jawawi. "Aspect-oriented model-driven code generation: a systematic mapping study". In: *Information and Software Technology* 55.2 (2013). Special Section: Component-Based Software Engineering (CBSE), 2011, pp. 395–411. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.09.003.

[MJL+06] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. *Acceleo user guide*. 2006.

[MKB+08] Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. "A generic weaver for supporting product lines". In: *Proceedings of the 13th International Workshop on Early Aspects*. EA '08. Leipzig, Germany: ACM, 2008, pp. 11–18. DOI: 10.1145/1370828.1370832.

[MOF15] *Meta object facility (MOF) core specification*. Standard. Object Management Group, June 2015.

[Mor09] Rajiv Mordani. *JSR 315: Java servlet 3.0 specification*. Java Specification Request. Oracle, 2009.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990, pp. I–XI, 1–101. ISBN: 978-0-262-63132-7.

[MVL+08] Brice Morin, Gilles Vanwormhoudt, Philippe Lahire, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. 'Managing variability complexity in aspect-oriented modeling'. In: *Model Driven Engineering Languages and Systems*. Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter. Vol. 5301. Lecture Notes in Computer Science. Springer, 2008, pp. 797–812. DOI: 10.1007/978-3-540-87875-9_55.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. In: *Compiler Construction*. Ed. by Görel Hedin. Berlin, Heidelberg: Springer, Apr. 7, 2003. Chap. Polyglot: An Extensible Compiler Framework for Java, pp. 138–152. ISBN: 978-3-540-36579-2. DOI: 10.1007/3-540-36579-6_11.

[NLS+12] S. Naujokat, A. L. Lamprecht, B. Steffen, S. Jörges, and T. Margaria. “Simplicity principles for plug-in development: the jABC approach”. In: *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*. June 2012, pp. 7–12. DOI: 10.1109/TOPI.2012.6229816.

[OCL06] *Object constraint language object constraint language, OMG available specification, version 2.0*. Standard. Object Management Group, 2006.

[OMG06] *CORBA component model 4.0 specification*. Specification Version 4.0. Object Management Group, Apr. 2006.

[OMG14] *Model-driven architecture (MDA)*. Standard. Object Management Group, 2014. URL: http://www.omg.org/mda/.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008. URL: http://www.artima.com/shop/programming_in_scala.

[Par07] Terence Parr. *The definitive ANTLR reference guide: building domain-specific languages*. Raleigh, NC: Pragmatic Bookshelf, 2007. ISBN: 978-0-9787-3925-6.

[PEM12] Vern Paxson, Will Estes, and John Millaway. *Lexical analysis with Flex, for Flex 2.5.37*. 2012. URL: http://flex.sourceforge.net/manual (visited on 03/02/2015).

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002, pp. I–XXI, 1–623. ISBN: 978-0-262-16209-8.

[Pie04] Benjamin C. Pierce. *Advanced topics in types and programming languages*. The MIT Press, 2004. ISBN: 0262162288.

[PKP13] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. 'Metamodelling for grammarware researchers'. English. In: *Software Language Engineering*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, 2013, pp. 64–82. ISBN: 978-3-642-36088-6. DOI: 10.1007/978-3-642-36089-3_5.

[Pro] Spoofax Project. *The type specification language*. URL: http://www.metaborg.org/ts/ (visited on 03/02/2016).

[QVT05] *MOF QVT final adopted specification*. Standard. Object Management Group, June 2005.

[RGL+13] Louis Rose, Esther Guerra, Juan de Lara, Anne Etien, Dimitris Kolovos, and Richard Paige. "Genericity for model management operations". English. In: *Software & Systems Modeling* 12.1 (2013), pp. 201–219. ISSN: 1619-1366.

[Rit93] Dennis M. Ritchie. "The development of the C language". In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: ACM, 1993, pp. 201–208. ISBN: 0-89791-570-4. DOI: 10.1145/154766.155580.

[RLL98] David Rowe, John Leaney, and David Lowe. "Defining systems evolvability - a taxonomy of change". In: *International Conference and Workshop: Engineering of Computer-Based Systems*. Maale Hachamisha, Israel: IEEE Computer Society, Apr. 1998, pp. 45+.

[Ros95] Guido Rossum. *Python reference manual*. Manual. Amsterdam, The Netherlands, 1995.

[RRM+11] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, eds. *The common component modelling example (CoCoME)*. Vol. 5153. Lecture Notes in Computer Science. Springer, 2011.

[SAM+14] Matthias Schöttle, Omar Alam, Gunter Mussbacher, and Jörg Kienzle. "Specification of domain-specific languages based on concern interfaces". In: *Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages*. FOAL '14. Lugano, Switzerland: ACM, 2014, pp. 23–28. DOI: 10.1145/2588548.2588551.

[SB99] R. Solingen and E. Berghout. *The goal/question/metric method: a practical guide for quality improvement of software development*. McGraw-Hill, 1999. ISBN: 9780077095536.

[SBP+09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: eclipse modeling framework*. 2. Boston, MA: Addison-Wesley, 2009. ISBN: 978-0-321-33188-5.

[SG08] Jesús Sánchez Cuadrado and Jesús Garcia Molina. "Approaches for model transformation reuse: factorization and composition". In: *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*. ICMT '08. Zurich, Switzerland: Springer, 2008, pp. 168–182. ISBN: 978-3-540-69926-2. DOI: 10.1007/978-3-540-69927-9_12.

[She88] Martin Shepperd. "A critique of cyclomatic complexity as a software metric". In: *Softw. Eng. J.* 3.2 (1988), pp. 30–36.

[SHN+13] H. Saada, M. Huchard, C. Nebut, and H. Sahraoui. "Recovering model transformation traces using multi-objective optimization". In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. Nov. 2013, pp. 688–693. DOI: 10.1109/ASE.2013.6693134.

[SJ07] Jim Steel and Jean-Marc Jézéquel. "On model typing". In: *Software & Systems Modeling* 6.4 (2007), pp. 401–413. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0036-6.

[Smi02] Felicity Smith. *Research methods in pharmacy practice*. Pharmaceutical Press, 2002. ISBN: 9780853694816.

[SMM12] *Architecture-driven modernization (ADM): structured metrics metamodel (SMM), v. 1.0*. Object Management Group, 2012. URL: http://www.omg.org/spec/SMM/.

[SRH+15] Misha Strittmatter, Kiana Rostami, Robert Heinrich, and Ralf H. Reussner. "A modular reference structure for component-based architecture description languages". In: *Proceedings of the 2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems co-located MoDELS 2015*. Sept. 28, 2015, pp. 36–41.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – Putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of VL/HCC*. San Jose, CA, USA, Sept. 2013.

[Ste12] Eike Stepper. *Connected Data Objects (CDO)*. 2012. URL: http://www.eclipse.org/cdo/documentation/index.php.

[SV06] Thomas Stahl and Markus Völter. *Model-driven software development – technology, engineering, management*. electronic version. Wiley & Sons, 2006.

[TFG+07] Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. "Fast best-effort pattern matching in large attributed graphs". In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '07. San Jose, California, USA: ACM, 2007, pp. 737–746. DOI: 10.1145/1281192.1281271.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. 'On the use of higher-order model transformations'. In: *Model Driven Architecture - Foundations and Applications*. Ed. by Richard F. Paige, Alan Hartman, and Arend Rensink. Vol. 5562. Lecture Notes in Computer Science. Springer, 2009, pp. 18–33. DOI: 10.1007/978-3-642-02674-4_3.

[TK09] Frank F. Tsui and Orlando Karam. *Essentials of software engineering, second edition*. 2nd. USA: Jones and Bartlett Publishers, Inc., 2009. ISBN: 9780763785345.

[UML15] *OMG unified modeling language*. Standard. Version 2.5. Object Management Group, Mar. 2015. URL: http://www.omg.org/spec/UML/2.5.

[VAB06] Bert Vanhooff, Dhouha Ayed, and Yolande Berbers. "A framework for transformation chain development processes". In: *Proceedings of the ECMDA Composition of Model Transformations Workshop* (2006), pp. 3–8.

[VB05] Bert Vanhooff and Yolande Berbers. "Supporting modular transformation units with precise transformation traceability metadata". In: *European Conference on Model Driven Architecture, European Conference on Model Driven Architecture, Nurnberg, Germany, November 7-10, 2005*. Nov. 2005, pp. 15–27.

[VJB+13] Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière. “Typing artifacts in megamodeling”. In: *Software & Systems Modeling* 12.1 (2013), pp. 105–119. DOI: 10.1007/s10270-011-0191-2.

[VNH+10] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. 'Incremental model synchronization for efficient run-time monitoring'. In: *Models in Software Engineering*. Lecture Notes in Computer Science 6002. Springer, Jan. 2010, pp. 124–139. DOI: 10.1007/978-3-642-12261-3_13. (Visited on 11/21/2013).

[Völ11] Markus Völter. *Xtext/TS - a Typesystem Framework for Xtext*. 2011. URL: http://www.infoq.com/articles/xtext%5C_ts.

[VP12] Markus Völter and Vaclav Pech. “Language modularity with the MPS language workbench”. In: *Proceedings of the 2012 International Conference on Software Engineering*. ICSE 2012. Zurich, Switzerland: IEEE Press, 2012, pp. 1449–1450. ISBN: 978-1-4673-1067-3.

[VVH+06] Bert Vanhooff, Stefan Van Baelen, Aram Hovsepyan, Wouter Joosen, and Yolande Berbers. 'Towards a transformation chain modeling language'. English. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Ed. by Stamatis Vassiliadis, Stephan Wong, and Timo D. Hämäläinen. Vol. 4017. Lecture Notes in Computer Science. Springer, 2006, pp. 39–48. DOI: 10.1007/11796435_6.

[VVJ+07] Bert Vanhooff, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. “Traceability as input for model transformations”. In: *Third ECMDA Traceability Workshop (ECMDA-TW) 2007 Proceedings,* ed. by Jon Oldevik, Goran K. Olsen, and Tor Neple. Trondheim, Norway: SINTEF, June 2007, pp. 37–46.

[W3C14] W3C XSL/XML Query Working Groups. *The XPath 3.0 standard*. W3C Recommendation. World Wide Web Consortium, 2014. URL: http://www.network-theory.co.uk/w3c/xpath/.

[WKR+11] Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. "Reusing model transformations across heterogeneous metamodels". In: *ECEASST* 50 (2011).

[WM96] Arthur H. Watson and Thomas J. McCabe. *Structured testing: a testing methodology using the cyclomatic complexity metric*. Report NIST Special Publication 500-235. National Institute of Standards and Technology (NIST), Sept. 1996.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured design: fundamentals of a discipline of computer program and systems design*. 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1979. ISBN: 0138544719.

This page is intentionally left blank

This page is intentionally left blank