Sven Jörges

Construction and Evolution of Code Generators

A Model-Driven and Service-Oriented Approach

Springer

Author

Sven Jörges
Technische Universität Dortmund
Department of Computer Science
Otto-Hahn-Straße 14
44227 Dortmund, Germany
E-mail: sven.joerges@tu-dortmund.de

This monograph constitutes a revised version of the author's doctoral dissertation, which was submitted to TU Dortmund, Department of Computer Science, Chair of Programming Systems, Otto-Hahn-Straße 14, 44227 Dortmund, Germany, under the original title "Genesys: A Model-Driven and Service-Oriented Approach to the Construction and Evolution of Code Generators," and which was accepted in December 2011.

Lots of figures in this book use icons from the excellent Tango Desktop Project (http://tango.freedesktop.org/).

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36126-5 e-ISBN 978-3-642-36127-2
DOI 10.1007/978-3-642-36127-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012955656

CR Subject Classification (1998): D.3.4, D.2.11, D.2.1-4, D.2.13, D.3.2-3, J.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Foreword

Domain-specific languages are key for canalizing the steadily growing demand for tailored applications. They help strengthen the role of application experts during the development process, sometimes to the extent that they can adapt or even develop special applications themselves. At the same time, tailored user-oriented hardware allows software applications to slowly conquer everyone's lives. Think of the iPhone, for example, whose immense portfolio of applications (train schedules including delay information, route planning, cinema programs, calendar, compass, camera, Internet browsing, and thousands more) makes clear that the phone itself is no longer the dominant part. Code generators are required to bridge the gap between these two independently growing dimensions. Although code generation is a well-developed discipline, the support to efficiently keep these two dimensions in synchrony is small. In essence, even though there are code generator workbenches that help the programming expert to design dedicated code generators, more high-level activities like process support, variability management, or product lining are not foreseen.

Sven Jörges's thesis is unique in addressing this important niche:

- It develops a code generation framework (Genesys) on the basis of the jABC framework with its underlying eXtreme Model-Driven Development (XMDD) style, which combines ideas from extreme programming, model-driven design, and service orientation. This allows one to manage product lines at the model level, and to control their features via constraint solving, e.g., via model checking.
- It illustrates its impact not only by sketching numerous case studies, but also by showing how arbitrary domain-specific languages can be captured as long as they have an Ecore-based meta-model. This part of the thesis is particularly elegant, as it exploits the reflexivity of Ecore for radical bootstrapping of code generator functionality, and the ease of Genesys to be integrated, e.g., into the Eclipse framework.

- It shows how to overcome typical problems of round-trip engineering by employing full code generation. In the corresponding case study this is achieved by integrating AndroMDA, a code generation framework aiming at the translation of static aspects of a programming language.

Each of these three parts has been realized with great care and a mature look at adequate design, practicality, and quality assurance. Particularly outstanding, however, is the way all this is composed: things fit together nicely, and there is a very careful discussion of related work that puts the contributions of the thesis into perspective. It addresses topics as diverse as model-driven architectures, meta-modeling, domain-specific languages, computer-aided software engineering, round-trip engineering, code generation, generative programming, extreme programming, aspect orientation, product line management, and quality assurance. This way the reader obtains a competent goal-oriented entry into all of these areas, which is not just conceptual, but profits from experience Sven Jörges gained while realizing Genesys as a robust and flexible framework. Indeed, Genesys has been used and extended by numerous students, and it has been applied in various industrial projects.

Characteristic of the thesis is the sovereign handling of models at different (meta) levels for varying purposes. Models are used for modeling applications, but, continuously following the underlying XMDD approach, also the code generators, temporal formulae, and the test cases are modeled graphically, all in a similar fashion. In addition there is the hierarchy of meta-models specified in Ecore. This elegant, homogeneous structuring is an entry hurdle for "newcomers" to orient themselves, but Sven Jörges takes great care to clarify the context in the various scenarios. On the other hand, the continuous use of models opens the door for bootstrapping, as impressively demonstrated in various settings: e.g., a just-modeled code generator can be applied to itself, now seen as the application to be translated, in order to obtain a code generator in native code. This technique, which is later also applied to generate the required functionality for the code generation for Ecore, simply exploits the homogeneous notion of models and the fact that in jABC, models are executable via interpretation.

Sven Jörges proposes a change from viewing the construction of code generators merely as a technical low-level activity to a truly building-block-oriented construction paradigm exploiting model-driven and service-oriented approaches at higher abstraction levels. This paves the way for "mass construction/customization" of code generators in a growing landscape of domain-specific languages running on increasingly diverse hardware. I am convinced that the proposed model-based construction, validation, maintenance, evolution, and migration significantly reduces cost, increases reliability, and helps master the management of immensely growing sets of code generators.

Bernhard Steffen

Acknowledgments

I am grateful to many people for supporting, inspiring, and encouraging me during the last six years.

First, I would like to express gratitude to my PhD supervisor Prof. Dr. Bernhard Steffen for enabling me to write this book as a member of his research group. His advice and enthusiastic support have always been a great source of inspiration and motivation for me. In particular, I am very thankful that he had confidence in me when he allowed me to gather valuable experience in numerous interesting projects.

Furthermore, thanks are due to Prof. Dr. Jens Knoop and Prof. Dr. Rainer Hähnle for reviewing the monograph. I am also much obliged to the other members of my committee, Prof. Dr. Jakob Rehof and Dr. Oliver Rüthing.

Special thanks go to my colleagues Markus Doedt, Maik Merten, and Johannes Neubauer, who sacrificed a significant amount of their time to proofread the text. I am happy about each and every mistake you spotted! Likewise, "Diolch!" to Julia Rehder for lending me her English vocabulary and grammar intuition where mine reached their limits.

I would like to thank Christian Wagner for the constant exchange of ideas, literature findings, for the confusing discussions that heavily featured the "meta" prefix, and for introducing me to the delightful world of Bodum-brewed coffee. Furthermore, I am grateful to Prof. Tiziana Margaria for allowing me to benefit from her manifold contacts and for showing me how to be "context-sensitive," Dunja Rauh for supporting me with her advice and her inexhaustible collection of wise sayings, Ralf Nagel for being my interceder and for keeping me close to pragmatism, Christian Kubczak for the humor and the music, as well as Marco Bakera and Clemens Renner for patiently answering my questions on model checking. Thanks to all the colleagues from the LS5 for the great working atmosphere – I will surely miss the bad Friday puns.

Moreover, I would like to thank all the students who contributed to the Genesys project, in particular my former student assistants Benjamin Bentmann and Akif Köse, and the authors of all related diploma, bachelor, and master's theses.

Last but not least, I am deeply grateful to my family for their unlimited support that always keeps me going.

August 2011 Sven Jörges

Abstract

Automatic code generation is an essential cornerstone of model-driven approaches to software development. It closes the gap that emerges when models are used to abstract from a concrete software system, and thus is to models what compilers are to high-level languages. Consequently, the simple and fast development of code generators is a key requirement of today's approaches to model-driven development, which are increasingly characterized by a strong focus on agility and domain-specificity. Currently, many techniques are available that support the specification and implementation of code generators, such as engines based on templates or rule-based transformations. All these techniques have in common that code generators are either directly programmed or described by means of textual specifications.

This monograph presents *Genesys*, a general approach, which advocates the *graphical* development of code generators for arbitrary source and target languages, on the basis of *models* and *services*. In particular, it is designed to support incremental language development on arbitrary metalevels. The use of models allows building of code generators in a truly platform-independent and domain-specific way. Furthermore, models are amenable to formal verification methods such as model checking, which increase the reliability and robustness of the code generators. Services enable the reuse and integration of existing code generation frameworks and tools regardless of their complexity, and at the same time manifest as easy-to-use building blocks that facilitate agile development through quick interchangeability. Both models and services are reusable and thus form a growing repository for the fast creation and evolution of code generators.

This book shows these and further advantages arising from the Genesys approach by means of a full-fledged reference implementation, which has been field-tested in a large number of case studies.

List of Abbreviations

ABS	Abstract Behavioral Specification
AO	aspect orientation
ASL	Action Specification Language
ASSL	Autonomic System Specification Language
API	Application Programming Interface
ASP.NET	Active Server Pages .NET
ATL	Atlas Transformation Language
BiBiServ	Bielefeld University Bioinformatics Server
BNF	Backus-Naur Form
BPEL	Business Process Execution Language
BPM	Business Process Modeling
BPMN	Business Model & Notation
CASE	Computer-Aided Software Engineering
CDR	Common Data Representation
CIM	Computation-Independent Model
CLDC	Connected Limited Device Configuration
CORBA	Common Object Request Broker Architecture
CTL	Computation Tree Logic
CWM	Common Warehouse Metamodel
DBC	design by contract
DDBJ	DNA Data Bank of Japan
DOM	Document Object Model
DSL	domain-specific language
DSM	Domain-Specific Modeling
EBI	European Bioinformatics Institute
EJB	Enterprise JavaBean
EL	expression language
EMBL	European Molecular Biology Laboratory
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
EMOF	Essential Meta-Object Facility

ENF	Engeler Normal Form
ERP	enterprise resource planning
FBB	Formula Building Block
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
GP	Generative Programming
GPS	Global Positioning System
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HUTN	Human-Usable Textual Notation
IDE	Integrated Development Environment
ITE	Integrated Test Environment
jABC EE	jABC Execution Engine
JAXB	Java Architecture for XML Binding
JCE	Java Class Extruder
JCG1	Java Class Generator 1
JCG2-SC	Java Class Generator 2, structured code variant
JCG2-LI	Java Class Generator 2, lightweight interpreter variant, reflective service calls
JCG2-LI-GS	Java Class Generator 2, lightweight interpreter variant, generated service calls
JDK	Java Development Kit
JEE	Java Enterprise Edition
jETI	Java Electronic Tool Integration
JME	Java Micro Edition
JMI	Java Metadata Interface
JML	Java Modeling Language
JSE	Java Standard Edition
JSP	JavaServer Pages
JSTL	JavaServer Pages Standard Tag Library
JVM	Java Virtual Machine
KTS	Kripke Transition System
MIL	model-in-the-loop
MPS	Meta Programming System
M2M	Model 2 Model
MD*	Umbrella term for any model-driven approaches to software development
MDA	Model Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MDTD	Model-Driven Test Development
MDT	Model Development Tools
MOF	Meta-Object Facility
MOFM2T	MOF Model to Text Transformation Language

MSA	Multiple Sequence Alignment
NXC	Not eXactly C
OASIS	Organization for the Advancement of Structured Information Standards
OCS	Online Conference Service
oDOBS	Dortmunder Online Bibliographieservice
OMG	Object Management Group
OTA	One-Thing-Approach
PIL	processor-in-the-loop
PIM	Platform-Independent Model
POJO	Plain Old Java Object
POM	Project Object Model
PSM	Platform-Specific Model
QVT	Query/View/Transformation
RCX	Robotic Command Explorer
RMI	Remote Method Invocation
RTE	round-trip engineering
SIB	Service Independent Building Block
SIL	software-in-the-loop
SLG	Service Logic Graph
SUT	system under test
SWT	Standard Widget Toolkit
UID	unique identifier
UML	Unified Modeling Language
URL	Uniform Resource Locator
V&V	Verification & Validation
VTL	Velocity Template Language
WSDL	Web Services Description Language
XMDD	Extreme Model-Driven Development
XMI	XML Metadata Interchange
XML	Extensible Markup Language

List of Figures

2.1 Simple Bootstrapping Example: Getting a Native Compiler for Language L 13
2.2 Basic MD* terminology (by Stahl et al. [Sta+07, p. 28], translated into English) 15
2.3 Four-Level Example of MDA's Metamodel Hierarchy (based on [Obj10a, p. 19]) 18
2.4 Using a template engine for code generation 30

3.1 The Extreme Model-Driven Development (XMDD) approach 40
3.2 The jABC user interface [JSM10] 44
3.3 Service adapter pattern for realizing a SIB's behavior 50
3.4 Hierarchical models in jABC 51
3.5 The LocalChecker plugin in jABC [JMS11] 55
3.6 Executing an SLG with the Tracer 62
3.7 User interface of the GEAR plugin 68

4.1 Genesys architecture and involved roles 75
4.2 Data type mapping infrastructure in Genesys 81
4.3 Specifying variants via hierarchical modeling 86
4.4 The `Docu Generator` main model (topmost hierarchy level) . . . 89
4.5 The `Initialize Docu Generator` model (second hierarchy level) 90
4.6 The `Load Models` model (third hierarchy level) 91
4.7 The `Generate Documentation` model (second hierarchy level) 91
4.8 The `Generate Model Pages` model (third hierarchy level) . . . 92
4.9 The `Generate Markup for SIBs` model (fourth hierarchy level) 93
4.10 The model hierarchy of the Documentation Generator 94
4.11 Translating the SLGs of the Documentation Generator to Java code 95

4.12 Left hand side: Inspector for editing a code generator's metadata, Right hand side: Setting up a benchmark 97
4.13 Benchmark results visualized in tabular or bar chart form . . . 98
4.14 Creating a configuration for using a code generator in jABC . . . 100

5.1 Generation concept of the Java Class Extruder 104
5.2 Java Class Extruder: Processing the input SLGs 106
5.3 Bootstrapping the Java Class Extruder by means of the Tracer 107
5.4 Generation concept of the Java Class Generator 112
5.5 Basic control flow patterns and their translation into code . . . 116
5.6 The fork-join control flow pattern and its translation to code 116
5.7 An unstructured SLG and its equivalent structured pendant 117
5.8 Transformation of a fork-join construct to hierarchical submodels 118
5.9 Java Class Generator: Root Service Logic Graph (SLG) (1), Model transformations (2) 119
5.10 Java Class Generator: Detection of Control Flow Patterns . . . 121
5.11 Lightweight data structure for SLGs (excerpt) 122
5.12 Generation concept of the Java Class Generator's "Interpreter Variant" 123
5.13 Transformation phase of the Genesys Code Generator Generator 125
5.14 Reuse of models among the Java Class Generator Variants . . . 130
5.15 Distribution of Service Independent Building Block (SIB) bundles in the different Java code generators 132
5.16 Genealogical tree of code generators for jABC 137

6.1 Verification & Validation in Genesys 156
6.2 Example: Verifying the Java Class Extruder with GEAR and the FormulaBuilder 159
6.3 Constraints from the `actionOccurrence` category 161
6.4 Constraints from the `actionOrder` category 164
6.5 The new "Handle By" pattern with scope "before", meaning "Handle every A by P before Q". 166
6.6 A constraint checking for the complete handling of input SLGs using the "Handle By" pattern 167
6.7 Model-Driven Test Development (MDTD) in jABC 169
6.8 Strategy for testing the jABC code generators 171
6.9 Example SLGs modeling test inputs 173
6.10 The testing concept from Fig. 6.8, modeled in jABC 174
6.11 Excerpt from the test suite graph of the Java Class Generator 176

7.1 Approach for constructing code generators for EMF with Genesys . . . 178
7.2 EMF workflow . . . 179
7.3 Overview of the Ecore metamodel [Ecl05] . . . 181
7.4 Example metamodel in Ecore: Simple taxonomy . . . 182
7.5 Model `Generate EMF SIBs` . . . 183
7.6 Model `Generate Code for Model Element` . . . 184
7.7 Model `Generate Code for Structural Features` . . . 185
7.8 SIBs generated from the “Simple taxonomy” metamodel . . . 186
7.9 Taxonomy POJO Generator main model . . . 187
7.10 Model `Generate Code for Object` . . . 187
7.11 Example taxonomy model (“Media Catalog”) and generated POJO . . . 188
7.12 Modeling on different metalevels . . . 189
7.13 Code generators of the case study, by metalevels . . . 190

8.1 Integration concept for combining jABC’s code generators and AndroMDA . . . 194
8.2 Code generation approach in AndroMDA . . . 195
8.3 jABC models for the Multiple Sequence Alignment (MSA) process . . . 197
8.4 UML diagrams for the MSA web application . . . 199
8.5 Associating SLGs with UML diagrams (left hand side) and configuring the AndroMDA SIB (right hand side) . . . 200
8.6 Screens of the Generated MSA Web Application . . . 202

List of Tables

3.1 Complex built-in data types in jABC 49

4.1 Services for data type mapping 82
4.2 Services for identifier generation 84
4.3 Service for the generation of variants 87

5.1 Java mapping for complex jABC data types 114
5.2 Quantitative comparison of the different Java code generators 128
5.3 Experimental performance results for classes produced by the different Java code generators 134
5.4 Size of the generated Java source classes 135
5.5 JME mapping for complex jABC data types 145

Contents

Part I Motivation and Fundamentals

1 Introduction . . . 3
1.1 Requirements of the Genesys Approach . . . 5
1.2 Organization of the Book . . . 9

2 The State of the Art in Code Generation . . . 11
2.1 Influences of Compiler Construction . . . 11
2.2 Models, Metamodels and Domain-Specific Languages . . . 13
2.3 The Role of Code Generation . . . 19
2.3.1 Computer-Aided Software Engineering . . . 19
2.3.2 Generative Programming . . . 20
2.3.3 Model Driven Architecture . . . 22
2.3.4 Domain-Specific Modeling . . . 24
2.3.5 Language Workbenches . . . 25
2.3.6 Approaches without Code Generation . . . 26
2.4 Code Generation Techniques . . . 27
2.4.1 Programming the Code Generator . . . 28
2.4.2 Template-Based Code Generation . . . 29
2.4.3 Rule-Based Transformation . . . 31
2.4.4 Round-Trip Engineering versus Full Code Generation . . . 32
2.5 Quality Assurance of Code Generators . . . 34
2.6 Classification of Genesys . . . 35

3 Extreme Model-Driven Development and jABC . . . 39
3.1 Extreme Model-Driven Development . . . 39
3.2 jABC . . . 43
3.2.1 Service Independent Building Blocks . . . 46
3.2.2 Service Logic Graphs . . . 50
3.2.3 Plugins . . . 54

3.3 Model Execution with the Tracer ... 57
3.3.1 Execution Semantics ... 57
3.3.2 Execution Context ... 58
3.3.3 Control SIBs ... 60
3.3.4 Tracer Plugin ... 62
3.4 Model Checking with GEAR ... 63
3.4.1 Specification of Global Constraints ... 63
3.4.2 GEAR Plugin ... 66
3.5 jABC as a Basis for Realizing the Genesys Approach ... 68

Part II The Genesys Framework and Case Studies

4 The Genesys Framework ... 75
4.1 Services for Building Code Generators ... 78
4.1.1 Contributions to the Common SIBs ... 78
4.1.2 Type Mapping ... 79
4.1.3 Identifier Generation ... 83
4.1.4 Variant Management ... 85
4.2 Simple Example: Documentation Generator ... 87
4.2.1 Structuring the Generation Process ... 88
4.2.2 The Initialization Phase ... 89
4.2.3 The Generation Phase ... 90
4.2.4 Finalizing the Generator ... 94
4.2.5 General Remarks on the Example ... 95
4.3 Genesys Tooling ... 96
4.3.1 Developer Tools ... 96
4.3.2 User Tools ... 99

5 Case Studies: Code Generators for jABC ... 101
5.1 Bootstrapping: Java Class Extruder ... 102
5.1.1 The Extruder Concept ... 103
5.1.2 Development of the Generator ... 105
5.1.3 Evaluation ... 107
5.2 Java Class Generator ... 109
5.2.1 Handling Service Calls ... 109
5.2.2 From Single Class to Multiple Classes ... 112
5.2.3 Data Type Mappings and Execution Context ... 113
5.2.4 Variant 1: Structured Code ... 114
5.2.5 Variant 2: The Interpreter Approach ... 121
5.2.6 Genesys Code Generator Generator ... 125
5.2.7 Remarks on Different Versions ... 126

5.3 Comparison and Evaluation of the Java Code Generators . 127
5.3.1 Code Generator Models 127
5.3.2 Code Generator Results 132
5.3.3 Conclusions 135
5.4 Further Code Generators for jABC 136
5.4.1 Servlet Extruder and Servlet Generator 138
5.4.2 SIB Extruder and SIB Generator 139
5.4.3 Web Service Generator 140
5.4.4 leJOS and NXC Generator 140
5.4.5 BPEL Generator 142
5.4.6 C# Generator 143
5.4.7 JME Generator 144
5.4.8 EE Process Definition Generator 146
5.4.9 JML Extension for Java Class Generator 147
5.4.10 iPhone Generator 148
5.4.11 Code Generators for Ruby, Perl and JavaScript 150
5.4.12 Robocode Generator 151

6 Verification & Validation of Code Generators 155
6.1 Local Constraints for Code Generators 157
6.2 Global Constraints for Code Generators 158
6.2.1 FormulaBuilder 158
6.2.2 The Constraint Library 160
6.2.3 Occurrence Constraints 161
6.2.4 Order Constraints 163
6.2.5 Deriving Patterns & Composing Constraints 165
6.3 Testing of Code Generators 168
6.3.1 Testing the jABC Code Generators 170
6.3.2 Generation of Test Scripts for JUnit 173

7 Case Study: Domain-Specific Code Generators for EMF 177
7.1 Eclipse Modeling Framework 179
7.2 The Ecore Metamodel 180
7.3 EMF SIB Generator 182
7.4 Example: Taxonomy POJO Generator 185
7.5 Evaluation 188

8 Case Study: Service-Oriented Combination of Code Generation Frameworks 193
8.1 AndroMDA 195
8.2 Example Application: Multiple Sequence Alignment (MSA) 196

8.3 Integrated Modeling 198
8.4 Integrated Code Generation 200
8.5 Evaluation 202

Part III Conclusions and Future Work

9 Conclusions 207
9.1 Requirements of the Genesys Approach Revisited 207
9.2 Further Applications of Genesys 210

10 Future Work 215

Bibliography 223

Index 243

Part I

Motivation and Fundamentals

1 Introduction

Today's software systems and the platforms they reside on "are often among the most complex engineering systems" [Sel03]. In their day-to-day work, software engineers are faced with a huge and constantly changing variety of languages, tools, frameworks and platforms that need to be mastered.

Abstraction is a traditional and very powerful means for shielding software engineers from this complexity. For instance, operating systems hide the complexity of underlying hardware platforms, and compilers allow the use of high-level programming languages that abstract from low-level languages such as assembly or machine code.

At an additional level of abstraction, *models* allow the specification of a software system independent of the concrete technologies that are used for its actual implementation. However, in order to fully exploit the potential of models, it is not sufficient to use them for documentation and visualization purposes only (called *model-based* development by Stahl et. al [Sta+07, p. 3]). Instead the gap that arises between the model of a system (the abstraction) and a corresponding implementation (a concrete incarnation of the abstraction) has to be bridged.

This is the task of *code generators*, which automatically derive an implementation from the model, and which thus relate to models in the same way in which compilers relate to high-level languages. Consequently, models are promoted to primary development artifacts. Ideally, the availability of automatic code generation relieves the developer from ever having to deal with the resulting code, which is considered a by-product just like the results produced by compilers. Due to the prominent role of models, corresponding approaches to software development are called *model-driven*. Today this realm has many incarnations referred to by a plethora of acronyms such as MDD, MDE, MDSD, MDA, DSM (cf. Chap. 2 for details on the single abbreviations) and so forth. Völter [Völ09] coined *MD** as an umbrella term that conveniently comprises all approaches to model-driven development.

Apart from bridging the gap between models and implementations, code generation provides further advantages. Herrington [Her03, p. 15], e.g., refers to the high quality of generated code, which is usually more consistent than hand-written code. For instance, a code generator automatically and consistently propagates bug fixes and other modifications to all corresponding parts of the code. Furthermore, Herrington points out that the generation of cumbersome boilerplate code allows the software engineer to concentrate on the important design aspects.

With the evolution of MD* approaches, code generation has successively gained in importance. Whereas in earlier approaches, such as Computer-Aided Software Engineering (CASE) [CNW89], code generators usually were fixed and static parts of general-purpose development environments, today's approaches increasingly focus on domain-specificity: From the modeling language through to the required tooling, all aspects of a development environment are tailored to each domain, which usually in particular includes the construction of dedicated code generators.

Consequently, there is a high demand for approaches that enable a simple and fast development of code generators. Currently, lots of techniques are available that support the specification and implementation of code generators, such as dedicated Application Programming Interfaces (APIs), template engines or rule-based transformation engines (cf. Sect. 2.4). All those techniques have in common that code generators are either directly programmed or described by means of textual specifications.

This book presents a novel approach called *Genesys*, which suggests the *graphical* development of code generators on the basis of *models* and *services*. It will be shown that this unique approach provides significant advantages for the construction and evolution of code generators.

Why models?

Just like for any software system, the advantages of models can also be applied to code generators. As code generators tend to be very complex, they can in the same way benefit from the abstraction provided by models. Especially *hierarchical* models (i.e., supporting a mechanism that allows embedding models into each other) are a powerful means for mastering complexity. Furthermore, apart from its structuring character, hierarchy enables the separation of concerns, e.g., by only showing those parts of the model hierarchy which are of current interest to the developer.

Another benefit arising from the abstraction provided by models is portability: A code generator can be developed independently from a particular implementation language or host machine. Via suitable generator generators, the same code generator can be translated for and deployed to arbitrary host systems without the need of adapting the generator's design.

Finally, models are amenable to formal verification methods such as model checking (cf. Sect 1.1), which allows the application of powerful tools that increase the reliability and robustness of code generators.

Why graphical models?

According to Sendall, “there are perceived cognitive gains” [SK03] when using graphical notations, and available research on visual programming languages underpins this (see e.g., [Bla96; Whi97; Bla01; Moo09]). For instance, a frequently used argument (e.g., [Kle08, p. 5; Moo09]) is that textual notations are one-dimensional due to their purely sequential character, whereas the spatial arrangements used in graphical notations may be more complex, but at the same time lead to a greater expressiveness. Furthermore, Moody [Moo09] points out that graphical notations are usually processed in parallel by the human mind in contrast to the serial processing of textual notations. Finally, Schmidt [Sch06] argues that graphical representations help flatten learning curves.

Why services?

Services [MSR05] are reusable units with simple and unified interfaces. As their usage does not require any knowledge about their actual implementation, services are, just like models, a suitable means for hiding complexity. Furthermore, assembling a code generator from services adds to its modularity and adaptability: Single services can be easily replaced by newer or alternative versions, thus allowing *agile* development and evolution. Existing code generation techniques, tools and frameworks can be made available as services, which can be freely used and combined in order to realize complex code generators. Moreover, models that are assembled from services are typically *executable*. Apart from supporting rapid prototyping and easy debugging, constructing code generators as executable models also enables the application of bootstrapping (cf. Sect. 1.1).

Genesys is a *general* approach for modeling code generators for arbitrary source and target languages. In particular, it is designed to support incremental language development on arbitrary metalevels (cf. Chap. 7). This monograph shows the feasibility of the Genesys approach by means of a fully-fledged reference implementation, which has been field-tested in a large number of case studies. The conceptual and technical basis of this implementation is given by the *Extreme Model-Driven Development* (XMDD) paradigm and its tool incarnation *jABC* (cf. Sect. 3), which enable model-driven and service-oriented development according to the mindset described above. The reference implementation meets a number of requirements which have been identified for the realization of the Genesys approach.

1.1 Requirements of the Genesys Approach

As described above, model-driven development and service orientation are the basic principles of the Genesys approach. Based on these two key demands, several general requirements can be formulated:

Requirement G1 - Platform Independence
The abstract nature of models allows designing a code generator independent of a specific programming language or particular host system on which it will be executed. Thus the same code generator model may be translated for execution on different host systems. Using services as building blocks of models further strengthens this abstraction: For the generator developer, those services are black boxes, and their usage does not require any knowledge about their concrete implementation. In order to guarantee a code generator's translatability for multiple platforms, service implementations need to be easily interchangeable, but by all means transparent to the generator developer. Furthermore, the employed service mechanism must not pose any restrictions on what can be made available as a service, so that there are no limitations on which libraries or tools (e.g., template engines, cf. Sect. 2.4.2) can be used for composing a code generator. Finally, a code generation framework must not be restricted to code generators that only support particular source or target languages.

Requirement G2 - Reusability and Adaptability
As programming languages, platforms and libraries change rapidly and new ones emerge frequently, code generator frameworks need to support *fast creation* and *adaptation*. Facilitating this kind of agility is one of the key concerns of service orientation: Services are made available in repositories in a way that allows to reuse them among different application contexts, thus avoiding the proverbial "reinvention of the wheel". This reusability, which is self-evident at the service level, is also desirable for entire models. If models themselves are easily reusable, code generators and their features can, once modeled, be collected in a repository just like services, so that modelers benefit from work that has already been done before. Furthermore, as every new code generator in turn contributes reusable assets to this repository, the potential of reuse is growing continuously.

Requirement G3 - Simplicity
In recent code generation approaches, often a huge variety of different languages is involved in the development of a code generator (cf. Sect. 2.4), such as template languages, transformation languages, grammar notations or constraint languages. This demands a high learning effort prior to getting started and inevitably slows down development. Thus it is desirable to keep the number of required formalisms, modeling notations and languages as small as possible in order to reduce the complexity of using the framework, of course without limiting its overall functionality. Choosing services as basic building blocks for models already is a first step towards simplicity, as services should have standardized and very simple interfaces. When replacing one service with another, it is not necessary to learn a new API or even programming language, as it is often the case when substituting libraries at the code level. Furthermore, as

already mentioned above, services can be used without knowing any details about their actual implementation or complexity.

Requirement G4 - Separation of Concerns

As code generators may get very complex, a suitable modeling language and its corresponding tools should support the separation of concerns. A generator developer should be able to focus on particular aspects rather than being confronted with the full amount of information all along. *Hierarchical models* have already been mentioned above as a possible way of supporting the separation of concerns.

Requirement G5 - Verification and Validation

Software verification and validation (often referred to as "V&V") are key activities for ensuring that a system meets all previously specified requirements and needs, and that it is built in an appropriate and correct way. When writing source code in an Integrated Development Environment (IDE), usually several checking mechanisms are executed in the background, performing, e.g., syntax checks or static code analysis. If any problems are detected, the developer is immediately alerted. Similar checks are often also supported by today's modeling environments. Given a metamodel that describes a modeling language's abstract syntax and static semantics (cf. Sect. 2.2), a corresponding editor with suitable checking facilities can be easily provided. Apart from syntactic checking, formal methods like model checking (cf. Sect. 3.4) go further by verifying whether a model conforms to a set of given constraints, often tailored to a specific domain. Provided that a suitable modeling language is chosen (i.e., one that is supported by existing model checkers) and a library of constraints for code generation is created, model checkers can perform sophisticated verification and thus increase the robustness and reliability of code generators. This verification potential goes beyond anything supported by most code generation approaches today (cf. Sect. 2.5). Beyond such checking mechanisms, *testing* is another important facet of V&V that needs to be supported by a code generation framework.

On the basis of those fairly general demands, several more specific requirements arise, especially when considering the current state of the art in code generation (cf. Chap. 2):

Requirement S1 - Domain-Specificity

Being themselves an enabling technique for domain-specific modeling as outlined above, it is also desirable to construct code generators in a way that respects the domain knowledge of the generator developer. Instead of using a general-purpose modeling formalism, the selected modeling language should be adaptable to a given domain, including suitable terminology and visualization. This adaptation could be initially performed by a domain expert, or even be automated on the basis of a given metamodel that describes the domain. A customized modeling language tailored to the domain knowledge of the modeler greatly improves the simplicity of the approach (cf. *Requirement G3 - Simplicity*).

Requirement S2 - Full Code Generation

A common technique that is involved in several code generation approaches is *round-trip engineering*. Essentially, this term refers to the automatic tool support for keeping models in sync with code generated from them. This is especially necessary when generated artifacts have to be manually modified or completed by developers. Changes in the generated artifacts need to be propagated to the corresponding models and vice versa, while assuring at the same time that no manual work is lost or overwritten. Implementing automatic tool support for this task is very cumbersome and increases the complexity of involved code generators (Sect. 2.4.4 elaborates on that). Consequently, it is desirable to avoid the need of round-trip engineering and to employ approaches that enable *full code generation*. Accordingly, all modifications are exclusively performed on the models, while generated artifacts are considered a by-product that never has to be touched. If there are any changes to the models, then the corresponding artifacts are simply regenerated. This deliberate avoidance of a bidirectional synchronization of models and generated artifacts considerably eases the work of a corresponding code generator and the overall development process (cf. Sect. 2.4.4).

Requirement S3 - Variant Management and Product Lines

The evolution of code generators based on reuse and adaptation (cf. *Requirement G2 - Reusability and Adaptability*) needs to be facilitated by corresponding tools. This includes tool support for the definition and creation of variants, using existing code generators and features as patterns for new *product lines*.

Requirement S4 - Clean Code Generator Specification

A typical code generator usually consists of two main aspects. First, its *generation logic* realizes, e.g., the traversal of the input model's elements and collects the data required for code generation. Second, the code generator includes an *output description* that specifies the resulting code or markup (cf. Sect. 2.4). Especially modern template languages often mislead to mixing up those two aspects, which may make it harder to understand, maintain and adapt a code generator, and to use it as a pattern for other code generators (cf. *Requirement G2 - Reusability and Adaptability* and *Requirement S3 - Variant Management and Product Lines*). Thus it is desirable to establish a way for clearly separating generation logic and output description when designing a code generator (cf. *Requirement G2 - Reusability and Adaptability*).

Requirement S5 - Bootstrapping

Bootstrapping (see Sect. 2.1) is a common development technique that emerged from compiler construction. Basically, a new compiler is developed in several stages by incrementally adding target language features and applying existing compilers to each other. As this approach is well-established and proven, it should be supported by a modern code generation framework.

Requirement S6 - Tool-Chain Integration

As code generators are usually built to be integrated into tools or productive development setups consisting of certain tool-chains, a code generation framework and the code generators based on it need to be compatible with build management tools like Apache Maven [Apa11b].

1.2 Organization of the Book

This book is divided into three main parts:

Part I:

After this introductory chapter, Chap. 2 presents the state of the art in code generation. Besides establishing terminology that is required for the rest of the monograph, the overview of the fundamentals of code generation provided by this chapter is at the same time intended as a presentation of related work. Accordingly, the chapter finishes with a comparison of Genesys with other approaches. Afterwards, Chap. 3 introduces the XMDD paradigm as well as jABC along with several plugins that play a central role for this book. The chapter also compares other MD* approaches with XMDD/jABC and evaluates their suitability as a basis of the Genesys approach.

Part II:

This part elaborates on the reference implementation of the Genesys approach called the *Genesys framework*, and on the case studies that were performed in order to field-test it. Chap. 4 presents the Genesys framework along with its services and tooling. Furthermore, it exemplifies the use of the framework by describing the construction of a simple code generator. Chap. 5 elaborates on a collection of case studies that aimed at providing various code generators for jABC itself. Chap. 6 illustrates the verification and validation of code generators in the Genesys framework by means of examples from the jABC case studies. Another case study is presented in Chap. 7, which describes the use of Genesys for the construction of domain-specific code generators for Eclipse Modeling Framework (EMF) on the basis of services that are generated from a given metamodel. Finally, Chap. 8 discusses the last case study that deals with the integration of services into Genesys, exemplified by means of the code generation framework AndroMDA.

Part III:

Chap. 9 sums up and draws several conclusions. In particular, it revisits the requirements of the Genesys approach and shows how they have been met by the reference implementation. In the end, Chap. 10 elaborates on future plans and possible extensions of Genesys.

2 The State of the Art in Code Generation

Some of the requirements for the Genesys approach presented in Sect. 1.1 are a direct result of examining and evaluating the work that has been done in the field of code generation so far. This chapter provides an overview of the current state of the art in code generation for MD*. It starts off with a brief retrospect on classical compiler construction (Sect. 2.1), which developed ideas and concepts that clearly influence current code generation techniques. Sect. 2.2 elaborates on the conceptual foundations of MD* and on how the associated terminology is used in this book. Afterwards, Sect. 2.3 examines the role of code generation in several existing MD* (and related) approaches, and Sect. 2.4 introduces techniques for actually realizing code generators. Sect. 2.5 presents the state of the art in verifying and validating code generators. Finally, Sect. 2.6 compares Genesys with the approaches and techniques described in the preceding sections.

2.1 Influences of Compiler Construction

Beyond doubt, compiler construction is one of the most well-grounded and well-proven fields in computer science. Having its seeds in the early 1950s, compiler construction promoted the evolution of important theoretical topics such as formal languages, automata theory and program analysis. The introduction of compilers had far-reaching effects on software development, as they enabled the use of high-level programming languages (such as FORTRAN) instead of tediously writing software in low-level languages like assembly or even machine code. By *raising the level of abstraction*, developers should be shielded from hardware-specific details.

Code generation approaches for MD* share these ideas. According to Selic, "most standard techniques used in compiler construction can also be applied directly to model-based automatic code generation" [Sel03]. However, as models are by their very nature more abstract than source code (cf. Sect. 2.2), corresponding code generators work on a much higher level of abstraction than compilers for source code. The following paragraphs highlight some similarities as well as differences between classical compilers and MD* code generators, focusing on concepts and notions that are important for the Genesys approach.

General Structure:

In essence, a compiler translates a program written in a given source language into a program in a given target language. Usually, modern compilers are organized into consecutive phases, such as lexing, parsing or data flow analysis, each of them often operating on their own intermediate language or representation [App98, p. 4]. Depending on whether such a phase is concerned with analysis (i.e., resolving the source program into its constituent parts, assigning a grammatical structure, etc.) or synthesis (i.e., constructing the desired target program), the phase is said to be part of the compiler's front-end or back-end [Aho+06, p. 4], respectively. One of these phases is called "code generation", which is situated in a compiler's back-end. It usually retrieves some intermediate form, such as an abstract syntax tree produced by a parser, and translates it to code in the desired target language, e.g., machine code or bytecode executable by a virtual machine. This translation typically raises issues such as instruction selection, register allocation or code optimization.

For MD* code generators, especially issues close to hardware are at most secondary, and can often even be considered commodity. When generating code from abstract models, target languages are in most cases high-level languages (such as Java or C++) with existing compilers, interpreters or execution engines that further process the generated output. Accordingly, compilers can be regarded as tasks or services that are incorporated in or postpositioned to code generators. In a similar fashion, MD* code generators employ parsers in order to translate models from their serialized form (e.g., XML Metadata Interchange, XMI [Obj07]) to an in-memory representation (e.g., an implementation of the Java Metadata Interface, JMI [Jav02]) prior to the actual code generation. As there is extensive tool-support for the development of compilers and their single components, e.g., parser generators such as ANTLR [PQ95] or Lex/Yacc [LMB92], code generator developers can resort to a rich repertoire of mature services.

Bootstrapping:

Apart from source and target language, the compiler's implementation language is relevant to the categorization of the compiler. For instance, a *self-compiling* (or *self-hosting*) compiler [LPT78] is a compiler that is written in the language it compiles, and a *cross-compiler* [Hun90, p. 8] targets a machine other than the host. Especially self-compiling compilers are often used for *bootstrapping* [Wat93, p. 44], which is a common technique for evolving compilers. Typically, this approach aims at decreasing the overall complexity of compiler development by separating the implementation process into consecutive stages.

Fig. 2.1 uses the established notation of *T-diagrams* [Hun90, p. 11] for visualizing an example of a very simple bootstrapping process. In this notation, blocks that look like the letter "T" represent compilers. The three text labels on the blocks indicate the compiler's source language (left), target language (right) and implementation language (bottom). Suppose we want to implement a native compiler for a fictitious programming language called L. As a start, we implement version 1 of this compiler using C, an existing programming language with an available native compiler (M is for "machine code"). Afterwards, we compile the newly written compiler, which results in a native L-to-M compiler. We could stop at this point, but as the maintenance of our L-to-M compiler now depends on the existence of a C-compiler, we implement a second version in L (rebuilding should not be as hard as building from scratch). Finally, we compile version 2 using version 1 and get a native L-to-M compiler that is no longer dependent on C.

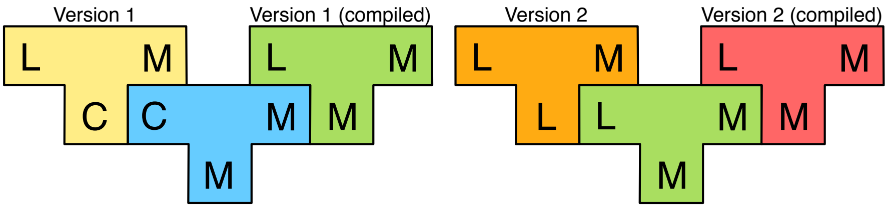


Fig. 2.1. Simple Bootstrapping Example: Getting a Native Compiler for Language L

The example in Fig. 2.1 is only a very small bootstrapping process. As mentioned above, bootstrapping is usually organized in stages in order to divide the implementation complexity into small manageable chunks. Instead of starting with the entire language L, a simple subset $L^* \subset L$ is identified, so that the first version of the compiler can be developed much easier. After building an L^*-to-M compiler in the manner described above, the compiler is enriched with the missing L-features and the procedure is repeated. Using several sublanguages with small feature additions in each stage further simplifies the implementation of the final compiler version.

The use of bootstrapping is also very common and desirable in MD* code generators, and thus an important technique used in the Genesys approach (see Sect. 1.1, 5.1 and 7.5). Throughout this work, T-diagrams will be used to visualize bootstrapping and other code generator evolution processes.

2.2 Models, Metamodels and Domain-Specific Languages

The existence of MD* approaches and numerous corresponding tools (cf. Sect. 2.3) indicates that there seems to be at least a common intuition of what a model actually is. However, there is still no generally accepted definition of the term "model". For instance, while Kleppe defines a model as "a linguistic utterance of a modeling language" [Kle08, p. 187], the Object Management Group (OMG) focuses on the role of the model as a means of specification [Obj03b, p. 12]:

> *"A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language."*

Kühne emphasizes the abstraction aspect of models [Küh06]:

> *"A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made."*

Another characterization of models that is cited frequently in the literature is the one of Stachowiak, who identifies three main features of models [Sta73, pp. 131–133]:

1. *Mapping feature:* A model is always a mapping of some natural or artificial original, which may in turn be a model.
2. *Reduction feature:* Generally, a model does not capture all attributes of the represented original, but only those relevant to the person who creates or uses the model.
3. *Pragmatic feature:* A model always serves a particular purpose.

This "fuzziness" or lack of precision can be observed for most of the vocabulary used in the context of MD*. There is still no established fundamental theory of modeling and related concepts that would be comparable to the maturity achieved in other disciplines of computer science, such as compiler construction (cf. Sect. 2.1). However, several publications (e.g., [BG01;Fav04;Küh06]) try to come up with precise definitions, and thus discuss issues like when it is appropriate to call a model a *metamodel*.

As a reflection of this discussion goes far beyond the scope of this monograph, all following chapters and sections resort to the terminology definitions described by Stahl et al. [Sta+07, pp. 28–32]. Fig. 2.2 uses the Unified Modeling Language (UML) [Obj10b; Obj10a] in order to illustrate the relevant concepts and their relationships, which are introduced in the following.

Domain:

A *domain* is a delimited field of interest or knowledge which consists of "real" things and concepts. It may also be divided into an arbitrary number of *subdomains*. For instance, the domain "hospital" contains, among other things, the subdomains "intensive care unit" and "coronary care unit", each capturing specific parts of the superordinate domain.

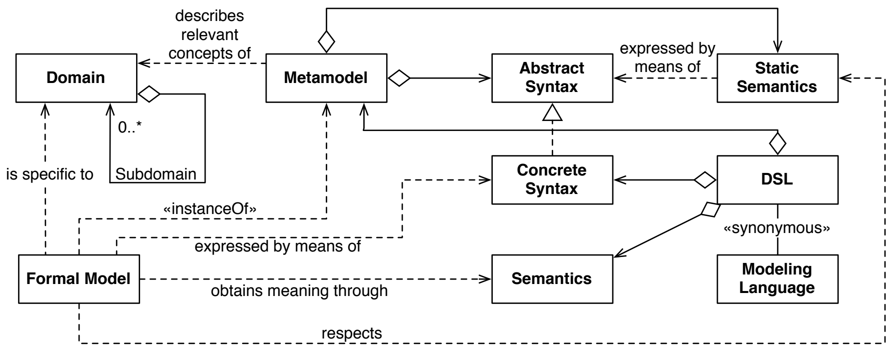


Fig. 2.2. Basic MD* terminology (by Stahl et al. [Sta+07, p. 28], translated into English)

Metamodel:

A *metamodel* is a formal description of a domain's relevant concepts. It specifies how formal models (or programs), that are specific to the given domain, can be composed. For this purpose, a metamodel comprises two important parts: the abstract syntax and the static semantics.

The *abstract syntax* defines the elements of the metamodel and their relationships, independent of the concrete representation of any corresponding formal model. For instance, the abstract syntax of an object-oriented language might define concepts like classes and interfaces, which have attributes such as a name and which are associated via relationships such as inheritance.

The *static semantics* specifies constraints for the well-formedness of a formal model. Accordingly, it is defined relative to an abstract syntax, i.e., it uses the contained terminology and concepts in order to describe the constraints. For instance, the static semantics of a metamodel for control flow graphs could specify constraints that demand the existence of exactly one start node.

Domain-Specific Language:

According to Fowler, the notion *domain-specific language* (DSL) refers to "a computer programming language of limited expressiveness focused on a particular domain" [Fow10, p. 27]. Stahl et al. [Sta+07] as well as this book use the notion synonymously with the term *modeling language*. As visible in Fig. 2.2, a DSL is based on a metamodel that comprises the abstract syntax and static semantics as described above.

Furthermore, a DSL provides a *concrete syntax*, which describes a particular representation of the elements and concepts specified by the abstract syntax. The concrete syntax can thus be considered an instance of the abstract syntax, and it is possible to define multiple concrete syntaxes for one abstract syntax. For instance, a UML class diagram [Obj10b] can be represented using at least three concrete syntaxes: the graphical UML notation itself, the Human-Usable Textual Notation (HUTN) [Obj04] and the XML-based interchange format XMI. In particular, this example illustrates that a concrete syntax – and thus the DSL and any formal model that follows the concrete syntax – can be textual or graphical.

The beginning of Chap. 1 presented several arguments that highlight advantages of graphical notations over purely textual notations (better cognitive accessibility, higher expressiveness, flatter learning curves etc.). However, there are also publications that argue in favor of textual notations. For instance, from the tool perspective, Völter [Völ09] points out that it requires more effort to build usable editors for graphical languages as opposed to textual editors. Stahl et al. [Sta+07, p. 103] exemplify this by means of the support for collaborative development: Whereas the synchronization of textual development artifacts is supported by a variety of tools (such as Subversion [Apa11e]), graphical notations often require the implementation of specific solutions.

Further positions advocate that graphical and textual notations are not mutually exclusive. Van Deursen et al. [DVW07] observe complementary strengths and thus propose a unification of both notations. Kleppe exemplifies UML class diagrams as such a hybrid concrete syntax, as they provide "a textual syntax embedded in a graphical one" [Kle08, p. 5]. Finally, Kelly and Pohjonen point out that the choice of a suitable concrete syntax "depends on the audience, the data's structure, and how users will work with the data" [KP09].

As the third component besides the metamodel and the concrete syntax, a DSL also provides *semantics* that assigns a meaning to any well-formed model written in the DSL. In practice, this semantics is often described by means of natural language as for instance performed in the UML specification [Obj10b]. However, in order to avoid the ambiguity and imprecision of natural languages, semantics can also be described formally, e.g., using a denotational [Sch86], operational [Plo81;Kah87], axiomatic [Hoa69] or translational approach [Kle08, p. 136f]. In the context of this book, the latter is most interesting: Following the translational approach, the semantics of a language is given by a translation into another language with well-known semantics. In MD*, such a translation can be provided by a model transformation, which may, e.g., be realized by a code generator. Sect. 2.3.5 elaborates on this role of code generation.

Fowler [Fow10, p. 15] distinguishes between internal and external DSLs. An *internal DSL* (also known as *embedded DSL*) forms a real subset of an existing (general-purpose) language, its "host language". It employs the syntactic constructs of the host language and maybe also parts of its available tooling support. Several languages like Lisp [McC60] or Ruby [FM08] support the creation of such internal DSLs. In contrast to this, an *external DSL* uses a separate custom syntax that is not directly derived from an existing host language. Consequently, with an external DSL it is usually not possible to resort to existing tools, so that, e.g., a specific parser for the language has to be implemented.

Formal Model:

The box labeled *formal model* in Fig. 2.2 represents a program or model written in a particular DSL. Consequently,

- it describes something from the domain for which the DSL is tailored,
- it is an instance of the metamodel contained in the DSL and in particular respects the metamodel's static semantics,
- it is written using the concrete syntax of the DSL, and
- its meaning is given by the DSL's semantics.

Due to its "formal" nature, such a model is a suitable basis for activities like verification, interpretation or code generation. For the sake of simplicity, this book uses the notion "model" in place of "formal model", implicitly including textual as well as graphical incarnations.

Metamodeling and Metalevels:

The notions and concepts depicted in Fig. 2.2 can be applied to arbitrary *metalevels*. For instance, considering "modeling" itself as a possible domain, one could create a "meta-DSL" for describing DSLs. Accordingly, when using the meta-DSL to specify a particular DSL *myDSL*, this new DSL is an instance of (i.e., a formal model conforming to) the metamodel given by the meta-DSL, or in other words: The meta-DSL provides the metamodel of *myDSL*. Continuing the example, *myDSL* can now in turn be used to create a particular model *M*, i.e., following the same argumentation as above, *myDSL* provides the metamodel of *M*. However, given the fact that *myDSL* itself is formally described by means of the meta-DSL, the meta-DSL provides the *metametamodel* of *M*. Thus the role of the meta-DSL is determined relative to the metalevel from which it is observed. Accordingly, the "metaness" of a model arises from its relations to other models (being its instances) rather than being an intrinsic model property [Sta+07, p. 63].

A well-known example of a metamodeling architecture which employs metalevels is the Model Driven Architecture (MDA) [Obj03b] (cf. Sect. 2.3.3) proposed by the OMG. MDA enables model-driven software development on the technological basis of standards that are also created by the OMG, such as the Meta-Object Facility (MOF) [Obj11d] and UML. Fig. 2.3 is a slightly extended version of an illustration from [Obj10a, p. 19], showing an example of metalevels in MDA. The single metalevels are typically labeled M0, M1, M2 and so on, with M0 designating the lowest level. M0 usually represents the actual system (existing or non-existing) that is to be modeled, or more precisely its runtime objects and user data. The models that represent this system are situated on level M1, e.g., concrete diagrams (class diagrams etc.)

modeled in UML. Level M2 holds the modeling language that is used for describing the models on M1, i.e., their metamodel. For instance, in the MDA context, this might be the UML along with its associated concepts. Finally, the metamodel on M2 is again formally described by a model which is situated on level M3, the metametamodel. In MDA, this role is played by MOF, and thus in order to be MDA-compliant, a modeling language has to be an instance of (i.e., it has to conform to) MOF. Please note that only levels M1–M3 (and maybe above) are actual modeling levels, as M0 represents the "real" system (which is why, e.g., Bézivin refers to the four-level example as a "3+1 architecture" [Béz05]).

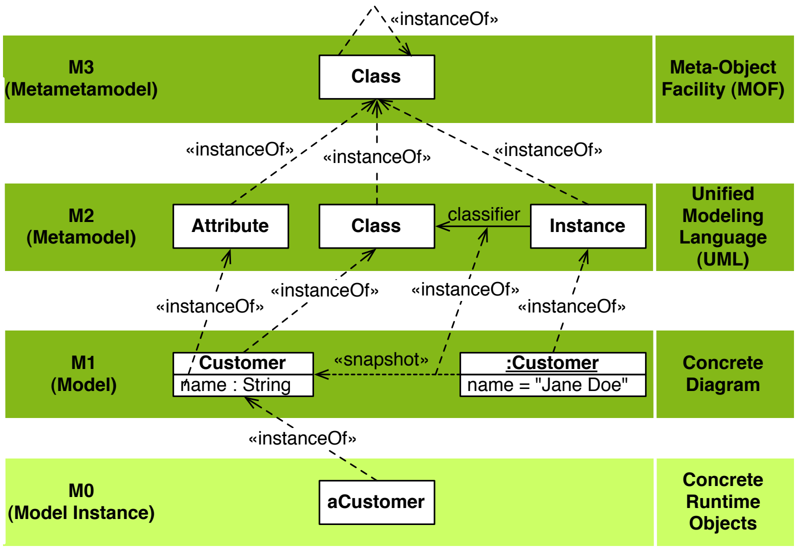


Fig. 2.3. Four-Level Example of MDA's Metamodel Hierarchy (based on [Obj10a, p. 19])

Except for the topmost metalevel, the elements of each level are instances of elements in the level above. Conceptually, there is no need for such a "hierarchy top" at all – the number of metalevels can be arbitrary [Obj10a, p. 19]. However, in practice, this potentially indefinite layering is usually avoided by means of a *reflexive* model, i.e., a model that is able to describe itself [Sei03; Sel09]. As indicated in Fig. 2.3, MOF is such a model that is defined in terms of itself, so that effectively no more metalevels are required. Another example of a reflexive model is *Ecore* from EMF (see Chap. 7).

It should be noted that the one-dimensional view on metalevels shown in Fig. 2.3 is subject to controversy. For instance, Atkinson and Kühne [AK02] pointed out that it fails to distinguish different types of "instance of" relationships and thus proposed a two-dimensional framework. However, a detailed discussion of those issues goes beyond the scope of this book.

Users of MD* tools usually only deal with a restricted view on the available metalevels. For instance, in typical UML tools like ArgoUML [Tig11], any modeling activity happens exclusively on level M1, i.e., the levels M2 and M3 are "hard-wired". Other tools such as language workbenches (see Sect. 2.3.5) also allow the user to define his own modeling languages and thus hard-wire only level M3 or above.

2.3 The Role of Code Generation

As pointed out in Chap. 1, code generation is key to any MD* approach to software development. It bridges the gap that arises when models are used to abstract from the technical details of a concrete software system. Code generation is thus an enabling factor for allowing real *model-driven* software development which treats models as primary development artifacts [Sei03; Béz05], as opposed to the approach termed *model-based* software development in Chap. 1 that is limited to using models for documentation purposes [Sta+07, p. 3].

Apart from the notion *MD**, which is used in this book (following Völter [Völ09]) as a generic term for referring to the variety of existing approaches to model-driven development, there are several further notions that are used in a similar way. Examples that can be frequently found in publications are Model-Driven Development (MDD) [Sel03; AK03], Model-Driven Engineering (MDE) [Sch06; Béz05; Fav04; DVW07] and Model-Driven Software Development (MDSD) [Sta+07], which are largely used synonymously. Among MD* approaches, code generation is usually considered a specific form of model transformation and thus often referred to as *model-to-text transformation* [CH06; Old+05] or *model-to-code transformation* [Sel03; Sta+07; Hem+10].

The following sections (2.3.1–2.3.5) provide examples of existing MD* and related approaches, with a particular focus on the respective role of code generation. Afterwards, Sect. 2.3.6 briefly sketches MD* approaches that do not resort to code generation.

2.3.1 Computer-Aided Software Engineering

The idea of automatically generating an implementation from high-level specifications is not really new. For instance, in the 1980s, the *Computer-Aided Software Engineering* (CASE) approach [CNW89] had very similar objectives, including the design of software systems by means of graphical general-purpose languages and the use of code generators for automatically producing suitable implementations [Sch06].

However, the CASE approach has not asserted itself in practice. As one reason for this, Schmidt [Sch06] especially designates the deficient translation of

CASE's graphical general-purpose languages to code for desired target platforms. The creation of corresponding code generators was very difficult as the produced code had to compensate the lack of important features, such as fault tolerance or security, in operating systems at that time. As a result, the code generators were very complex and thus hard to maintain. Moreover, CASE tools focused on proprietary execution environments, which resulted in low reusability and integrability of the generated code. Schmidt also names further problems of CASE, such as the lack of support for collaborative development and the fact that the employed graphical languages were too generic and too static to be applicable in a large variety of domains. Especially as a result of the insufficient code generation facilities, CASE tools were often used for model-based software development only [Sch06].

Today's MD* approaches benefit from the fact that programming languages and platforms significantly evolved since that time. Apart from the fact that code generation technologies have matured [Sel03], code generation has become much more feasible, as generators "can synthesize artifacts that map onto higher-level, often standardized, middleware platform APIs and frameworks, rather than lower-level OS APIs" [Sch06], which decreases their complexity significantly.

Moreover, as another lesson learned from CASE, lots of MD* approaches advocate the use of DSLs rather than general-purpose languages, thus turning away from CASE's "one size fits all" idea [Sta+07, p. 44]. The focus on DSLs further increases the significance of code generation, as the specification of a DSL often entails the demand for a corresponding code generator – or multiple ones if several target platforms are used –, the creation of which also needs to be supported by appropriate frameworks and tools.

2.3.2 Generative Programming

Generative Programming (GP) [CE00], also known as *Generative Software Development*, is an approach that "aims at modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages" [Cza04].

Accordingly, it puts particular emphasis on two main aspects. First, GP focuses on developing families of systems instead of only single systems. A system family is a set of systems based on a common set of assets [CE00, p. 31], which are used for building the single family members. Among other things, such a system family might form the basis for creating product lines [Sta+07, p. 35]. Second, GP involves the automatic assembly of the final system via generators. Inspired by industrial manufacturing, the generated system should resemble a complete, "highly customized and optimized intermediate or end-product" [CE00, p. 5].

The common model that is used for generating the single members of a system family is called the *generative domain model*. This model essentially describes three components: the problem space, the solution space as well as a mapping between both. The *problem space* can be considered the domain, and it contains one or more domain-specific languages that provide the concepts and terminology for specifying system family members. For instance, *feature models* [CHE04] are frequently used in connection with GP as a means for describing the common features of system family members along with those features that are variable. Feature models also capture how variable features depend on each other. The *solution space* consists of elementary implementation components which are used to assemble a system. The mapping between problem space and solution space is given by *configuration knowledge*, which includes illegal combinations of features, default settings and dependencies as well as construction rules and combinations [CE00, p. 6]. This configuration knowledge is implemented by means of one or more generators.

Based on this generative domain model, a system is essentially specified via configuration: An application programmer creates such a configuration by selecting desired features in the problem space, and the generator uses the configuration knowledge for automatically mapping it to a configuration of implementation components in the solution space. Besides this *configuration view* [Cza04] further describes a *transformational view* on the generative domain model. In this view, the problem space is resembled by a domain-specific language which is transformed into an implementation language situated in the solution space. Independent of the particular view, GP does not dictate which technologies are used for actually implementing the single elements of the generative domain model [CØV02].

GP is strongly related to MD* approaches as both advocate the use of DSLs for creating high-level specifications along with corresponding generators that automatically produce a system from those specifications. However, GP's strong focus on the development of software system families distinguishes it from several MD* approaches such as MDA (see the following section). Whereas MDA mainly addresses technical variability by aiming at portability, GP also takes application domain variability into account [Cza04]. Furthermore, Stahl et al. [Sta+07, p. 39] point out that GP traditionally focuses more on textual DSLs rather than on graphical notations.

In particular, lots of research in the realm of software product line engineering [CN01; PBL05] relates to GP's mindset. A recent example is the HATS project [Cla+11], which employs Abstract Behavioral Specification (ABS) in order to model system families. To this end, ABS consists of five textual languages for specifying

1. core modules of the system in a behavioral fashion,
2. the system's features and their attributes via feature modeling,
3. variability of the system by means of delta modeling [Sch+10],
4. product line configurations that link features with delta modules, and
5. concrete product selections.

From the GP perspective, those specifications provide the required concepts in the problem space as well as the configuration knowledge required for the mapping into the solution space. Finally, a concrete product is generated via a dedicated compiler, which, for instance, is able to translate an ABS model into Java code.

2.3.3 Model Driven Architecture

As mentioned in Sect. 2.2, *Model Driven Architecture* (MDA) [Obj03b] is an initiative of the OMG. It has been introduced in 2001 and primarily aims at "portability, interoperability and reusability through architectural separation of concerns" [Obj03b, p. 12]. Conceptually, MDA defines three models that represent different viewpoints on a system:

- *Computation-Independent Model* (CIM): Also termed "domain model" or "business model" [Fra02, p. 192], the CIM describes the pure business functionality including the requirements of and rules for the system. Any technical aspects of the system are ignored. CIMs are supposed to be created and used by business experts (or "domain practitioners" [Obj03b, p. 15]) and thus use familiar terminology of the respective domain. They are intended as a bridge between business experts who are versed with a particular domain, and IT experts who have the technical knowledge for realizing a system. CIMs provide a very broad view as they also may contain aspects of a domain that are not automated at all [Fra02, p. 194].
- *Platform-Independent Model* (PIM): In contrast to CIMs, PIMs also consider technical aspects of a system, but only those which are independent of a concrete platform. This platform-independence is key to achieving the goal of portability, however it should be noted that it is a relative notion. Frankel [Fra02, p. 48f] exemplifies this by means of OMG's middleware standard, the Common Object Request Broker Architecture (CORBA) [Obj11b], which can be considered platform-independent as it does not depend on particular programming languages or operating systems. However, when viewing CORBA as one among many existing middleware technologies, it also can be considered a specific platform. From this perspective, platform-independence is only achieved by not depending on a concrete middleware technology. Accordingly, a PIM "exhibits a specified degree of platform-independence so as to be suitable for use with a number of different platforms of similar type" [Obj03b, p. 16].
- *Platform-Specific Model* (PSM): A PSM augments a PIM by further technical details that are specific to a particular platform. Please note that the above comments on the relativity of platform-independence can be similarly applied to platform-specificity.

Further OMG standards provide the technological basis for creating such models: Any modeling language that conforms to MOF (see Sect. 2.2) can be used, such as UML or the Common Warehouse Metamodel (CWM) [Obj03a].

PIMs, PSMs and the actual implementation code of the system are connected by means of *transformations*. For instance, a PIM could be successively refined by one or several consecutive model transformations producing either further PIMs or PSMs, the last of which being the most concrete or specific model that is used as the basis of a final code generation step. However, the creation of intermediate models is not mandatory, as it might also be possible (e.g., depending on the abstractness of the employed PIM) to produce code directly from a PIM [Obj03b, p. 25]. The exact nature of the transformation is not dictated by MDA: A transformation may, e.g., be entirely manual, semi-automatic by marking the models with additional information, or fully automatic [Obj03b, pp. 34–36].

Model transformations (PIM to PIM, PIM to PSM, PSM to PSM) can, e.g., be realized by using any implementation of OMG's Query/View/Transformation (QVT) [Obj11c] specification. Another example for a language that supports such model transformations is the Atlas Transformation Language (ATL) [JK06]. Both QVT and ATL are, e.g., implemented in the context of the Model 2 Model (M2M) project which is part of the Eclipse Modeling Project (EMP) [Gro09].

For code generation (PIM to code, PSM to code), there exists a plethora of tools and frameworks such as AndroMDA (which has been used for a case study in the context of this monograph and thus will be described in more detail in Sect 8.1), MOFScript [Old+05], Fujaba [GSR05] or XCoder [Car11]. Moreover, there are implementations of OMG's MOF Model to Text Transformation Language (MOFM2T) [Obj08] like Acceleo [Obe11], and integrated code generation facilities in tools that support UML modeling, such as Altova UModel [Alt11] and Together [Bor11].

Although the MDA has gained lots of attention and is, in the author's assessment, perhaps the most widely known MD* approach, some of its related standards are subject to criticism. For instance, Sect. 2.2 already pointed out that the one-dimensional metamodeling architecture specified by MOF was controversial – however, the situation improved significantly with the introduction of UML 2.0 and MOF 2.0 (though still some issues remain [AK03]).

Maybe the most contentious part of MDA is UML. A major point of criticism is its lack of a clearly and formally described semantics [Tho04; BC11]. Furthermore, Kelly and Tolvanen point out the low abstraction provided by UML models, which "are at substantially the same level of abstraction as the programming languages supported" [KT08, p. 19f], because "the modeling constructs originate from the code constructs" [KT08, p. 14] instead of deriving them from the domain of the modeled system. Another problem arises from the practical difficulty of synchronizing the various UML models that describe different aspects of a system: When changes to a model are not propagated to dependent models, this may lead to inconsistencies that hamper the system's evolution [Hör+08]. In particular, this issue also concerns *round-tripping*, i.e., the synchronization of UML models and the code generated from them – Sect. 2.4.4 further elaborates on this.

2.3.4 Domain-Specific Modeling

Domain-Specific Modeling (DSM) [KT08] explicitly focuses on the creation of solutions that are entirely tailored to a particular domain. According to Kelly and Tolvanen, DSM typically includes three components: a domain-specific modeling language, a domain-specific code generator and a domain framework [KT08, p. xiii f]. Once those components are in place, developers use the domain-specific modeling language for creating models which are automatically translated into code. The use of the term "domain-specific *modeling* language" (instead of just DSL) can be considered to reflect a tendency of DSM towards visual notations "such as graphical diagrams, matrices and tables" [KT08, p. 50], that are used along with text (i.e., hybrid concrete syntaxes as described in Sect. 2.2). Furthermore, DSM clearly aims at *full code generation* (cf. Sect. 2.4.4), so that the generated code is complete and does not have to be touched [KT08, p. 49f]. In order to reduce the complexity of code generators, the produced code often is executed on top of a dedicated domain framework. Such a domain framework provides elementary implementations that do not have to be generated and thus relieve and simplify the code generator.

Kelly and Tolvanen point out that full code generation is achievable, because the language and the generator employed in DSM "need [to] fit the requirements of only one company and domain" [KT08, p. 3], thus strictly following the tenet that "Customized [sic] solutions fit better than generic ones" [KT08, p. xiv]. As a consequence of this orientation, DSM typically does not involve shipping of ready-made DSLs or code generators, because both are developed in-house as a part of implementing a DSM solution for a particular domain. In [TK09], Tolvanen and Kelly state that based on their industry experiences, this implementation phase is usually very short, with the time required for implementing the generator often outweighing the time for realizing the language.

In order to enable this modus operandi, proper tooling is required that supports both the definition and the usage of a DSM environment for creating a particular domain-specific solution. Consequently, tools for DSM usually have a hard-wired metametamodel (i.e., level M3, see Sect. 2.2), thus allowing the definition of new metamodels, ergo new domain-specific modeling languages. In this respect, DSM tools contrast with CASE or UML tools [KT08, p. 60], which usually dictate the use of a particular modeling language.

Perhaps the most prominent DSM tool is MetaEdit+ [TK09; KLR96]. As further tools that can be considered realizations of the approach, Kelly and Tolvanen [KT08, p. 390–396] mention the Generic Modeling Environment (GME) [Led+01] (originally developed in the context of Model-Integrated Computing [SK97]), Microsoft's DSL Tools [Coo+07] (a part of the Software Factories [Gre+04] initiative) and the EMF-based Graphical Modeling Framework (GMF) [Ecl11a; Gro09].

2.3.5 Language Workbenches

In 2005, Martin Fowler coined the term *language workbench* [Fow05] for referring to a class of tools that specifically focus on DSLs. This is not restricted to providing an IDE for creating a DSL (e.g., features for creating a metamodel or generating a parser): Language workbenches also support building a specialized IDE that is equipped with, e.g., custom editors and views for using the created DSL. Consequently, similar to tools for DSM mentioned in Sect. 2.3.4, language workbenches significantly differ from CASE and UML tools, which usually are based on a fixed metamodel [KT08, p. 60]. Altogether, a language workbench enables the definition of a DSL environment by specifying the metamodel, an editing environment and the semantics of the DSL ([Fow10, p. 130], adapted to the terminology introduced in Sect. 2.2).

For the custom editing environment, language workbenches usually employ either source editing or projectional editing [Fow10, p. 136]. *Source editing* uses one single representation for editing and for storing, which is usually text. The creation of such text does not depend on a particular tool but can be performed with any text editor. In contrast to this, with *projectional editing* the primary representation of a program or model is specified and tightly coupled with the employed tool. The tool provides the user with an editable projection of this representation, which might follow any concrete syntax (textual or graphical). Editing the projection then directly modifies the primary representation. In consequence, in this scenario, the user never works directly with the primary representation, and the tool is imperatively required for editing, as it has to perform the projection.

Projectional editing provides several advantages over direct source editing, such as the possibility to provide multiple (e.g., user-specific) projected representations. Graphical modeling tools naturally employ projectional editing, as the actual model is usually kept separate from its graphical representation. Thus the differentiation makes most sense for textual DSLs. Language workbenches that are based on projectional editing are also termed *projectional language workbenches* (see, e.g., [VV10]).

Code generation plays a central role for most language workbenches as it is frequently used for providing the semantics of a created DSL. According to Fowler, the semantics of the DSL is most commonly specified in a translational way (cf. Sect. 2.2), i.e., by means of code generation, and more rarely on the basis of interpretation [Fow10, p. 130]. Consequently, most workbenches provide means for specifying code generators, some of which will be exemplified in Sect. 2.4.

The rationale behind language workbenches is often associated with language-oriented programming (see, e.g., [Fow05; **?**]). The term has been coined by Ward [War94] in 1994 and refers to the general approach of solving a problem with one or more domain-specific languages rather than with general-purpose languages.

Many existing tools meet the characteristics of language workbenches described above. For instance, MetaEdit+ (presented in Sect. 2.3.4) can be considered a language workbench which supports the creation of graphical (or visual) DSLs along with projectional editing. Other language workbenches mainly focus on textual DSLs, providing either projectional editing like the Meta Programming System (MPS) [Jet11] or parser-based source editing like Xtext [Ecl11h], Spoofax [KV10] or Rascal [KSV09].

2.3.6 Approaches without Code Generation

For the sake of completeness, it should be noted that code generation is not the only way to obtain a running system from a model. Another common solution is the use of an interpreter which directly executes a model without previous translation.

Business Process Modeling (BPM) is an example of a field which predominantly employs model execution. Such models are usually business processes that are described by means of dedicated languages, and that are typically executed (i.e., interpreted) by a process engine. Examples are Business Model & Notation (BPMN) [Obj11a] with corresponding process engines like jBPM [Red11b] or Activiti [Act11b], and the Business Process Execution Language (BPEL) [OAS07] which can be executed by engines such as ActiveVOS [Act11a] or Apache ODE [Apa11c]. Typically, process engines provide features like scalability, long-running transactions (e.g., via persistency of process instances), support for human interactions and monitoring of running processes.

A major feature of interpreters is late binding. In BPM this is used, among other things, for running multiple versions of a process. It also allows, e.g., the realization of multi-tenancy capabilities, or of process adaptations at runtime. The latter is also a major goal of the "models@run.time" approach [BBF09] which aims at exploiting the advantages of models not just for software development, but also in the running system. For instance, models can be useful at runtime for realizing (self-)adaptive software systems.

Furthermore, an interpreter may play the role of a reference implementation that specifies the semantics of a DSL, as an alternative to describing the semantics in a formal way (cf. Sect. 2.2). Kleppe [Kle08, p. 135] refers to this as *pragmatic semantics*.

The choice between code generation and interpretation is not exclusive, as both approaches can be combined. For instance, the execution of generated Java code can be considered such a combination, as the Java Virtual Machine (JVM) [LY99] can be regarded an interpreter for bytecode. This book will show several further combinations of code generation and interpretation, such as interpreter-based bootstrapping of a code generator (Sect. 5.1) and the use of an interpreter via API in order to realize the execution of generated code (Sect. 5.1.1).

2.4 Code Generation Techniques

Similar to a compiler, a code generator can be characterized as a "T-shape" in a T-diagram (cf. Sect. 2.1): It supports a particular source language, translates to a desired target language and is implemented using a specific implementation language. Each of these three facets may be based on a different language. While the source and the target language are usually given by initial requirements, the implementation language has to be selected advisedly. For instance, it may be advantageous to use the same language as source and implementation language in order to enable bootstrapping (cf. Sect. 2.1).

Apart from the selection of an appropriate implementation language, there are also several approaches for the actual implementation of a code generator. Generally, each approach covers two aspects of the code generator. First, the *output description* specifies the structure and the appearance of the generated code. Second, the *generation logic* describes the logic of the code generator, i.e., how the mapping from the source language to the target language is actually performed. This may also include further actions such as pretty-printing, assembling code fragments or writing the code to corresponding files.

In the literature, different classifications are used for categorizing the existing approaches to code generation. For instance, Kleppe [Kle08, pp. 151–156] makes the following interrelated distinctions:

1. *Model transformation rules versus hard-coded transformation:* In the first case, the code generator is described by means of a set of transformation rules. These rules are processed by a corresponding tool which performs the actual translation from source to target language, and which thus realizes a large part of the generation logic via a generic transformation engine. In the second case, the transformation is implemented explicitly, e.g., using an imperative language.
2. *Source-driven versus target-driven transformation:* With source-driven transformation, the structure of the input model in the source language drives the code generation: The generator processes the input model and produces corresponding code in the target language for each model element. For instance, this might result in a set of code fragments that are assembled in a final step. If the translation is target-driven, the code generator is oriented towards the structure of the desired output. In such an approach, the code is, e.g., generated sequentially into some kind of stream, and each time any information from the input model is required, the model is specifically queried for it.
3. *Concrete form versus abstract form target:* A code generator may either translate into the concrete syntax of the target language or into a representation of its abstract syntax. Accordingly, in the latter case, the result is again a model resembling an abstract form of the code (see Sect. 2.4.3 for more details on this).

Czarnecki and Helsen [CH06] employ a much more coarse-grained and technical categorization as they only distinguish *visitor-based* and *template-based* approaches. The former use a form of the well-known visitor design pattern [Gam+95, pp. 331ff] for realizing the traversal of the input model and for mapping elements of the source language to elements of the target language (see also [Kle08, pp. 158f]). The approaches associated with the second category describe the code generation by means of templates, a combination of static text (i.e., the output description) and dynamic portions (which realize parts of the generation logic). In order to produce the actual code, a template engine evaluates the dynamic portions on the basis of the input model (see Sect. 2.4.2 for more details).

Fowler [Fow10, p. 124] also introduces two categories, called *transformer generation* and *templated generation.* Basically, templated generation equals Czarnecki and Helsen's category of templated-based approaches. With transformer generation, Fowler refers to any approach that processes the input model and emits code in the target language for each model element.

The following sections describe different techniques for realizing code generators and, where applicable and useful, assign them to the different categories outlined above. Finally, Sect. 2.4.4 elaborates on different types of outputs that can be produced with code generation.

2.4.1 Programming the Code Generator

The most minimalistic way to implement a code generator is to write it using a general-purpose programming language. As in this case the transformation from source language to target language is explicitly implemented, the resulting code generators belong to Kleppe's "hard-coded transformation" category. In the sense of Fowler's classification, those generators are an application of transformer generation.

Implementing a code generator this way only requires an API for accessing the models programmatically. The actual output is typically assembled by means of basic string concatenation. Accordingly, output description and generator logic are usually mixed up in such implementations. Moreover, depending on the selected programming language, the required handling of strings may increase the complexity of the implementation: If, e.g., Java is selected as the implementation language, special characters (such as quotation marks) have to be escaped and explicit operators (e.g., +) have to be employed for the concatenation of strings [Sta+07, pp. 150f].

In parts, this complexity can be hidden by means of dedicated code generation APIs. As described by Völter [V03], such an API is designed to resemble the abstract concepts of the target language. For instance, if Java is the target language, a corresponding code generation API would provide concepts like classes, methods, modifiers etc. as manipulable objects. After manipulation, each of those objects would be able to produce its own code in the target language. Consequently, the generator developer only has to deal with the API, which relieves him of tedious tasks such as low-level string concatenation.

Additionally, the visitor pattern (see above) can be applied for realizing the mapping of the API objects to corresponding code non-invasively and at a central place. Czarnecki and Helsen [CH06] mention the code generator framework Jamda [Boo03] as an example of an API- and visitor-based approach.

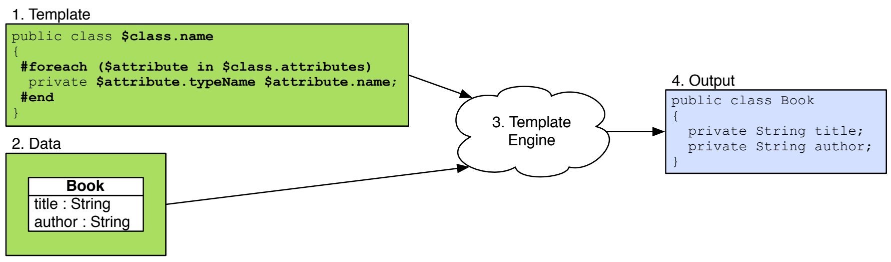
Code generators which are implemented "per pedes" based on a general-purpose programming language and APIs are usually sufficient for small application scenarios, which do not require generating a large amount of complex code. However, for larger scenarios such code generators usually do not scale well as in this case they are much harder to write [V03] and to maintain. Furthermore, Kelly and Tolvanen [KT08, p. 271] point out that many general-purpose languages do not provide convenient support for the navigation of complex models and the production of text at once.

A possible solution to the latter problem is the selection of a programming language which provides facilities that are specifically designed to support the implementation of code generators. An example of such a language is Xtend 2 [Ecl11g] which is used in recent versions of Xtext (version 2 at the time of writing this text). As another solution, Kelly and Tolvanen propose the use of a dedicated DSL, which allows a more concise description of a code generator than a general-purpose language. Furthermore, a DSL enables the specification of the code generator on a higher level of abstraction, thus hiding low-level issues. An example of such a DSL is MERL [KT08, p. 273] which is used for creating code generators in MetaEdit+. As a disadvantage of this solution, it is not possible to resort to existing tool support, which is typically readily available for general-purpose languages. Consequently, if the DSL is not an internal DSL, the implementation of specific tools for, e.g., executing and debugging the code generator may be required. For MERL, MetaEdit+ provides corresponding tools [TK09].

2.4.2 Template-Based Code Generation

This technique is based on the use of *templates*. Similar to a form letter [Sta+07, p. 146], a template consists of static text with embedded dynamic portions that are evaluated by a *template engine*. This approach is especially common in web development, where it is used by techniques such as Active Server Pages .NET (ASP.NET) [Mic11] or JavaServer Pages (JSP) [Jav09b] for dynamic server-side generation of web site contents.

Fig. 2.4 shows an example of a template and the general modus operandi of the approach. It is visible that apart from the actual template, a template engine also requires concrete data as an input. In order to generate the actual output, the dynamic portions of the template are evaluated on the basis of this data and replaced by corresponding static text.



```
public class $class.name
{
 #foreach ($attribute in $class.attributes)
  private $attribute.typeName $attribute.name;
 #end
}
```

```
public class Book
{
  private String title;
  private String author;
}
```

Fig. 2.4. Using a template engine for code generation

The template depicted in Fig. 2.4 describes the translation of a class noted as a UML class diagram into corresponding Java code. The dynamic portions of the template (visualized in bold face) are written in a *template language*. Such languages typically use dedicated control characters (in the example `$` and `#`) for distinguishing static from dynamic contents. In the example, it is visible that the template accesses the elements of the class diagram via the diagram's abstract syntax (defined in the corresponding metamodel). For instance, a class contained in the diagram is referenced by means of the expression `$class`, and also the properties of the class are accessed via suitable expressions such as `$class.name` or `$class.attributes`. Moreover, apart from such facilities for data access, most template languages support the use of control flow statements like conditionals, loops as well as method- or macro-calls. The example in Fig. 2.4 shows a `foreach` loop which iterates over all attributes of a class. For each attribute, the template describes the generation of a private member variable in the resulting Java class.

There is a large number of ready-made template engines which can be used for implementing a template-based code generator, such as StringTemplate [Par04], Velocity [Apa10], FreeMarker [Fre11b], Xpand [Ecl11f] and JET [Ecl11d]. Usually each template engine defines its own template language. For some template engines there is also sophisticated IDE support. For instance, Xpand is supported by an Eclipse-based editor that provides features such as syntax highlighting and code completion.

Template-based code generators are very common [KT08, p. 272], which can also be witnessed by the fact that Fowler as well as Czarnecki and Helsen consider them a category of their own. Examples of tools which employ template-based code generation are ANTLR (StringTemplate), EMF (JET), AndroMDA (Velocity, FreeMarker), Fujaba (Velocity), Acceleo (own template language) and former versions of Xtext (Xpand).

Similar to code generator implementation by means of a programming language (as described in Sect. 2.4.1), templates mix generation logic and output description. However, with a template-based approach, the generator developer is not confronted with issues such as escaping and string concatenation. Especially the latter is specified implicitly in the template and performed automatically and transparently by the template engine. As the structure of a template follows the structure of the output, the transformation is, in Kleppe's terminology, target-driven. Furthermore, template-based approaches belong to the category of hard-coded transformations [Kle08, p. 151].

Kelly and Tolvanen [KT08, p. 273] point out that working with templates can be inefficient if the generated output is distributed among multiple files (or locations). As a template usually resembles one file, a separate template is required for each output file and all templates have to be evaluated sequentially in order to produce the entire set of resulting files. This may lead to unnecessarily frequent traversals of the input model, even if information that is relevant for multiple files is located at the same place in the model.

2.4.3 Rule-Based Transformation

As mentioned above in Kleppe's categories, an alternative to hard-coding the transformation performed by a code generator is the use of *transformation rules*. In this approach, a set of such rules describes how each element in the source language is translated to a corresponding element in the target language. For the actual transformation, a *transformation engine* processes those rules and applies them to the input model given in the source language.

A code generator can be realized as a chain of such transformations. For instance, according to the MDA approach (cf. Sect. 2.3.3), such a chain is a sequence of model-to-model transformations on several intermediate representations, eventually ending with a final model-to-text transformation. Essentially, this idea is based on the classical "divide-and-conquer" paradigm: A complex transformation is handled by dividing it into smaller, simpler and thus more manageable steps.

Furthermore, approaches using chains of rule-based transformations often aim at an abstract form of the target language rather than at its concrete syntax (see Kleppe's "abstract form target" category). Instead of directly translating the original input model or any of the intermediate representations along the transformation chain to the concrete syntax of the target language, a structured representation (i.e., a model) of the target language is produced. The actual code is then produced by means of a final abstract-form-to-concrete-form transformation within the target language [Kle08, p. 155]. As the major advantage of targeting an abstract form, the abstract representation of the code is still available after the code generation. Thus it can be used for further processing steps and transformations, e.g., for extending the target language with additional constructs [Hem+10].

An example of rule-based transformations is described by Hemel et al. in [Hem+10]. They use Stratego/XT [Bra+08] (also employed by the language workbench Spoofax) for specifying the code generation via rewrite rules in combination with strategies for applying those rules. Another example is the language workbench MPS, which also allows rule-based transformation with abstract form target.

2.4.4 Round-Trip Engineering versus Full Code Generation

With regard to their results, code generators can be distinguished by means of two further categories: those which produce complete code and those which only generate stubs or skeletons that have to be completed by a developer.

Round-Trip Engineering:

Due to the fact that in the latter case models and code are both editable development artifacts, it is required to keep them mutually consistent. Performing this by hand is error-prone and increases the workload, because the same information has to be maintained at multiple locations. Consequently, a technique called *round-trip engineering* (RTE) [HLR08;SMW10] (also called *round-tripping* [KT08, p. 5]) aims at automating the synchronization between models and code. The both directions of this synchronization are also referred to as *forward engineering* (higher level model to lower level model or code) and *reverse engineering* (lower level model or code to higher level model) [MER99]. Accordingly, code generation belongs to the forward engineering techniques.

However, RTE has several problematic aspects. For instance, the forward engineering part has to ensure that the code can be regenerated safely when the model has been modified. This task is not trivial, especially when the code also has been subject to modification: In order to protect the developer's work, such changes must not be overwritten or invalidated by the regeneration.

According to Frankel, one possible solution is *partial round-trip engineering* [Fra02, p. 233–235], which restricts the allowed code modifications to additive changes. In this scenario, it is not allowed to overwrite or delete any code that has been generated from the model. At the same time, it is forbidden to add any code that could have been generated from a corresponding description in the modeling language. Consequently, the developer and the code generator only touch code for which they are exclusively responsible. This form of RTE is partial because it is unidirectional only – it does not support iterative reverse engineering [Fra02, p. 234].

Protected regions [KT08, p. 295f; Fra02, p.234]are a means for supporting such strictly additive code changes. Those regions are specific parts of the code that are, e.g., marked with dedicated comments. As a general rule, the developer must not perform any modifications outside of the protected regions. In turn, the code generator is able to detect the protected regions and leaves them untouched in case of a regeneration. However, as this feature needs to be supported by the code generator, this inevitably increases the complexity of the generator's implementation. Further problems with protected regions include modifications to the model which lead to the invalidation of manually written code (such as renaming of classes, methods etc.) [KT08, p. 66], or developers who do not stick to the rules and perform modifications outside of the protected regions [Fra02, p. 234].

An alternative to protected regions is the use of the *generation gap* pattern [Vli98, pp. 85ff]. Based on this pattern, manually written code can be added non-invasively by means of inheritance: The “hand-made” classes simply extend the generated classes. On regeneration, the code generator can safely overwrite the superclasses, and the manually written subclasses are not affected at all. Hence the code generator is less complex than for the protected regions approach, because it only has to ensure that, e.g., suitable visibilities in the generated code support the inheritance.

Besides partial RTE, there is also *full round-trip engineering* [Fra02, pp. 235f] which allows arbitrary changes to model and code along with a bidirectional synchronization of both. However, in practice, full RTE is very hard to realize due to the fact that “transformations in general are partial and not injective” [HLR08]. As a consequence, full RTE often only works if model and code are at the same level of abstraction [Sta+07, p. 45; KT08, pp. 5f]. This contradicts the very purpose of a model, that is, to be an abstraction of the code (cf. Sect. 2.2).

MDA is a prominent example of an approach that is frequently realized on the basis of RTE. Many code generators for UML, such as AndroMDA which is presented in more detail in Sect. 8.1, mainly produce stubs and skeletons that have to be completed manually. Hence lots of UML modeling tools like Together or Altova UModel provide support for RTE. As many UML models are very close to the code in terms of abstraction, even full RTE is possible – however, as already pointed out in Sect. 2.3.3, UML is often criticized exactly for this lack of abstraction.

Full Code Generation:

An alternative approach that aims at avoiding the problems arising from stub/skeleton generation and RTE is *full code generation* [KT08, p. 49 f]. This refers to the generation of fully functional code which does not require any manual completion. More precisely, the manual modification of the generated code is explicitly forbidden: Any change to the system has to be performed at the modeling level, followed by a regeneration of the code. As the code is never edited, the code generator can overwrite it blindly (similar to the superclasses of the generation gap pattern, see above) which strongly simplifies the generation. In this scenario, the generated code is considered a by-product, analogous to the results of a compiler for a programming language [Sel03].

Please note that full code generation usually is not equivalent to generating a full application, though in some cases the generated source code may already resemble a complete application or system. Typically, the generated parts coexist with other code and software components, such as hand-written code (e.g., specialized GUIs, legacy code, a domain framework in the sense of DSM), frameworks (e.g., a web framework like Struts [Apa11d]), libraries (e.g., a template engine like StringTemplate, see Sect. 2.4.2), or an application server like JBoss [Red11a].

It largely depends on the source language whether full code generation is possible or not. The challenge is to design the language in such a way that it contains enough information for the generation of complete code, but at the same time is not forced to align its abstraction level with the code.

For instance, the latter can be observed with Executable UML [MB02; Rai+04], which aims at making UML models executable via precisely defined action semantics, using a compliant action language like the Action Specification Language (ASL) [Ken03]. Although this technique improves the results of code generation, it comes at the cost of less abstract and more technical models: Executable UML is virtually using UML itself as a programming language. [KT08, pp. 56f]. Similar arguments apply to other approaches that, e.g., try to generate the dynamic aspects from collaboration diagrams [Eng+99].

One approach for achieving full code generation is specifically tailoring the language and the code generator to each domain, as, e.g., advocated by DSM and MDSD. This book will show that another solution is the combination of model-driven development and service-orientation that is proposed by the XMDD paradigm (cf. Chap. 3).

2.5 Quality Assurance of Code Generators

Just like any other software product, code generators have to be the subject of quality assurance measures such as verification and validation (V&V). Bugs in code generators may lead to drastic problems such as uncompilable code or unexpected behavior of the generated system. This is particularly unacceptable for safety-critical systems that can be found, e.g., in the automotive or aviation industry. In consequence, it is essential that the automated translation provided by a code generator is dependable and always leads to the desired results.

In compiler construction, there has been lots of research on V&V, including compiler verification (e.g., based on techniques like theorem proving [Str02; Ler06], refinement algebras [MO97], translation validation [PSS98; Nec00], program checking [GZ99] and proof-carrying code [Nec97]) as well as compiler testing [KP05]. In particular, the "verifying compiler", i.e., one that proves the correctness of the compilation result, has been the subject of a grand challenge proposed by Tony Hoare in 2003 [Hoa03]. Moreover, compiler verification in general is still an active topic (see, e.g., the workshop on "Compiler Optimization Meets Compiler Verification", COCV; or the conference on "Verified Software: Theories, Tools and Experiments", VSTTE).

Sect. 2.1 already pointed out that existing tools from the realm of compiler construction (e.g., parser generators) can be reused for the construction of code generators in MD* approaches. Similarly, insights and techniques from compiler verification often serve as the basis of V&V for such code generators. For instance, theorem proving is used by Blech et al. [BGL05] to verify the translation of statecharts to a subset of Java, and in the Gene-Auto [Rug+08] project for verifying the generation of C code from data-flow and state models. Ryabtsev and Strichman [RS09] apply translation validation to a commercial code generator that translates Simulink [The11] models to optimized C code. Denney and Fischer [DF06] propose an evidence-based approach to the certification of generated code that is similar to the ideas of proof-carrying code.

Concerning testing, Stürmer et al. described "a general and tool-independent test architecture for code generators" [Stü+07; SC04]. Sect. 6.3 further elaborates on this testing approach, as parts of it have been realized in the context of the Genesys framework presented in this book. Beyond the publications of Stürmer et al., the author could not find any further substantial research on code generator testing.

Stürmer et al. categorize V&V of code generators as *analytical procedures* [SWC05]. Apart from this, they also identify further approaches to the quality assurance of code generators termed *constructive procedures.* Such approaches advocate the implementation of code generators along the lines of systematic development processes. According to Stürmer et al., this includes, e.g., the adoption of standards like SPICE (Software Process Improvement and Capability Determination, ISO/IEC 15504).

2.6 Classification of Genesys

This section locates Genesys on the scale of approaches and techniques presented in the previous sections. For this purpose, it focuses on highlighting the differences and similarities – for any details on the single aspects of Genesys there will be cross-references to the corresponding chapters in this book.

As pointed out in Chap. 1, the Genesys approach propagates the construction of code generators on the basis of graphical models and services. This approach is, to the knowledge of the author, unique in the realm of code generation.

Generally, the advantages of service orientation are typically not exploited for building code generators. For instance, this is also true for the field of BPM, which is traditionally closely connected to the ideas of service orientation. Furthermore, it frequently features the combined use of graphical models and services (e.g., in BPMN, cf. Sect. 2.3.6). However, those notations are typically used for higher-level business processes, and not for lower-level technical domains such as code generation.

If a program written in a DSL is considered a model (cf. Sect. 2.2), one could argue that some approaches (e.g., MERL in MetaEdit+) indeed employ modeling for realizing code generators. However, none of the code generation approaches known to the author of this book uses *graphical* models for this purpose: Textual specifications of code generators are the rule.

A reason for this might be that code generation generally seems to be attributed to a lower level of abstraction. Code generators are mainly implemented by developers who are used to textual languages and APIs – so why bother them with graphical models and services? This book argues that the use of both can be highly beneficial for the development of code generators.

The previous sections showed that existing approaches are usually restricted to the use of specific code generation techniques (e.g., templates engines in AndroMDA, rule-based transformations in Spoofax, or the language Xtend in Xtext). In contrast to this, Genesys does not dictate which techniques or tools are used for building a code generator. This is a direct consequence of service orientation: Any tool or framework can be incorporated as a service and directly used in Genesys. Modeling on the basis of the available services is not fixed to any specific procedure, and thus the generator developer is free to choose any technique and modus operandi for the code generator.

For instance, most of the Genesys code generators exemplified in this monograph (cf. Sect. 4.2 and Chap. 5) employ template engines and thus can be considered template-based. Each template engine is an available service, so that the generator developer can freely select which engine should be used. He could even mix several template engines in one single code generator.

It should be noted that in order to obtain a clean separation of generation logic and output description (*Requirement S4 - Clean Code Generator Specification*), many Genesys code generators employ template engines in a different manner than typical template-based generators. For instance, as a convention in Genesys, advanced features of template languages such as control flow statements or function calls should be avoided: Instead the corresponding logic is specified explicitly in the code generator models, so that it can, e.g., be captured by verification tools (see Sect. 4.2.5). As a result of this convention, those Genesys code generators typically use rather small templates that are distributed over the code generator, producing code fragments that need to be assembled at some point of the code generation process. This is similar to, e.g., the rule-based transformation approach described by Hemel et al. [Hem+10], which employs a similar fragmentation of the output description.

Apart from separating generation logic and output description, a further advantage arising from this different use of template engines in Genesys is the fact that code generators can be source-driven and template-based at the same time. As mentioned above in Sect. 2.4.2, code generators employing template engines are typically restricted to target-driven transformation. However, because Genesys imposes no restrictions on the order in which the code fragments have to be produced, the generation of the output can be performed in a source-driven as well as in a target-driven manner, or even with a combination of both. This flexibility also helps to overcome the typical problems of template-based code generators that occur when dealing with multiple files (cf. Sect. 2.4.2). The Documentation Generator described in

Sect. 4.2 is an example which employs templates, is both source-driven and target-driven, and deals with multiple output files.

This book also shows examples of Genesys code generators which are not template-based at all. For instance, the *FormulaBuilder* (cf. Sect. 6.2.1) employs a rule-based transformation with a concrete form target, and the *BPEL Generator* (cf. Sect. 5.4.5) performs a transformation to an abstract form target and then serializes this to code.

Furthermore, this book illustrates the flexibility arising from service orientation by integrating and using the code generation framework AndroMDA as a service (in this case even paving the way for full code generation, cf. Chap. 8). Consequently, Genesys may be considered "a code generator construction kit which allows the (re)use and combination of existing heterogeneous tools, frameworks and approaches independent of their complexity" [JS11]. In this role, Genesys does not complement, but supplement and unify existing approaches.

Additionally, Genesys is not limited to any particular source language (see Chap. 7) or representation of the source language, like the bulk of language workbenches which strongly focus on textual source languages. Likewise, there are no restrictions of supported target languages whatsoever.

The development of code generators in Genesys is characterized by the reuse of existing components, as it relies on a library of models and services (cf. Chap. 4). Accordingly, Genesys strives for a balanced approach that aims at:

1. providing fast creation of code generators via customization and reuse, in contrast to, e.g., DSM and language workbenches, which usually achieve their high domain-specificity by developing an entirely new code generator for each domain (thus repeatedly starting from scratch), and at the same time
2. being more flexibly adaptable to different domains than, e.g., CASE or UML tools with their rather fixed and inextensible code generators.

As another major difference in comparison to other approaches, Genesys provides a holistic view on code generator construction that supports all phases including the specification, execution, generation, debugging, verification and testing of a code generator[1]. While specification, execution and generation are typically supported, facilities for debugging a code generator are more rare. Among the examples listed in the previous sections, only MetaEdit+ supports this by means of a dedicated tool [TK09], and in the case of code generators implemented with a programming language, existing debuggers can be used. However, for testing and in particular for verification, most approaches do not provide integrated and dedicated solutions.

Furthermore, Genesys aims at retaining simplicity along all phases of code generator development. Following *Requirement G3 - Simplicity*, the goal is that constructing a code generator demands learning as few languages as

[1] For specification, execution, generation and debugging see Chap. 4 and 5, for verification and testing see Chap. 6

possible. In other approaches, the knowledge of multiple languages (or at least dialects of a language) are required, apart from the actual source and target language of the code generator. For instance, the language workbench Xtext has separate languages for specifying grammars, transformations and workflows of transformations [Ecl11h]. Genesys uses the same simple modeling language (cf. Sect. 3.2.2) for all artifacts required in the single phases. In consequence, artifacts like test cases, test suites (cf. Sect. 6.3) or constraints (cf. Sect. 6.2) are specified by means of the same language employed for developing the actual code generators. Aside from this, only a (freely selectable) template language might have to be learned, given the case that a template-based code generator is to be developed.

Concerning verification, Genesys is also unique in that it applies model checking for proving the correctness of code generators relative to a set of constraints. Although in particular model checking and another facility called *local checking* (i.e., checking of constraints attached locally to single services, cf. Sect. 6.1) are in the focus of this book, other verification techniques can be easily incorporated into Genesys (cf. Sect. 10).

The reference implementation of the Genesys approach presented in this monograph is conceptually and technically based on another MD* approach called XMDD and its tool incarnation jABC (cf. Chap. 3). Sect. 3.5 evaluates the feasibility of other MD* approaches and tools with regard to their aptitude for realizing the requirements of the Genesys approach (cf. Sect. 1.1), and in doing so it illustrates why XMDD and jABC are a suitable basis for reaching those goals.

Finally, the combination of XMDD, jABC and Genesys can be considered a realization of GP (cf. Sect. 2.3.2). In this combination, services resemble the elementary implementation components situated in GP's solution space, and models provide a particular configuration of services in the problem space. Those models are a suitable basis for the evolution of system families, as exemplified with a family of code generators in Chap. 5. Variability can be specified by means of the variant management features presented in this book (cf. Sect. 4.1.4 and 10). The code generators provided by Genesys in conjunction with a library of constraints (cf. Sect. 3.1) embody GP's configuration knowledge.

3
Extreme Model-Driven Development and jABC

This chapter introduces the jABC framework and the underlying *Extreme Model-Driven Development* (XMDD) paradigm, which form the technical and conceptual (respectively) basis for the reference implementation of the Genesys approach described in this book. Historically, the demand for code generation in jABC provided the initial motivation for starting the Genesys project. In the beginning, the project started off as one of many application scenarios of jABC and XMDD. However, during its evolution, the approach showed great potential for application beyond the jABC context, so that the scope of the Genesys project has broadened.

Nevertheless, XMDD and jABC remained the integral core of the Genesys framework. Accordingly, reading the following sections is highly recommended, as they introduce the notions and concepts that are required for understanding all remaining chapters of this book. Sect. 3.1 first introduces the XMDD approach, which establishes a basic mindset for model-driven and service-oriented software engineering. Afterwards, Sect. 3.2 introduces jABC as a concrete framework and toolset for software development along the lines of XMDD. Sect. 3.3 describes the execution of models in jABC, and Sect. 3.4 elaborates on jABC's support for system verification via model checking, which has been used intensively in the Genesys framework (cf. Chap. 6). Finally, Sect. 3.5 discusses the feasibility of jABC/XMDD as a basis for realizing the Genesys approach in comparison to related approaches.

3.1 Extreme Model-Driven Development

Today's software projects are facing several kinds of gaps that may lead to serious problems or even to failure. First, there are *social gaps* between people involved in developing a software product. Those gaps may be of cultural nature, e.g., for development teams scattered globally, or they result from miscommunication between project members with differing mindsets and professional backgrounds, such as customers, developers and the management.

Charette [Cha05] lists “poor communication among customers, developers, and users” as one of the most common reasons for failed projects. Cerpa and Verner add further communication-related factors such as “customer/users [sic] had unrealistic expectations” or “process did not have reviews at the end of each phase” [CV09].

Second, as described by Margaria and Steffen [MS08], *system gaps* hamper the construction of software, especially of heterogeneous large-scale systems that cross organizational boundaries. Typically, such systems are composed of components, such as libraries or entire products like enterprise resource planning (ERP) systems, that evolve independently of each other, e.g., because they are created by different manufacturers. Consequently, updates of single components are very problematic, as they may affect the correct functionality of the overall system in an unexpected – and often also unpredictable – way, thus making the problem diagnosis a very cumbersome task.

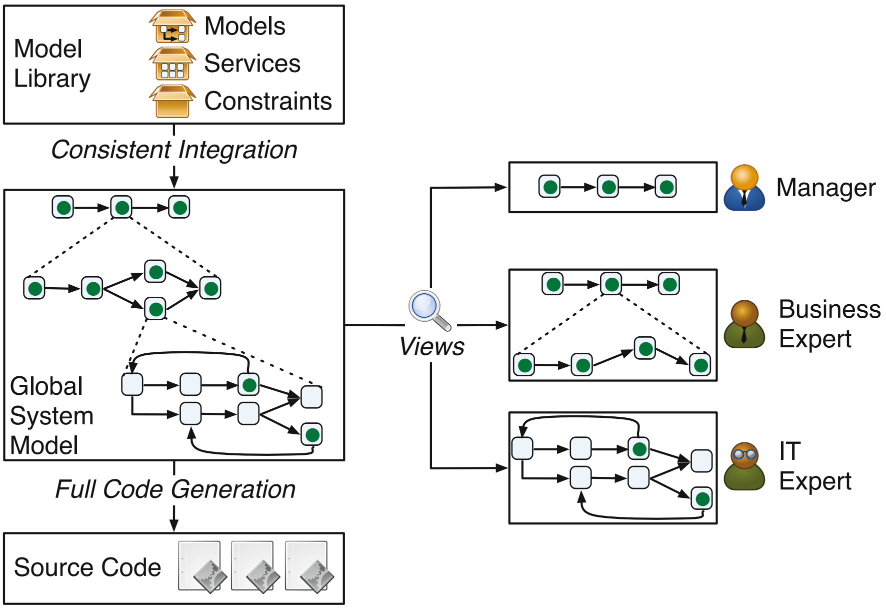


Fig. 3.1. The Extreme Model-Driven Development (XMDD) approach

The XMDD [MS08; MS09a; MS09b] approach illustrated in Fig. 3.1 has been proposed by Steffen et al. for bridging those gaps. In order to achieve this, it combines the ideas of model-driven development, service orientation, extreme programming and aspect orientation [MS09b]. By promoting models to central and primary development artifacts, XMDD exploits the fact that models typically raise the level of abstraction (cf. Chap. 1). Moreover, the approach postulates that the integration of a system is performed at the modeling level rather than at the level of software components [MS08].

As shown in Fig. 3.1, modeling is performed on the basis of a *library* of models, which are combined to one large artifact, the *global system model*, that reflects the actual system. Due to the approach's focus on this single central artifact it is also called the *One-Thing-Approach* (OTA) [SN07;MS09a].

Integrating the system at the modeling level provides the significant advantage that models are amenable to formal methods. Accordingly, the consistency of the global system model (i.e. the valid combination of its constituent models) can be ensured by means of corresponding tools that perform, e.g., model checking (see Sect. 3.4). Such mechanism help overcoming the compatibility and interoperability issues of software components outlined above.

A necessary condition for this modus operandi is the consequent restriction of any system changes to the modeling level. As visible in Fig. 3.1, the global system model is automatically translated into code for a desired target platform via full code generation (cf. Sect. 1.1). Due to the fact that this generated code is complete (i.e. no stubs or skeletons), it can be considered a by-product that does not have to be modified. Any system changes are directly performed at the modeling level by adapting the global system model, followed by a regeneration of the corresponding code.

As the code generator does not need to take care of any manual changes on the code, results from old generation runs can be overwritten safely. Accordingly, there is no need for any round-trip engineering mechanisms which may introduce additional complexity or problems that jeopardize the system's consistency (cf. Sect. 2.4.4). XMDD shares this tenet with other MD* approaches such as MDSD and DSM (cf. Sect. 2.3).

The global system model is used as a common basis for *collaboratively* designing the actual system. As a central objective of XMDD, this process is not solely performed by developers. Instead the customers and/or business experts are continuously involved in the entire project evolution, with the global system model being the shared language for bridging the social gaps described above. Again, the abstract nature of models is beneficial for achieving this, as any technical details of the final target platform are faded out. However, several additional mechanisms are required in order to provide a model that is capable of combining both the developer's technical perspective and the business expert's functional perspective of a system.

One of such mechanisms in XMDD is *hierarchical modeling* [Ste+97]. As visible in Fig. 3.1, the global system model is constructed in a hierarchical fashion, which, apart from reducing the model's complexity through modularization, enables a collaborative creation of the model by means of iterative refinement. Starting at a rather abstract level, which only specifies the global tasks or features of the system, the model is continuously refined and enriched, getting more and more concrete with each hierarchy level. Finally, the models at the lowest level consist of atomic *services*, which are elementary building blocks of models, and which represent real functionality of the system.

This functionality is either reused from existing systems or software components (e.g., commercial off-the-shelf or libraries), or it is implemented by a developer. In order to be able to integrate the represented functionality in different application domains, services are usually configurable. However, at the modeling level, the details of the service implementations are not visible, so that the "system development becomes in essence a user-centric orchestration of intuitive service functionality" [MS08]. This orchestration of "real" functionality has another important advantage: Models composed of services are immediately *executable*. Among other things, this enables rapid prototyping, debugging and tangibly trying out the modeled system, even at very early stages of development. As shown at the top left of Fig. 3.1, services are an integral part of the model library.

Analogous to the reusability of atomic services, hierarchical modeling also allows the reuse of entire models. Once created, a model can be seen as a ready-made *aspect* or *feature*, which becomes a part of the model library and thus can be reused in other application scenarios. Consequently, the model library plays the role of a steadily growing repertoire: With each new application domain or developed system, this library is enriched by new models and services, thus constantly increasing the potential of reuse for any following projects.

This steady growth even applies to *constraints*, which are, besides models and services, the third part of XMDD's model library. Constraints specify the consistency rules of the global system model and its constituent parts. Typically, there are two types of such constraints: local and global constraints. *Local constraints* concern the atomic parts of a model, i.e. the contained services. In contrast to this, *global constraints* relate to an entire model and usually ensure the consistent interplay of all contained models and services, in order to guarantee, e.g., well-formedness and executability.

Both types of constraints are verified by corresponding check tools such as model checkers (see Sect. 3.4) for global constraints. With each such constraint that is added to the library, the impact of the check tools on the integrity of the global system model as well as the automatic guidance of the user while modeling is increased, a process which is also referred to as *incremental formalization* [Ste+96; MS06].

Another mechanism for facilitating different perspectives on the global system model in XMDD is the support of *views* [SM99]. Such views are typically provided by means of transformations on the representation of the global system model. Fig. 3.1 shows some examples of different views:

- The manager's view only contains the topmost hierarchy level of the model, as he is only interested in the global features or process of the system.
- The business expert's perspective on the system is more detailed. He is able to see more hierarchy levels, but some of the models may be simplified

(e.g., by hiding error paths). Furthermore, his view does not include the lowest hierarchy levels that contain the concrete services.
- The IT expert's view contains the most concrete models at the lowest hierarchy levels.

MDA also captures different perspectives on a system with its concepts of CIM, PIM and PSM (cf. Sect. 2.3.3). However, in MDA these different models are connected by means of transformations of the *actual* model structure. The views in XMDD are *projections* of the global system model. This means that when a view is created, the global system model itself is not altered, but only its representation to the user.

By means of hierarchical modeling and views, the global system model becomes a suitable basis for the collaborative system development outlined above, as the perspectives of all involved persons are adequately supported. Apart from bridging the social gaps, the close cooperation with the customer/business experts in XMDD leads to shorter feedback cycles and a higher flexibility for dealing with changing requirements [MS09b]. XMDD shares those objectives with other agile approaches such as Extreme Programming[1] [BA04] or Scrum [SB01].

3.2 jABC

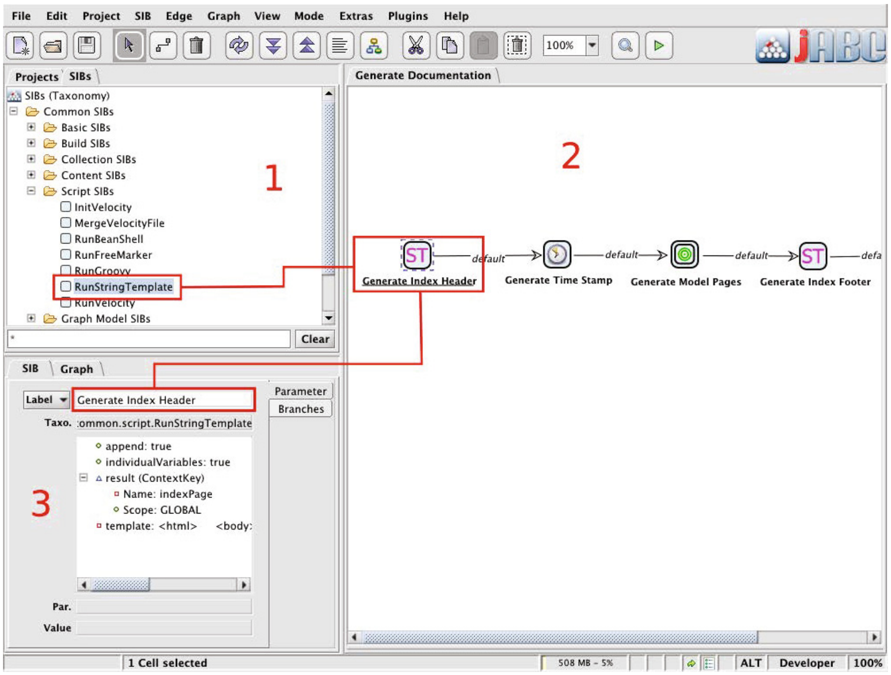
jABC [Ste+07; MS08] is a highly customizable Java-based framework that realizes the tenets of XMDD described above. Previous versions were based on C++, and the earliest precursors appeared almost two decades ago [Ste+94; SM99]. Currently, the framework is developed and maintained by the Chair of Programming Systems at the TU Dortmund.

jABC provides a tool that allows users to graphically develop systems in a behavior-oriented manner [MS06] by means of models called *Service Logic Graphs* (SLGs). As advocated by XMDD, SLGs are constructed hierarchically, and their elementary building blocks represent concrete services. Those building blocks are called *Service Independent Building Blocks* (SIBs). The jABC tool provides facilities for composing and manipulating SLGs as well as for instantiating and configuring the contained SIBs. Further functionality can be added by means of plugins.

Fig. 3.2 shows a screenshot of the tool's user interface, which consists of three main areas (corresponding to the numbers in the screenshot):

1. *Project and SIB browsers:* The project browser (not visible in the figure) provides an overview of all available projects, which are the basic organization units in jABC. Each system that is under development is reflected by a corresponding project which collects all models, services and constraints that are of interest for and employed by this system. For enabling the collaborative work on a project, its contents are usually

[1] In fact, the "X" in XMDD is a reference to Extreme Programming.



Label Generate Index Header
Taxo. :ommon.script.RunStringTemplate
- append: true
- individualVariables: true
- result (ContextKey)
  - Name: indexPage
  - Scope: GLOBAL
- template: <html> <body:

Fig. 3.2. The jABC user interface [JSM10]

managed by a revision control system such as Subversion [Apa11e]. Besides the project browser, the SIB browser enlists all services that can be used in the current project. The corresponding SIBs are organized by means of a *taxonomy*. In this context, taxonomies are graphs in which the sinks represent services and the intermediate nodes are groups or categories that subsume several services according to common properties or characteristics. Such a taxonomy is usually adapted to fit the current application domain in terms of which SIBs are visible (and thus applicable), and how they are named and organized. As visible from Fig. 3.2, jABC's SIB browser displays taxonomies as simple tree structures.

2. *Canvas:* On the canvas, the user graphically models the SLGs that represent the actual system. After selecting a service from the SIB browser (1), it can be instantiated by dragging it onto the canvas, where it is integrated into the SLG.
3. *Inspectors:* The inspectors provide detailed information on the currently displayed SLG and on its constituent parts. For instance, the *SIB inspector* allows to view and modify the current configuration of a particular SIB in the SLGs. This is visible in Fig. 3.2: The SIB inspector shows the configuration of the SIB `Generate Index Header` which has been selected in the canvas. As a further example, the *Graph inspector* allows the configuration of an entire model. Furthermore, it supports the

modification of metadata associated with an SLG, such as its name. Finally, plugins are able to add further inspectors that provide specific information and functionality.

Usually, when working with jABC in a team, several roles with different responsibilities can be identified [MS06; MS04]. First, the *application expert* or *business expert* has deep knowledge about the tasks and processes of a particular application, and he uses jABC for modeling this knowledge as SLGs. For this task, the application expert does not have to be familiar with the target platform, i.e., the technical infrastructure of the resulting system. In particular, no programming skills are required.

Second, the *domain expert* has detailed knowledge about the application *domain* including corresponding concepts and terminology. At the beginning of a new project, the domain expert customizes jABC in order to produce a variant that is tailored to the concrete domain and that optimally fits the needs of the application expert. For instance, this customization includes the adaption of the SIB taxonomy by selecting, naming and categorizing services on the basis of domain-specific terminology and characteristics. Further customization tasks are the selection of suitable plugins as well as the specification of additional domain knowledge such as supported data types or constraints [NLS11]. Finally, and most relevant in the context of this book, the domain expert selects required code generators for the translation of the SLGs for the desired target platform. If no appropriate code generator is available, he even may play the role of the *generator developer* (cf. Chap. 4) and use the Genesys framework in order to create a code generator that is tailored to the requirements of the chosen target platform. Accordingly, it may be advantageous (though not mandatory) if the domain expert is also roughly familiar with the technical characteristics of the target platform.

Third, the *IT expert* or *SIB expert* is technically versed and usually a classical software developer. Using his IDE of choice (e.g., Eclipse), he creates new SIBs on demand by integrating existing services or by implementing new ones. The SLGs created in collaboration with the application expert provide him with corresponding requirements. Accordingly, the IT expert also participates in the actual modeling, in particular in the creation of the more concrete models on the lower hierarchy levels (cf. Sect. 3.1). Furthermore, the IT expert is responsible for the software infrastructure and the runtime environment of the target platform.

The customization mentioned in the context of the domain expert is a central concept in jABC, as it provides the means for flexibly using the framework in arbitrary application scenarios. In contrast to the DSM approach described in Sect. 2.3.4, domain-specificity in jABC is not achieved by constantly creating entirely new languages, but by adapting the actual framework to the specific needs of the domain. The feasibility of this modus operandi is witnessed by a large variety of domains that have been covered in practical projects with jABC. For instance, jABC has been used for modeling telecommunication services [SM99], experiments in bioinformatics [MKS08; LMS08], large-scale web applications [KM06], remote configuration management [BM06], supply chain management processes [Hör+08], robot control programs [Jör+07], game strategies [BJM09], test cases [MS04; Raf+08], compiler optimizations [MRS06], property specifications [JMS06] and web services [Kub+09]. This book presents code generation as another application domain for jABC.

The following sections elaborate on jABC's core constituents that realize the ideas of XMDD: Sect. 3.2.1 further describes the concept of SIBs, Sect. 3.2.2 focuses on SLGs and Sect. 3.2.3 enlarges upon plugins. Parts of those sections are based on a previous publication [JSM10].

3.2.1 Service Independent Building Blocks

Service Independent Building Blocks (SIBs) are the elementary building blocks of models in jABC. The notation originates from the telecommunication realm [IIT97; IIT93]. It refers to a SIB being an abstract representation that is independent of

a) its context of usage, i.e., reusable for composing different applications or systems (which in turn can be considered services from a compositional perspective [IIT97]) spanning arbitrary domains, and of
b) the technical realization of the actual service functionality or behavior it stands for.

As already pointed out in Sect. 3.1, the granularity of the service represented by a SIB is arbitrary. It ranges from low-level functionality like string concatenation or database access to remotely available web services, or even to the interaction with more complex systems such as ERP software.

The Application Expert's View on SIBs

From the perspective of an application expert who is modeling a system, the concrete manifestation of the represented services is entirely transparent and irrelevant: In order to use a SIB, it is only necessary to know which behavior it represents, but not how the behavior is implemented. As advocated by XMDD, application experts resort to a modeling repertoire that contains ready-made libraries of SIBs, which can be used as simple black boxes.

jABC already ships with a library of such SIBs, the *Common SIBs* [TU10], that represent very general and basic services that are of interest for almost any application domain. The Common SIBs are organized as bundles according to their tasks. For instance, the "IO SIBs" represent I/O functionality such as reading text from a file or creating directories, and the "Collection SIBs" provide services that deal with different collections such as hash tables or lists.

In order to enable an intuitive usage in jABC, SIBs have a simple interface [MS04] that provides:

- a set of *parameters* which enable the configuration of the SIB's behavior,
- a set of *branches*, which reflect the possible execution results of a SIB and which are used to connect SIB instances in a model via directed edges (cf. Sect. 3.2.2),
- an *icon* and a *label* which are used for visualizing the SIB in the canvas, and
- a *documentation* that informs the user about the behavior represented by the SIB and about the purpose of its parameters and branches.

In order to illustrate this, Fig. 3.2 highlights an example of a SIB used in a model displayed in the canvas (2). From the SIB browser (1) it is visible that the SIB is an instance of `RunStringTemplate`, which is categorized as part of the Common SIBs bundle called "Script SIBs". `RunStringTemplate` integrates the template engine StringTemplate (cf. Sect. 2.4.2) as a service. Accordingly, the task of this service is the evaluation of a template. The corresponding SIB instance in the canvas is labeled `Generate Index Header`, and there is another instance of the SIB contained in the model (labeled `Generate Index Footer`), which illustrates the reusability of the building blocks. The icons and labels that are used to visualize the SIB instances in the canvas can be customized by the application expert.

By selecting a particular SIB instance in the canvas, its details are displayed by the SIB inspector (3). As visible in Fig. 3.2, the SIB `RunString-Template` provides four parameters, with one of them ("template") being the template that should be evaluated by StringTemplate. Furthermore, the SIB has two branches (not visible in the figure): *default*, reflecting the case that the template evaluation succeeded, and *error*, indicating that the template could not be evaluated (e.g., due to syntax errors).

In order to enable SIB instances in a model to communicate with each other, i.e., to share data, the concrete service implementations keep track of an *execution context* which acts as a shared memory. Technically, this context is like a hash table containing a set of key-value pairs. Thus a SIB instance is able to read and manipulate data that has been stored in the context by other SIB instances, provided that both SIB instances agree on the key which identifies the data. While the concrete implementation of the execution context is irrelevant for (and invisible to) the application expert, it is his responsibility to specify the keys used by SIB instances for accessing shared data. These keys are specified by means of special SIB parameters. For instance, the SIB instance shown in Fig. 3.2 provides a parameter "result" (3) which specifies the key used to store the evaluation result of the StringTemplate service in the execution context (in the example, this key is `indexPage`). Sect. 3.3.2 elaborates on the concept of the execution context in more detail.

The IT Expert's View on SIBs

Among other tasks (cf. Sect. 3.2), the IT expert provides the application expert with required SIBs, either by continuously extending the ready-made libraries such as the Common SIBs, or as a reaction to a direct request. Implementing a SIB basically consists of two parts: the SIB itself and its service adapters.

SIB:

The *SIB* is the (graphical) building block used by the application expert for composing models in jABC. As pointed out above, it is an abstract representation of a specific behavior or functionality, providing parameters for configuration and branches for reflecting the possible execution results. Technically, such a SIB is described by means of a very simple Java class which defines the SIB's constituents via programming conventions:

- Parameters are defined by all public fields of the Java class. All parameters have to be initialized with default values (i.e. in particular `null` values are not allowed).
- Branches are separated into *final* and *mutable branches.* Final branches cannot be modified by the application expert when configuring the SIB, whereas mutable branches can be renamed or deleted. The latter may be useful if one or more execution results emerge dynamically when the represented service is executed. The final branches of a SIB are defined by a final static String array called `BRANCHES`, and the mutable branches are specified in a non-constant String array named `branches`. Furthermore, if the `branches` array is initialized at least as an empty array, this enables the possibility for the application expert to add new mutable branches via the jABC tool.
- An icon and the SIB's documentation are specified by implementing special methods.
- Optionally, plugins may introduce further information or functionality by means of corresponding interfaces, which then can be implemented by the IT expert (see Sect. 3.2.3 and 3.3.1 for examples).

Finally, the class is marked as a SIB via an annotation (`@SIBClass`), which also declares a *unique identifier* (UID) in order to reliably distinguish the SIB from other SIBs.

The jABC framework only allows a restricted set of data types for specifying a SIB's parameters. The set is separated into *simple* and *complex types.* The simple types consist of standard Java data types such as `Boolean`, `String`, numeric types (e.g., `Integer`, `Float`), `File`, arrays and various collections (e.g., `ArrayList`, `HashMap`).

Furthermore, the framework supports several complex types which are listed in Table 3.1. The data types written in italics have been designed for very specific application scenarios and are rarely used, thus they will not be considered further in this book. The remaining data types are used more commonly and serve different purposes. Some of those complex types represent data that allows to deal with jABC-specific concepts. For instance, `ContextExpression` and `ContextKey` enable working with contents of the

Table 3.1. Complex built-in data types in jABC

Complex Data Type	**Represented data**
ContextExpression	EL expression evaluated on the execution context (cf. Sect. 3.3.2)
ContextKey	Key for accessing contents of a particular execution context (cf. Sect. 3.3.2)
ExtendedFile	File located relative to the current jABC project, may be restricted to specific file types
JavaBeanReference	Arbitrary Java object following the JavaBeans programming model [Ora11d]
ListBox	Single object/item selected from a fixed list of values
MultiObject	Aggregated object, similar to a record in Pascal or a struct in C
ObjectReference	Arbitrary object
Password	Password string
SIBLink	Link to another SIB instance in an arbitrary SLG
StrictCollection	Typed collection
StrictList	Typed list
Variable	Typed variable

execution context (Sect. 3.3.2 elaborates on the usage of those types). Further complex types mainly have the purpose of indicating the display of particular user interface elements in the jABC tool, in order to adequately support the application expert in specifying the corresponding data. Accordingly, the type `ListBox` signals the use of a combo box that allows the selection of one value from a fixed list of items, and the type `Password` leads to a typical password input field which only shows asterisks instead of the entered text. Finally, other complex types compensate the lack of particular data types or features in Java, such as `MultiObject`, which resembles an aggregated object similar to records in Pascal. Furthermore, `StrictCollection` and `StrictList` represent typed collections which, in contrast to Java's generics that are subject to type erasure [Gos+05, p. 56], also retain and use their type restriction at runtime. As another convention in jABC, all collection and array types must only contain values which are in turn assignable to one of the supported simple or complex types.

Please note that this set of data types represents the *technical* grounding for realizing the parameters of a SIB. Accordingly, those data types are typically only visible to the IT expert who implements the SIB. The application expert who uses a SIB in the jABC tool usually only deals with abstract types. During the customization described above, the domain expert defines such abstract types and maps them to the corresponding technical data types [NLS11].

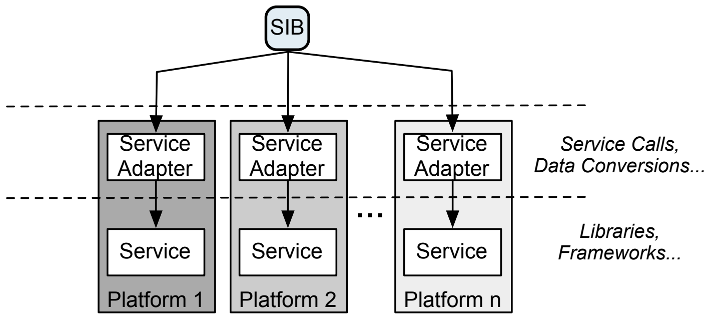


Fig. 3.3. Service adapter pattern for realizing a SIB's behavior

Service Adapters:

A *service adapter* realizes the SIB's behavior for a concrete target platform using the object adapter pattern [Gam+95]. Particularly, as one SIB may be executable on multiple target platforms, an arbitrary number of service adapters can be attached to a SIB. Fig. 3.3 illustrates this pattern. Each service adapter is implemented in a programming language supported by the desired target platform, so it may for instance be a Java class, a C# class or a Python script. Typically, it contains calls to, e.g., platform-specific third party libraries or systems, that realize the actual service represented by the SIB, along with corresponding data conversions. Decoupling the concrete platform-specific implementations from the SIB description assures that the SIB itself is entirely platform-independent. As soon as at least one service adapter is implemented for a SIB, the SIB is executable, thus enabling interpretation and code generation for models that contain the SIB. Sect. 5.2.1 elaborates on the high importance of the service adapter concept for code generation in jABC.

3.2.2 Service Logic Graphs

As mentioned above, the models in jABC are called Service Logic Graphs (SLGs). Basically, SLGs are directed graphs that represent the flow of actions in an application, thus focussing on its behavioral (dynamic) aspects [MS06]. In formal terms, SLGs are *Kripke Transition Systems* (KTS) [MOSS99; MS09a], a combination of Kripke structures and labeled transition systems. Accordingly, nodes as well as edges are labeled in a KTS:

Definition 1 (Kripke Transition System, KTS). *A KTS over a set of atomic propositions AP is a four-tuple* $\mathcal{M} = (S, \mathcal{A}, \rightarrow, I)$ *where*

- S *is a finite set of states (nodes),*
- $\mathcal{A}$ *is a finite set of action labels,*

- *the transition relation $\rightarrow \subseteq S \times \mathcal{A} \times S$ describes possible action-triggered transitions between states, and*
- *the interpretation function $I : S \rightarrow 2^{AP}$ specifies which atomic propositions hold at which node.*

AP is assumed to always contain the propositions true *and* false*, and for any state $s \in S$,* true $\in I(s)$ *and* false $\notin I(s)$. *A finite path π in $\mathcal{M}$ is considered a sequence of states and action labels $\pi = \langle s_1, a_1, s_2, a_2, s_3, \ldots, s_n \rangle$ with $s_i \in S$ and $a_i \in \mathcal{A}$, such that $(s_i, a_i, s_{i+1}) \in \rightarrow$ (also written $s_i \xrightarrow{a_i} s_{i+1}$) for $i = 1, \ldots, n-1$. Infinite paths are defined in a similar manner [MOSS99]. Furthermore, π_i refers to the state s_i on the path.*

The set of atomic propositions AP and the interpretation function I are mostly required for verification, which is described in more detail in Sect. 3.4. In the context of SLGs, the nodes in such a graph are SIB instances or, in order to enable hierarchical modeling (cf. Sect. 3.1), *macros* that point to other SLGs. For instance, in the example model depicted on the top of Fig. 3.4, the nodes labeled `Print Exception` and `Print Success` are SIB instances, while the nodes labeled `Initialize Docu Generator` and `Generate Documentation` are macros (indicated by the big dot on their icons).

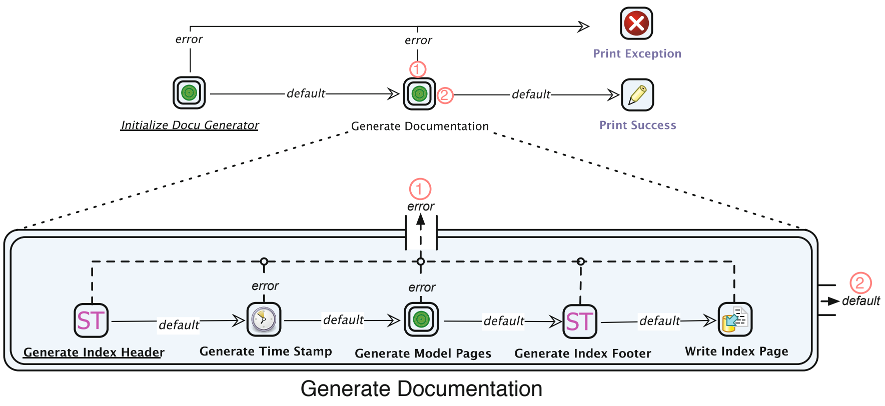


Fig. 3.4. Hierarchical models in jABC

The directed edges between the nodes, described by the transition relation $\rightarrow$, indicate the flow of actions. In SLGs , the actions connecting two nodes are reflected by branches: Each edge is labeled with one or more branches, whereas the source node of the edge defines the set of possible branches that can be assigned to that edge. In other words, the wiring of SIB instances and macros in models is performed on the basis of possible execution results. Thus roughly speaking, branches could also be seen as potential “exits” of a SIB/macro.

If a node has more than one outgoing edge, the edges represent alternative execution flows. For instance, the node `Initialize Docu Generator` in Fig. 3.4 has two branches “default” and “error”, each assigned to one outgoing edge. This reads as follows: *If the result of `Initialize Docu Generator` is “default” proceed with `Generate Documentation`, if the result is “error” execute `Print Exception`.* If `Initialize Docu Generator` produces another result, it is considered an undefined behavior. In order to specify where the execution of a model starts, a node can be defined as an entry point. This is indicated by the node’s label being underlined (e.g., `Initialize Docu Generator` in Fig. 3.4). Please note that an SLG could potentially have more than one entry point, depending on whether this is supported by the selected interpreter or code generator.

Hierarchical Modeling:

In order to enable seamless hierarchical modeling, macros are used just like normal SIBs. For the application expert, the SLG referenced by a macro is also considered a service (with the difference that its realization is available as another model) that follows the same simple interface that is also imposed on SIBs. In consequence, macros also have parameters and branches. However, as these parameters and branches belong to an entire model associated with the macro, they are called *model parameters* and *model branches*, respectively.

An SLG’s set of model parameters and model branches is defined by selectively *exporting* parameters and branches of SIB instances or macros in that SLG. Fig. 3.4 illustrates this for model branches. The bottom part of the figure shows the submodel that is associated with the macro `Generate Documentation`. This submodel has again one designated entry point, as indicated by the underlined label of `Generate Index Header`. For proper execution semantics, we also need to specify at which points the submodel can be left in order to return to the parent model. This is done by means of model branches, which, analogous to SIB branches, define the “exits” of models. In the jABC tool, such exits are not visualized by concrete edges pointing to the parent model, but instead the information is displayed in one of the inspectors (the *Graph inspector*). For illustration, Fig. 3.4 indicates the model exits via dashed arrows.

In the example, each SIB instance contained in the submodel has a branch labeled “error”, all of them exported and mapped to a model branch which is also called “error” ①. The name of a model branch or model parameter can be defined freely by the application expert. Resulting from this specification of the “error” model branch, the error handling for all execution steps in the submodel is delegated to the parent model. Furthermore, the SIB instance `Write Index Page` exports its “default” branch as a model branch which is also called “default” ②. In the parent model, the macro `Generate Documentation` provides exactly those two exits “default” and “error”, which are defined as model branches in the underlying submodel. As with normal SIBs, these branches then can be assigned to outgoing edges of the macro.

Likewise, it is possible to define model parameters of a submodel, which then become the parameters of an associated macro.

Technical Realization:

Technically, jABC manages SLGs by means of a data structure called `SIBGraphModel` which strongly focuses on robustness. In particular, the data structure ensures that SLGs can always be opened and modified, even if some or all of the contained SIBs are not available (e.g., because a particular SIB bundle has not been installed). For this purpose, any affected SIBs are replaced by a *proxy SIB*, which is a special generic SIB that is able to emulate any other SIB's interface. Thus the proxy SIB that replaces another SIB has the same parameters and branches, and both can be manipulated in the jABC tool as usual. Consequently, it is entirely transparent to the application expert whether all SIBs contained in his models are actually available.

While being an adequate replacement at modeling time, proxy SIBs are not able to emulate the runtime behavior of a SIB. As proxy SIBs lack corresponding service adapters, they are not executable. Thus proxy SIBs can only compensate the effects of missing SIBs in a way that the application expert's work is not interrupted – however, as missing SIBs threaten the executability of the modeled system, they pose a problem which has to be fixed by the IT expert.

Besides its focus on robustness, jABC's `SIBGraphModel` data structure is very extensible. For instance, plugins are allowed to attach arbitrary information, called *user objects*, to all constituent parts of a model (i.e., nodes, edges and the model itself). Similar to a hash table, any user object associated with a model element is identified by a unique key.

Metamodeling:

From the metamodeling perspective, SLGs and their associated concepts (SIBs, branches etc.) are jABC's metamodel. As this metamodel is hard-wired in the framework, it is not interchangeable and thus cannot be substituted by an entirely different one, i.e., jABC is not a language workbench in the sense described in Sect. 2.3.5. In terms of the metalevels proposed by the OMG (cf. Sect 2.3.3), modeling in jABC mostly happens on level M1.

However, this is a deliberate design decision that aims at reducing the framework's overall complexity in favor of simplicity. In contrast to CASE tools with their fixed modeling languages (cf. Sect. 2.3.1), jABC's metamodel is not static, as it can be customized in a number of ways. For instance, its abstract syntax can be dynamically extended by new SIBs which thus increase the size and expressibility of the modeling language. Additionally, the domain expert's customization possibilities outlined above allow tailoring the modeling language to particular domains, e.g., by adjusting the available language elements via taxonomies, or by adapting the concrete syntax of SLGs (terminology, icons etc.). By means of a special class of SIBs called "control SIBs" (cf. Sect. 3.3.3) it is even possible to introduce entirely new modeling constructs. However, such new constructs usually also require a corresponding adaptation of jABC's tooling (e.g., plugins). In this respect, the modeling languages derived from jABC's metamodel show characteristics of both internal (addition of new SIBs, customization by the domain expert) and external (addition of new modeling constructs) DSLs (cf. Sect. 2.2).

Apart from the abstract syntax, the static semantics of jABC's metamodel can also be customized via

- local constraints (cf. Sect. 6.1) which are attached to each SIB,
- global constraints (cf. Sect. 6.2) which are (domain-specifically) defined for models, and
- plugins (cf. Sect. 3.2.3) which may, e.g., introduce additional well-formedness rules.

This clearly contrasts the approach of language workbenches, which usually generate a domain-specific environment from a metamodel specification. jABC achieves this domain-specificity by means of customization and adaptation of a generic metamodel and tool. This idea is comparable to UML's profile mechanism (cf. Sect. 8.1).

In summary, with this *customizable and extensible metamodel*, jABC defines a *class of domain-specific languages* that share a fixed (and thus controlled) common core, the SLG concept.

SLG Types:

In this book, SLGs are used for many different purposes. Hence in the following chapters and sections, any figure containing an SLG is marked with a small icon in one of its corners, in order to avoid confusion. The icons indicate the type of the depicted model:

CG	Code generator	A	Any application
F	Formula	TC	Test case
TS	Test suite	TD	Test data

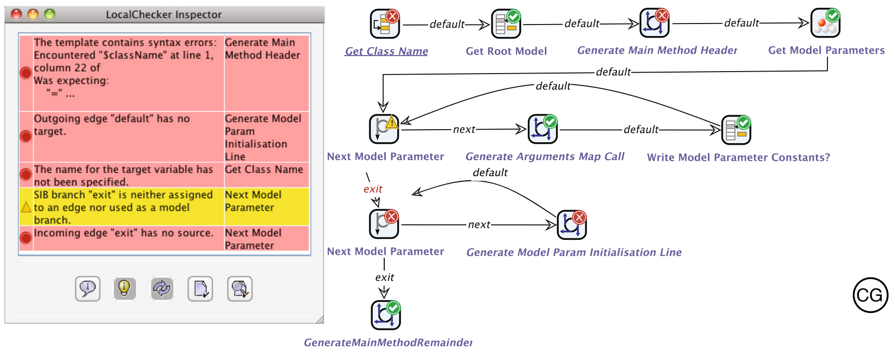
3.2.3 Plugins

Plugins allow adding functionality to jABC, e.g., by extending the jABC tool with further menus or inspectors, or by enriching SLGs and their constituents with additional information (i.e., user objects). Plugins are even able to associate different semantics with one SLG. While the previous section already anticipated the most common semantics according to which an SLG is a control flow graph, there are also plugins that interpret SLGs differently, e.g., as an entity-relationship model [Win06] or even as a formula (cf. Sect. 6.2.1). Like jABC itself, plugins are implemented in Java and then integrated in jABC via a simple interface.

As mentioned above, the selection of suitable plugins is an important task of the domain expert who customizes jABC for a particular application domain. For this purpose, a multitude of plugins is available, supporting different aspects of system development with jABC. Two plugins that are very important from the perspective of this book are presented in detail in separate sections: the Tracer, which allows the execution of SLGs (Sect. 3.3), and GEAR, which enables SLG verification via model checking (Sect. 3.4). Furthermore, the Genesys framework also provides a jABC plugin in order to support code generation for SLGs (cf. Sect. 4.3). The *FormulaBuilder*, another plugin that extends jABC by the ability of modeling and generating formulas, has been realized on the basis of Genesys and thus is also presented later on in Sect. 6.2.1. Further relevant plugins are briefly introduced below.

LocalChecker:

The *LocalChecker* plugin [Neu07; Ste+07] ensures the correct use of SIB in models by checking local constraints (cf. Sect. 3.1) that are attached to each SIB. The checks that realize those constraints are directly implemented as Java code: The IT expert adds them to the SIB's Java class (see Sect. 3.2.1) by implementing a special interface (`LocalCheck`). For convenience, the LocalChecker provides a set of general standard checks that are domain-independent and thus can be used for most SIBs. The bulk of those checks concern the well-formedness of the SLGs (i.e., they check an SLG's conformance with the static semantics specified by the metamodel, cf. Sect. 2.2). Examples of such checks include the correct parametrization of a SIB (valid codomain, input syntax etc.), branches that are not assigned to any outgoing edges, or unconnected edges that do not have a valid source or target node. The standard checks already cover the most common modeling mistakes.


LocalChecker Inspector

●	The template contains syntax errors: Encountered "$className" at line 1, column 22 of Was expecting: "=" ...	Generate Main Method Header
●	Outgoing edge "default" has no target.	Generate Model Param Initialisation Line
●	The name for the target variable has not been specified.	Get Class Name
⚠	SIB branch "exit" is neither assigned to an edge nor used as a model branch.	Next Model Parameter
●	Incoming edge "exit" has no source.	Next Model Parameter

Fig. 3.5. The LocalChecker plugin in jABC [JMS11]

While modeling in jABC, all checks specified for the employed SIBs are performed continuously. Any feedback obtained from the checks is immediately reported to the user via an additional inspector, which is depicted on the left side of Fig. 3.5. The figure also shows several examples of messages produced by different checks, and it is visible that those messages are classified by severity as errors, warnings or plain informations. For instance, the SIB "Next Model Parameter" causes a warning (second line from bottom in the inspector), and for a second instance of the SIB, an incoming edge without a proper source node leads to an error message (last line in the inspector). Besides the messages in the inspector, the check results are also indicated directly in the SLG on the icon of each SIB. Sect. 6.1 elaborates on how the LocalChecker has been employed in the context of this book.

Annotation Editor:

The *Annotation Editor* [Nag09] allows attaching almost any kind of information to jABC projects as well as to SLGs and their constituent parts (such as SIB instances or edges). By means of grammars (which are also specified as SLGs), the Annotation Editor can be tailored to specific application scenarios. Such a grammar determines which kind of information is allowed to be attached to which elements. From the grammar, the Annotation Editor dynamically assembles a correspondingly adapted user interface for creating and editing suitable information in jABC. An important application of the Annotation Editor is, e.g., the documentation of SLGs and their contained elements.

Taxonomy Editor:

This plugin is mainly used by the domain expert when customizing jABC for a particular application domain. It allows naming and organizing the available SIBs in a way that they fit the terminology and concepts of the targeted domain. For this purpose, the Taxonomy Editor provides a user interface that shows all available SIBs listed according to their physical Java package structure. Starting from this view, the domain expert renames and rearranges the SIBs as required.

jETI plugin:

Java Electronic Tool Integration (jETI) [SMN05] augments jABC by enabling the inclusion of *remotely* available services and tools as SIBs. Such SIBs can be used just like any other SIBs, i.e., the communication with the remote services is seamless and transparent to the application expert. The corresponding remote services or tools either have to be available on the internet in some standardized form (e.g., as Web Services [Pap08]), or they are provided by means of jETI's own tool server. In both cases, jETI is able to automatically generate the corresponding SIBs [SMN05;Kub+09], that allow the application expert to integrate those remote services or tools in his SLGs.

3.3 Model Execution with the Tracer

The *Tracer* [Doe06; Ste+07; JMS08] enables the direct execution of models and can thus be considered an interpreter for SLGs . As such it is key to activities such as rapid prototyping, debugging and monitoring. The Tracer is separated into two parts: a general execution environment for SLGs and a corresponding jABC plugin.

The *execution environment* is an integral part of the jABC framework, and it provides

- an *execution semantics* for SLGs , incarnated by a corresponding interpreter (thus being an example of pragmatic semantics, cf. Sect. 2.3.6),
- the concept of the *execution context* (already mentioned in Sect. 3.2.1) which acts as a shared memory for SIB instances, as well as
- an interface for adding new control flow mechanisms via specific SIBs called *control SIBs.*

The *Tracer plugin* augments the jABC tool with facilities for starting, controlling, observing and debugging SLG executions.

The Tracer is particularly essential to the realization of the Genesys approach. First, any code generator modeled with Genesys (and thus in jABC) can be, just like any other SLG, immediately tested by executing it with the Tracer. Second, when using Genesys for building code generators for jABC (cf. Sect. 5), the Tracer is the technical basis for an entire class of generators called *Extruders* (see Sect. 5.1). Finally, the execution semantics defined by the Tracer is a guideline for the behavior of the generated code, which should coincide with the behavior of the traced SLG (*execution equivalence*, see Sect. 5.1).

The following sections elaborate on the constituent parts of the Tracer.

3.3.1 Execution Semantics

In parts, the execution semantics for SLGs which is used by the Tracer has already been anticipated above in Sect. 3.2.2. The execution of an SLG always starts with one designated entry point (also called *start SIB*). When a SIB instance is reached by the interpreter, the underlying service is executed using the configuration that results from the SIB's parameter values. How this happens exactly is specified by the IT expert: In order to be executable by the Tracer, the SIB's Java class (see Sect. 3.2.1) has to implement the Tracer interface `Executable`. This implementation describes the SIB's behavior when it is executed by the interpreter. For this purpose, it may call a particular service or service adapter, it may directly implement the service itself (though this has several disadvantages for code generation, see Sect. 5.2.1), or it may contain mock code or delegate to a mock service. The latter is an option for enabling executability even if the actual service represented by a SIB is not available yet. If the reached SLG node is not an instance of a SIB but a macro, the execution descends to the referenced submodel and proceeds with the submodel's entry point. The submodel is configured with the values specified for the macro's parameters.

By all means, the result of executing a SIB instance or macro should always be one of the available branches (i.e., branches defined for the SIB or macro). The interpreter then determines the outgoing edge that is labeled with this branch and continues executing the corresponding successor. If the result reflects a model branch, i.e., an "exit" of the SLG, the execution proceeds with the parent model, and searches for a suitable outgoing edge at the macro which previously caused the interpreter to descend the hierarchy. Correspondingly, in this execution semantics, multiple outgoing branches of a SIB instance or macro usually represent alternative execution flows (though this pattern may be changed, see "Control SIBs" below).

The execution of an SLG is finished regularly (i.e., successfully) as soon as it reaches a SIB or macro that is contained in the topmost model of the hierarchy and that does not have any outgoing edges. The following situations may cause an execution to stop irregularly:

- The execution does not end at the topmost hierarchy level.
- The execution of a SIB instance produces a result which is not among the branches defined for the SIB.
- The execution of a SIB instance fails (e.g., because it has been replaced by a proxy SIB, see Sect. 3.2.2).
- An SLG defines multiple entry points or none at all.

In those cases, the Tracer aborts the execution and reports an error to the user.

3.3.2 Execution Context

As mentioned above, the execution context is a shared memory that enables communication of SIB instances in a model. Any SIB instance is able to put data into the execution context and to read, edit and delete data stored by other SIB instances. Similar to a hash table, each datum contained in the execution context is identified by a unique identifier called *context key*.

In order to enable modeling recursion with an SLG hierarchy, the Tracer also allows stacked execution contexts. In this setting, each SLG in the hierarchy is associated with its own execution context, similar to a local namespace. Each time the execution descends to a submodel referenced by a macro, a new execution context is put on the stack of contexts, and when the execution of the model is finished, the context is removed from the stack and thus ceases to exist. This concept allows shadowing of context keys, which enables full support for recursion as known from programming languages. A recursive SLG can, e.g., be constructed by means of a macro which in turn references the SLG in which it is contained. The application expert takes the decision whether a submodel should be executed with its own local execution context by selecting a corresponding type of macro (see Sect. 3.3.3). Consequently, stacking of execution contexts can be flexibly enabled or disabled for each hierarchy level.

Independent of execution context stacking being enabled or not, the Tracer always provides one designated global execution context which is accessible from anywhere in the SLG hierarchy. This global context is even maintained after the actual execution has been terminated, e.g., in order to avoid the loss of data that resulted from the SLG execution.

Besides being able to work with this global execution context and an optional local execution context associated with their model, SIB instances are also allowed to access any other context that may be present on the stack. For identifying the particular execution context that should be accessed, the Tracer defines four *scopes*:

- *global* designates the global context,
- *local* is the local context associated with the currently executed SLG,
- *parent* refers to the superordinate context on the stack, which usually belongs to the current SLG's parent model, and
- *declared* references the first context on the stack that contains a specific key (starting from the local context).

When working with execution contexts, SIB instances that are supposed to share data have to agree on the corresponding keys. Accordingly, SIBs may provide parameters of type `ContextKey` (see Sect. 3.2.1) which allow the application expert to configure how the execution contexts are accessed by a SIB. For this purpose, such a `ContextKey` parameter demands the specification of a name for the key that identifies the data in the context, as well as the selection of a scope for determining which context should be accessed. For instance, the SIB inspector depicted in Fig. 3.2 (3) in Sect. 3.2 shows an example of a context key parameter called "result" which specifies the key "indexPage" along with the scope "global". Most SIBs in jABC's ready-made libraries (such as the Common SIBs) provide this configurability.

Another type of SIB parameter for working with contents of the execution contexts is `ContextExpression` (see Sect. 3.2.1). This data type uses *context expressions* written in the expression language (EL) introduced with the JavaServer Pages Standard Tag Library (JSTL) [Jav06] in order to enable *dynamic* access to the execution contexts. The Tracer's execution environment provides corresponding resolvers for evaluating such expressions during the execution. For instance, the simple expression `${keyName}` is resolved to the value identified by the context key `keyName`. In this expression, anything between the characters `${` and `}` is interpreted as a context key, and upon evaluation the resolvers search the correspondingly referenced value in the execution contexts using the scope "declared". Furthermore, context expressions can be used to perform more complex tasks such as concatenating character data (e.g., `${key1}${key2}`), accessing attributes of objects stored in the context (e.g., `${keyOfObject.attributeName}`), performing comparisons (e.g., `${key1 <= key2}`) or calling external functions (e.g., `${key}_${Math:random()}` for suffixing the string value of `key` with an underscore and a random number). In consequence, as expressions may contain static strings along with the dynamically evaluated parts (such as `_` in the last example), context expressions may even be used as a simple template language[2] (cf. Sect. 2.4.2). However, as working with context expressions requires certain technical skills, they are mostly intended to be used by IT experts on the lower levels of an SLG hierarchy, rather than by application experts.

As to the experience of the author, advanced features like stacked execution contexts or context expressions are not required in most application scenarios. Especially for applications which do not require recursion, resorting to one single (i.e., *flat*) execution context instead of the stack variant is often preferable. This way, application experts do not have to consider scopes at all, which noticeably eases the creation of SLGs . Furthermore, for application experts, context expressions complicate the usage of corresponding SIBs as they require to learn and to apply the expression language. Hence instead of producing generic, highly configurable SIBs with context expressions, the development of more domain-specific SIBs is preferable, because such SIBs are usually significantly easier to use.

For the sake of simplicity, the remainder of this book will use the singular notion "the execution context" for both a flat execution context and for the stack variant.

3.3.3 Control SIBs

As a further important feature, the Tracer's execution environment supports the extension and adaptation of the standard execution semantics for SLGs described above. Such extensions can be performed by means of specific SIBs called *control SIBs*. In contrast to normal SIBs, control SIBs do not represent an underlying service that is executed by the Tracer. Instead, when the execution arrives at a control SIB, the Tracer completely hands over the execution control. As control SIBs are granted full access to the execution environment,

[2] In fact, the EL language combined with a corresponding implementation, such as the resolvers provided by the Tracer's execution environment, is just another template engine.

they are, e.g., able to create new execution contexts or even new execution threads. Given these competences, control SIBs are able to alter existing control flow patterns and to define new ones. Currently, jABC provides a small set of standard control SIBs that add control flow patterns for hierarchy, multi-threaded execution as well as event handling.

The previous sections already stated that *hierarchy* in SLGs is created by means of macros. In fact, those macros are control SIBs which tell the Tracer to continue the execution with corresponding submodels. As mentioned above in the description of the execution context, there are several types of macros that represent different ways for executing a submodel. The standard macro is the `MacroSIB`, which simply executes the referenced submodel using a flat execution context. In contrast to this, the `GraphSIB` creates a new local context for the execution of the submodel (i.e., stacked execution contexts are used). The `ThreadSIB` behaves similarly, but it additionally executes the submodel in a separate thread. Thus by selecting one of those three macros, the application is able to determine how the referenced submodel should be executed and in particular which type of execution context (flat or stacked, cf. "Execution Context" above) is employed.

Further control SIBs allow the use of *multi-threading* in SLGs. For this purpose, the `ForkSIB` modifies the standard execution semantics that usually treats multiple outgoing edges as alternative execution paths. For the `ForkSIB`, each outgoing edge represents a separate thread, so that upon execution, the Tracer follows all specified outgoing edges in parallel. Via another control SIB, the `JoinSIB`, threads can be synchronized and remerged as one single thread. SLG 4 in Fig. 6.9 (Sect. 6.3.1) illustrates the use of those two control SIBs. When executing this model, the `ForkSIB` creates one thread for each outgoing edge ("Thread1" and "Thread2"), so that the macros `Sequence` and `Recursion` are executed in parallel. Afterwards, the `JoinSIB` synchronizes the two threads, i.e., it waits for both threads to finish and then proceeds with the execution in one single thread.

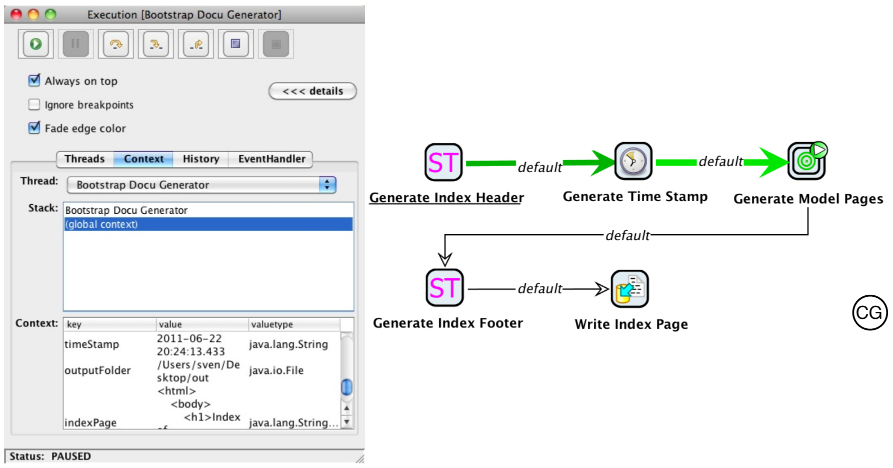
In order to enable proper execution, the Tracer defines strict rules for the well-formedness of such fork-join constructs. First, fork and join are only allowed to occur pairwise, i.e., for each `ForkSIB` there has to be exactly one corresponding `JoinSIB` that synchronizes exactly those threads created by the `ForkSIB`. Correspondingly, the number of outgoing edges of a `ForkSIB` (i.e., the number of created threads) has to match the number of incoming edges of the corresponding `JoinSIB`. In particular, this entails that the modeled threads between the `ForkSIB` and the `JoinSIB` must not contain any "exits", such as model branches or other paths that bypass the `JoinSIB`. Furthermore, fork-join constructs have to be nested in a way that does not violate any of the above rules. If a fork-join construct violates one or more of these rules it is not well-formed and thus will lead to an irregular termination of the execution.

Finally, there are also control SIBs that support event handling. Those control SIBs allow to suspend the execution of a model until it receives a specific event. Furthermore, a submodel can be specified that handles the incoming event. In order to avoid infinite waiting, the suspension of the execution can be restricted by means of a timeout.

The control flow mechanisms supported by the standard execution semantics and by the existing control SIBs have proven sufficient for all application scenarios of jABC so far. However, further control flow patterns (e.g., those from [VDA+03] that are not yet supported) can be added on demand by implementing corresponding control SIBs.

3.3.4 Tracer Plugin

The *Tracer plugin* can be considered a debugger for SLGs . Besides allowing to simply execute an SLG straightaway by pushing a "play" button, it is also able to graphically visualize the current state of an execution. This is depicted in Fig. 3.6. In the SLG on the right hand side, the SIB that is currently executed (`Generate Model Pages`) is marked by a little icon, and the previous path of the execution is also made visible by means of highlighted edges.



Execution [Bootstrap Docu Generator]

☑ Always on top
☐ Ignore breakpoints
☑ Fade edge color

Thread: Bootstrap Docu Generator

Stack: Bootstrap Docu Generator, (global context)

Context:

key	value	valuetype
timeStamp	2011-06-22 20:24:13.433	java.lang.String
outputFolder	/Users/sven/Desktop/out	java.io.File
indexPage	<html> <body> <h1>Index	java.lang.String...

Status: PAUSED

Fig. 3.6. Executing an SLG with the Tracer

The dialog that is shown on the left hand side of Fig. 3.6 has two purposes. First, it allows to control the execution: By means of the buttons on the top, the user is able to start, pause and stop the execution at any point. Second, the dialog displays information about the current execution, such as the currently running threads, the current contents of the execution contexts and a history reflecting the execution path so far.

Instead of visually executing the SLG step by step, the user may also define breakpoints which cause the execution to stop at specified points of interest.

3.4 Model Checking with GEAR

As outlined in Sect. 3.1, safeguarding the consistency of the global system model while it is composed from models and services is a central requirement of the XMDD approach, as it helps overcoming typical system gaps. Furthermore, only a consistent global system model can be properly executed and translated to working code. The rules that have to be followed (or, in other words, the properties that have to be satisfied) in order to ensure this consistency are specified by means of (temporal) constraints, which are an integral part of XMDD's model library. Those constraints form a steadily growing library and thus are the basis of incremental formalization (cf. Sect. 3.1). Sect. 3.2.3 already presented the LocalChecker plugin, which focusses on verifying local constraints that concern the single SIB instances in a model. However, an additional check tool is required that takes care of global constraints which describe the rules addressing entire models.

As also stated above, a central advantage of shifting all activities of system development to the modeling level is the fact that models are amenable to formal methods. Among those, *model checking* [CGP99; QS82; MOSS99] is an established technique for checking whether a given model satisfies a specified property. The *GEAR plugin* [Bak+09] enables user-friendly model checking of global constraints for SLGs in jABC based on a game-based approach [Bak+07; MOY04]. The following sections introduce this plugin from the user perspective: Sect. 3.4.1 elaborates on how global constraints for GEAR are specified and Sect. 3.4.2 describes how the GEAR model checker is actually used via the corresponding jABC plugin. Please note that the sections only focus on those parts of GEAR which are important in the context of this book. For more details on GEAR's conceptual and algorithmic aspects, in particular on the realization and impact of the employed game-based approach, please refer to [Bak+07; MOY04; Yoo07]. Finally, Sect. 6.2 elaborates on how GEAR has been applied for model checking of code generators.

3.4.1 Specification of Global Constraints

When working with model checkers, constraints or properties that should be verified are typically specified by means of temporal logics [CGP99]. This approach is quite general, as temporal logic is sufficient to express any required property [Ste89; SI94]. GEAR essentially is a model checker for the *modal μ-calculus* [Koz83], which is its core logic for formulating global constraints. However, GEAR also supports "derived, more user-friendly logics" [Bak+09] (or also "domain-specific specification languages" [Bak+09]) such as the Computation Tree Logic (CTL) [BAPM83; CE81]. Such alternative logics and specification formalisms can be flexibly added to GEAR on the basis of so-called *macros* (called *GEAR macros* in the following in order to avoid confusion with the macros used in SLGs for modeling hierarchy, cf. Sect. 3.2.2). Basically, a GEAR macro is an abbreviation or a pattern that represents a specific formula, and that can be used like a simple black box (similar to the idea of SIBs), thus leading to more readable and concise constraints. When using a GEAR macro, GEAR automatically expands it and internally – i.e., transparently to the user – translates it to the modal μ-calculus. This even allows mixing several formalisms and languages in one constraint specification (see Sect. 6.2 for example constraints).

The flexibility and extensibility of GEAR's input syntax is a powerful feature as it allows to continuously improve the accessibility and simplicity of property specifications. For instance, this mechanism is used to incorporate the property specification patterns proposed by Dwyer et al. [DAC99]. Those patterns capture common constraints concerned with the occurrence or order of actions in a formalism-independent way, and thus are, in terms of their abstraction level, on par with GEAR macros. Sect. 6.2 shows several examples of those patterns, which are frequently used in this book for specifying constraints for code generators.

The FormulaBuilder is an extension of GEAR that aims at further easing the specification of constraints. This jABC plugin allows to graphically formulate constraints as SLGs, which can be seamlessly used, just like any other constraints, for performing model verification with GEAR. Since the FormulaBuilder is also an application of the Genesys approach, it is presented in detail in Sect. 6.2.1.

Finally, as this book uses a variant of CTL for the formal description of constraints, this logic is briefly introduced in the following.

Computation Tree Logic (CTL)

CTL can be used to formulate temporal constraints of a model or, more formally, of a computation tree, which can be represented, e.g., as a Kripke structure [CGP99] or as a Kripke Transition System (cf. Sect. 3.2.2). For instance, such constraints are concerned with the reachability or order of certain states. For the specification of constraints, CTL employs path quantifiers and temporal operators [CGP99]. The former refer to the branching structure of a model, and the latter describe properties that hold on a single path. The possible path quantifiers in CTL are A meaning "for all paths", and E meaning "for some paths". Furthermore, there are four temporal operators:

- $X\ \phi$ ("next"): The property ϕ holds in the next state of the path.
- $G\ \phi$ ("globally"): ϕ holds in all states of the path.
- $F\ \phi$ ("finally"): ϕ holds eventually at some state of the path.
- $\phi\ U\ \psi$ ("until"): ψ holds at some state of the path and ϕ holds in all preceding states. This operator is sometimes also called "strong until" as it requires ψ to hold finally [MOSS99]. There is also a commonly used variant of this operator called "weak until" ($\phi\ WU\ \psi$) which does not demand the occurrence of ψ, but instead also allows ϕ to hold forever.

Due to the fact that its operators quantify over paths that start in a specific state of the model, CTL belongs to the *branching-time logics* [MOSS99]. For the formal description of temporal constraints, this book uses a variant of CTL based on a logic described by Schmidt and Steffen [SS98]. This variant adds the box ($[\,]$) and diamond ($\langle\rangle$) operators known from Hennessy-Milner logic [HM85;Mil89] along with corresponding backward counterparts ($\overline{[\,]}$ resp. $\overline{\langle\rangle}$). Those operators are basically parametrized versions of the "next" operator that allow to refer to certain actions. The following definition describes how formulas of the CTL variant are composed syntactically:

Definition 2 (Syntax of CTL variant, based on [SS98] and [MOSS99]). *The syntax of a CTL formula is defined as follows*[3]*:*

$$\begin{aligned}\phi\ ::= {} & p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \\ & \mid [a]\ \phi \mid \overline{[a]}\ \phi \mid \langle a\rangle\ \phi \mid \overline{\langle a\rangle}\ \phi \\ & \mid AG\ \phi \mid EG\ \phi \mid AF\ \phi \mid EF\ \phi \mid A[\phi\ U\ \phi] \mid E[\phi\ U\ \phi]\end{aligned}$$

with p being an element of a set of atomic propositions AP and a being an element of a finite set of action labels $\mathcal{A}$.

Please note that the definition is reduced to those syntactic elements that are required for the descriptions in this book. Consequently, it does not contain all CTL extensions introduced in [SS98] (e.g., the parametrized version of the AG operator).

The semantics of those formulas can be defined with respect to Kripke Transition Systems (KTS, see Definition 1), which are the formal basis of SLGs :

Definition 3 (Semantics of the CTL variant, based on [HR04, p. 211] and [MOSS99]). *Let $\mathcal{M} = (S, \mathcal{A}, \rightarrow, I)$ be a KTS. The relation $\mathcal{M}, s \models \phi$ (read as "ϕ holds in state s of $\mathcal{M}$") is defined by structural induction on ϕ:*

[3] The notation of the syntax description is based on the well-known Backus-Naur Form (BNF) [Bac+63].

1. $\mathcal{M}, s \models p$ $\Leftrightarrow p \in I(s)$
2. $\mathcal{M}, s \models \neg\phi$ $\Leftrightarrow \mathcal{M}, s \not\models \phi$
3. $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ $\Leftrightarrow \mathcal{M}, s \models \phi_1$ *and* $\mathcal{M}, s \models \phi_2$
4. $\mathcal{M}, s \models \phi_1 \vee \phi_2$ $\Leftrightarrow \mathcal{M}, s \models \phi_1$ *or* $\mathcal{M}, s \models \phi_2$
5. $\mathcal{M}, s \models \phi_1 \Rightarrow \phi_2$ $\Leftrightarrow \mathcal{M}, s \not\models \phi_1$ *or* $\mathcal{M}, s \models \phi_2$
6. $\mathcal{M}, s \models [a]\ \phi$ $\Leftrightarrow$ *for all* t *with* $s \xrightarrow{a} t$ *we have* $\mathcal{M}, t \models \phi$
7. $\mathcal{M}, s \models \overline{[a]}\ \phi$ $\Leftrightarrow$ *for all* t *with* $t \xrightarrow{a} s$ *we have* $\mathcal{M}, t \models \phi$
8. $\mathcal{M}, s \models \langle a \rangle\ \phi$ $\Leftrightarrow$ *there exists a* t *with* $s \xrightarrow{a} t$ *such that* $\mathcal{M}, t \models \phi$
9. $\mathcal{M}, s \models \overline{\langle a \rangle}\ \phi$ $\Leftrightarrow$ *there exists a* t *with* $t \xrightarrow{a} s$ *such that* $\mathcal{M}, t \models \phi$
10. $\mathcal{M}, s \models AG\ \phi$ $\Leftrightarrow$ *for all paths* π *with* $\pi_1 = s$ *and all* s_i *along the path, we have* $\mathcal{M}, s_i \models \phi$
11. $\mathcal{M}, s \models EG\ \phi$ $\Leftrightarrow$ *there exists a path* π *with* $\pi_1 = s$ *and for all* s_i *along the path, we have* $\mathcal{M}, s_i \models \phi$
12. $\mathcal{M}, s \models AF\ \phi$ $\Leftrightarrow$ *for all paths* π *with* $\pi_1 = s$ *there exists an* s_i *such that* $\mathcal{M}, s_i \models \phi$
13. $\mathcal{M}, s \models EF\ \phi$ $\Leftrightarrow$ *there exists a path* π *with* $\pi_1 = s$ *for which there exists an* s_i *such that* $\mathcal{M}, s_i \models \phi$
14. $\mathcal{M}, s \models A[\phi_1\ U\ \phi_2]$ $\Leftrightarrow$ *for all paths* π *with* $\pi_1 = s$ *there exists an* s_i *such that* $\mathcal{M}, s_i \models \phi_2$ *and for each* $j < i$ *we have* $\mathcal{M}, s_j \models \phi_1$
15. $\mathcal{M}, s \models E[\phi_1\ U\ \phi_2]$ $\Leftrightarrow$ *there exists a path* π *with* $\pi_1 = s$ *for which there exists an* s_i *such that* $\mathcal{M}, s_i \models \phi_2$ *and for each* $j < i$ *we have* $\mathcal{M}, s_j \models \phi_1$

It is often convenient to use $[Act]$ and $\langle Act \rangle$ for a set of action labels $Act \subseteq \mathcal{A}$ [MOSS99]:

$$[Act]\ \phi \overset{def}{=} \bigwedge_{a \in Act} [a]\ \phi \qquad\qquad \langle Act \rangle\ \phi \overset{def}{=} \bigvee_{a \in Act} \langle a \rangle\ \phi$$

Using this notation, the original (unparametrized) “next” operator of CTL can be written as:

$$AX\ \phi \overset{def}{=} [\mathcal{A}]\ \phi \qquad\qquad EX\ \phi \overset{def}{=} \langle \mathcal{A} \rangle \phi$$

The “weak until” operator mentioned above can be expressed by means of the “until” operator [Lar95]:

$$A[\phi_1\ WU\ \phi_2] \equiv \neg E[\neg\phi_2\ U\ (\neg\phi_1\ \wedge \neg\phi_2)]$$
$$E[\phi_1\ WU\ \phi_2] \equiv \neg A[\neg\phi_2\ U\ (\neg\phi_1\ \wedge \neg\phi_2)]$$

3.4.2 GEAR Plugin

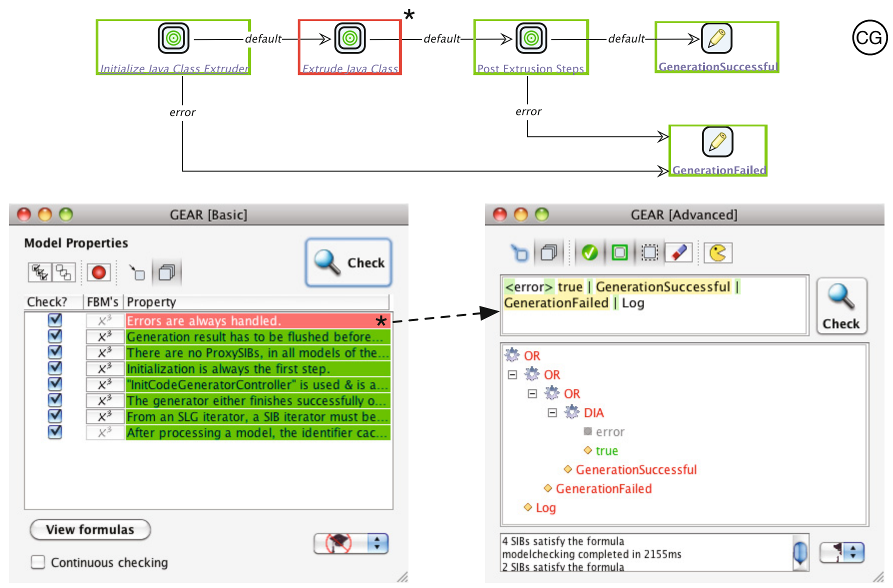
The GEAR plugin provides a user interface for verifying SLGs in jABC. In order to account for the various roles involved in working with jABC (cf. Sect. 3.2) and their different skill sets, the plugin supports two different inspectors: the basic and the advanced view.

The *basic view* is visible on the bottom left of Fig. 3.7. It is intended for application experts and other users who want to check whether their modeled SLGs satisfy a given set of constraints. This set consists of a selection of constraints from the constraint library, which are applied to the current domain. This selection is usually performed by the domain expert (or a dedicated specification expert as described in [Bak+09]). The basic view provides a simple list of those active constraints which are displayed in form of natural language descriptions. While assembling an SLG, the active constraints are checked continuously[4], and the list entries in the basic view are marked green or red depending on whether the corresponding constraint is satisfied or not, respectively. For further inspection, the application expert may select a particular constraint in the list: Consequently, each SIB in the currently displayed SLG is marked with a green or red box, depending on whether the selected constraint holds at this node. For instance, in the basic view in Fig. 3.7, the first constraint from the list is marked red (indicated by a "*"), meaning that it is not satisfied by the current SLG. The constraint description says "*Errors are always handled.*". By clicking the constraint in the list, the source of the error can be identified directly in the SLG shown at the top of Fig. 3.7. All SIBs in this SLG are marked with a green box, except for `Extrude Java Class` (again indicated by a "*"). Unlike the others, this SIB misses an outgoing edge labeled "error" which models how errors should be handled, thus violating the constraint. In this case, the error can be corrected by simply adding the missing edge.

For domain experts and specification experts who are versed in the employed specification formalisms, the *advanced view* provides means for further diagnosis. As visible on the bottom right of Fig. 3.7, this view shows the formula underlying a constraint, in this case the formula associated with the constraint exemplified above. Furthermore, the view displays the syntax tree of the formula which may be used for further inspection: Each node in this tree corresponds to a subformula, and on selecting a particular node, again green or red boxes highlight the SIBs in the SLG based on whether they satisfy the selected subformula. This also works in the opposite direction: When selecting a particular SIB in the SLG, the single nodes in the syntax tree are marked green or red, so that the user is able to see which subformulas are satisfied by the SIB and which are not satisfied. Bakera et al. refer to this as *reverse checking* [Bak+09].

The advanced view supports the creation and modification of constraints by means of features such as syntax highlighting, on-the-fly syntax checking and auto completion. An additional formula manager dialog (not visible in Fig. 3.7) allows adding constraints to or removing them from the set of active constraints, and it is also used for providing the natural language descriptions displayed to

[4] Alternatively, the check can be triggered manually via the "Check" button.



Check?	FBM's	Property
✓	X^3	Errors are always handled. *
✓	X^3	Generation result has to be flushed before…
✓	X^3	There are no ProxySIBs, in all models of the…
✓	X^3	Initialization is always the first step.
✓	X^3	"InitCodeGeneratorController" is used & is a…
✓	X^3	The generator either finishes successfully o…
✓	X^3	From an SLG iterator, a SIB iterator must be…
✓	X^3	After processing a model, the identifier cac…

```
<error> true | GenerationSuccessful |
GenerationFailed | Log
```

```
4 SIBs satisfy the formula
modelchecking completed in 2155ms
2 SIBs satisfy the formula
```

Fig. 3.7. User interface of the GEAR plugin

the application expert in the basic view. Furthermore, the advanced view allows access to GEAR's game-based facilities [Bak+09].

Finally, the GEAR plugin provides an inspector (also not shown in the figure) for manually equipping SIBs in an SLG with atomic propositions. Please note that atomic propositions may also be automatically attached to SIBs or, e.g., derived from the domain knowledge specified by the domain expert [Lam+10].

3.5 jABC as a Basis for Realizing the Genesys Approach

As mentioned at the beginning of this chapter, the demand for code generation facilities in jABC was the initial motivation for starting the Genesys project. Beyond that, jABC in general provides a suitable basis for realizing the requirements of the Genesys approach presented in Sect. 1.1. In fact, jABC and its plugins contribute to most requirements:

- *Requirement G1 - Platform Independence*: jABC's service mechanism embodied by SIBs facilitates platform independence as demanded by this requirement: SIBs represent arbitrary services (i.e., there are no restrictions whatsoever), and the corresponding implementations can be easily interchanged. The latter is performed by adding new or replacing existing service adapters, which requires no changes at the modeling level and which is thus transparent to the generator developer.

- *Requirement G2 - Reusability and Adaptability*: As pointed out in Sect. 3.1, both reusability and adaptability are rooted as very basic principles in XMDD, which are obtained by establishing libraries of models and services. The reusability of models is enabled by means of hierarchical modeling.
- *Requirement G3 - Simplicity*: As jABC essentially aims at involving non-programmers in the design and development of software systems, it is specifically designed towards simplicity.
- *Requirement G4 - Separation of Concerns*: The separation of concerns is supported by jABC via hierarchical modeling.
- *Requirement G5 - Verification and Validation*: The LocalChecker plugin and the GEAR plugin provide powerful verification mechanisms.
- *Requirement S1 - Domain-Specificity*: jABC can be tailored to a specific domain by a domain expert, using mechanisms such as SIB taxonomies, plugins and code generators.
- *Requirement S2 - Full Code Generation*: Due to the combination of models and services, jABC's SLGs represent complete and executable (sub-) systems, and thus contain all information required for the generation of complete code (cf. Sect. 2.4.4).
- *Requirement S3 - Variant Management and Product Lines*: As will be shown in Sect. 4.1.4, the mechanisms for hierarchical modeling in jABC also form an adequate technical basis for specifying variability.
- *Requirement S5 - Bootstrapping*: Bootstrapping is enabled by the executability of jABC models and by the availability of the Tracer (cf. Sect. 5.1).

In sum, jABC was an obvious choice for the reference implementation of the Genesys approach. However, the question arises as to whether other MD* approaches and tools would also be suitable for such an implementation. The remainder of this section exemplarily discusses this for several approaches related to XMDD and jABC. Please note that the presented list is not exhaustive, as the consideration of all related work[5] of XMDD and jABC is beyond the scope of this book. Instead the following discussion is intended to provide an intuition for which general characteristics and features are necessarily required for realizing the Genesys approach.

MDA:

As pointed out in Sect. 2.3.3, platform independence and reusability (Genesys requirements *G1* and *G2*) are primary concerns of MDA. Furthermore, UML's profile mechanism (see Sect. 8.1 for details) is designed to support domain-specificity (requirement *S1*), and there are also various approaches to the verification (requirement *G5*) of UML models (e.g., [LMM99; Sch+04;

[5] See, e.g., [Ste+07] or [Nag09] for more discussions on related work of XMDD and jABC.

JS04]). However, especially when using UML, MDA is not a suitable basis for the Genesys approach due to several reasons.

First, UML does not per se include concepts for modeling with services. There have been several proposals for extending it correspondingly (e.g., [HLT03; LS+08]), but those extensions usually add bare modeling constructs and terminology only. In particular, this means that at modeling time, there is usually no direct connection between the services reflected as corresponding constructs in the models, and their actual implementation(s). In jABC, this relationship is established by means of the service adapters attached to each SIB.

The second problem of UML is the lack of model executability. In contrast to this, jABC models are executable by design, and the Tracer provides a corresponding interpreter as well as a debugging environment. The lack of comparable facilities in UML particularly impedes the realization of the Genesys requirements *S2 - Full Code Generation* and *S5 - Bootstrapping.* Sect. 2.4.4 already pointed out that initiatives like Executable UML aim at adding executability to UML, but typically come at the expense of the abstraction provided by the models.

Finally, this lack of abstraction, which is often criticized for UML in general (cf. Sect. 2.3.3), also impinges upon simplicity (requirement *G3*).

BPM:

As mentioned in Sect. 2.6, approaches in the realm of BPM very often employ graphical modeling on the basis of services. In consequence, executability is a natural characteristic of the models occurring in those approaches. Sect. 2.3.6 listed several examples of process engines that execute the various notations available in BPM. Just like for UML, there are also several approaches to verification that apply to different process notations used in BPM [Mor08].

However, the Genesys approach typically requires executability for rapid prototyping, fast debugging during development as well as for realizing full code generation and bootstrapping. After a code generator is completed, it is typically translated into a desired implementation language running on a particular host platform (such as the JVM). In contrast to this, in BPM approaches, process engines *are* the host platform rather than primarily being a helpful device during modeling. Consequently, those engines usually provide features that aim at supporting the operation of large software systems, such as scalability, persistence or long-running transactions. While useful for the typical application scenarios of BPM, those features are not required for the development of code generators and thus provide a significant overhead. The lightweight execution enabled by jABC's Tracer is much more suitable in this context.

Another fundamental difference can be observed for domain-specificity. Due to the fact that jABC can be tailored to specific domains by design (as described in Sect. 3.2), it can in particular be easily customized for the domain "code generation". However, tools for BPM usually are optimized for and thus restricted to one specific domain – business processes. Consequently, BPM tools are not intended at all for the development of code generators, and thus are by their very nature not able to support a generator developer like a customized domain-specific tool.

DSM and Language Workbenches:

Sect. 2.3.4 and 2.3.5 pointed out that the DSM approach and the class of tools called language workbenches include domain-specificity by definition: Both are based on providing dedicated development environments for the creation of domain-specific solutions. Due to this very general orientation, all requirements of the Genesys approach could be realized with DSM or language workbenches (excluding those language workbenches that only focus on textual DSLs, such as Xtext or Spoofax).

Realizing the Genesys approach this way would include the design of a graphical DSL that allows modeling code generators on the basis of services, and, where required, the implementation of corresponding specific tooling (e.g., an interpreter for execution, and tools for verification and testing). Consequently, such an approach would have cost significantly more development time than required for the jABC-based solution presented in this book. As jABC already provides a customizable modeling language, a corresponding modeling environment, useful plugins and several libraries of ready-made services (e.g., the Common SIBs), it was clearly preferable to DSM and language workbenches.

Part II

The Genesys Framework and Case Studies

4

The Genesys Framework

The Genesys code generation framework is a reference implementation of the ideas that constitute the Genesys approach presented in this book. As mentioned in Sect. 3, the jABC framework and its underlying XMDD approach form the technical and conceptual basis for this implementation. At the same time, jABC is also an appropriate domain for applying the Genesys framework in case studies (see Chap. 5).

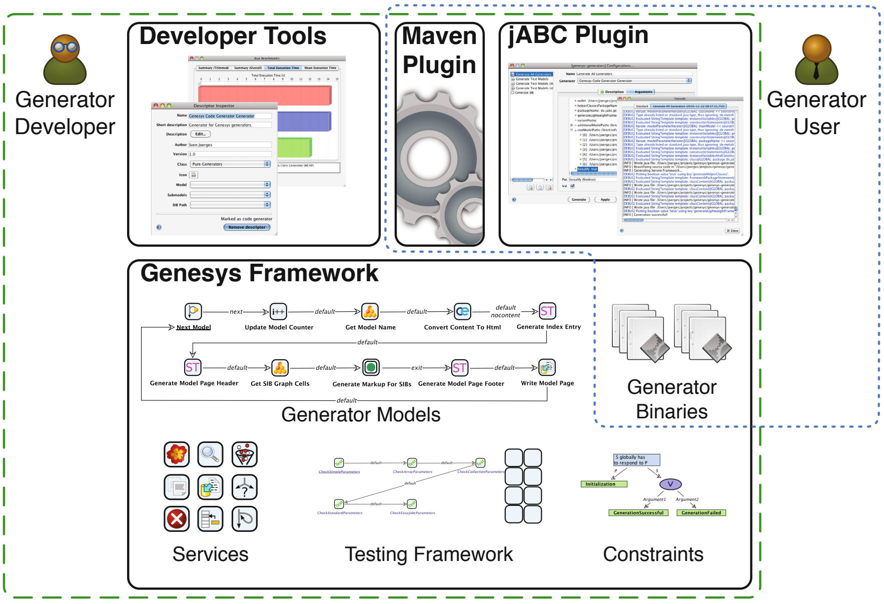


Fig. 4.1. Genesys architecture and involved roles

Fig. 4.1 shows how the reference implementation is organized, the central part being the actual *framework*, which consists of the following components:

Services: The framework provides a library of services that cover typical functionality required for most code generators, such as type conversion, identifier generation, model transformations and code beautification (*Requirement G2 - Reusability and Adaptability*). These services are available as SIBs (cf. Sect. 3.2), so that they can be used as atomic building blocks for code generator models built with jABC. Sect. 4.1 further elaborates on this service library.

Generator Models: As described in Sect. 3.2, XMDD does not only support the reuse of services, but also of entire models (*Requirement G2 - Reusability and Adaptability*). Consequently, the framework contains a library of code generator models which realize further typical functionality such as loading and traversing input models, e.g., jABC's SLGs or EMF models (cf. Chap. 7). Just like the atomic services mentioned above, these models can be directly reused as macros when building a new code generator (thus representing ready-made *features* in the sense of XMDD, see Sect. 3.1). They can also serve as patterns which are instantiated or adapted for new code generators. Furthermore, the model library contains most code generators created in the case studies which will be presented in Chap. 5–8. The rationale behind this is that each new code generator contributes to this library of models, so that the available repertoire and the potential for reuse is growing continuously (*Requirement G2 - Reusability and Adaptability*).

Generator Binaries: In order to be accessible by tools and users, the framework also includes all code generators as compiled Java classes. For this purpose, each modeled code generator is translated to an appropriate Java class via the *Genesys Code Generator Generator* (see Sect. 5.2.6).

Testing Framework: As an addition to manually testing a code generator by executing its models with the Tracer (cf. Sect. 3.3), the Genesys framework also includes an approach for the automated model-driven testing of code generators. In this approach, test cases as well as the corresponding test inputs are also modeled as SLGs. Subsequently, dedicated code generators (again developed with Genesys) translate these SLGs into test scripts or test programs running on a desired test platform. Sect. 6.3 elaborates on the details of this approach, which is currently realized for jABC code generators, i.e., those which support SLGs as input models (see Chap. 5). Accordingly, the provided facilities include a library of SIBs for building test cases and test data models, a collection of standard test cases covering the domain of jABC models as well as a testing strategy for checking whether the execution semantics of a model is retained by the code generation (cf. Sect. 6.3.1).

Constraints: For verifying code generators via model checking, the framework provides a library of global constraints which specify required properties such as the complete processing of the input data or proper handling of errors. These constraints are specified graphically as *formula graphs* and then translated to temporal logics by the FormulaBuilder (cf. Sect. 6.2.1), which is also an application of jABC and Genesys. Sect. 6.2 presents examples of such global constraints for code generators and shows how they are specified.

Above the framework, Fig. 4.1 shows *tools* that support the usage and the development of code generators. The *jABC plugin* allows the configuration and invocation of code generators within the jABC editor, and the *Maven Plugin* enables the integration into a Maven-based project setup. Finally, the *developer tools* bundle utilities that support the creation of code generators. All tools will be presented in more detail in Sect. 4.3.

Fig. 4.1 also indicates the roles that are targeted by Genesys. First, Genesys addresses *generator developers* (left hand side of Fig. 4.1) by providing facilities that support creating, adapting, verifying and testing code generators. For this purpose, they use a specific jABC bundle tailored to the domain "code generation", containing all services, models and tools depicted in Fig. 4.1. In combination, the ready-made services and models form a *DSL for building code generators*. Second, Genesys offers a library of ready-made code generators which can be used without deeper knowledge of the Genesys approach or the internals of the framework. *Generator users* (right hand side of Fig. 4.1) may access these generators by means of the tools outlined above (jABC plugin, Maven plugin), or integrate the generator binaries into their own applications via API.

When Genesys is applied to jABC itself, i.e., used for developing a code generator whose source language is given by SLGs, the above two roles can be related to the jABC roles introduced in Sect. 3.2. In this specific scenario, which is examined in greater detail in Chap. 5, the generator developer equals the domain expert and the generator user is congruent with the application expert. When creating a domain-specific jABC variant for the application expert, the domain expert also has to define how SLGs are translated to code in the target domain by creating new code generators with Genesys or selecting existing ones. The application expert then uses the jABC variant customized by the domain expert to model and to generate an application for the target domain.

The following sections will elaborate on Genesys' constituent parts, starting with the atomic services for creating code generator models (Sect. 4.1). In order to give an idea of modeling a code generator with Genesys, Sect. 4.2 describes a complete example of a simple code generator. Finally, Sect. 4.3 presents all tools provided by Genesys.

4.1 Services for Building Code Generators

The Genesys framework is equipped with many ready-made atomic services which are made available as SIBs, the most important building blocks for creating code generator models. Following the principles of service orientation, these services are black boxes with very simple interfaces, and, in consequence, easy to use without any knowledge of their actual implementation (*Requirement G3 - Simplicity*). The availability of such ready-to-use services saves the generator developer from having to start from scratch, thus speeding up the overall development (*Requirement G2 - Reusability and Adaptability*). The following sections present services which are especially relevant to the code generation domain. As the description of all available SIBs (more than 200 Common SIBs and around 40 Genesys-specific SIBs) would go beyond the scope of this book, there is a focus on those services which are essential to most code generators. For an exhaustive list of all available SIBs please refer to the documentation of the Common SIBs [TU10] and the Genesys SIBs [Jör10].

Please note that the following descriptions focus on the parameters when explaining the structure of the SIBs. This is due to the fact that all presented SIBs provide the same two branches: `default`, meaning that the service has been executed successfully, and `error`, indicating that the execution of the service has failed.

4.1.1 Contributions to the Common SIBs

During the development of Genesys, many SIBs had to be created, in particular at the very beginning of the project. At this time, Genesys significantly pushed the development of jABC's Common SIBs (cf. Sect. 3.2.1), thus contributing to the availability of general jABC services. Any created SIB which was suitable to be used with a broader scope than code generation has been added to the Common SIBs.

The following SIB bundles have been heavily influenced by Genesys and thus are used in nearly any code generator:

Basic SIBs: The SIBs contained in this bundle provide very basic functionality which is required by nearly any application modeled in jABC (especially by any Genesys code generator). This mainly includes building blocks for manipulating the execution context (see Sect. 3.3.2), e.g., creating, modifying, removing or copying context entries, as well as for thread-safe access, and locking or unlocking context entries. Furthermore, the bundle provides SIBs that realize control flow patterns such as conditional constructs (“if” and “switch”) and loops. There are also building blocks for the basic support of application-level logging and performance measurement. Due to their elementary character, most SIBs in this bundle are very generic and represent rather fine-grained functionality.

Graph Model SIBs: This bundle provides building blocks for handling SLGs and hence is relevant to code generators that are built for jABC (cf. Chap. 5). It includes SIBs for loading and traversing hierarchical SLGs as well as for retrieving information from contained SIBs (e.g., SIB labels, parameters, branches, outgoing edges, successors in the SLG, or user objects). Due to this reflective character, the bundle can be considered a metamodel API for accessing SLGs and their constituent parts. The more than 60 SIBs of this bundle were originally a part of Genesys, but have been contributed to the Common SIBs in order to enable SLG accessibility for other jABC applications. Please note that in order for a code generator to be able to deal with another modeling language than SLGs, another SIB bundle suitable for the new modeling language has to be employed. This is exemplified by the case study presented in Chap. 7, which required the development of a SIB bundle for accessing EMF models.

IO SIBs: These SIBs mainly support dealing with files and directories, e.g., creating and scanning directories or writing text files. It is used by Genesys for writing generated code to actual files. Furthermore, the bundle contains simple building blocks for exception and error display on the console.

Script SIBs: This is a set of more specialized SIBs that enable the execution of scripts as well as the integration of template engines, the latter being particularly relevant to Genesys. Currently, the template engines Velocity [Apa10], StringTemplate [Par04][1] and FreeMarker [Fre11b] are supported. Consequently, the generator developer can freely decide upon an appropriate template engine by just using the corresponding SIB (*Requirement G1 - Platform Independence*). It is also possible to mix several template engines in one code generator model, e.g., in order to benefit from a specific feature of a template language without having to use the corresponding engine for the entire code generator.

Apart from those very general SIBs, which can be used in any jABC application and mostly realize small, fine-grained tasks, there are also services specifically designed for the code generation domain. Several examples of those SIBs, which are part of the Genesys framework (cf. Fig. 4.1) and form a bundle called "Genesys SIBs" [Jör10], are presented in the following sections.

4.1.2 Type Mapping

An essential task of a code generator is mapping the data types found in the source language to corresponding data types of the target language. In middleware techniques like CORBA and Web Services [Pap08], data type mapping usually includes the use of intermediate exchange formats such as the Common Data Representation (CDR) for CORBA or XML dialects like

[1] The `RunStringTemplate` SIB described in Sect. 3.2 is also contained in the "Script SIBs".

SOAP [W3C07] for Web Services. Thus marshalling and unmarshalling is required, i.e., the conversion of the data types into the exchange format and vise versa. In the case of Web Services, e.g., this conversion is performed by XML binding libraries like the Java Architecture for XML Binding (JAXB) [Jav09a].

Type Mapping Scenarios:

For code generators, this task is usually much simpler, as it involves a direct unidirectional translation of the data type to a corresponding text-based representation (e.g., an initializer) in the target language. The overall complexity of this task strongly depends on the given combination of source and target language. For this book, the following cases are distinguished:

1. *Identity:* The exact same data type exists in the source language as well as in the target language. For instance, when translating jABC's SLGs to Java, the data type `java.lang.String` exists in both languages, due to the fact that SIBs are realized as Java classes.
2. *Direct Mapping:* The data type can be mapped to an equivalent type in the target language without any loss or omission of information. For instance, when generating Java code from EMF models, the data type `EString` in Ecore (cf. Sect. 7.2) can be directly mapped to the Java data type `java.lang.String`. Such a mapping does not necessarily have to be injective, as it is possible to map several data types which are different in the source language to one single data type in the target language.
3. *Reductive Mapping:* If the data type has no direct equivalent in the target language, it possibly can be reduced, provided that the source data type contains information that is not required for the generated result. For instance, the built-in data type `ContextKey` in jABC's SLGs (cf. Sect. 3.2.1) has no direct counterpart in any target language. However, it contains a context scope, which might be omitted if the target language does not support a stacked execution context. In this case, `ContextKey` can be reduced to the name of the context key, which can be easily represented by a simple string in the target language. This method only works if the omitted information is irrelevant, otherwise a solution without information loss is preferable. An additive counterpart of the reductive mapping is imaginable, but as no practically relevant examples have been found in the context of Genesys, this case is not considered here.
4. *New Data Type:* There is no equivalent counterpart for a source data type in the target language, neither for a direct nor for a reductive mapping. In this case, a possible solution is the introduction of a new data type in the target language. The corresponding code of the new data type has to be emitted by the code generator. As will be presented in Sect. 5.2, this solution is applied by the code generator which translates SLGs to plain Java code. For every complex jABC data type (cf. Sect. 3.2.1), such as `ContextKey` or `ListBox` which are not built-in Java types, the code generator emits a corresponding counterpart.

5. *Data Type Exclusion:* If neither of the above cases is applicable, a source data type may be excluded from the mapping, at the expense of no longer being able to translate any inputs containing that data type. In most cases, this solution is only a compromise and thus should be avoided. For instance, the `ContextExpression` data type in jABC's SLGs (cf. Sect. 3.3.2) requires an implementation for resolving EL expressions, which is not available for non-Java languages such as Objective-C [Koc09]. Instead of spending effort on providing such an implementation, the exclusion of the `ContextExpression` data type might be a pragmatic alternative.

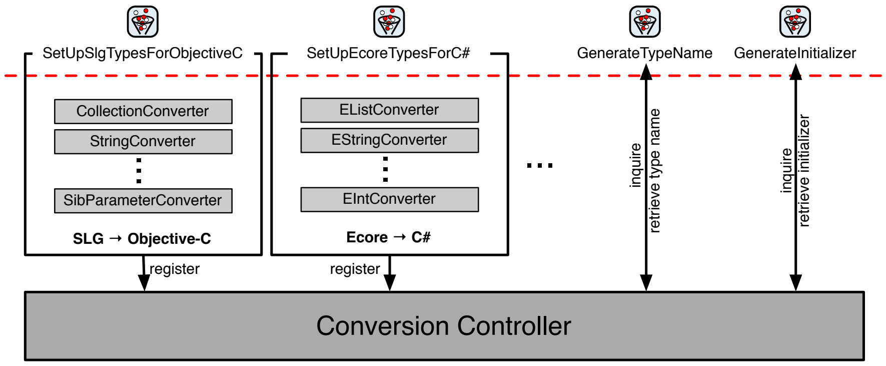


Fig. 4.2. Data type mapping infrastructure in Genesys

Type Mapping Services:

Genesys provides a simple infrastructure for establishing data type mappings, which is depicted in Fig. 4.2. For each combination of source and target language, a mapping has to be specified by means of a set of *converters*. A converter is responsible for one or more source data types and is able to produce corresponding type names, initializers etc. for the target language. For instance, Fig. 4.2 shows a data type mapping from SLGs to Objective-C, containing a *String Converter*. For a given `String` instance found in an SLG, this converter is, among other things, able to produce a corresponding type name (`NSString`) or initializer (`@"myStringValue"`[2]), which is used by the code generator.

All registered converters are managed by the *Conversion Controller* (bottom of Fig. 4.2). Given a concrete primitive value or object instance found in the source language, the controller determines the responsible converter, applies it and returns the result. In order to keep the determination mechanism

[2] `@"myStringValue"` is Objective-C's way of creating a new constant string object with the value `myStringValue`.

simple, each possible data type in the source language has to be assigned to exactly one converter. While multiple data types may be handled by one converter, it is not allowed to assign more than one converter to a data type.

From the perspective of the generator developer, the data type mapping infrastructure is accessed via simple services shown at the top of Fig. 4.2. In the initialization phase of the code generator, the system has to be set up by registering required converters, which is performed by a dedicated, usually parameterless, SIB (e.g., `SetUpSlgTypesForObjectiveC`). Such a SIB and its corresponding converters have to be created for every new combination of a source and a target language, either by the generator developer himself or by a SIB expert. Technically (and transparent to the generator developer), the SIB registers all required converters with an instance of the conversion controller, which is then stored in the execution context (cf. Sect. 3.3.2) so that it can be accessed by other SIBs.

After this initialization step, the generator developer may use the SIBs `GenerateTypeName` and `GenerateInitializer` to apply the data type mappings for the code generation. Table 4.1 shows the parameters of those services that are available to the generator developer. Besides context keys for the input and output, especially the `GenerateTypeName` SIB provides several additional configuration flags that, e.g., allow the omission of any package or namespace information in the resulting type name. Except for the case in which the generator developer writes converters by himself, everything below the dashed line in Fig. 4.2 is transparent to him and virtualized by the SIBs.

Table 4.1. Services for data type mapping

GenerateInitializer		Generates an initializer string for the given object/primitive value.
Parameters	object	Context key for reading the object.
	initializer	Context key for storing the generated initializer.

GenerateTypeName		Generates a type name string for the given object/primitive value.
Parameters	object	Context key for reading the object.
	typeName	Context key for storing the generated type name.
	generateSimpleTypeName	If this flag is set to true, the simple type name (without the package or namespace information) will be generated.
	preferInterface	If this flag is set to true, the type name for a preferred interface will be generated (e.g., `java.util.List` instead of `java.util.ArrayList`), which is useful for declarations.

4.1.3 Identifier Generation

Another important task of code generators is the output of valid identifiers. Identifiers are tokens that name elements in programming languages such as classes, variables, labels or methods. The generation of appropriate identifiers depends on the given target language, as each programming language defines its own syntactic restrictions for correct identifiers. For instance, a valid identifier in Java is composed of characters such as lowercase or uppercase ASCII Latin letters, digits, underscores or dollar signs, but must not begin with a digit, contain any blanks or have the same spelling as a reserved language keyword such as "`public`" [Gos+05]. Furthermore, some programming languages even assign semantics to single characters in identifiers. In Perl, identifiers beginning with a dollar sign (`$`) indicate scalars, and those starting with the percent sign (`%`) denote hashes [Wal00]. As another example, in Ruby, variables with identifiers that start with an upper case letter are considered immutable [FM08]. Thus with each new target language, identifier generation needs to be specified appropriately.

Apart from the syntactic restrictions, a central characteristic of identifiers is their uniqueness. For example, in most programming languages it is forbidden to declare two variables with the same identifier within the same scope. When generating code from models, identifiers are often produced from the names of model elements, which usually are subject to various syntactic restrictions, or which are not regulated at all. For instance, the labels of SIB instances in jABC's SLGs are not restricted and thus may contain blanks, or even may equal reserved keywords of a target language. Simply using such names as identifiers and ignoring the rules of the target languages may lead to faulty generation results, such as code which is not compilable or which yields unexpected execution behavior.

In order to cope with this recurring task, the Genesys framework provides a backend for identifier generation, which can be accessed by corresponding services. In essence, this backend keeps track of a blacklist of reserved and previously used identifiers. When a new identifier is given, this blacklist is checked: If the new identifier is already on the blacklist, then it is *unified*, e.g., by adding a suffix like "`_<serial number>`", and finally added to the blacklist. Otherwise, the unmodified identifier is added to the blacklist. For instance, if "`public`" is given as an identifier and it is contained in the blacklist due to being a reserved keyword, it will be unified to "`public_1`", which can be safely used in generated code.

In the initialization phase of a code generator, reserved identifiers such as keywords of the target language can be added to the blacklist via the SIB `RegisterReservedKeywords`. At this point it is necessary to distinguish between two kinds of identifiers that occur in code generation: those which are derived from the code generator's input (*generated identifiers*) and those which are fixed, e.g., because they are hard-coded in a template (*fixed identifiers*). In contrast to generated identifiers, fixed identifiers are static at generation time

Table 4.2. Services for identifier generation

RegisterReservedKeywords		Creates a blacklist containing keywords and prefixes which are forbidden to be used for identifiers in generated code.
Parameters	reservedKeywords	Set containing the reserved keywords.
	reservedPrefixes	Set containing prefixes for reserved keywords. All words starting with one of these prefixes are not allowed.
UnifyString		Unifies a given string.
Parameters	string	Context key for reading the string that should be unified.
	uniqueString	Context key for storing the unified string.
GenerateJavaIdentifier		Converts a given string into a valid Java identifier.
Parameters	string	Context key for reading the string that should be converted.
	identifier	Context key for storing the resulting identifier.
GenerateUniqueJavaIdentifierForSlg		Generates a unique Java identifier for the given SLG, derived from its name.
Parameters	model	Context key for reading the SLG.
	uniqueIdentifier	Context key for storing the resulting unique identifier.

and are specified by the generator developer when modeling a code generator. In order to avoid clashes between such generated and fixed identifiers, the generator developer might add each fixed identifier to the blacklist as a separate reserved keyword. As this is rather uncomfortable, it is, apart from prohibiting entire words, also possible to define reserved prefixes. Each identifier starting with a reserved prefix is treated as if it would already be contained in the blacklist. Consequently, as a convention for the generator developer, any fixed identifier has to start with such a reserved prefix (e.g., "`cg_`").

After initializing the blacklist, any given string may be unified by using the SIB `UnifyString`. This SIB expects a string in the execution context, invokes the backend for identifier generation to unify it, and finally stores the resulting unique string in the execution context for further usage. However, before a string can be unified, it has to be converted to a valid identifier, adhering to the rules of the target language for which the code is generated. Consequently, for each different target language, a corresponding specific service is required.

For instance, the SIB `GenerateJavaIdentifier` converts a given string into a Java identifier according to the rules outlined above. Just as with the converters for data types described in Sect. 4.1.2, such a specific SIB has to be created if it does not exist yet, either by the generator developer himself, or by a SIB expert. However, once the SIB has been created, it can be reused for every code generator that targets the corresponding language (*Requirement G2 - Reusability and Adaptability*), which is a major benefit of service orientation.

Optionally, even more specific SIBs may be added as desired, e.g., to provide more convenient identifier generation for specific combinations of source and target languages (*Requirement S1 - Domain-Specificity*). As an example, the SIB `GenerateUniqueJavaIdentifierForSlg` creates an identifier from a given SLG by extracting the name of the SLG, converting it to a Java identifier and then unifying it in one step. Furthermore, it additionally caches all identifiers that have already been generated for SLGs, so that executing the service repeatedly for the same SLG always yields the same identifier. This is very useful if the identifier occurs at multiple places in the generated code.

Table 4.2 sums up the services presented in this section.

4.1.4 Variant Management

When reusing existing code generators as drafts or templates for building new ones, it is an obvious approach to start with a code generator that is as similar as possible to the one that should be developed. For instance, in the context of code generators for jABC, it was easy to derive a code generator for Java Servlets [Jav11b] from an existing code generator for ordinary Java classes with only few modifications (cf. Sect. 5.4.1), as the structure and generated output of both is very similar.

Genesys is designed to facilitate software product line engineering [CN01; PBL05] (*Requirement S3 - Variant Management and Product Lines*). Using the terminology established in this realm, the models and services contained in the Genesys framework are *core assets* that form the basis for building *product lines* and deriving *variants*.

Accordingly, appropriate tool support is required that enables specifying and managing variability, e.g., via the definition of *variation points* [PBL05, p. 62]. The Genesys framework enables this by offering a dedicated service. Please note that as its implementation was driven by the demands raised during the realization of Genesys, this service does not exploit the full potential of applying product line engineering in jABC yet – Sect. 10 elaborates on further prospects and possibilities.

The service enables the specification of variants on the basis of aspect orientation (AO) [Kic+97] provided by jABC's hierarchical modeling facilities. As described in Sect. 3.2.2, hierarchical modeling is performed in jABC via macros, which enable embedding SLGs into each other. On this basis, models can act as reusable aspects (*Requirement G4 - Separation of Concerns*): For managing variability, macros are used as variation points (or *joinpoints* in AO nomenclature) to which multiple submodels may be assigned, each of them representing a single variant. This is an extension of jABC's standard

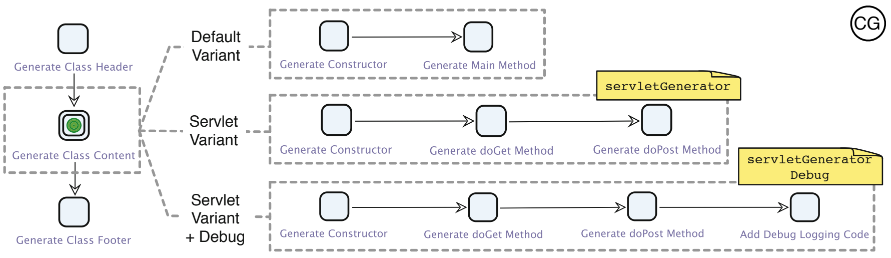


Fig. 4.3. Specifying variants via hierarchical modeling

semantics for hierarchical models, which normally only allows assigning exactly one submodel to a macro.

Fig. 4.3 shows a simplified and schematic example of this concept. On the left hand side, there is a simple model that represents the generation of a Java class: First, the header of the class is generated, followed by the content, and finally the remainder of the class is added. The SIB for generating the class content is a macro and thus a potential variation point.

In order to obtain a complete model that is well-formed in terms of executability, at least one submodel, which represents the default behavior, has to be assigned to the macro. In Fig. 4.3, this submodel is labeled "Default Variant" and generates the content of an ordinary Java class, consisting of a constructor and a `main` method[3].

The modeler may now specify further variants, which are kept apart by unique names. The example in Fig. 4.3 shows two variants. The first ("Servlet Variant") with the unique name `servletGenerator` produces the content of a Servlet class by generating a `doGet` and a `doPost` method[4] instead of the `main` method defined in the default behavior. The second ("Servlet Variant + Debug") with the unique name `servletGeneratorDebug` extends the Servlet variant by adding extra code for debugging.

Please note that the unique name of a variant does not necessarily refer to one single variant model only. Instead it *globally* identifies a variant (or product line) of an entire SLG hierarchy (or even several hierarchies). For instance, the variant name `servletGeneratorDebug` in Fig. 4.3 may refer to a set of variant models that are associated with variation points distributed over the SLGs hierarchy.

As the assignment of multiple submodels to a macro is not supported by jABC, this feature is added by the Genesys jABC plugin (see Sect. 4.3.2). Please note that technically, only the submodel that represents the default behavior is a physical part of the model hierarchy. All other variants assigned to a macro are just attached to it as additional information, but are not incorporated into the model hierarchy.

[3] `main` is a dedicated method used for starting the execution of Java programs.
[4] See the Servlet specification [Jav11b] for details on the `doGet` and `doPost` methods.

Table 4.3. Service for the generation of variants

BuildVariant		Builds the variant identified by the given name for the given models.
Parameters	models	Context key for reading the models that will be transformed.
	variantName	The unique name of the variant that will be built.
	transformedModels	Context key for storing the transformed models.

For achieving this, the specified variants are automatically generated by means of an accordingly equipped generator generator. To this end, the generator generator needs to incorporate the `BuildVariant` SIB (see Table 4.3), which builds a variant via a simple model-to-model transformation. As its input, the service requires the unique name of the variant that will be built along with the list of models to be transformed. For each macro contained in each of the given models, the service checks whether a variant with the given unique name is assigned to it. If so, the detected variant is set as the default submodel of the macro (thus now representing the new default behavior). Otherwise the macro is not modified, leaving the default behavior unchanged. The role of this model-to-model transformation is very much comparable to the one of aspect weavers [Kic+97] in AO, except that the weaving is performed on the model level (*model weaving* [Béz+04]). After this transformation step, the generator generator translates the models, which now represent the desired variant of a code generator, to code.

The current implementation of the variant management service imposes one restriction on models in order to be suitable as variants: They have to provide the same model interface (i.e., model parameters and branches) as the default variant. This is due to the fact that there is currently no support for separately parametrizing variants – the implementation of a corresponding GUI will remove this restriction (cf. Sect. 10 for details).

A more complex example showing the application of variant management is the Java Class Generator for jABC, which will be presented in detail in Sect. 5.2. This generator offers different generation strategies, each of them realized by a corresponding variant.

4.2 Simple Example: Documentation Generator

In order to give the reader an idea of modeling a code generator with Genesys, the following sections present a complete example of a simple code generator called the "Documentation Generator". For the most part, this example is based on a tutorial introduction which originally has been published in [JSM10]. The Documentation Generator is designed to be used in jABC, i.e., SLGs are its source language. The generator's task is to produce an HTML documentation website (comparable to the output of Java's Javadoc Tool) from those models according to the following requirements:

1. The generator should process all models in a given directory.
2. For each model, a separate HTML page should be generated, containing the following information:
   - the documentation of the model and
   - a list of all SIB instances contained in that model. Each list entry should display the corresponding SIB's label. Furthermore, each entry should be linked to a detail page (described in 3) containing the documentation of the particular SIB instance, as well as to the corresponding online SIB documentation (e.g., [TU10]).
3. For each SIB instance in each SLG, a detail HTML page will be generated, displaying the SIB instance's documentation. This page should be linked to the corresponding model page.
4. An index page should be generated, listing all processed models along with links to their respective model pages.
5. Each generated HTML page should contain a timestamp in order to retain the time of the last generation.

Based on these requirements, the code generator is modeled by a generator developer who has to have knowledge of using jABC and Genesys, of SLGs (the source and implementation language) along with their associated concepts such as SIBs and branches, and of the HTML format (the target language). The following sections will show how to model the complete Documentation Generator, which is built almost entirely on the basis of jABC's Common SIB library. For the sake of simplicity, not all the parameterizations of the employed SIBs will be explained in detail, but instead the descriptions will focus on which SIBs are used to solve the task and how they are connected to each other. For each employed SIB, the corresponding class will be named, so that it can be easily related to the online documentation [TU10], which contains detailed information for all SIBs. If no class is given, then the name of the SIB class equals the SIB label displayed in the model.

4.2.1 Structuring the Generation Process

Modeling a code generator with Genesys usually proceeds top-down. Accordingly, as the first step of modeling the Documentation Generator, the generation process is divided into two abstract coarse-grained phases: the *initialization phase* and the *generation phase*. In the initialization phase, the code generator will set up the generation by verifying the input parameters and loading the input SLGs, and in the generation phase the actual HTML website is produced.

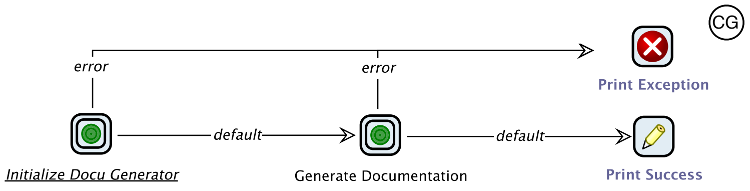


Fig. 4.4. The `Docu Generator` main model (topmost hierarchy level)

Fig. 4.4 shows the resulting model, containing the SIB `Initialize Docu Generator` for the initialization phase and `Generate Documentation` for the generation phase. Both SIBs are macros (SIB class `MacroSIB`), and both phases will be refined and concretized in the following. Please note that all models of the Documentation Generator only use `MacroSIB`s (cf. Sect. 3.3.3) as macros, i.e., a flat execution context is employed (cf. Sect. 3.3.2). Along with the two macros, the model contains two other SIBs emitting either a success message (`Print Success`, SIB class `PrintConsoleMessage`) when the two phases have been finished successfully, or an error message (`Print Exception`) if anything failed during the execution of the code generator.

Note that in this example, the error handling is in most cases delegated to the main model depicted in Fig. 4.4 (*error delegation*). All SIBs used in the Documentation Generator's models have "error" branches, most of which lead to the SIB `Print Exception`, either as direct edges in the main model, or as model branches in all the other models. Consequently, `Print Exception` is the central (though very simple) error handling step for the entire generator. Another (but more rare) way of handling errors is *in-place handling*, whereby the handling is usually performed by subsequent SIBs (or even models) in the same model. This distinction between in-place handling and error delegation is very much comparable to exception handling in Java, where a caught exception may either be handled directly or thrown again in order to be handled somewhere else.

4.2.2 The Initialization Phase

Subsequently, the initialization phase is concretized. Basically, this phase has to verify the input parameters provided by the user and to set up the generation process. The Documentation Generator will have two input parameters:

outputFolder, the absolute path to the output directory for the generated HTML files, and

modelPath, a list of absolute paths to directories containing the jABC models for which the documentation should be generated.

Fig. 4.5 shows the refined model for the initialization phase.

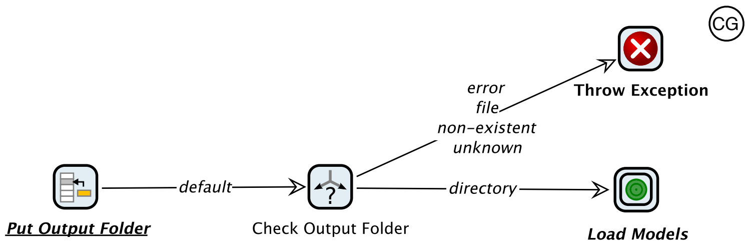


Fig. 4.5. The `Initialize Docu Generator` model (second hierarchy level)

The generator starts by processing the parameter "outputFolder", whose value is first put into the execution context (`Put Output Folder`, SIB class `PutFile`) in order to be accessible by the following SIBs. Afterwards, the SIB `Check Output Folder` (SIB class `CheckPath`) verifies this value: If it does not denote a proper (i.e., existent and writeable) directory, the following step `Throw Exception` issues an error. Otherwise, the initialization phase continues with handling the second input parameter "modelPath" (macro `Load Models`), which is again performed in a submodel.

Referring to the different error handling techniques described above, the SIB `Throw Exception` is an example for in-place handling of errors. In this case, the exit branch of a preceding step (`Check Output Folder` in Fig. 4.5) reflects an undesired result. When such an error is detected, it is directly handled by `Throw Exception`, which performs the error handling in-place at the service level, rather than delegating it to a higher model hierarchy level.

The submodel referenced by `Load Models` is displayed in Fig. 4.6. As loading SLGs is a standard task for many code generators (at least for those dealing with SLGs as their source language), this model is, among many others, contained in the Genesys framework and thus can be entirely reused for the Documentation Generator without any changes. A detailed discussion of this loading process is not required at this point: The generator developer does not have to know the technical details of loading SLGs anyway, as from his perspective, the model is used just like a ready-made service in a black box fashion.

4.2.3 The Generation Phase

For modeling the generation phase, the macro `Generate Documentation` in the main model (Fig. 4.4) is refined. Fig. 4.7 shows the resulting model. The generation process starts with producing the static header of the index page (`Generate Index Header`) using StringTemplate (see Sect. 2.4.2). Please note that all SIBs with "ST" on their icon are instances of the SIB class `RunStringTemplate` described in Sect. 3.2.1. The header of the index page only consists of static text, for instance, containing the opening `html` and `body` tags for the document. Afterwards, a time stamp is generated (`Generate Time Stamp`, SIB class `GetTimeStamp`), which is inserted into the footer of

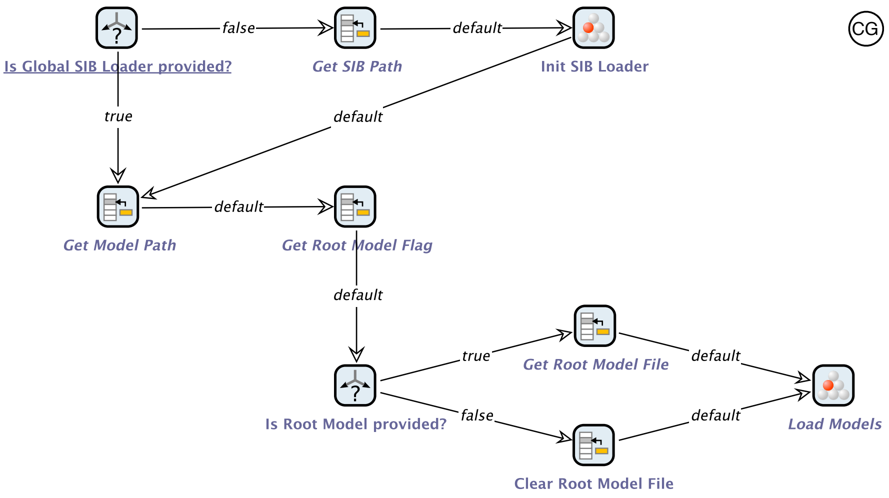


Fig. 4.6. The `Load Models` model (third hierarchy level)

each generated HTML page. The generation of the index page content and the detail pages for the models and the contained SIBs is again modeled in a submodel (referenced by the macro `Generate Model Pages`). After the detail pages are produced, the generator finalizes the index page. For this purpose, the SIB `Generate Index Footer` is parameterized with the following simple template:

```
    </ul>
    <hr>
    <i>Generated: $timeStamp$</i>
  </body>
</html>
```

Besides some closing tags, this template contains a placeholder called "timeStamp", which is enclosed by dollar signs. When the generator is executed, StringTemplate replaces this placeholder by the timestamp produced by the step `Generate Time Stamp`. The generation phase finishes with writing the index page to a file (`Write Index Page`, SIB class `WriteTextFile`).

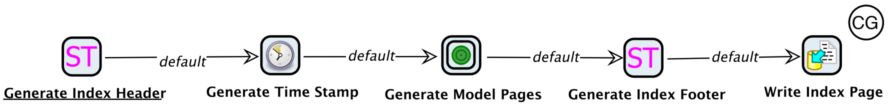


Fig. 4.7. The `Generate Documentation` model (second hierarchy level)

The submodel that refines the macro `Generate Model Pages` is depicted in Fig. 4.8. It starts by iterating over all input SLGs that have been loaded in the initialization phase (`Next Model`, SIB class `IterateElements`). Please note that the Documentation Generator produces HTML pages for *all* SLGs in a given directory, which particularly includes all referenced submodels. Consequently, we do not need to expand the macros or to use any recursion - a simple iteration of the models is sufficient. As long as there are still models left to be processed, the "next" branch of the SIB will be used. Otherwise, the execution proceeds with the parent model (Fig. 4.7), which is connected via a model branch (i.e., the "exit" branch of `Next Model` is exported as a model branch that leads to the parent model). The following step `Update Model Counter` (SIB class `UpdateCounter`) keeps track of a model number that is incremented each time the SIB is executed. This number is required to construct the names for the model detail pages. Then the generator extracts some information from the current model. The SIB `Get Model Name` stores its name in the execution context, and the SIB `Convert Content to Html` reads annotated documentation and converts it to proper HTML markup, which is also stored in the execution context. Now the generator has collected enough information for generating an index page entry for the current model (basically the model name, linked to the model detail page, step `Generate Index Entry`) as well as the header of the model detail page containing the model's documentation and name.

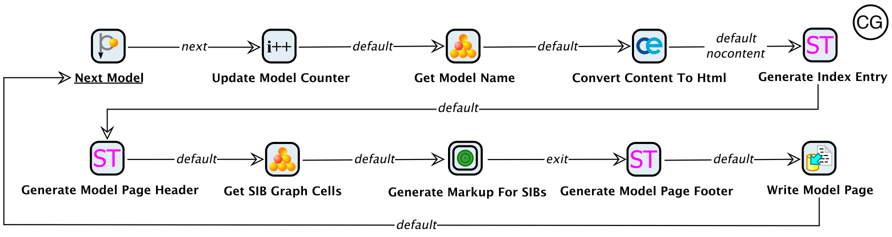


Fig. 4.8. The `Generate Model Pages` model (third hierarchy level)

In order to generate the list of SIBs in the current model along with the SIB detail pages, the generator retrieves all contained SIB instances (`Get SIB Graph Cells`) and then again delegates the production of all SIB-specific HTML markup to a submodel (macro `Generate Markup for SIBs`). Finally, the footer of the detail page is generated (`Generate Model Page Footer`) and the entire page is written to a file (`Write Model Page`).

The last model required for the Documentation Generator refines the macro `Generate Markup for SIBs` and is depicted in Fig. 4.9. In the first step, it iterates over all SIB instances contained in the current model (`Next SIB Graph Cell`, SIB class `IterateElements`), which works just like the `Next Model` step in Fig. 4.8. Furthermore, analogous to the model detail pages, the SIB `Update SIB Counter` (SIB class `UpdateCounter`) keeps track of a SIB counter that is used for the file names of the resulting SIB detail pages. Then again, some information is collected from the current SIB found

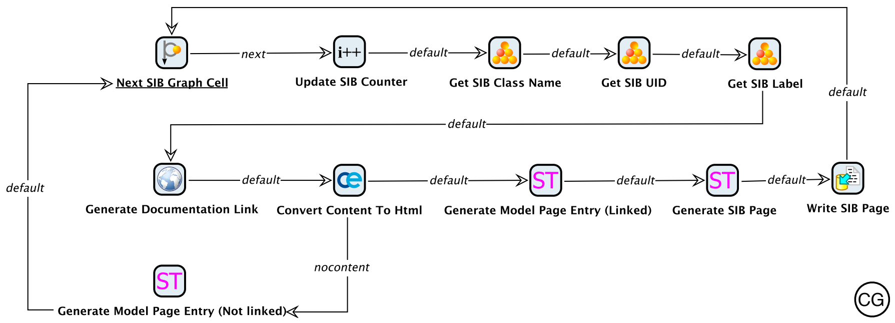


Fig. 4.9. The `Generate Markup for SIBs` model (fourth hierarchy level)

in the execution context: its class name (`Get SIB Class Name`), unique identifier (`Get SIB Class Name`) and instance label (`Get SIB Label`).

The following SIB `Generate Documentation Link` differs from the other SIBs used in the Documentation Generator, as it is the only one that calls a remote functionality, in this case a Web Service. This Web Service takes a SIB's class name and UID (cf. Sect. 3.2.1) as input and uses this information to construct the URL of the online documentation [TU10] that describes the SIB class. As such a SIB was not provided by the Common SIBs, it had to be created by implementing an appropriate service adapter (cf. Sect. 3.2.1), that realizes the communication with the already existing Web Service. This implementation was an easy one-time task.

In the following step, the code generator reads the documentation annotated to the current SIB (`Convert Content To Html`). Depending on whether such a documentation could be found, a list entry for the current SIB on the model page is generated. In case a documentation exists, this entry is linked to a SIB detail page which is generated in the step `Generate SIB Page`. This SIB is parameterized with the following template:

```
<html>
  <body>
    $sibDoc$
    <a href="model_$modelCounter$.html">back to "$modelName$"</a>
    <hr>
    <i>Generated: $timeStamp$</i>
  </body>
</html>
```

Again, the static text contains placeholders that are replaced by StringTemplate, using information collected by the code generator:

sibDoc: The current SIB's HTML documentation retrieved by the `Convert Content To Html` step in Fig. 4.9.

modelCounter: The number of the current model assigned by the SIB `Update Model Counter` in Fig. 4.8.

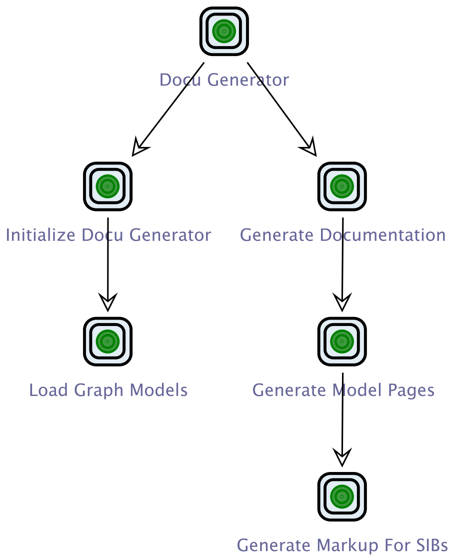


Fig. 4.10. The model hierarchy of the Documentation Generator

modelName: The name of the current model retrieved by the SIB `Get Model Name` in Fig. 4.8.
timeStamp: The time stamp produced by the SIB `Generate Time Stamp` in Fig. 4.7.

The usage of this information in the template shows how SIB instances in submodels can easily access information left in the execution context by SIB instances at arbitrary levels of the model hierarchy, which is due to the flat nature of the execution context.

Finally, if a SIB detail page has been generated, it is also written to a file (`Write SIB Page`).

4.2.4 Finalizing the Generator

In summary, the demonstrated models constitute a complete code generator according to the requirements listed above. The resulting generator consists of six models (five new, one could be reused from Genesys' model library), containing 43 instances of 23 different SIBs. Only one SIB had to be implemented, as the rest of the required functionality could be covered with existing ones. The resulting model hierarchy (see Fig. 4.10) spans four levels.

While modeling a code generator, it is possible at any time to execute, debug and test it using jABC's Tracer (Sect. 3.3). However, for productive use of the code generator, it should be translated to code itself, for instance in order to be able to use it via the Genesys jABC Plugin or to integrate it into a Maven-based tool-chain (both options will be described in more detail

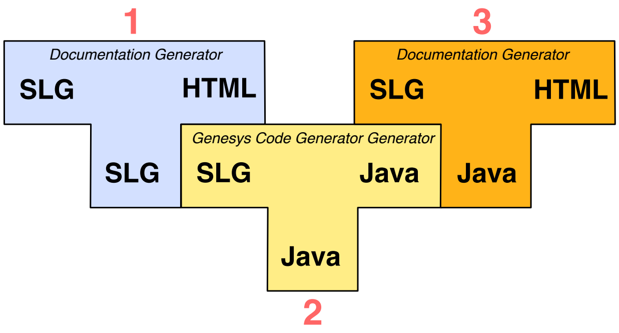


Fig. 4.11. Translating the SLGs of the Documentation Generator to Java code

in Sect. 4.3.2). This finalization of the code generator usually consists of two steps: *editing the generator's metadata* and finally *generating the generator*. The former includes metadata such as the generator's name, version, author and usage documentation, and is edited via a corresponding GUI that is part of Genesys' developer tools presented in Sect. 4.3.1.

In the second step, a corresponding generator generator is responsible for translating the generator models into code. The T-diagram depicted in Fig. 4.11 shows that for the Documentation Generator, this translation is performed by means of the *Genesys Code Generator Generator* (see Sect. 5.2.6). It translates a set of given generator models to Java code that contains all necessary information (e.g., metadata and corresponding interface implementations) to be useable with the appropriate tools named above.

4.2.5 General Remarks on the Example

The example above demonstrated how a code generator is modeled with Genesys. Several aspects of the example are particularly noteworthy as they illustrate some characteristics of the general approach. In particular, the SLGs of the Documentation Generator show that there is a strict separation between the generation logic and the output description (cf. Sect. 2.4), as demanded by *Requirement S4 - Clean Code Generator Specification*. The generation logic is given by the code generator SLGs, and the output description is given by the templates, which are parameters of dedicated SIBs such as `RunStringTemplate`.

In order to achieve this strict separation, it is not advisable to describe parts of the generation logic in the templates. This would be easily possible, as most template languages also support control structures like conditionals, loops or function calls (cf. Sect. 2.4.2). However, describing parts of the generation logic in the code generator SLG and other parts in the templates has several serious drawbacks. First, the models are more difficult to understand as the generation logic of the code generator cannot be fully grasped by just looking at the flow of actions in the SLGs. Second, as parts of the generation logic are hidden in templates and not modeled explicitly, they are not considered by tools such as a model checker (see Sect. 3.4), thus impeding a proper verification of the code generator.

Consequently, the use of templates in Genesys is mostly restricted to those facilities of a template language that allow accessing data (e.g., placeholders or simple expressions). Employing advanced control structures is discouraged, as they most likely would be used to describe logic which should rather be modeled explicitly by means of corresponding SIBs provided by Genesys. Likewise, templates should not be misapplied for making function calls. Instead either an existing SIB realizing the function should be used or, if no such SIB exists yet, a new one should be created in order enable the reusability of the function.

Accordingly, the templates shown in the previous sections are not particularly simple examples. Instead they are characteristic of how templates are generally used in Genesys.

The Documentation Generator also exemplifies the mixed application of source-driven and target-driven transformation. The generator is target-driven as, apart from the actual templates, its generation logic roughly follows the structure of the output. For instance, the headers of the single HTML pages are always generated before the footers (see, e.g., Fig. 4.7 and 4.8). At the same time, the generator can be considered source-driven, because its generation logic avoids multiple traversals: Each model and each contained SIB is only visited once, and when processing a model or SIB, all required output is produced at once, so that no additional visit is necessary. This is, e.g., visible in the SLG shown in Fig. 4.9, which generates an entry for the model detail page as well as the SIB detail page for a particular SIB instance.

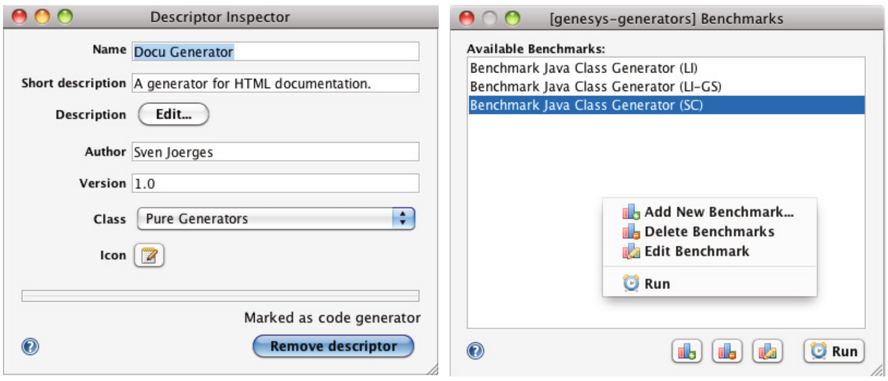
Furthermore, this flexibility of structuring the transformation allows the Documentation Generator to produce multiple output files without any problems, thus overcoming the typical inefficiencies attributed to template-based code generators (cf. Sect. 2.4.2).

4.3 Genesys Tooling

As outlined at the beginning of this chapter and depicted in Fig. 4.1, Genesys provides tools supporting generator developers as well as generator users. The following sections briefly introduce those tools.

4.3.1 Developer Tools

The developer tools provide facilities that assist the generator developer in building code generators. They are realized as jABC plugins, but are not necessarily restricted to code generators for jABC (i.e., having SLGs as their source language).



Name: Docu Generator
Short description: A generator for HTML documentation.
Author: Sven Joerges
Version: 1.0
Class: Pure Generators
Marked as code generator

Available Benchmarks:
Benchmark Java Class Generator (LI)
Benchmark Java Class Generator (LI-GS)
Benchmark Java Class Generator (SC)

Fig. 4.12. Left hand side: Inspector for editing a code generator's metadata, Right hand side: Setting up a benchmark

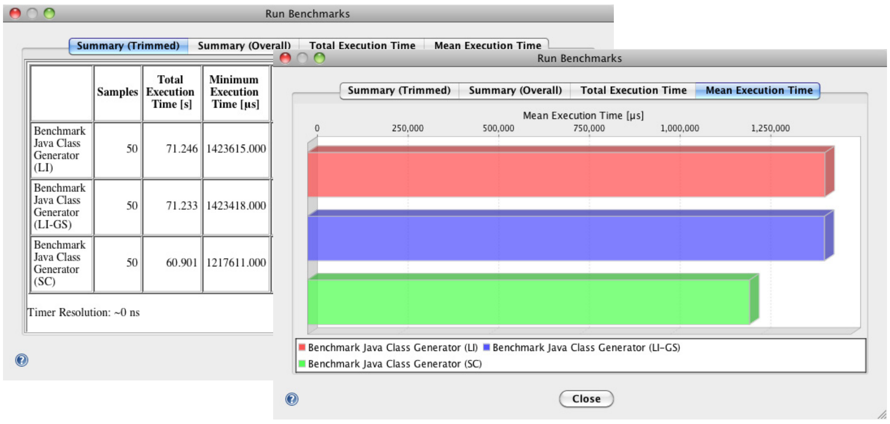
Descriptor Inspector:

The *Descriptor Inspector* allows the generator developer to add metadata to a code generator. This includes information such as the name of the code generator, a short and a long description, the author's name, a version, the category of the code generator as well as an icon. This metadata is attached to the topmost model of the code generator as a user object (cf. Sect. 3.2.2) and serves multiple purposes. First, generator generators may incorporate the metadata into a generated version of the code generator and thus may allow tools using the code generator to display the information to users. For instance, when translating a code generator for jABC using the *Genesys Code Generator Generator* (see Sect. 5.2.6), any metadata is added to the resulting Java class. When using a code generator translated this way in jABC, the information is displayed to the user by means of the Genesys jABC Plugin presented in the next section. Second, the metadata may be used to automatically generate documentation (e.g., an HTML website) for the code generator. Fig. 4.12 (left hand side) shows the contents of the Descriptor Inspector for the Documentation Generator presented in Sect. 4.2.

Benchmark Framework:

By means of the *benchmark framework*, a generator developer is able to compare different code generators or generator variants in terms of the performance of their generated results. For instance, one could compare the two Servlet Generator variants exemplified in Sect. 4.1.4 in order to examine whether the addition of extra debugging code negatively influences the performance of generated results. The benchmark framework performs this comparison by

1. using each participating code generator to translate a set of input models which act as objects of investigation,



	Samples	Total Execution Time [s]	Minimum Execution Time [µs]
Benchmark Java Class Generator (LI)	50	71.246	1423615.000
Benchmark Java Class Generator (LI-GS)	50	71.233	1423418.000
Benchmark Java Class Generator (SC)	50	60.901	1217611.000

Timer Resolution: ~0 ns

Fig. 4.13. Benchmark results visualized in tabular or bar chart form

2. executing the result of each generation (after eventually compiling it, if necessary),
3. measuring the time duration of each execution and
4. finally visualizing the measurements.

For setting up a benchmark, the generator developer first creates a configuration for each participating code generator. Such a configuration includes information such as the code generator that will be used, the input models which are translated to code as well as a so-called *execution runner* that specifies how the generated code is executed. The latter strongly depends on the code generator that is used, as, e.g., a Java class with a main method is executed in a different way than a Servlet. Technically, an execution runner is a Java class following a simple interface, which has to be implemented for every generation result that should be supported by the benchmark framework.

Furthermore, the generator developer may also specify the number of runs for a configuration. This avoids benchmarks which execute so fast that they are hardly observable, e.g., due to small input models or very high-end host machines. Another motivation for repeated executions is dealing with statistical deviations which might prevent reliable comparison of results, e.g., resulting from possible effects of memory caching, just-in-time compilation or hot-spot optimizations. Fig. 4.12 (right hand side) shows the graphical interface for setting up a benchmark, which contains prepared configurations for different variants of a code generator for Java classes (cf. Sect. 5.3).

After this preparatory configuration, the benchmark can be executed. The benchmark framework follows the procedure described above and then displays the results. As shown in Fig. 4.13, the measurements can be viewed in either tabular form or as bar charts.

Please note that although only the execution time of the generated artifacts is measured, it is nevertheless possible to similarly benchmark the execution performance of the code generators. For setting up such a benchmark, the used code generator has to be a generator generator, that is then applied to the models of the code generators whose performance is to be measured.

VTL Editor:

As many Genesys users choose Velocity (cf. Sect. 2.4.2) as their template engine, the Developer Tools provide a textual editor for templates written in the Velocity Template Language (VTL). This editor allows in-place editing of Velocity templates in jABC with line numbering and syntax highlighting. Please note that although Velocity is specifically supported that way, Genesys is not restricted to any particular template engine.

4.3.2 User Tools

jABC Plugin:

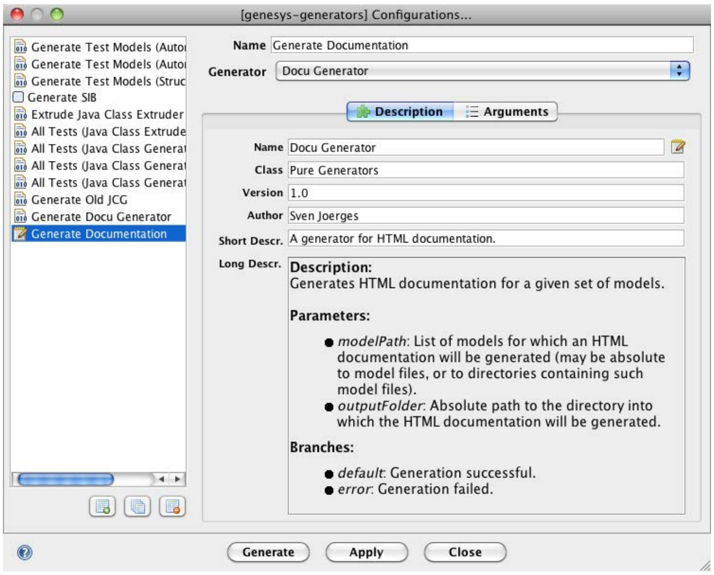
The Genesys Plugin for jABC provides a graphical interface, depicted in Fig. 4.14, for configuring and executing code generators in order to translate SLGs to code. For an existing jABC project, a user may create an arbitrary number of code generator configurations. After selecting the desired code generator, the metadata specified by the generator developer (see Sect. 4.3.1) is displayed and the generator can be configured (e.g., its input models and output directory). The code generation can then be started via this configuration. An integrated console informs the user about the generation progress and about any errors.

Furthermore, apart from generator users in jABC, this jABC plugin is also useful for generator developers. The graphical interface allows them to conveniently generate their code generator models by creating a configuration for an appropriate generator generator. In order for a code generator to be usable with the Genesys Plugin for jABC, it has to be generated with the *Genesys Code Generator Generator*, which is described in more detail in Sect. 5.2.6.

Maven Plugin:

Apache Maven [Apa11b] is a popular and powerful tool for creating build environments. It supports, among other things, building and deploying artifacts, dependency management, automatic testing and release management. All these activities are realized as plugins. In a special XML file, the so-called Project Object Model (POM), such plugins are configured and assigned to particular phases of a build lifecycle managed by Maven.

As code generators are often integral parts of such build environments they need to be compatible with corresponding management tools (*Requirement S6 - Tool-Chain Integration*). For being able to integrate a code generator developed with Genesys into a Maven-based tool-chain, Genesys provides a


[genesys-generators] Configurations...
Generate Test Models (Auto
Generate Test Models (Auto
Generate Test Models (Struc
Generate SIB
Extrude Java Class Extruder
All Tests (Java Class Extrude
All Tests (Java Class Generat
All Tests (Java Class Generat
All Tests (Java Class Generat
Generate Old JCG
Generate Docu Generator
Generate Documentation
Description
Arguments
Generate
Apply
Close

Name: Generate Documentation
Generator: Docu Generator

Name: Docu Generator
Class: Pure Generators
Version: 1.0
Author: Sven Joerges
Short Descr.: A generator for HTML documentation.
Long Descr.:

Description:
Generates HTML documentation for a given set of models.

Parameters:

- *modelPath*: List of models for which an HTML documentation will be generated (may be absolute to model files, or to directories containing such model files).
- *outputFolder*: Absolute path to the directory into which the HTML documentation will be generated.

Branches:

- *default*: Generation successful.
- *error*: Generation failed.

Fig. 4.14. Creating a configuration for using a code generator in jABC

corresponding Maven plugin. In analogy to the Genesys jABC plugin, this Maven plugin is able to execute any code generator that has been translated with the *Genesys Code Generator Generator*. Like any other Maven plugin, it is configured and added to a build environment via a project's POM.

As code generators produced by the *Genesys Code Generator Generator* are just plain Java classes, it is moreover easy to use them with other build management tools such as Apache Ant [Apa11a] or GNU make [Fre11a].

5 Case Studies: Code Generators for jABC

Besides being the basis of the Genesys framework, jABC itself is an application field for code generation. As described in Sect. 3.1, the XMDD paradigm underlying jABC postulates an unidirectional code generation approach that deliberately avoids round-tripping (cf. Sect. 2.4.4). Consequently, jABC is in need of code generation facilities that are powerful enough to support this tenet, and at the same time are easy to use in a way that respects jABC's users, who are typically application experts without deep technical knowhow. Similar objectives can also be found among the basic requirements of the Genesys approach (*Requirement S2 - Full Code Generation*, *Requirement G3 - Simplicity*). Furthermore, as jABC is used in a broad range of very heterogeneous application scenarios (cf. Sect. 3.2), it provides the opportunity of examining and comparing the construction of code generators for very different target platforms, each of them representing a separate case study with its own set of specific requirements.

In fact, the Genesys framework is itself one of jABC's application scenarios, because it is based on jABC. Accordingly, developing a code generator with Genesys is the same as applying jABC for the domain "code generation". At the same time, Genesys is used to develop code generators for jABC. Both projects strongly profit from this "self-application": jABC provides concrete motivation and demand for case studies with Genesys, while each new code generator enriches jABC's functionality and its practical range in terms of supported target platforms.

Consequently, a large number of case studies presented in this book focus on developing code generators for jABC. The source language of these code generators is generally given by SLGs (cf. Sect. 3.2.2). This does not restrict the validity of the case studies whatsoever, as the high diversity of target platforms still allows checking whether the Genesys framework meets the requirements set in Sect. 1.1. However, the case study presented in Chap. 7 also demonstrates the application of Genesys for source languages other than SLGs.

The following sections elaborate on the different code generators that have been developed for jABC. As a full description of all generators goes beyond the scope of this book, the descriptions focus on important aspects that show the application of the Genesys framework as well as the fulfillment of the basic requirements. Sect. 5.1 and 5.2 start off with the development of the first code generator, a generator for plain Java classes, which proceeded in several stages by means of bootstrapping and which gave rise to different generation strategies and variants. Sect. 5.2.6 elaborates on the *Genesys Code Generator Generator* which allows translating code generator models to code, e.g., in order to be useable with the tools provided by Genesys (cf. Sect. 4.3). Sect. 5.4 proceeds with further generators that resulted from other case studies and diploma theses. In particular, the reader will get an impression of how every code generator enriched Genesys' repertoire of services and models, and how the code generators evolved from each other on the basis of reuse (*Requirement G2 - Reusability and Adaptability*).

5.1 Bootstrapping: Java Class Extruder

One of the central ideas of Genesys is the iterative, evolutionary development of code generators based on previously built and thus ready-made services and models (*Requirement G2 - Reusability and Adaptability*). However, at the very beginning of the project, no such services and models existed, let alone any tool support specific to code generation, as presented in Sect. 4.3.

Consequently, the first code generator had to be developed from scratch, using only jABC and its plugins such as the Tracer (cf. Sect. 3.3). This first code generator was the *Java Class Extruder*. It translates a hierarchical SLG[1] modeled in jABC into a Java class which may either be executed via a `main` method or by means of an appropriate API method. The former allows the class to be invoked, e.g., via a command-line interface, and the latter enables embedding the class in other programs, comparable to using a library. The execution behavior of the generated class should coincide exactly with the execution behavior of the SLG executed with the Tracer (*execution equivalence*).

There were two reasons for the choice of Java as the target language of the first code generator. First, all SIBs available at that time (e.g., from other jABC projects) offered a Java implementation of their underlying services in order to support execution with the Tracer. Accordingly, calling the services used in an SLG from correspondingly generated Java code was very simple. Second, as the Tracer could be used as a standalone Java library for the execution of SLGs (without the jABC tool), it was possible to apply the Tracer in the generated code and thus benefit from its features, such as

[1] In the following, the notion "an SLG" is assumed to denote a hierarchical SLG, i.e., an SLG along with all its contained submodels (0 or more).

the execution context, calling the services, parallel execution, or management of SLG hierarchies.

The following sections elaborate on a concept for translating SLGs to Java code that uses the Tracer, and on how the Java Class Extruder has been developed with jABC. Finally, the result is evaluated.

5.1.1 The Extruder Concept

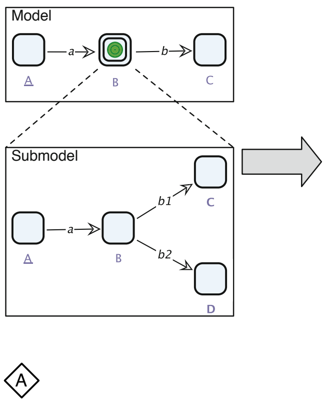
As mentioned above, code generated by the Java Class Extruder uses the Tracer for execution. This considerably shifts the task of the code generator: Instead of translating the flow of actions modeled by an SLG to correspondingly structured code (i.e., a composition of sequential statements, conditional statements and loops), it is sufficient to generate code that is able to reconstruct the SLG's structure in a way that it can be processed by the Tracer.

As the input for starting an execution, the Tracer expects an SLG in its deserialized in-memory representation, which is defined by jABC's `SIBGraph-Model` (see Sect. 3.2.2). Consequently, when code generated by the Java Class Extruder is executed, it reconstructs the `SIBGraphModel` data structure of the original SLGs and passes it to the Tracer. There is no need to generate any code that, e.g., realizes the execution flow, establishes the execution context or calls the single services, because all these features are already provided by the Tracer. Furthermore, no mapping of data types is required: As the data types occurring in SLGs (cf. Sect. 3.2.1) are exactly those expected by the Tracer, it is always the "identity" case (cf. Sect. 4.1.2) that applies.

In other words, the code generator does not care about the execution semantics of the SLG, and thus the generated code does not explicitly contain corresponding statements that realize any execution semantics. Instead the code generator maps the data structure of the SLG to code, and the execution semantics are left to an execution library or engine like the Tracer. In this book, such a code generation approach is called *extrusion*, and a code generator realizing it is called *Extruder*.

Fig. 5.1 illustrates how an SLG is mapped to Java code using this concept. On the left hand side, the figure depicts a hierarchical SLG containing one submodel, that serves as an input for the Java Class Extruder. The right hand side of the figure shows the Java code generated from this input. Please note that both the SLG and the generated code are simplified for illustration purposes.

The code generation results in a single class that is named after the topmost model of the SLG hierarchy. This class contains one method for each model that is contained in the SLG hierarchy (in the example: `createModel` and `createSubmodel`). In the following, these methods are referred to as *model methods*. Each model method is responsible for reconstructing the data structure of the corresponding SLG, as, e.g., shown for the SLG "Model" in lines 4–13 of the generated code in Fig. 5.1.



```
 1 public class Model {
 2
 3   private SIBGraphModel createModel() {
 4     SIBGraphModel model = new SIBGraphModel();
 5
 6     SIB A = new SIB("A", new SibA(...), ...);
 7     A.setIsStartSib(true);
 8     model.add(A);
 9     SIB B = new SIB("B", createSubmodel(), ...);
10     [...]
11     model.connect(A, B, "a");
12     [...]
13     return model;
14   }
15
16   private SIBGraphModel createSubmodel() {
17     [...]
18   }
19
20   public Map<String,Object> execute() {
21     return Tracer.execute(createModel());
22   }
23
24   public static void main(String[] args) {
25     new Model().execute();
26   }
27 }
```

Fig. 5.1. Generation concept of the Java Class Extruder

First, a new empty `SIBGraphModel` data structure is created. Afterwards, all contained SIBs are created as instances of the generic container `SIB`. This container collects information such as the SIB's name and label, and the Java class that implements the SIB (e.g., `SibA` in line 6). Macros like `B` in the SLG "Model" are not implemented by a Java class, but are instead assigned to the model method that reconstructs the corresponding submodel (e.g., `createSubmodel` in line 9).

Subsequently, the SIB instances are added to the data structure (line 8) and connected via edges labeled with the corresponding branch names (line 11). Further information usually contained in the generated code, but not displayed in the simplified example, includes the configuration of SIB and model parameters as well as the assignment of branches.

Besides the model methods, the code contains a `main` method and an API method (`execute`, lines 20–22), which allow the execution of the class. For this purpose, the Tracer is applied to the data structure created by the model methods, as visible in line 21. The map returned by the `execute` method (line 20) represents the execution context and, for instance, allows the retrieval of computed results after the execution.

Fig. 5.1 does not illustrate the translation of model parameters, which nevertheless play an important role for code generation. As a reminder, model parameters enable the configuration and reuse of entire SLGs (see Sect. 3.2.2). When generating code from such a configurable SLG, its model parameters are used as parameters of the corresponding model method. For instance, for a model called "MyModel" with two model parameters "param1" of type String and "param2" of type Integer, the signature of the corresponding model method will look like this: `createMyModel(String param1, Integer param2)`.

That way, the generated code is kept more compact by avoiding the generation of redundant model methods. Even if an SLG is reused multiple times across the given SLG hierarchy, the generated code will contain only one configurable model method for this SLG. The model parameters of the topmost model in the SLG hierarchy (e.g., "Model" in Fig. 5.1) are a special case, as they additionally determine the parameters of the whole Java class that is generated. Such model parameters are translated to parameters of the `execute` API method and are also expected as command-line arguments when using the `main` method.

Although the Extruder approach to code generation is simple yet powerful due to the sophisticated features of the Tracer, it is not applicable for every code generator. Sect. 5.1.3 elaborates on that.

5.1.2 Development of the Generator

Following the concept of translating SLGs to Java code described in the previous section, the Java Class Extruder had to be developed from scratch. As jABC had never been used for constructing code generators before, there were no ready-made services or models which could be reused. In particular, all required SIBs and their underlying services had to be implemented, the most important among them being the "Graph Model SIBs", which allow processing SLGs and their constituent parts (see Sect. 4.1.1), and the `RunVelocity` SIB (later part of the "Script SIBs"), which made the template engine Velocity available as a service. The Tracer was used for debugging and testing the Java Class Extruder throughout the entire modeling activity.

Fig. 5.2 shows the part of the Java Class Extruder that is responsible for processing the input SLGs for which code will be generated.[2] The SLG marked with number 1 first iterates over all input models (`Next Model`). Via an instance of the `RunVelocity` SIB, it then generates the header of the model method. Afterwards, there are two macros containing further SLGs that generate code for the SIBs (the SLG marked with 2), for the SLG's edges and for its model parameters. Finally, another `RunVelocity` SIB is used to generate the remainder of the model method.

SLG 2 generates code for all SIB instances contained in an input model. First, the current input model and its SIBs are retrieved from the execution context. Afterwards, the SLG iterates over the SIBs and generates code for each by calling SLG 3, which is again embedded via a macro. SLG 3 first retrieves the current model, the current SIB and its parameters from the execution context. Subsequently, it iterates over the SIB's parameters and generates corresponding code via `RunVelocity` SIBs, depending on whether

[2] Please note that the depicted SLG hierarchy is truncated for the sake of presentability.

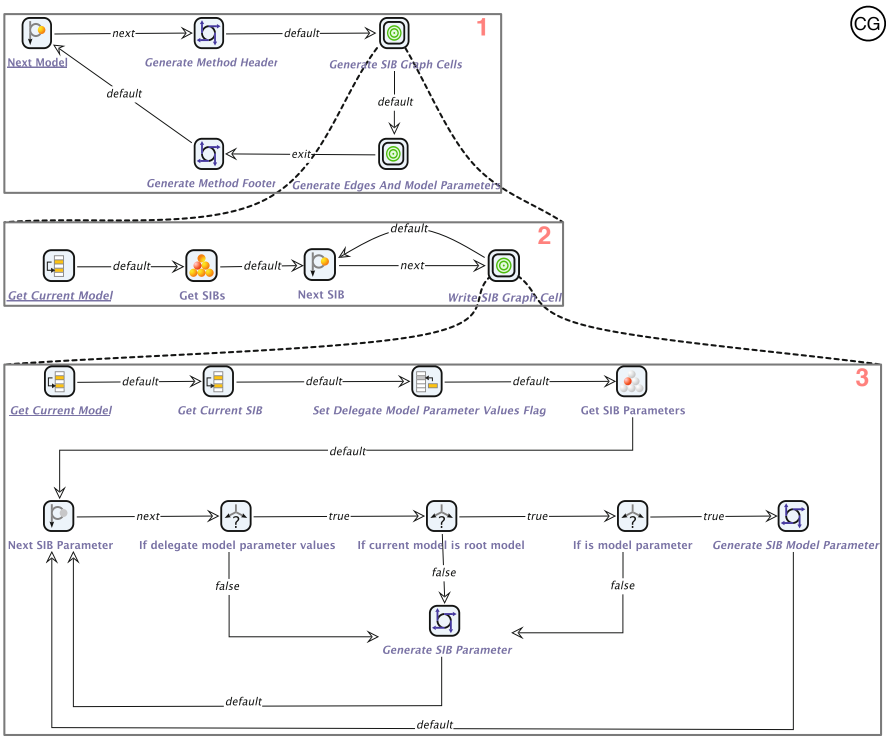


Fig. 5.2. Java Class Extruder: Processing the input SLGs

the current parameter is a normal SIB parameter or exported as a model parameter. The SLGs depicted in Fig. 5.2 are not limited to the Java Class Extruder: In fact, they are shared by all code generators following the Extruder approach and targeting a Java-based platform, such as the Servlet Extruder and the SIB Extruder which are presented in Sect. 5.4.

Due to the fact that the code generator is realized as SLGs, the only way to use it in earlier development phases was opening its models in jABC and then executing it with the Tracer. However, for being able to make the generator accessible via a jABC plugin or to integrate it in a tool-chain (cf. Sect. 4.3.2), it needs to be available as an executable Java class. As illustrated by the T-diagram in Fig. 5.3, this is achieved by means of bootstrapping. Note that the box marked with the number “3” is the T-diagram notation for an interpreter, with text labels indicating the source language that is interpreted (top) and the interpreter’s implementation language (bottom). The interpreter shown in Fig. 5.3 is the Tracer, with SLGs as its source language and Java as its implementation language. In essence, the Java Class Extruder generates itself by executing it with the Tracer, using its own models as input. This bootstrapping step yields the Java Class Extruder as an executable Java Class.

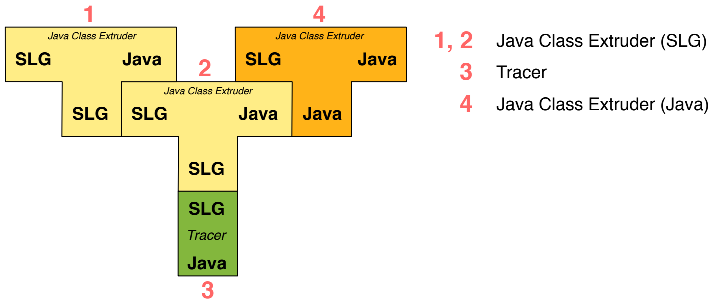


Fig. 5.3. Bootstrapping the Java Class Extruder by means of the Tracer

In order to make this Java class accessible for the Genesys tools, it needs to implement an interface called `CodeGenerator`. This interface is used by the tools to detect code generators and to acquire metainformation, such as the generator's name, author and documentation, that can be displayed to the user. This metainformation is usually provided by the generator developer via the Descriptor Inspector (cf. Sect. 4.3.1). In order to enable the Java Class Extruder to produce such a special Java class for a code generator, corresponding generator generator features had to be added to its models.

The bootstrapping process used for the Java Class Extruder is comparably simple, as it only involves one single stage. There are two reasons for this. First, as the Java Class Extruder's source and implementation language are both SLGs, it is a *self-generating generator* (analogous to the notion of a self-compiling compiler established in Sect. 2.1) and thus highly amenable to bootstrapping. Second, due to the availability of the Tracer as an interpreter for SLGs, the self-generation is performed easily without the need of an additional compiler. Especially the latter shows that jABC's executable modeling language eases and facilitates bootstrapping, thus meeting *Requirement S5 - Bootstrapping*.

Furthermore, the self-generation of the Java Class Extruder is an example of full code generation (*Requirement S2 - Full Code Generation*). The Java class resulting from the bootstrapping process does not require any manual adjustment or completion, and can be directly compiled and executed using a Java Development Kit (JDK). This is due to the service-oriented modeling approach in jABC: All required services have been implemented before the actual code generation, so that there is no missing information or unspecified behavior.

5.1.3 Evaluation

The Extruder approach to code generation provides several advantages, the most evident of them being the fact that the actual code generation is very simple, as described above. The direct reuse of the Tracer's features also significantly improves the maintainability of the code generator, because it does not need to be adapted to new Tracer versions (except for the case the API for invoking the Tracer changes). Any new feature introduced by the Tracer, such as a new control SIB realizing a novel execution pattern, is immediately supported by any Extruder. Furthermore, the execution equivalence of the models and the generated code is guaranteed automatically, because the Tracer is used for the execution in both cases.

Besides those advantages, there are also some drawbacks when using an Extruder. The biggest problem clearly is the overall size of the generated artifact. Although an Extruder-generated Java class usually does not grow beyond critical boundaries, it entails a lot of dependencies, such as libraries which need to be present at compilation time as well as at runtime. This is due to the Tracer which requires jABC's `SIBGraphModel` data structure, which in turn requires the jABC framework, a graph library, libraries for XML serialization etc. Furthermore, the `SIBGraphModel` data structure always contains the complete SIBs, including icons, documentation and code for jABC plugins such as the LocalChecker (cf. Sect. 3.2.3). Such information is not really required for executing the generated code, but it is induced by the employed libraries and thus cannot be omitted. This plethora of (for the generated code in large part unnecessary) information and dependencies may lead to a big size of the resulting artifact.

Furthermore, the runtime performance of the generated code is decreased, as the necessary construction of the data structure delays the actual execution. The Tracer's execution features are also expensive in terms of performance and cause some runtime overhead (cf. benchmark results in Sect. 5.3).

While being uncritical in lots of environments, those issues are especially problematic for target platforms with strong performance requirements or memory limitations, such as embedded systems. Sect. 5.4.4 describes the *leJOS Generator*, that is an example of a code generator for an embedded system to which the Extruder approach was not applicable due to the problems described above.

Furthermore, generating one single Java class does not scale for very large SLG hierarchies. The number of different models determines the number of model methods in the generated class, and the size of each model method depends on the overall number of elements contained in the model. Those elements include SIB instances, SIB parameters, model parameters, edges, branches, model branches and user objects, and each of those elements leads to one or more lines of code. For instance, when the Java Class Extruder (see Sect. 5.3.1 for the size of the corresponding models) generates itself, this results in a Java class with 6366 lines of code and a file size of 410 KB. More complex SIB hierarchies can easily lead to even bigger Java classes, which may in the worst case exceed the limits set by the JVM [LY99, p. 152].

Apart from these technical issues, another reason for refusing an Extruder approach might be the prevention of a possible vendor lock-in. Code generated by an Extruder highly depends on the jABC framework and the Tracer, i.e., on third-party software. If such dependencies are unwanted or even deprecated, the Extruder approach is clearly not appropriate.

Due to these drawbacks, an alternative approach had to be developed, which is presented in the following sections.

5.2 Java Class Generator

The weak points of the Extruder approach, which made it inapplicable for several target platforms like embedded systems, led to the development of alternative code generation strategies. Resulting from the experience with the Java Class Extruder described above, these alternatives are based on the following requirements:

1. **Small deployment size:** The overall size of the generated artifact, including its (compile-time and runtime) dependencies, should be kept small.
2. **Scalability:** The code generator should be able to produce valid results, even for large inputs.
3. **Better performance:** The performance should be improved in comparison to Extruder results.
4. **No vendor lock-in:** The generated artifact should not per se depend on third-party tools or libraries, except for those induced by the services used in the models.

Requirements 1 and 4 enforce the relinquishment of the Tracer as well as of jABC's `SIBGraphModel` data structure, due to the huge amount of dependencies resulting from both. Furthermore, the size of SIBs has to be reduced to those constituent parts relevant to the generated artifact, e.g., by sorting out unnecessary information. Requirement 3 also inhibits the use of the Tracer as it is a performance bottleneck of Extruder-generated code (cf. benchmark results in Sect. 5.3). Finally, as a consequence of requirement 2, the single class generation performed by Extruders has to be replaced by a more scalable concept. In order to distinguish code generators adhering to these principles from those that follow the Extruder approach, previous publications established the notion of *Pure Generators* for referring to the former [JMS08; Jör+07]. The Pure Generator counterpart for the Java Class Extruder is called the *Java Class Generator.*

The following sections elaborate on the Java Class Generator as well as on its underlying concepts and different variants.

5.2.1 Handling Service Calls

The development pattern for SIBs described in Sect. 3.2.1 was noticeably influenced by the experiences made with the Java Class Extruder. In the very beginning of the Genesys project, when jABC did not yet provide any facilities for code generation, the Tracer was the only way to execute SLGs. At this time, when designing a SIB it was general practice to directly call the underlying service from the SIB. This was viable because most SIBs provided only Java implementations anyway, and because the Tracer, which was primarily designed for debugging and (visual) animation in jABC, required the entire SIB for execution.

However, as outlined above in Sect. 5.1.3, this practice was not equally feasible for code generation. The large amount of information which is contained in a SIB but irrelevant for generated code, such as the icon, documentation, constraints or plugin annotations, leads to unnecessary dependencies and increases the generated artifact's size. Consequently, a code generator needs to be able to isolate only the required portion of a SIB, consisting of

1. the information on how the SIB's underlying service is called, and
2. the values of those SIB parameters which are used for configuring the underlying service.

The concept that emerged from these considerations was the decoupling of the SIB and the underlying service by means of a service adapter (cf. Sect. 3.2.1). This removes any unnecessary dependencies from the generated code, which has to contain direct calls to the service adapters only. In order to be able to generate such a call to a SIB's service adapter, a code generator has to inquire some information, as indicated in 1 and 2 above.

As an example, assume that the Java Class Generator detects an instance of the SIB `WriteTextFile` from the "IO SIBs" bundle (see Sect. 4.1.1) in an input model. The task of this SIB is to write a text, found in the execution context, to a file. For this purpose, the SIB provides two parameters: `text` denotes the execution context key for the actual text, and `file` specifies the path to the output file to which the text will be written. In order to generate a call to the SIB's Java implementation, the code generator first has to determine the corresponding service adapter and the name of the actual service that will be called. For `WriteTextFile`, the service adapter is a Java class called `JavaServiceAdapter`, containing a method named `writeTextFile`, which provides the actual implementation of the SIB's behavior. In the next step, the code generator identifies the SIB parameters required for configuring the service call, which are `text` and `file`. With this information, the code generator is finally able to produce a direct call to the SIB's service adapter, e.g.:

```
de.jabc.adapter.common.io.JavaServiceAdapter.writeTextFile(
    "This is a test text.", new java.io.File("/path/to/file.txt"));
```

The aggregate of this information is called the *service adapter descriptor*.

In order to realize this concept, an appropriate mechanism was required that connects a SIB with its available service adapters in a way that allows a code generator to deduce the necessary information for generating a service call from a given SIB instance, as described above. Two solutions emerged for achieving this, the first of which uses interfaces to establish the connection:

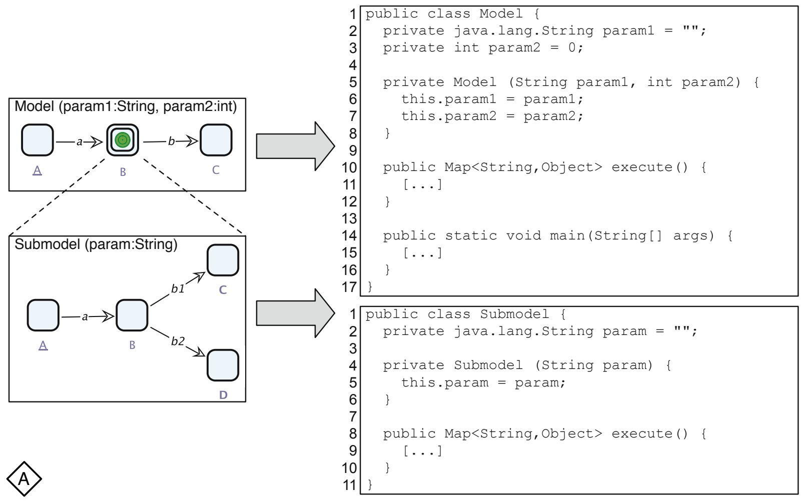
For any new service adapter that realizes a SIB, the SIB has to implement a corresponding interface. This interface implementation provides the information on how to call the service adapter, i.e., the service adapter descriptor. Although this solution works well and has been used extensively by the first Genesys code generators for jABC, a major disadvantage is the effort that is required in order to attach a new service adapter to a SIB. For each new service adapter, the following steps are required: a corresponding interface has to be implemented if it does not already exist, the SIB has to be adapted so that it implements the new interface and exposes the corresponding service adapter descriptor, and finally the SIB has to be recompiled. Those modifications are usually rather cumbersome and sometimes even hardly possible, especially if the affected SIB is contained in a third-party bundle that is only available in binary form.

Thus the second solution avoids the need for modifying and recompiling the SIB by using a separate XML file. Such a file contains the mapping between SIBs and their service adapters for an entire SIB bundle and is placed in a standardized location, so that it can be easily found by a code generator. Listing 5.1 shows an excerpt of such an XML file, which associates a service adapter implemented in Java to the SIB `WriteTextFile` mentioned above. The mapping between the SIB and the service adapter is established via the SIB's UID in order to avoid ambiguities (line 1). The `adapter` tag in line 2 declares the name of the service adapter that contains the implementation of the SIB, which is, in this example, a Java class. For other cases, it may, e.g., denote the name of an Objective-C class or a Perl script. The following lines 3–7 specify the name of the particular service that implements the SIB's behavior (in this case the name of a Java method) as well as the names of the SIB parameters which provide the input values for the service.

```
1 <serviceAdapter uid="io-sibs/WriteTextFile">
2     <adapter>de.jabc.adapter.common.io.JavaServiceAdapter</adapter
        >
3     <service>writeTextFile</service>
4     <arguments>
5         <argument name="text" />
6         <argument name="file" />
7     </arguments>
8 </serviceAdapter>
```

Listing 5.1. Excerpt from an XML file that maps service adapters to SIBs

The XML file that contains the excerpt from Listing 5.1 attaches Java implementations to all SIBs in the "IO SIBs" bundle. If further implementations are available, e.g., for Ruby, they are mapped to the SIBs via an additional XML file. Each such file is identified by a unique name. For instance, the file containing the Java mappings is called `adapters_java.xml`, and the one for the Ruby mappings is called `adapters_ruby.xml`. For convenience and in order to shield the generator developer from these technical issues, the Genesys framework contains SIBs that retrieve the desired service adapter descriptor for a given SIB instance. This includes finding the corresponding



```
public class Model {
  private java.lang.String param1 = "";
  private int param2 = 0;

  private Model (String param1, int param2) {
    this.param1 = param1;
    this.param2 = param2;
  }

  public Map<String,Object> execute() {
    [...]
  }

  public static void main(String[] args) {
    [...]
  }
}
```

```
public class Submodel {
  private java.lang.String param = "";

  private Submodel (String param) {
    this.param = param;
  }

  public Map<String,Object> execute() {
    [...]
  }
}
```

Fig. 5.4. Generation concept of the Java Class Generator

XML file, parsing it, extracting the relevant information and storing it in the execution context, so that it can be further processed by the code generator.

Due to this concept, adding new implementations to a SIB is very easy, as it is reduced to augmenting an existing XML file or adding a new one. Recompilation is not required, so that this solution also works if a SIB's source code is not available. There are no limitations on the number of implementations (and thus target platforms for code generation) that can be assigned to one SIB. Please note that this flexible service concept is applicable for all SIBs, including those used for building code generators with Genesys, consequently meeting *Requirement G1 - Platform Independence* and *Requirement G2 - Reusability and Adaptability*.

5.2.2 From Single Class to Multiple Classes

Sect. 5.1.3 argued that generating one Java class per SLG hierarchy, as performed by the Java Class Extruder, does not scale well. Thus an alternative approach had to be found, that produces results respecting the limitations set by the JVM, even for large inputs. A natural solution is partitioning the generated artifact: Instead of one single class for an entire SLG hierarchy, the Java Class Generator produces one class for each (different) model that is contained in the hierarchy.

Fig. 5.4 illustrates this concept by means of the sample SLGs already used in Fig. 5.1. Additionally, the example now displays the model parameters of the involved SLGs: "Model" has two model parameters "param1" and "param2" of type `String` and `int` respectively, and "Submodel" provides one `String` parameter named "param". The right hand side of the figure sketches the code resulting from the generation process, again in a simplified and truncated form. It is visible that each SLG in the hierarchy yields a separate Java class, called the *model class*, each containing an `execute` method for executing the behavior represented by the corresponding model. Only the class `Model` has been equipped with a `main` method, as it corresponds to the topmost model in the SLG hierarchy and thus is considered the application's entry point.

In contrast to results produced by the Java Class Extruder, the `execute` methods are always parameterless, as the configuration of each class is performed via its constructor. Generally, the model parameters of an SLG lead to corresponding constructor parameters (line 5 in `Model` and line 4 in `Submodel`) and private instance variables (lines 2–3 in `Model` and line 2 in `Submodel`) in the resulting code. Just like the model methods in the Extruder pendant, this configurability allows the reuse of generated code, in this case of entire Java classes, thus avoiding redundancies. For instance, although "Submodel" may be reused multiple times in the SLG hierarchy, the code generation will nevertheless always produce exactly one `Submodel` class, which will be reused in an analogous manner in the generated code.

Although this concept does not reduce the overall size of the generated artifact, it produces Java classes which are considerably smaller and thus do not exceed any limits set by the JVM. For generating the code that realizes the actual behavior of the SLGs, i.e., the content of the `execute` methods, two variants have been developed which will be presented in Sect. 5.2.4 and 5.2.5, respectively.

5.2.3 Data Type Mappings and Execution Context

As most of jABC's built-in data types are in fact Java data types, the mappings required for the Java Class Generator are very simple. Consequently, in terms of the five cases of data type mapping defined in Sect. 4.1.2, all jABC data types referred to as "simple types" in Sect. 3.2.1 belong to the "identity" category.

In contrast to this, the complex data types provided by jABC have no direct pendants in Java, so that corresponding mappings are required. Table 5.1 lists those mappings, and it is visible that most of them belong to category 4 ("new data type"). For those data types, new lightweight pendants are introduced (suffixed with `Foundation`), which preserve all necessary information, but at the same time do not induce any unwanted dependencies. For instance, the jABC data type `ContextKey`, which specifies a location for accessing data in the execution context (cf. Sect. 3.3.2), contains a name for the context key as well as information about the context scope. This data type is translated to `ContextKeyFoundation`, which carries the same information, but is implemented without the use of any jABC-specific APIs. The Java Class Generator emits those lightweight types on demand, i.e., only if the corresponding jABC data type is used in the input models.

Table 5.1. Java mapping for complex jABC data types

jABC Data Type	Java Data Type
ContextExpression	ContextExpressionFoundation
ContextKey	ContextKeyFoundation
ExtendedFile	File
ListBox	ListBoxFoundation
MultiObject	MultiObjectFoundation
Password	String
StrictCollection	StrictCollectionFoundation
StrictList	StrictListFoundation

The types `ExtendedFile` and `Password` belong to category 3 ("reductive mapping"). Both contain information which is used by jABC for properly displaying the parameter in the SIB inspector. For instance, `ExtendedFile` allows the definition of a filter for restricting the modeler to the selection of particular file types. However, such information is solely required for modeling, but it has no effect on the execution of a SIB using a correspondingly typed parameter. Hence such data types could be reduced to the information relevant for execution: `ExtendedFile` is reduced to a simple `File` object specifying an absolute file path, and `Password` is mapped to a simple string containing the password.

Apart from data types, the stacked execution context provided by the Tracer (cf. Sect. 3.3.2) also has to be supported by the generated code. Again, the Tracer's original execution context cannot be reused as it induces too many unwanted dependencies. In consequence, a lightweight version of the execution context has been designed, which basically uses hash tables for the single contexts, that are managed in a stack. Just like the lightweight data types described above, this lightweight implementation of the execution context is emitted by the code generator.

The lightweight data types and the lightweight execution context are not only important for the generated code, but also for the SIBs contained in the input SLGs of the code generator: The Java service adapters of the SIBs must exclusively use the lightweight types and context in order to assure compatibility with the generated code (which contains corresponding service calls, see Sect. 5.2.1), and to avoid unnecessary dependencies to jABC. This has to be considered by the SIB expert who is responsible for developing the service implementations.

5.2.4 Variant 1: Structured Code

The first variant for mapping the behavior modeled by an SLG to corresponding code involves the generation of *structured code*. This refers to the translation of the control flow resembled by an SLG to corresponding code statements with equivalent semantics. As described in Sect. 3.3.1, the semantics for the execution of SLGs is determined by the Tracer.

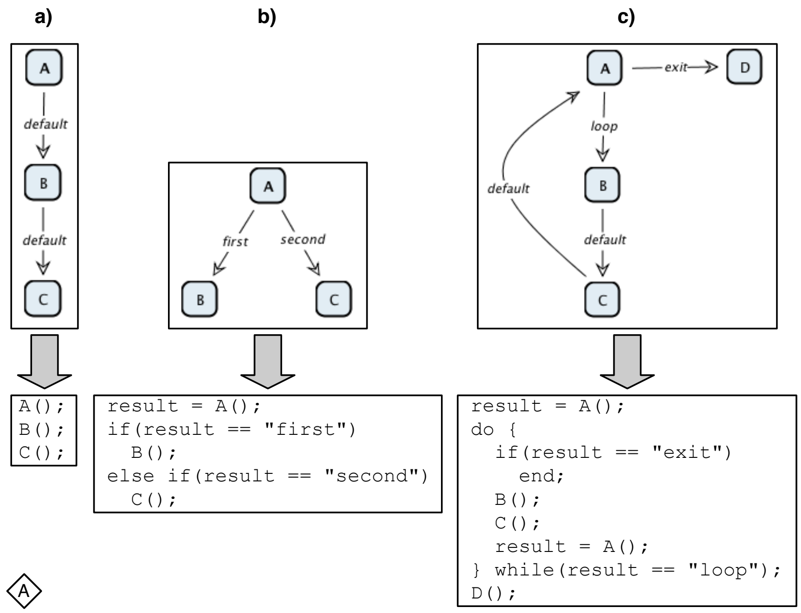
Such translations are usually based on the identification of structured regions, which can be easily converted to corresponding code statements. However, due to the trade-off between graph-based (like SLGs) and block-oriented (like Java) languages, models may contain unstructured parts which cannot be directly translated to code. Both the identification of structured regions and the treatment of unstructuredness are well-researched fields.

Identifying Control Flow Patterns:

Essentially, the control flow structures occurring in an SLG are very similar to those found in elementary flow charts used to represent procedural programs. Williams [Wil77] refers to these structures as the "three basic structures of structured programming", consisting of sequences, selections (i.e., decisions) and repetitions. The translation of those three control flow patterns to corresponding block-oriented code is usually straightforward. Fig. 5.5 shows SLG examples for the patterns and illustrates the translation of each into code. The latter is written as simplified Java-like code. The generated code for SIBs is represented by simple function calls such as "`A();`" for the SIB `A` (meaning that the service underlying `A` is called), instead of a full service call as described in Sect. 5.2.1.

In terms of the translation, sequences (part a) are the most simple case, as they are directly mapped to consecutive statements. For decisions, branches are used to distinguish different alternatives, in accordance with the SLG semantics defined by the Tracer. This is depicted in part b of Fig. 5.5: First, the service underlying SIB `A` is called and its result, which is a branch name, is obtained. Subsequently, an if-else-statement is constructed according to all branches of the SIB that are assigned to outgoing edges, in the example "first" and "second". For each alternative, the service underlying the corresponding successor is called. Finally, part c of Fig. 5.5 exemplifies the code resulting from a repetition by means of a do-while-loop. In the SLG, the SIB labeled `A` decides whether the loop is continued (branch "loop") or exited (branch "exit"). Thus it is the service underlying `A` which is called first. Afterwards, the result is checked: For "exit", the loop is aborted immediately, otherwise the contents of the loop are processed (in the example a sequence of `B` and `C`). Finally, the service underlying `A` is called again at the end of the loop, prior to evaluating the loop condition.

Besides the three basic patterns described above, SLGs may also contain hierarchy and fork-join constructs. However, the former do not introduce a new control flow pattern to SLGs. As described in Sect. 3.2.2, macros are a structural means for hiding parts of an SLG. Although such a macro resembles an entire submodel, it appears just like any other SIB in a model. This is even true for recursion, which can, e.g., be built by means of a macro that in turn references the same model in which it is contained. For a code generator, the only difference is the generation of a corresponding service call: Instead of producing a call to an existing service (cf. Sect. 5.2.1), a macro is translated into a call to the generated code corresponding to the referenced

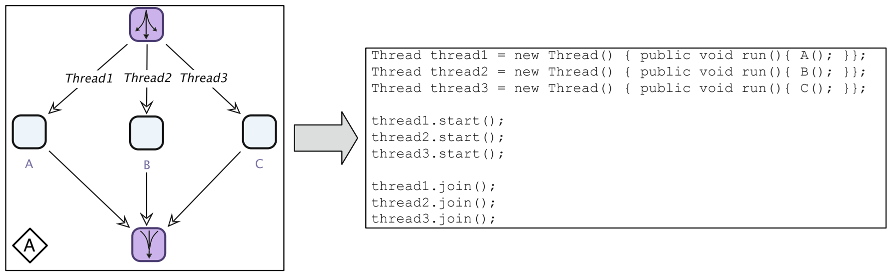


```
A();
B();
C();
```

```
result = A();
if(result == "first")
  B();
else if(result == "second")
  C();
```

```
result = A();
do {
  if(result == "exit")
    end;
  B();
  C();
  result = A();
} while(result == "loop");
D();
```

Fig. 5.5. Basic control flow patterns and their translation into code



```
Thread thread1 = new Thread() { public void run(){ A(); }};
Thread thread2 = new Thread() { public void run(){ B(); }};
Thread thread3 = new Thread() { public void run(){ C(); }};

thread1.start();
thread2.start();
thread3.start();

thread1.join();
thread2.join();
thread3.join();
```

Fig. 5.6. The fork-join control flow pattern and its translation to code

submodel. Consequently, hierarchy constructs are ignored for the following considerations.

In turn, fork-join constructs introduce an additional control flow pattern to SLGs. However, those constructs are unproblematic for the generation of structured code, as they are subject to strict well-formedness rules described in Sect. 3.3.3 (such as pairwise occurrence of fork and join, or correct nesting). The single threads between a fork and a join may only contain the three basic patterns as well as further well-formed fork-join constructs.

Fig. 5.6 shows the translation of a well-formed fork-join construct into corresponding code. Like in Fig. 5.5, the code is written as simplified Java-like code. It is visible that for each of the parallel steps modeled in the SLGs

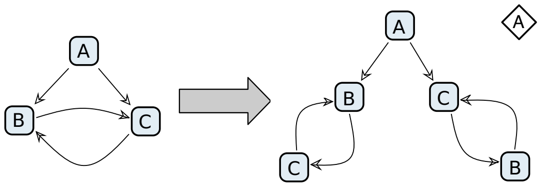


Fig. 5.7. An unstructured SLG and its equivalent structured pendant

on the left hand side, a separate thread is constructed in the generated code. After the threads have been started, the generated program waits until all threads are finished.

Handling Unstructuredness:

Any control flow structure that cannot be assigned to the basic patterns described above is considered unstructured. The SLG on the left hand side of Fig. 5.7 is an example of such a case, as it contains one loop with multiple entry points (`B` and `C`), which is one of five basic unstructured forms identified by Williams [Wil77]. The remaining forms named by Williams are abnormal selection paths, loops with multiple exit points, overlapping loops and parallel loops.

For flow charts, it is well established that any unstructured form can be transformed to an equivalent structured form. For instance, apart from identifying unstructured forms that may occur in flow charts, Oulsham [Oul82] also provided transformations that convert each to equivalent structured forms. This is typically achieved by means of node duplication or by introducing auxiliary variables. As SLGs are basically a condensed version of flow charts that omits split and join nodes (especially those for reflecting alternative flows), similar transformations can also be adapted and applied to SLGs.

The right hand side of Fig. 5.7 shows a structured pendant for the unstructured SLG on the left hand side. In this model, the unstructuredness is resolved by duplicating the loop, so that each outgoing edge of `A` leads to one loop with a single unique entry point. Furthermore, the transformation preserves the model's execution behavior, which shows that it can be safely used as a pre-processing step for code generation.

Further relevant research emerged from BPM (cf. Sect. 2.3.6), which recently focused on this topic, e.g., when translating BPMN to BPEL. The results include efficient techniques for the detection of control flow patterns [VVK08;GB08], for transforming unstructured BPMN models and other workflow specification languages to structured pendants [DGBP11; KHB00] as well as strategies for code generation [Ouy+06; ABL08]. In the case of BPMN, the translation to structured code is much more complex than for SLGs. This is due to the fact that BPMN provides much more syntactic constructs which mostly are not subject to strict well-formedness rules like the SLGs' fork-join, thus entailing a higher potential for unstructuredness. Consequently, the techniques mentioned above also cover SLGs, which may be considered a subset of BPMN in terms of control structures.

Engeler Normal Form (ENF):

The structured code variant of the Java Class Generator does not yet employ the advanced techniques mentioned above, but it is available as a simple prototypical realization. For eliminating unstructuredness, it uses an algorithm which employs node duplication in order to transform SLGs to their Engeler Normal Form (ENF) [Eng71]. This normal form only contains structured instances of the three basic control flow patterns described above, and thus can be easily translated into code.

As an example, the structured model on the right hand side of Fig. 5.7 is the ENF of the SLG on the left hand side. The transformation algorithm was described and implemented in a diploma thesis [Dra06]. For generator developers, the ENF transformation is available as a SIB called `BuildEngeler-NormalForm`, which transforms a given set of SLGs to their corresponding ENFs.

In order to be applicable in the Java Class Generator, the algorithm was extended for supporting fork-join constructs, which are handled in a prior transformation step. Fig. 5.8 illustrates this transformation, which maps a well-formed fork-join construct (left hand side) to a hierarchical graph (right hand side): Each thread modeled between fork and join is moved to a separate submodel. In the example, the sequences `B-C` and `D-E` are single threads which are converted to submodels. In the original model, the entire fork-join construct is replaced by a special macro which allows the assignment of multiple models (in contrast to standard macros in SLGs, which are able to reference one single submodel only). Finally, the new submodels representing the threads are assigned to this macro. During execution of the model, the special macro behaves just like an ordinary fork-join construct: It executes each assigned submodel in parallel and waits until each thread finishes, before the next step (in the example `F`) is executed.

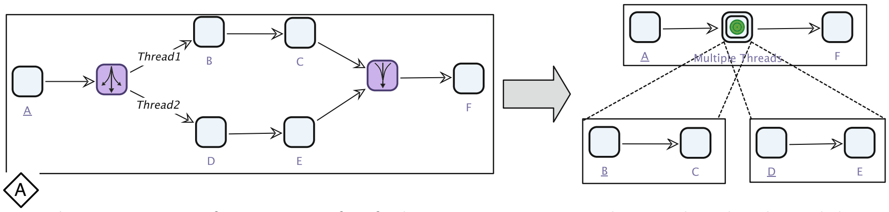


Fig. 5.8. Transformation of a fork-join construct to hierarchical submodels

As a consequence, code generators need to be aware of the special macro in order to generate corresponding code as described above. However, as the

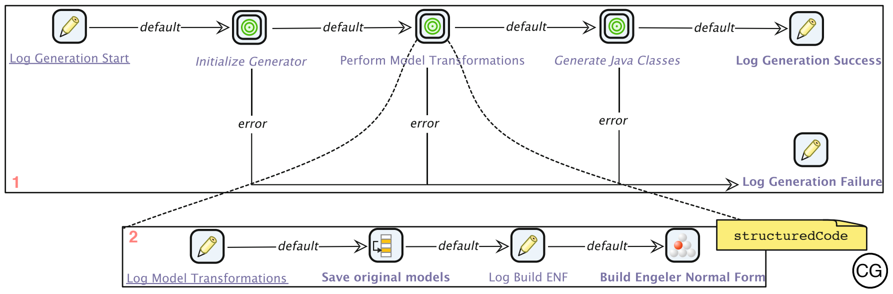


Fig. 5.9. Java Class Generator: Root SLG (1), Model transformations (2)

major advantage of this modus operandi, model-to-model transformations such as the ENF transformation do not have to explicitly support the fork-join control flow pattern.

After the ENF transformation, each node in an SLG can be assigned to one of the basic control flow patterns by simply counting its incoming and outgoing edges:

- *sequence:* 0 to 1 incoming edges and 0 to 1 outgoing edges
- *decision:* 0 to 1 incoming edges and 2 to n outgoing edges
- *repetition:* $n > 1$ incoming edges (or 1 if the node is a start node) and 1 to n outgoing edges

The identification of the control flow patterns and the actual translation into code can be performed simultaneously during a depth-first traversal.

Modeling the Structured Code Approach:

The first version of the Java Class Generator exclusively followed the structured code approach (cf. Sect. 5.2.7). Subsequently, further alternative approaches (see Sect. 5.2.5) were added and the parts of the generator that realize the structured code approach became one variant among others. In Fig. 5.9, SLG 1 at the top shows the root model of the Java Class Generator. Besides several steps that perform logging (those SIBs which show a pen on their icon), it is visible that the generator is separated into three phases:

1. *Initialization Phase* (macro `Initialize Generator`): In this phase, the code generator is initialized. For instance, this includes the verification of the generator arguments provided by the user as well as loading the input SLGs from the file system. Furthermore, the required data type mappings are set up (cf. Sect. 4.1.2 and 5.2.3) and all keywords reserved by Java are added to the identifier blacklist (cf. Sect. 4.1.3).
2. *Transformation Phase* (macro `Perform Model Transformations`): If the input SLGs need to be pre-processed or transformed prior to the actual code generation process, this can be performed in the transformation phase. For the structured code approach, this is the place where the input

SLGs are transformed to their corresponding ENFs, as shown by SLG 2 in Fig. 5.9. Preceding the `BuildEngelerNormalForm` SIB that performs the transformation, SLG 2 contains a SIB that saves backup copies of the original (unmodified) input SLGs, and two SIBs that issue log messages.
3. *Generation Phase* (macro `Generate Java Class`): This is where the actual code generation is performed, i.e., the elements of the input SLGs are processed in order to collect all information necessary for generating code, which is assembled and finally written to corresponding output files.

Fig. 5.9 also exemplifies the use of Genesys' variant management feature described in Sect. 4.1.4. SLG 2 represents a variant that realizes the transformation phase, identified by means of the unique name `structuredCode`. In the parent SLG 1, this variant is assigned to the macro `Perform Model Transformations`, which serves as a variation point, i.e., other variants may realize the transformation phase differently. For instance, the alternative generation approach presented in Sect. 5.2.5 does not require any model transformations, so that the model specifying this phase is empty for this variant. At the same time, there are also parts that are shared by all variants such as SLG 1 and all models realizing the initialization phase. This illustrates the reuse of models (*Requirement G2 - Reusability and Adaptability*) on the one hand and the specification of variability and product lines (*Requirement S3 - Variant Management and Product Lines*) on the other hand.

Furthermore, by means of the SIB `BuildEngelerNormalForm`, SLG 2 in Fig. 5.9 shows how the abstraction provided by SIBs (and services in general) simplifies the development process: A generator developer is able to apply the ENF transformation without knowing all of its theoretical, algorithmic or implementation details. All he has to know is the purpose of the ENF transformation explained in the SIB's documentation, in order to recognize it as a necessary pre-processing step for the detection of control flow patterns. If the generator developer subsequently decides to use another algorithm for the ENF construction, or even another transformation, the SIB can be easily replaced by another one (*Requirement G3 - Simplicity*). Similarly, the code generator can be improved with the advanced techniques from the BPMN/BPEL community outlined above. In order to achieve this, the corresponding algorithms, libraries etc. have to be made available once as corresponding SIBs, so that they can be used by generator developers for modeling.

Fig. 5.10 shows the part of the structured code variant that is responsible for detecting the control flow patterns. The depicted SLGs are situated in the generation phase of the code generator, and thus work on input models which already have been converted to their ENF. Each input model is traversed in a depth-first manner. For each SIB instance in an input model, the steps modeled by SLG 1 in Fig. 5.10 are executed. After retrieving the current SIB instance, the generator checks whether it has already been visited. If this is the case, then the SIB instance represents a loop entry and as it already has

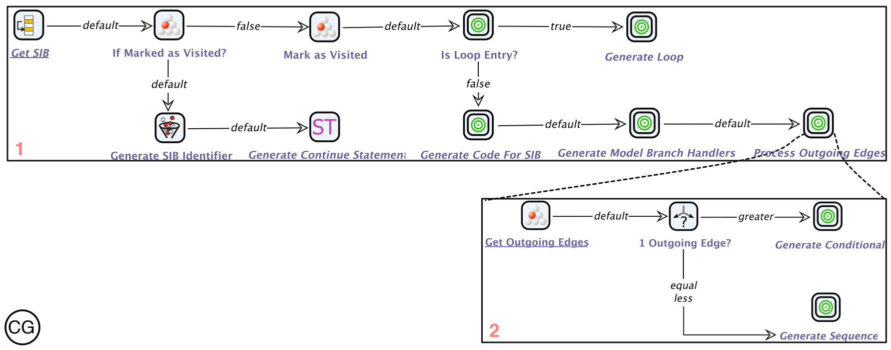


Fig. 5.10. Java Class Generator: Detection of Control Flow Patterns

been visited, the traversal just followed a back edge. Consequently, the generator produces a `continue` statement for jumping back to the loop entry[3]. If the SIB instance has not been visited before, it is checked for being a loop entry. If it is a loop entry, corresponding code is generated, otherwise the generator first produces the corresponding service call along with code for handling any model branches specified by the SIB instance. Afterwards, its outgoing edges are examined in order to decide whether it is a sequence or a decision that should be generated. This check is modeled in SLG 2, and it resembles the criteria for identifying sequential and conditional control flow patterns outlined above.

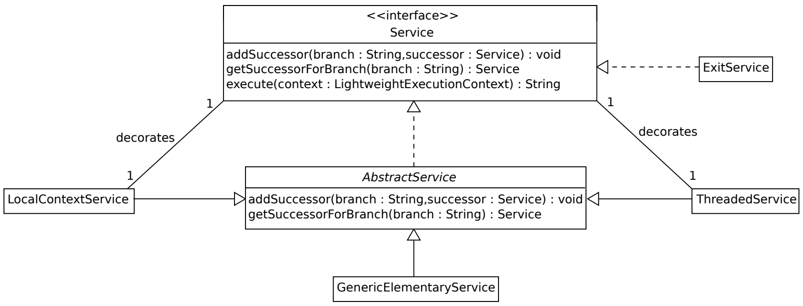
5.2.5 Variant 2: The Interpreter Approach

The second variant of the Java Class Generator is influenced by the ideas of the Extruder approach described above in Sect. 5.1.1. It aims at translating an SLG into a corresponding data structure, which is then executed by means of an interpreter. However, in contrast to Extruders, the resulting code has to be self-contained in order to meet the requirements for the Java Class Generator formulated above. This means that both the data structure used for representing the SLG as well as the interpreter itself have to be generated in a way that avoids the problems of the Extruder approach.

The Lightweight Data Structure:

Similar to the handling of data types and the execution context (cf. Sect. 5.2.3), an SLG is translated into a simplified lightweight data structure. The class diagram in Fig. 5.11 shows an excerpt of this data structure, which is a highly condensed version of jABC's `SIBGraphModel` (see Sect. 3.2.2), that is reduced

[3] As a result of simplifying the models of the generator, the technical implementation of how the Java Class Generator produces code for loops slightly differs from the simplified code in Fig. 5.4.



<<interface>>
Service
addSuccessor(branch : String,successor : Service) : void
getSuccessorForBranch(branch : String) : Service
execute(context : LightweightExecutionContext) : String
ExitService
decorates
1
decorates
1
1
1
AbstractService
addSuccessor(branch : String,successor : Service) : void
getSuccessorForBranch(branch : String) : Service
LocalContextService
ThreadedService
GenericElementaryService

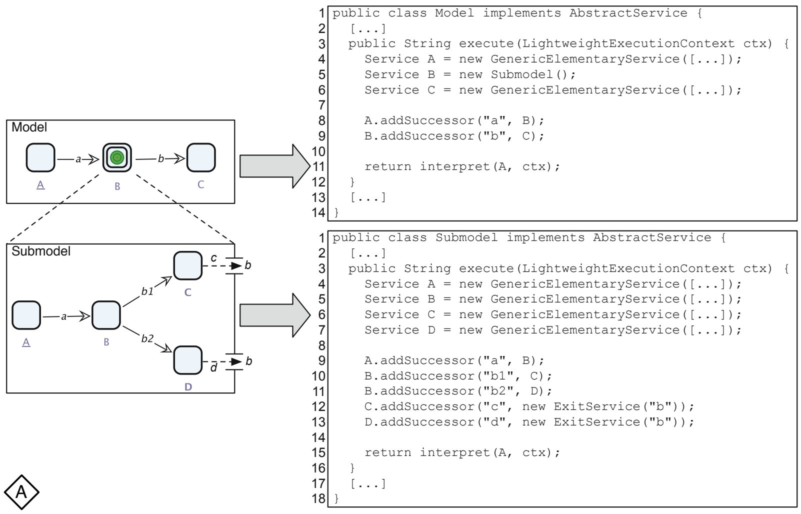
Fig. 5.11. Lightweight data structure for SLGs (excerpt)

to contain only information which is relevant for execution. Anything else that is, e.g., specific to displaying (such as icons or positioning information) or verifying (i.e., local and global constraints) SLGs is omitted.

In the lightweight data structure, any contained service has to be constructed as specified by the `Service` interface. According to this interface, a service has to define its execution behavior by implementing the `execute` method, which works on the lightweight version of the execution context outlined in Sect. 5.2.3. As its result, this method always returns the name of a branch which reflects the service's execution result. Furthermore, the `Service` interface defines that a service can have zero or more successors, each of them identified by a corresponding branch name (methods `addSuccessor` and `getSuccessorForBranch`). As the management of the successors is the same for almost any service, this functionality is realized by an abstract class called `AbstractService`. Thus any concrete service only needs to extend this abstract class and to implement the remaining `execute` method.

Fig. 5.11 also provides examples for concrete services that may occur in the lightweight data structure:

- `GenericElementaryService` is the lightweight pendant to a SIB. Basically, such a service is parametrized with the information contained in a SIB's service adapter descriptor (see Sect. 5.2.1). When the service is executed, it uses the information to call the corresponding service adapter by means of Java's Reflection API.
- `LocalContextService` and `ThreadedService` are counterparts of the `GraphSIB` and the `ThreadSIB`, respectively. In jABC, both are macros and control SIBs, so that they cause another service (realized by a submodel) to be executed in a certain way. In order to reflect this, the lightweight pendants are designed on the basis of the well-known *Decorator* pattern [Gam+95]. Accordingly, these services mainly "decorate" another service (such as a `GenericElementaryService`) with additional execution behavior: execution with a stacked context in case of the



```
public class Model implements AbstractService {
  [...]
  public String execute(LightweightExecutionContext ctx) {
    Service A = new GenericElementaryService([...]);
    Service B = new Submodel();
    Service C = new GenericElementaryService([...]);

    A.addSuccessor("a", B);
    B.addSuccessor("b", C);

    return interpret(A, ctx);
  }
  [...]
}
```

```
public class Submodel implements AbstractService {
  [...]
  public String execute(LightweightExecutionContext ctx) {
    Service A = new GenericElementaryService([...]);
    Service B = new GenericElementaryService([...]);
    Service C = new GenericElementaryService([...]);
    Service D = new GenericElementaryService([...]);

    A.addSuccessor("a", B);
    B.addSuccessor("b1", C);
    B.addSuccessor("b2", D);
    C.addSuccessor("c", new ExitService("b"));
    D.addSuccessor("d", new ExitService("b"));

    return interpret(A, ctx);
  }
  [...]
}
```

Fig. 5.12. Generation concept of the Java Class Generator's "Interpreter Variant"

`LocalContextService` and execution in a separate thread in case of the `ThreadedService`.

- An `ExitService` is equivalent to a model branch which is used in SLGs to mark the exit of a model. It is also responsible for mapping the branch name returned by the previously executed service to the branch name that reflects the corresponding execution result of the entire model[4]. An example for this will be provided below.

Similar lightweight pendants exist for all control SIBs (e.g., for event handling), but for the sake of clarity, only a representative subset of them is presented in Fig. 5.11.

If the use of reflection is undesired, e.g., due to performance reasons (or for target languages other than Java, if no reflection capabilities are available), the `GenericElementaryService` can be easily replaced by correspondingly generated services: For each SIB class, the code generator has to produce a suitable implementation of `Service` that contains the direct call to the SIB's service adapter, as described in Sect. 5.2.1. When comparing and evaluating the different Java code generators in Sect. 5.3, both alternatives (i.e., with and without reflection) will be considered.

Fig. 5.12 illustrates the translation to the lightweight data structure by means of the sample SLGs already used in Fig. 5.1 and Fig. 5.4. In addition to the previous examples, the edges in the SLGs have been augmented by corresponding branch labels. Furthermore, model branches leaving the model are indicated for the SLG "Submodel". The resulting model classes

[4] As described in Sect. 3.2.2, the name of a model branch may differ from the name of the actual SIB branch that has been exported.

extend `AbstractService`, i.e., in the generated code, models are treated just like any other services. In the implementation of the `execute` method, the corresponding lightweight data structure is constructed. First, the services resembling the single SIBs are created. The example mostly contains standard SIBs which are translated to `GenericElementaryServices`. As an exception, the macro `B` in the SLG "Model" is resembled by an instance (class `Model` at the top, line 5) of the generated "Submodel" class. Second, the connections between the services are established by declaring the successors of each service. Those connections reflect the edges in the original SLG. Accordingly, each successor is set along with the branch name which labels the corresponding incoming edge in the SLG. For instance, in Fig. 5.12, the edge labeled "a" between `A` and `B` in the SLG "Model" is translated to "`A.addSuccessor("a", B)`" (class `Model`, line 8).

Fig. 5.12 also exemplifies the use of exit services. In the SLG "Submodel", it is visible that the SIB `C` provides a branch "c", which is in turn exported as a model branch called "b", used to label the outgoing edge of macro `B` in the superordinate SLG "Model". In the generated code, this is reflected by an exit service, which returns the name of the model branch "b", and which is set as a successor of `C` and a virtual outgoing edge labeled "c" (see class `Submodel`, line 12).

After constructing the lightweight data structure, it is executed via an interpreter.

The Interpreter:

Given the lightweight data structure, the interpreter for the execution is very simple. In order to start an execution, the interpreter just requires the start service of the data structure (`A` in the examples in Fig. 5.12) along with the lightweight execution context.

```
 1  public String interpret(Service startService,
        LightweightExecutionContext ctx) {
 2    Service service = startService;
 3    String result = "";
 4    while (service != null) {
 5      result = service.execute(ctx);
 6
 7      if (isExitService(service))
 8        break;
 9
10      service = service.getSuccessorForBranch(result);
11    }
12    return result;
13  }
```

Listing 5.2. Generated interpreter for the lightweight data structure

Listing 5.2 describes the interpretation algorithm by means of simplified Java-like code. Basically, this algorithm uses a while loop to traverse an execution path through the data structure (lines 4–11). Commencing with the given start service, each service is executed, and afterwards a corresponding

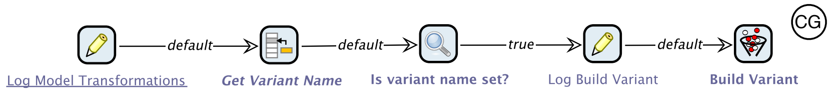


Fig. 5.13. Transformation phase of the Genesys Code Generator Generator

execution result is obtained (line 5). This result reflects a branch name that is used to determine the next service that is to be executed (line 10). The loop terminates if there is no such successor service or if the next service is an exit service. The former is equivalent to reaching the end of the topmost model in an SLG hierarchy, meaning that the overall execution is finished. The latter indicates the end of a submodel, which causes the execution to continue by following a model branch that leads to the superordinate model.

The simplicity of the interpreter results from the fact that the actual execution work is delegated to the single services. The interpreter is emitted by the Java Class Generator along with the generated code.

5.2.6 Genesys Code Generator Generator

The purpose of the *Genesys Code Generator Generator* is to make a modeled code generator accessible for the Genesys tools (such as the jABC plugin or the Maven plugin, cf. Sect. 4.3.2). It translates the models of a code generator into a special Java class which implements the interface `CodeGenerator` (cf. Sect. 5.1.2). The Genesys tools utilize this interface to detect code generators and to make them available for users. Furthermore, the interface is intended to expose the metainformation of a code generator, such as the generator's name, author or general documentation to the tools.

The Genesys Code Generator Generator is realized as a simple extension of the Java Class Generator. It directly reuses the latter's functionality for translating an SLG hierarchy into corresponding Java classes and contains further templates which add the implementation of the `CodeGenerator` interface. The metainformation that is generated into the resulting Java class is taken from the data provided by the generator developer via the Descriptor Inspector (cf. Sect. 4.3.1).

Additionally, the Genesys Code Generator Generator is able to selectively produce a specific variant of a given code generator. Via an additional parameter, the user may provide the name of the variant that should be generated. This parameter is evaluated in the generator generator's transformation phase, which is depicted in Fig. 5.13. If the name of a variant has been provided by the user, the `BuildVariant` SIB is used to build the corresponding variant via a model transformation, as described in Sect. 4.1.4. If no variant name has been specified, the default variant is used as an input for the code generation. In this case no model transformation is necessary.

Sect. 4.2.4 exemplifies the application of the generator generator for translating the Documentation Generator.

5.2.7 Remarks on Different Versions

The first version of the Java Class Generator did not realize all the concepts and variants described in the previous sections. In particular, the first Java Class Generator exclusively followed the structured code approach. This is partly due to the fact that the services for variant management, which considerably ease experimentation with alternative generation strategies, have been developed at a later stage of the Genesys framework. Furthermore, as the first version was directly derived from the Java Class Extruder, and as the "Scalability" requirement formulated above was not a pressing issue at that time, it also followed the single class approach rather than producing multiple classes. In the following, this first version will be referred to as *Java Class Generator 1.* The Java Class Generator 1 was in active use and shipped with jABC bundles for about three years. Consequently, it formed the basis of numerous code generators that have been derived from it, e.g., in the context of diploma theses (cf. Sect. 5.4).

The second and current version of the Java Class Generator, the *Java Class Generator 2*, realizes all concepts and variants as described above. Based on experiences made with the code generators developed so far, on updated requirements (e.g., the additional "Scalability" requirement) as well as on new features of the Genesys framework (such as the variant management and new SIBs), it is a complete reworking of the Java Class Generator 1. As it also adheres to new modeling guidelines and best practices (see Sect. 5.3.1 for examples) that resulted from the jABC core project, only SIBs have been reused in the usual manner, but almost no models of existing code generators. Though this led to a higher effort for the reworking, it was a deliberate design decision with respect to the Java Class Generator's role as the reference for derived code generators.

Additionally, similar to the Java Class Extruder (cf. Sect. 5.1.2), the Java Class Generator 1 was also a generator generator, as it contained features for translating modeled code generators into corresponding Java classes accessible for the various Genesys tools. This significantly increased the generator's complexity. In order to keep the Java Class Generator 2 clean and simple, this task was separated from the actual Java class generation process and moved to a dedicated code generator, the Genesys Code Generator Generator described in the previous section.

The distinction between Java Class Generator 1 and 2 is especially important for the comparison of the different variants and versions in the following Sect. 5.3. Furthermore, it has to be taken into account when examining how the code generators evolved from each other (see Sect. 5.4).

5.3 Comparison and Evaluation of the Java Code Generators

The previous chapters introduced various Java code generators created with Genesys. In the following, these different variants will be compared and evaluated on the basis of metrics and experiments that concentrate on two main aspects. First, the code generator models are examined in order to show the effects of reuse and variant management (Sect. 5.3.1). Second, the results produced by the code generators will be compared in terms of performance and source code size (Sect. 5.3.2). For referring to the different Java code generators, the following abbreviations are used:

- *JCE*: Java Class Extruder (cf. Sect. 5.1),
- *JCG1*: Java Class Generator 1 (cf. Sect. 5.2.7),
- *JCG2-SC*: Java Class Generator 2, variant for generating structured code (cf. Sect. 5.2.4),
- *JCG2-LI*: Java Class Generator 2, lightweight interpreter variant with reflective service calls (cf. Sect. 5.2.5) using the `GenericElementary-Service`,
- *JCG2-LI-GS* Java Class Generator 2, lightweight interpreter variant with explicitly generated service calls (cf. Sect. 5.2.5).

5.3.1 Code Generator Models

Size of the Code Generators:

Table 5.2 shows the number of models and SIBs used for the construction of each Java code generator. The numbers are divided into the *total* number of uses and the number of *unique* uses. The former corresponds to the number of model and SIB instances, whereas the latter indicates how many different models and SIBs have been employed. For instance, the JCE contains a total number of 215 SIB instances, which are instantiated from 34 different SIBs. The span between the total and unique number is an indicator for reuse of models or SIBs *inside one code generator*. This reuse can be measured by means of the well-known *reuse percent* metric [FT96], which is calculated as

$$reuse\ percent = \frac{reused}{total} \times 100\% \text{ , with } reused = total - unique.$$

In this context, reuse means using an existing component more than once without any modification, except for parametrization.

In Table 5.2, it is visible that the JCE is the smallest code generator, as it consists of a total number of only 28 models. However, with 3.6%, the reuse of models is very small – only one model could be used more than once. This is due to the fact that the contained models have been constructed specifically for the JCE, without any optimizations for reusability. For SIBs,

Table 5.2. Quantitative comparison of the different Java code generators

Generator	Variant	Models		SIBs		Recursion
		total	unique	total	unique	
JCE	–	28	27	215	34	–
JCG1	–	86	40	986	43	+
JCG2	SC	65	32	464	65	+
	LI	43	27	312	61	–
	LI-GS	45	29	326	68	–

the potential for reuse was much higher, as the JCE mainly employs the Common SIBs library which provides a lot of generic services. Consequently, the reuse among SIBs in the JCE is 84.2%.

As mentioned in Sect. 5.2.7, the JCG1 evolved from the JCE. With 86 models and 986 SIBs, it is the largest and most complex Java code generator, but at the same time, the reuse percentage is high: 53.5% for models and 96% for SIBs. This good reuse percentage results from the fact that in contrast to the JCE, not only the SIBs but also the models used in the JCG1 were optimized for high reusability.

However, this came at the expense of simplicity, as it resulted in very generic models and SIBs. In order to assure a broad applicability, such models and SIBs often provide a lot of parameters (i.e., configuration possibilities), which have to be considered by the generator developer and thus are more difficult to use.

An extreme example is the `ContextExpression` parameter for SIBs (cf. Sect. 3.3.2), which even allows the modeler to add dynamic behavior via a dedicated expression language. Although this is quite powerful and improves the versatility of a SIB, this mixture of modeling and programming clearly contradicts the goals formulated in *Requirement G3 - Simplicity*. Furthermore, it could be observed that more generic SIBs tend to be less coarse-granular, which decreases their abstraction value. In consequence, more SIBs are required for modeling, which is witnessed by the huge size of the JCG1.

Another reason for the higher complexity of the JCG1 in comparison to the JCE is the fact that the structured code approach is generally more elaborate than the Extruder approach. Moreover, Table 5.2 shows that the JCG1 employs recursion. As this enforces the use of a stacked execution context (cf. Sect. 3.3.2), the generator developer has to work with context scopes, which further increases the complexity of the code generator.

In terms of the employed models and SIBs, the JCG2 and its variants are situated between the JCE and the JCG1. Among the variants, SC is the biggest one, which is again due to the structured code approach being more elaborate. The SC variant is significantly smaller than the JCG1, although it uses recursion which generally adds complexity. This results from the fact that, in contrast to the approach followed when building the JCG1, the reworking did not solely focus on maximum reusability, but instead tried to keep the balance between reusability and simplicity. In consequence, SIBs became less generic and more domain-specific, and rather technical parts of the code generator previously modeled with several "small" generic SIBs have been replaced by more coarse-granular services. This approach had a slightly negative effect on the SIB reuse, which dropped by 10% to 86% in comparison to the JCG1, i.e., effectively more different SIBs were required for building the SC variant. However, this is outweighed by the significant reduction of the overall complexity and code generator size. The model reuse even only decreased by 2.7%.

The LI and LI-GS variants of the JCG2 are very small and more simple than the SC variant and the JCG1, that follow the structured code approach. However, they are not as small and simple as the JCE, which directly profits from the features provided by the Tracer and the `SIBGraphModel` data structure, which have to be emulated by LI and LI-GS. Similar to the SC variant, the reuse percentages have decreased (e.g., LI: 37.2% for models and 80.4% for SIBs), which is also a consequence of the stronger focus on domain-specificity.

Another reason for the reduced size of all JCG2 variants is the fact that any generator generator features have been moved to the Genesys Code Generator Generator (cf. Sect. 5.2.7), thus keeping the JCG2 models clean and simple. The JCE and the JCG2 directly contained those features.

Reuse among Code Generators:

As formulated in *Requirement G2 - Reusability and Adaptability*, reusability is a central goal of the Genesys approach. Due to the application of model-driven development and service orientation, it is expected that the potential for reuse should increase with a growing library of models and services. In order to understand the role of reuse for code generators that evolve from each other, e.g., via derivation or variant specification, it is important to consider the reuse of components among code generators. This has to be performed with respect to the order in which the code generators have been created, that is: JCE → JCG1 → JCG2-SC → JCG2-LI → JCG2-LI-GS. With each new code generator, the Genesys library of models and SIBs is extended, so that each evolution step constitutes the available repertoire for the succeeding steps. Accordingly, it is interesting to observe the development of reuse along this chain of evolution steps.

Unfortunately, for SIBs, such an analysis is not easily possible. Due to the sheer number of available SIBs, it is very difficult to determine (especially retroactively) which SIBs had to be newly implemented and which ones have been reused in the single evolution steps. In particular, this is difficult for SIBs originating from the Common SIBs library, which contains contributions from numerous other jABC projects and thus advances entirely independent of Genesys.

For models, the situation is much better, as the code generators do not contain models that originate from any source other than the Genesys library.

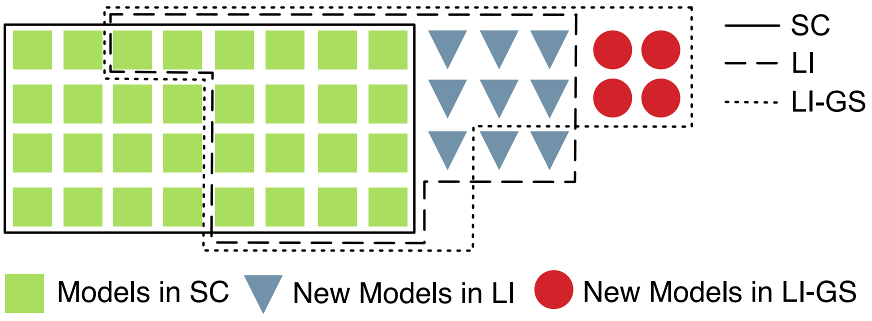


Fig. 5.14. Reuse of models among the Java Class Generator Variants

However, the analysis has to be divided into two parts: As the JCG2 is a complete reworking (cf. Sect. 5.2.7), it deliberately does not reuse any models from the previous code generators. Accordingly, model reuse has to be examined separately for JCE → JCG1 on the one hand, and for JCG2-SC → JCG2-LI → JCG2-LI-GS on the other hand.

Although the JCG1 emerged from the JCE, the model reuse is at first sight rather small. Only 5 models from the JCE have been reused without modification, i.e., 12.5% of the 40 different models contained in the JCG1 can be directly attributed to reuse. One reason for this is the lack of any variant management support, which did not exist at the time the JCG1 was created. Consequently, the derivation of new code generators often involved copying and modifying models instead of reusing them. Clearly, workarounds of that kind are not desirable, as they always lead to new files that "pollute" the Genesys library with very similar models and also impede maintenance. For instance, when fixing an error on the original model, it also has to be performed on each copied derivative that contains the erroneous parts, which is very difficult and cumbersome. In order to avoid such workarounds and the resulting problems, Genesys' variant management facilities have been created later on.

Fig. 5.14 visualizes the model reuse in the development of the different JCG2 variants. SC was the first variant that has been developed, because conceptually, it could be built along the lines of the JCG1. However, as mentioned above, the JCG2 was a complete reworking and thus no models have been directly reused from previous code generators. Altogether, the SC variant consists of 32 different models, which are depicted in Fig. 5.14 as square shapes. In the next step, the LI variant was modeled using 27 different models, marked by the dashed box in Fig. 5.14. Among those models, 18 were reused from the SC variant and 9 new models (visualized as triangular shapes) had to be created, so that 66.7% of the models in the LI variant can be attributed to reuse. The LI-GS variant has been built by means of 29 different models marked by the dotted box in Fig. 5.14. Again the variant reuses 18 models from SC, which are actually the same models that have already been reused by the LI variant. Moreover, LI-GS reuses 7 models from LI and adds only 4 new models (visualized as circles), which results in a reuse percentage of 86%. This shows an important advantage of Genesys' variant management feature: It facilitates reuse, in contrast to the "copy and modify minimally" approach outlined above.

SIB Usage:

Fig. 5.15 shows the distribution of SIBs in the different Java code generators with respect to SIB bundles. The bundle labeled "Genesys" denotes SIBs which originate from Genesys' library of services that are specific to code generation. "Control" refers to control SIBs (such as macros), which are mostly delivered by the Tracer or by the jABC framework. All remaining bundles are part of the Common SIBs library.

The usage of services from the "Script SIBs" bundle is especially interesting for code generators, as the bundle contains services for different template engines (cf. Sect. 4.1.1). As a rule of thumb, the more "Script SIBs" are used, the more templates are contained in the code generator, which increases its overall complexity. In Fig. 5.15, it is visible that according to absolute usages, the code generators that follow the structured code approach (i.e., JCG1 and JCG2-SC) contain the highest number of templates. For all code generators, at least a fifth of all used SIBs originate from the "Script SIBs" bundle (JCE: 25%, JCG1: 26%, JCG2-SC: 24%, JCG2-LI: 20%, JCG2-LI-GS: 22%), which shows that templates generally form a substantial part of each generator.

The stronger focus of the JCG2 on using domain-specific services is witnessed by the numbers for the "Genesys" bundle. While the JCE and the JCG1 contain only few of such dedicated services (JCE: 1.9%, JCG1: 0.5%), around one quarter of the different JCG2 variants consists of those SIBs (JCG2-SC: 23.7%, JCG2-LI: 27.2%, JCG2-LI-GS: 27.3%). Conversely, the JCG1 makes significantly more use of services from the "Basic" bundle, which mostly contains very basic and generic SIBs. 54.9% of the SIBs in the JCG1 originate from this bundle, which illustrates the generator's preference for generic services over domain-specific ones.

Altogether, the "Script", "Genesys" and "Basic" bundles are the sources for most SIBs used in the code generators (JCE: 65.1%, JCG1: 81.4%, JCG2-SC: 64.9%, JCG2-LI: 68.6%, JCG2-LI-GS: 71.2%). The remaining SIBs are mostly "Graph Model SIBs" for processing the input models, control SIBs and other bundles like the "Collection SIBs" or the "IO SIBs" (referred to as "Other" in Fig. 5.15). For those code generators which are not recursive (JCE, JCG2-LI, JCG2-LI-GS), the "Control" category consists of normal macros (i.e., of type `MacroSIB`) only, whereas the JCG1 and the SC variant of the JCG2 also contain `GraphSIB`s for the recursion. On average, all Java code generators manage on SIBs from 8 of around 20 existing SIB bundles.

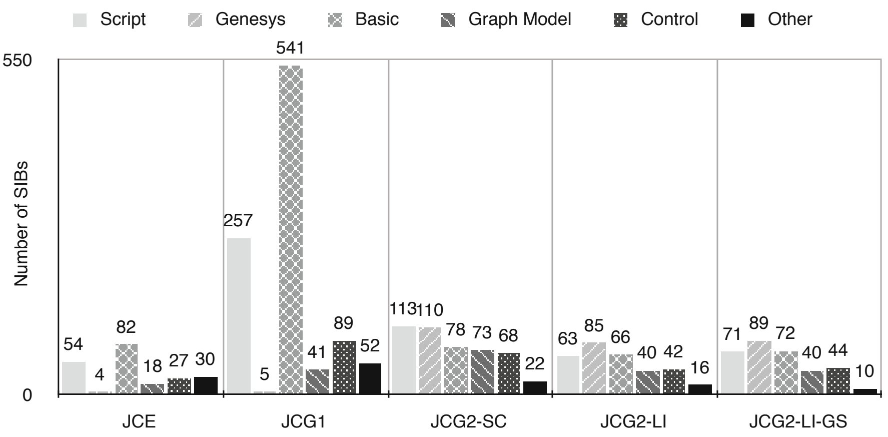


Fig. 5.15. Distribution of SIB bundles in the different Java code generators

5.3.2 Code Generator Results

This section examines and compares the generated results (i.e., Java classes) of the different Java code generators in terms of their performance and size. For the code generators themselves, these characteristics are not considered explicitly. However, all following observations and conclusions about the generation results are true for the code generators as well, because they are also results of code generation: Either the code generators generate themselves as it is the case for the JCE, or they are produced by means of a generator generator (cf. Sect. 5.2.6) that is also developed with Genesys.

Related Experiments:

Concerning the performance of generated artifacts, Lamprecht et al. [LMS09] describe several interesting results. Based on a reference process from the realm of bioinformatics, they compared the performance of a corresponding hand-written program, of directly executing the process with the Tracer in jABC, and of the artifacts produced by (amongst other Genesys generators) the JCE and the JCG1.

As the reference process also contained remote services, which were subject to network delays, the results distinguish between workflow execution time and local execution time. The former indicates the total time required for executing the process, including the time required for contained remote services, whereas the latter provides the execution time without the remote service runtimes.

Lamprecht et al. observed that the local execution time just takes between 2.7%–11.6% of the workflow execution time. The results also show that the local execution times of the generated artifacts only vary little (within a range of 3%) among the different code generators. Furthermore, the difference between the local execution times of the hand-written program, the Tracer-executed process and the generated artifacts “stays within 5 s and below 5% of the workflow execution time” [LMS09].

Regarding the size of the code, Lamprecht et al. state that on average, the generated artifacts contain around twice as much lines of code compared to the hand-written version. However, they describe the generated code as “very tense and regular, the correspondence of its structure with the models’ structure explicit (through systematically introduced comments and annotations) and [. . .] easily understandable” [LMS09].

Benchmarks:

For this book, further experiments have been conducted in order to measure the performance and size of the artifacts produced by the different Java code generators. As the basis for comparison, the experiments use reference models with specific characteristics:

- “NoOps” contains a long sequence of 100 services, which perform no operation at all.
- “Loops” includes a simple loop, that decrements a given counter with each iteration. The loop terminates if the value of the counter equals zero. At the beginning, the counter is initialized with a value of 10000. This model is realized with three services.
- “Recursion” essentially decrements a counter as well, but instead of a loop a recursive call is used. Just like for “Loops”, the recursion ends if the counter equals zero. The start value of the counter is 100. “Recursion” consists of five services.
- “All Tests“ combines all test models of Genesys’ test suite (see Sect. 6.3) by means of a common supermodel. The test models contain all kinds of control SIBs, SIB parameter types etc. that may occur in SLGs. Altogether, “All Tests” consists of 38 submodels and 168 SIBs.

All models have in common that the contained services represent either no or only very small functionality, in order to be able to focus on the local execution time. While the first three models aim at basic control flow aspects, “All Tests” is intended to represent a big hierarchical model.

Table 5.3 shows the results of the experiments[5], which have been determined with Genesys’ benchmark framework (cf. Sect. 4.3.1). In order to increase the expressiveness of the measurements, the mean execution times are determined from 100 executions of each generated artifact. For all reference models, the code generated by the JCE is the slowest, which is due to the overhead resulting from the use of the Tracer, and from the necessary construction of the `SIBGraphModel` data structure (cf. Sect. 5.1.3). The fastest artifacts are those produced by the JCG1, which is specifically optimized

[5] The experiments have been performed on a MacBook Pro with a 2.33 GHz Core 2 Duo processor, 2GB of RAM, MacOS X 10.6.6 and Java 1.5.0_19.

Table 5.3. Experimental performance results for classes produced by the different Java code generators

Generator	Variant	Mean Execution Time (s), 100 samples			
		NoOps	Loops	Recursion	All Tests
JCE	–	2.728	37.898	2.149	106.139
JCG1	–	0.002	1.388	0.096	100.686
JCG2	SC	0.002	1.470	0.103	100.571
	LI	0.122	15.316	0.371	100.792
	LI-GS	0.100	15.355	0.369	100.729

for producing results with good performance characteristics. The differences between the JCE and the JCG1 are conspicuous: The Java classes generated by the JCG1 are 27 times faster for “Loops”, 22 times for “Recursion” and even 1360 times for “NoOps” than the JCE results. Especially the high factor for the latter case is plausible, as in contrast to using a heavyweight interpreter like the Tracer, executing a sequence as consecutive statements is clearly faster and more simple.

The results for the SC variant of the JCG2 are very close to those of the JCG1, as it also uses the structured code approach. This observation also shows that in terms of performance, generating multiple classes rather than one single class has no significantly negative effects.

For LI and LI-GS, the latter variant performs slightly better, which may be due to the fact that LI-GS relinquishes the use of reflection. The results of both variants are situated between those of the Extruder and structured code approaches. For instance, for “NoOps”, the generated artifacts are 22 times faster than the JCE results, but still 61 times slower than corresponding structured code (JCG1 or JCG2-SC). In the other scenarios, the gaps are more moderate, e.g., for “Recursion”, artifacts produced by LI/LI-GS are 6 times faster than those from the JCE and 4 times slower than those generated by JCG1/JCG2-SC. This indicates that the removal of the Tracer overhead is effective, but at the same time the execution via a lightweight interpreter is still more expensive than structured code.

Interestingly, for the reference model “All Tests” there are only very small differences between all Java code generators. The reason for this is the fact that, although the reference model consists of a lot of submodels, most of them are rather small and simple in terms of control flow (e.g., short sequences and loops with few iterations). This shows that a high number of models or SIBs alone does not significantly affect the performance of the resulting artifacts.

In contrast to this observation, the size of the models clearly affects the size of the generated Java class, as visible in Table 5.4. For comparison, the size of the generated classes are measured in lines of code. The “All Tests” scenario is excluded from these considerations, as it is a hierarchical model which leads to multiple classes with the JCG2 on the one hand, and to one single

Table 5.4. Size of the generated Java source classes

Generator	Variant	Lines of Code		
		NoOps	Loops	Recursion
JCE	–	705	156	187
JCG1	–	848	98	176
JCG2	SC	426	47	76
	LI/LI-GS	445	76	89

class with the other code generators on the other hand, thus complicating the comparison.

Especially for "NoOps", i.e., for models containing many SIBs, JCE and JCG1 produce relatively big classes. Code generated by the JCE suffers from the complex construction of the `SIBGraphModel` data structure, whereas the JCG1 implements an ineffective translation that results in a method for each SIB. For "Loops" and "Recursion", both generators perform better, as the models contain less SIBs.

All JCG2 variants produce significantly smaller classes. The SC variant generates the smallest classes, which is a result of condensing and improving the code patterns used by the JCG1. LI and LI-GS produce equally sized classes close to the results of SC. In particular, this shows the lower complexity of the lightweight data structure in comparison to the `SIBGraphModel`.

5.3.3 Conclusions

Overall, the JCG2 and its variants are a clear improvement of the JCE and the JCG1. In particular, it is superior to the JCE in all disciplines. In comparison with the JCG1, the JCG2 produces smaller classes, and at least artifacts generated by the SC variant provide a similar performance. Furthermore, from a generator developer's perspective, the models of the JCG2 are significantly smaller and simpler, as they are less generic and relinquish complicated performance optimizations that only had little impact. Generally, the results above show that using genericity for increasing the potential of reuse reduces simplicity, especially for SIBs. Thus it is desirable to achieve a reasonable balance between reusability and simplicity, e.g., by means of domain-specificity. The increased model reuse in the JCG2 supports the thesis that the potential for reuse increases with each evolution step of the model library, thus meeting *Requirement G2 - Reusability and Adaptability.* Furthermore, the results also show that Genesys' variant management features used by the JCG2 facilitate model reuse (*Requirement S3 - Variant Management and Product Lines*).

LI is a good compromise between performance and size of the generated artifact, which should be appropriate for most target languages. If the target language lacks reflection capabilities, the slightly different LI-GS variant may be used instead. The approach is very simple, thus it is easy to model and to comprehend. However, a major disadvantage is the fact that for each programming language, the corresponding lightweight data structure and interpreter have to be specifically constructed. At the very least, the existing Java version may serve as a reference implementation. If the effort is considered too high or if the target language is too limited (e.g., because it misses adequate data structures such as lists or hash tables), SC is a more appropriate variant.

SC is also the better variant if performance is considered critical. However, the structured code approach is generally more complicated and thus harder to comprehend, especially for novices. Furthermore, depending on how the approach is realized, it may run the risk that the generated code diverges too much from the structure of the original input models, e.g., due to preceding model transformations. This is especially problematic for the realization of monitoring and debugging features, which need to establish the connections of the generated code to corresponding parts of the original models. In order to maintain this possibility, the generated code can, e.g., be enriched with appropriate annotations or comments which provide the necessary information (cf. Chap. 10). This entails additional effort and a higher complexity of the code generator. The lightweight data structure used by LI/LI-GS is guided by the structure of the input models, which significantly eases the establishment of the connection between the generated code and the original models.

5.4 Further Code Generators for jABC

Besides the five Java code generators presented above, 19 code generators targeting other platforms and languages have been developed. Fig. 5.16 provides an overview in form of a genealogical tree that shows how the generators were derived from each other. The lines connecting the code generators indicate two different types of evolution that may occur. First, solid lines show a direct derivation, which means using the original code generator as a basis and then modifying it in order to obtain a new one. Second, dashed lines represent the creation of a code generator by selectively reusing some (parts of) models from the original generator. Accordingly, the second type of evolution clearly involves less reuse. Furthermore, the different shapes and colors used for the code generators in Fig. 5.16 indicate the targeted platforms and languages as well as the context in which a particular code generator has been developed.

The genealogical tree consists of five connected components which represent different branches of evolution. The largest family is formed by those generators descending from the Java Class Extruder. With four cases of direct derivation and two cases of partial reuse, the Java Class Generator 1 is, in terms of being a basis for new generators, the most productive code generator. Only one generator was derived from the Java Class Generator 2 so far, which is due to the fact that it is a complete reworking which has been created later than all code generators in the Java Class Extruder family. However, the Java Class

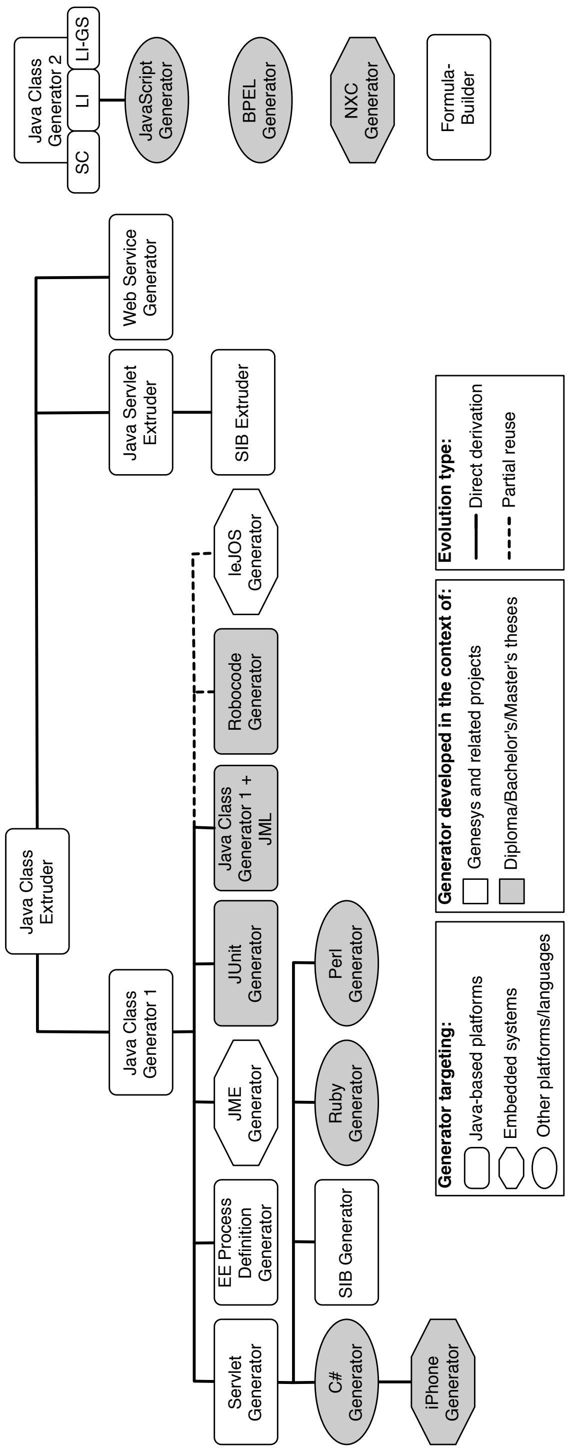


Fig. 5.16. Genealogical tree of code generators for jABC

Generator 2 is a direct result of all experience made with the previous code generators, thus it is the reference for any further code generators. It is also likely that in the future, most code generators from the Java Class Extruder family will be ported to the new concepts of the Java Class Generator 2.

The following sections elaborate on the remaining code generators depicted in Fig. 5.16 (except for the *FormulaBuilder* and the *JUnit Generator*, which will be presented later in Sect. 6.2.1 and 6.3.2, respectively). The descriptions focus on how the generators were derived, on which features of the Genesys framework have been (re)used, and on which specific challenges were posed by each code generator. Parts of these descriptions are based on [JMS08]. For illustrating the evolutionary development of the library, the order of the sections roughly reflects the order in which the single code generators emerged.

5.4.1 Servlet Extruder and Servlet Generator

In the Java Enterprise Edition (JEE) [Ora11b], a Servlet [Jav11b] is a server-side component that, e.g., runs on an application server such as the JBoss AS [Red11a]. It is realized as a Java class that is responsible for receiving, processing and answering client requests.

By resorting to the existing Java code generators, it was easy to derive corresponding pendants that generate a Servlet from an SLG [Ste+07]. Two versions have been created: the *Servlet Extruder* which was derived from the Java Class Extruder, and the *Servlet Generator*, which is based on the Java Class Generator 1 and thus uses the structured code approach. In both cases, the templates of the generators (i.e., the parameters of corresponding contained "Script SIBs", cf. Sect. 4.1.1) had to be adjusted, so that the resulting Java class extends the super class `HttpServlet` and provides the methods `doPost` and `doGet` instead of a `main` method. Please note that only the method signatures needed to be changed – the content of the `main` method could be reused for the bodies of the new methods without any modification. Furthermore, the models had to be adapted slightly, e.g., in order to remove the generator generator features of the Java Class Extruder and Java Class Generator 1, which were not required for the Servlet generators.

For further details on the Servlet Generators please refer to [JMS08], which also shows excerpts from models and templates of the Servlet Extruder. In practice, the Servlet Generator was used more frequently than its Extruder counterpart, which is due to the fact that less dependencies were required for the generated Servlets. It is visible in Fig. 5.16 that the Servlet Generator is highly productive as a basis for new code generators (in terms of direct derivation, it is just as productive as the Java Class Generator 1). As the Servlet Generator does not contain any generator generator features, which are especially difficult to comprehend for beginners, it often has been preferred to the Java Class Generator 1 as the ancestor for derivation.

5.4.2 SIB Extruder and SIB Generator

The basic idea of the generators presented in this section is the translation of an SLG into an executable SIB, which may in turn be used as a service in other SLGs. In contrast to using hierarchical modeling and embedding the SLG via a macro, such a SIB does not allow the user to inspect the model from which the SIB was generated. Accordingly, translating a model into a SIB is a simple way of protecting (sub)models from unsolicited insights. For instance, this might provide a basis for realizing different responsibilities of departments that own separate parts of a business process.

Similar to the Servlet generators presented above, two versions have been developed, the *SIB Extruder* and the *SIB Generator*, which have been derived from the Servlet Extruder and from the Servlet Generator, respectively. That way the SIB generators profited from the fact that for the Servlet generators the generator generator features of the Java pendants already have been removed (see Sect. 5.4.1).

Among the two versions, the SIB Extruder is more simple, as it focuses on generating a SIB that is only compatible with the Tracer and with code generators that follow the Extruder approach. Due to this restricted task, the Servlet Extruder only had to be modified minimally in order to derive the SIB Extruder. Essentially, the templates had to be changed so that the resulting Java class carries the `@SIBClass` annotation with a generated UID and implements the interfaces required for the Tracer as well as for all Extruders. The latter includes replacing the `doPost` and `doGet` methods by corresponding ones that realize the execution behavior of the SIB (e.g., a `trace` method for the Tracer). Similar to the Servlet generators, only the signatures of the methods had to be changed, and the content of the method bodies could be reused entirely. Additionally, two new models had to be created that translate the model parameters and model branches of the input SLG to parameters and branches of the resulting SIB.

The SIB generator goes one step further, as it produces a SIB that is compatible with all Java code generators (i.e., not only with those using the Extruder approach). Such a SIB follows the development pattern described in Sect. 3.2.1. Consequently, the generated result consists of two parts: first, a service adapter which contains the actual execution behavior of the input model and second, a SIB class which implements all required interfaces, and which delegates to the service adapter. The realization of the SIB generator required the adjustment of several templates (similar to the Extruder pendant) as well as the addition of eight new models.

Among others, the SIB generator has been used as a part of a WSDL-to-SIB importer [Lem+09], which produces corresponding SIBs from a Web Services Description Language (WSDL) file describing a Web Service [Pap08]. The importer is contained in the jETI framework (see Sect. 3.2.3).

5.4.3 Web Service Generator

The *Web Service Generator* has been developed in the context of the Semantic Web Services (SWS) Challenge [Kub+09] and is a part of the jETI framework described in Sect. 3.2.3. The generator's task is to translate an SLG to a remotely accessible Web Service. As described in [Kub+09], this could be achieved by entirely reusing the Java Class Extruder non-invasively, i.e., without any modification: Instead it was simply embedded into the model of the Web Service Generator as a macro.

For the actual generation of the Web Service, the generator first uses the Java Class Extruder to produce an executable Java class from the SLG (cf. Sect. 5.1). Afterwards, it generates wrapper classes which contain the necessary information (e.g., annotations) for publishing this generated Java class as a Web Service using the JAX-WS [Jav11a] API. Furthermore, the generator collects all necessary library dependencies and produces an Ant [Apa11a] script for comfortably starting the Web Service. Finally, all generated sources and dependencies are packaged into a ZIP archive for distribution.

In order to derive a fully-functional Web Service from an SLG, additional metadata is required that is not directly inferable from the flow-graph structure of the SLG. For instance, this information includes XML Schema Definitions (XSD [W3C11]) that specify data types and input/output types of the Web Service. In jABC, such meta-information is usually provided via the Annotation Editor (cf. Sect. 3.2.3). The Web Service Generator extracts the meta-information added via the Annotation Editor from the input SLGs and correspondingly incorporates it into the generated result.

For a detailed description of the Web Service Generator please refer to [Kub+09].

5.4.4 leJOS and NXC Generator

The *leJOS Generator*, which has originally been introduced in [Jör+07], was the first Genesys code generator that targeted an embedded system: the Lego Mindstorms Robotic Command Explorer (RCX) [Bag02]. Lego Mindstorms is a construction kit for combining robotics with the famous Lego bricks. It enjoys great popularity among hobby craftsmen, but is in particular also used for teaching in schools and universities as well as for projects and experiments with scientific or industrial background (e.g., [Ive+00], [FVZ03]).

The most important brick of such a construction kit is the RCX, which is the central command unit of a robot. It enables the attachment of different sensors, such as light or touch sensors, and motors. This brick can be controlled by means of programs, which are deployed to the RCX, e.g., via its infrared interface. Once started, such a program is able to process sensor events and to steer attached motors. Suitable programs may be implemented with a variety of supported programming languages. For instance, leJOS [Bag02] provides a Java-based firmware that replaces the original firmware of the RCX, and a simple API to control motors, sensors etc.

The basic idea presented in [Jör+07] was to enable jABC as an environment for modeling programs for the RCX, thus benefitting from advantages like the graphical SLG notation and the availability of formal verification methods. Accordingly, a code generator was required, which translates such SLGs into code that uses the leJOS API and that is executable on the RCX.

At this time, the Extruders for Java classes, Servlets and SIBs were the only code generators that were fully implemented (the Java Class Generator 1 was still being developed). Thus the first attempt of creating the leJOS Generator started off by modifying the Java Class Extruder, as this approach was already successful for the Servlet Extruder (see Sect. 5.4.1). However, the Extruder approach turned out to be unsuitable due to restrictions imposed by the RCX. In particular, the memory size of the RCX posed a problem: After deploying the leJOS replacement firmware, there is only about 16kB of free memory left for the actual programs. This is problematic both for the rather big artifacts generated by the Java Class Extruder as well as for the implied dependencies (cf. Sect. 5.1.3). As an example for the latter, the jABC framework, required by the generated artifact due to the use of the Tracer, already has a size of around 800kB. Although the leJOS linker only incorporates classes directly used by the main program[6], the sheer number of the dependencies implied by an Extruder-generated artifact most likely exceeds the RCX's memory limits. Furthermore, any classes that should be deployed to the RCX have to conform to the leJOS API, which only supports a limited subset of the Java Standard Edition (JSE). For instance, only a very restricted set of Java data types is allowed. Programs that imply incompatible dependencies are denied by the leJOS compiler. Accordingly, any third-party library used by the generated artifact, including its transitive dependencies, would have to be adapted to the restricted API, which is impracticable.

Due to these restrictions, the leJOS Generator could not directly evolve from an Extruder. Consequently, a more suitable approach has been employed, which consists of two parts. First, the complexity of the generated artifact had to be decreased by using a structured code approach (see Sect. 5.2.4) instead of the Tracer and the large `SIBGraphModel` data structure. For this purpose, at least some parts of the necessary models could be reused from the (at this time unfinished) Java Class Generator 1, indicated by the "partial reuse" relationship in Fig. 5.16. Second, apart from the leJOS API itself, any external dependencies were forbidden for the generated artifact.

The latter is a very strong restriction, as it entirely impedes the use of service adapters. In consequence, the leJOS Generator handles services differently: Instead of generating calls to the services, it directly incorporates the code that realizes the service behavior into the generated Java class. Thus, any SIB used for modeling a robot control program in jABC had to be able to expose its execution code as a string, rather than just providing

[6] Previous publications like [Jör+07] mistakenly claimed that it is only possible to upload one single class to the RCX, which is not correct.

information on how to call its underlying service functionality (cf. Sect. 5.2.1). This execution code has to be self-contained and must not imply any dependencies other than the leJOS API. This part of the leJOS Generator had to be modeled from scratch. As a positive side-effect, due to the fact that the code retrieved from a SIB has to be self-contained, no data-type mappings (such as the ones described in Sect. 5.2.3) were required for the translation: The developer who adds the code to the SIBs only employs suitable types supported by the leJOS API.

Finally, this improved approach led to considerably smaller generated artifacts, that are suitable to be deployed on the RCX. Please refer to [Jör+07] for an example of a robot program modeled with jABC.

The *NXC Generator*, which resulted from a diploma thesis [Sch07], focuses on Lego's more recent Mindstorms generation called NXT. With this new version, several drawbacks of the RCX have been remedied. For instance, the light sensors were very sensitive to the surrounding light, so that the same program led to different results, depending on the brightness of the room illumination or the solar irradiation. Furthermore, the motor movement was very imprecise.

In contrast to the leJOS Generator, the NXC Generator does not produce code for a Java-based API, which made it the first Genesys code generator to target a programming language other than Java. Instead it translates SLGs into the C-like language Not eXactly C (NXC) [Han07]), which was the most advanced and stable solution for programming NXTs at this time [Sch07, p. 33]. For NXC, a compiler is available that produces programs which are directly executable on the NXT brick, without the need for any firmware replacement. Due to the strong differences between NXC and the (at this time purely Java-based) generation results of existing code generators, no SLGs could be reused for the NXC Generator. Consequently, Fig. 5.16 visualizes it as a separate connected component. However, the generator reuses the concept of SIBs that are able to return their execution code as a string, which has already been successfully employed for the leJOS Generator. Furthermore, by restricting allowed input models to specifically structured ones (e.g., only special, explicitly compatible control SIBs are allowed), the generator rules out any unstructured models and thus does not require a structured code approach with preceding model transformations. Although this significantly simplifies the code generator itself, it has the obvious disadvantage that not every model constructed with jABC is supported.

Please refer to [Sch07] for example models and for further details on the NXC Generator.

5.4.5 BPEL Generator

The *BPEL Generator* has also been developed in the context of a diploma thesis [Gae07]. It translates an SLG into BPEL (cf. Sect. 2.3.6), which is a standard of the Organization for the Advancement of Structured Information

Standards (OASIS) for describing business processes on the basis of Web Services. Corresponding process descriptions can be directly deployed and executed on an execution engine (such as ActiveVOS [Act11a]). As BPEL is based on XML, the structure of a process description essentially differs from, e.g., the one of Java classes. In consequence, no SLGs of existing code generators could be reused for the BPEL Generator, which is thus depicted as a separate connected component in Fig. 5.16.

Furthermore, like the FormulaBuilder (cf. Sect. 6.2.1), the BPEL Generator is not based on the usage of templates. Instead it uses an implementation of the Document Object Model (DOM) [W3C09] in order to construct the XML document in-memory as an abstract form target (cf. Sect. 2.4), prior to finally writing it to a file. For this purpose, several new SIBs had to be implemented that support working with DOM.

Similarly to the NXC Generator presented above, the BPEL Generator considerably constrains the models that are allowed as an input for the translation. First, only specific SIBs, that are designed to reflect corresponding BPEL constructs such as `Assign`, `Invoke` or `Flow`, can be used. Second, the modeling style is guided by the structure of a BPEL description. For instance, some SIBs like `Flow` or `Switch` have to be used strictly pairwise. Again, such restrictions decrease the complexity of the code generator, as it does not have to deal with arbitrary models and thus does not require an approach for generating structured code. However, this comes at the expense of significantly limiting the capabilities of jABC for this application scenario, e.g., by excluding the Common SIBs library.

Currently, a new version of the BPEL Generator is being developed, which is able to handle arbitrary SLGs by using a structured code approach. With this generator, any control SIBs contained in the input models are translated into corresponding BPEL constructs. Any other SIBs (e.g., from the Common SIBs library) are automatically wrapped as Web Services and reflected by corresponding `Invoke` constructs in the resulting BPEL process description. This new BPEL Generator will be more generally applicable than the one developed in [Gae07].

5.4.6 C# Generator

The *C# Generator* [Hös08] was the first code generator which, although it targets a programming language other than Java, considerably profited from existing generators. Even though it generates code for C#, which is part of Microsoft's .NET [Mic11] framework, it was directly derived from the Servlet Generator. This can be attributed to the strong structural similarities of classes in C# and Java.

As the main task of the derivation, all templates contained in the Servlet Generator had to be translated into C# syntax. This succeeded without any problems, as for each required Java construct a semantically equivalent C# pendant could be identified (e.g., `package` vs. `namespace`, see [Hös08, p. 113] for more examples). Of course, the `doPost` and `doGet` methods produced by the Servlet Generator also had to be replaced by a C# `Main` method.

In the context of the C# Generator, the need for general data type mapping mechanisms in Genesys became apparent for the first time, as jABC's built-in data types had to be converted to corresponding data types in C#. For this purpose, Hösel [Hös08, pp. 64ff] describes a converter infrastructure which has been implemented specifically for the C# Generator, and which can be considered an early precursor of Genesys' type mapping features (cf. Sect. 4.1.2). Similar to the Genesys mechanism, the data type converter has been integrated into the generator model as a new SIB. However, the proposed infrastructure is still very much driven by the needs of C# and is not as generic as the corresponding mechanism meanwhile provided by the Genesys framework.

Hösel [Hös08, p. 113f] also lists all required type mappings: Essentially, all of jABC's simple types and most of the complex types could be translated to C# pendants, mostly by means of direct mapping (category 2 in Sect. 4.1.2, e.g., `java.lang.Boolean` to `System.Boolean`, or `java.util.HashMap` to `System.Collections.Generic.Dictionary`). Unfortunately, some of the complex types such as `ContextKey` or `MultiObject` were excluded from the conversion (category 5 in Sect. 4.1.2), which means that any models containing SIBs which use these data types are not supported by the C# Generator. In order to nevertheless include the data types, a future version of the generator could introduce corresponding new C# data types (category 4 in Sect. 4.1.2), similar to the lightweight types created for the Java Class Generator 2 (cf. Sect. 5.2.3).

In order for an SLG to be translatable to C#, each contained SIB has to provide a C# implementation via a corresponding service adapter. SIBs which are not equipped with such a service adapter lead to the abortion of the code generation process. In addition to the generated C# class, the generator also produces a C#-based version of the stacked execution context, which is emitted by the code generator and can then be used by the services (respectively their service adapter) in order to share data at runtime.

Beyond these changes, virtually no modifications of the SLG's workflow were necessary for transforming the Servlet Generator into a code generator for C#.

5.4.7 JME Generator

The *JME Generator* targets the Java Micro Edition (JME) [Ora11c], which is a special variant of Java tailored to mobile and embedded devices, such as cellphones or multi-functional printers. The resulting code employs the API defined by the Connected Limited Device Configuration (CLDC) [Jav07]. Such a configuration defines the minimum of library and virtual machine functionality that has to be available on a target device.

In comparison to normal Java classes for the JSE which are produced by the Java code generator presented above, CLDC only provides a very restricted API. In particular, this API only supports a limited subset of the data types contained in the JSE. For instance, `Vector` is the only type of collection, and the different maps are only supported via hash tables. Some data types such as `File` are not supported at all. Again, data type mappings were required in order to translate jABC's simple and complex types to CLDC-compatible ones. Consequently, the first generic version of Genesys' type mapping features was developed for the JME Generator.

For the most part, jABC's simple data types could be easily translated via identity or direct mappings, except for collections and maps, which had to be mapped to `Vector` and `Hashtable`, respectively. Table 5.5 shows the required data type mappings for jABC's complex types. Just like for the Java Class Generator 2 (cf. Sect. 5.2.3), new data types have been introduced in order to support `ContextKey` and `ListBox`, and `Password` is translated into `String` via a reductive mapping. All remaining supported data types are reduced to corresponding collection and map types. Furthermore, `ContextExpression` and `File` are excluded due to the lack of comparable functionality in CLDC. Of course, when implementing JME-compliant service adapters, a SIB expert has to pay attention that only compatible data types are used.

Table 5.5. JME mapping for complex jABC data types

jABC Data Type	**JME Type**
ContextExpression	–
ContextKey	MicroContextKey
ExtendedFile	–
ListBox	MicroListBox
MultiObject	Hashtable
Password	String
StrictCollection	Vector
StrictList	Vector

As visible in Fig. 5.16, the JME Generator was derived from the Java Class Generator 1. The first necessary change was the introduction of the type mapping mechanism described above. For this purpose, a new SIB, which establishes the data type mappings, was implemented and added to the JME Generator. Beyond the mapping of data types, the creation of the JME Generator was straightforward. Besides only a few model changes (such as removing the generator generator functionality of the Java Class Generator 1), again mainly the templates had to be modified. Instead of providing a `main` method, the generated class only features the `execute` methods (cf. Sect. 5.2.2). After deployment on the target device, those methods can, e.g., be used within a MIDlet [Jav09c], which is an artifact that realizes a JME application. Such a MIDlet is not produced by the JME Generator and thus has to be created by a developer in an additional step.

5.4.8 EE Process Definition Generator

The *EE Process Definition Generator* was developed in the context of the jABC Execution Engine (jABC EE). The jABC EE is an execution environment that deals with "large scale processes", i.e., processes which require powerful features such as scalability, failover, compensation, user interaction, versioning, multi-tenancy and hot deployment. It aims at business-critical applications deployed in big companies with a large user basis, and thus plays a role similar to process engines in BPM (cf. Sect. 2.3.6).

Bajohr and Margaria describe an early conceptual precursor of the jABC EE called the *JobFlow-Engine* [BM08], which basically extended the Tracer by the features mentioned above. In contrast to this, the jABC EE is based on JEE, as this framework already provides the foundations for some of the required features (such as hot deployment and failover). Since the beginning of 2010, the project is discontinued, but nevertheless the contribution of Genesys to the jABC EE is noteworthy.

In the context of the jABC EE, a code generator was required which translates SLGs into corresponding *process definitions*, which are the artifacts deployable on a running instance of the engine. A process definition is a simple Java class which maps an SLG to a specific data structure that is defined and understood by the jABC EE. In fact, this data structure is very similar to the lightweight data structure used for the interpreter variant of the Java Class Generator 2 (see Sect. 5.2.5) later on, thus the EE Process Definition Generator is a precursor of this approach.

Once deployed, such a process definition can be queried (e.g., "get the start SIB", "get a specific successor of a particular SIB") in order to steer the process execution on the engine. As all execution and control flow mechanisms (such as multi-threading, hierarchy or event-handling) are provided by the engine, a process definition does not have to contain any logic or code that realize these features, just like the artifacts produced by the Java Class Extruder, which delegate the actual execution to the Tracer.

Another requirement for the EE Process Definition Generator was given by the fact that a process definition may contain much more information than a plain SLG, such as:

- *Security information*: This includes roles and rights that determine whether a specific person is allowed to execute a process definition or service on the jABC EE.
- *Client-specific information*: Process definitions may contain steps demanding an interaction with the user, e.g., in order to ask for information necessary to proceed with the process. These interaction points are independent of how a concrete client interface is realized. For instance, the user may interact with one and the same process definition via a rich

client (e.g., based on Java Swing) or a thin web-based client. The client-specific information establishes mappings that tell the jABC EE which client interface (e.g., which web form or which dialog) can be used for interaction at a particular point in the process definition.
- *Version information*: When a process definition has to be modified (e.g., because a workflow has been changed), the jABC EE allows hot deployment on a running instance of the engine without requiring to restart or stop the server. This is especially interesting for business-critical applications which do not allow for any downtimes. In order to be able to safely finish the execution of "old processes" without being disrupted by hot deployment, versioning of process definitions is required. Consequently, every process definition carries version information, which is adjusted in case a process has to be modified.

Just as for the Web Service Generator presented above, this metainformation is provided via the Annotation Editor (cf. Sect. 3.2.3).

The EE Process Definition Generator collects all information attached to the jABC project and to the SLGs along with their constituent parts, and incorporates it into the resulting process definition. The generated artifact usually consists of multiple files:

- one Java class for each process definition,
- an Enterprise JavaBean (EJB) [Ora11a], which allows the registration and the deployment of the generated process definitions on the jABC EE, as well as
- an XML file declaring the main processes (i.e., process definitions which may be used to start an application, usually corresponding to the topmost SLGs of a hierarchy). Furthermore, this file contains the security, version and client-specific information described above.

Although conceptually closer to the Extruder approach, the EE Process Definition Generator was derived from both the Java Class Extruder and the Java Class Generator 1. From the former, it borrowed the structure of the generation processes and combined it with the generic and performance-focused modeling style of the latter. Altogether, 23 new models had to be created.

5.4.9 JML Extension for Java Class Generator

This extension of the Java Class Generator 1 was realized as a part of a diploma thesis [Fis09] that aimed at enabling the validation of design by contract (DBC) [Mey92] specifications for jABC models. According to DBC, software components are annotated with contracts, e.g., consisting of preconditions, postconditions and invariants, which are specified in a formal and thus precise way. For instance, corresponding tools then may perform runtime checking in order to validate whether a given software system fulfills the contracts associated with its components.

The Java Modeling Language (JML) [Bur+05] enables the use of DBC for Java. For this purpose, contracts specified in JML are embedded into the Java code as special annotation comments marking, e.g., classes, interfaces and methods. Several tools are available that work with JML-annotated Java programs, such as jmlc/jmlrac [CL02; Bur+05] for runtime checking or ESC/Java2 [Cha+05] for static checking.

In order to realize the DBC concept for SLGs, the solution created in the diploma thesis consists of two parts:

1. A method has been developed that allows an application expert to specify contracts for an SLG and its constituent parts (especially SIBs). The contracts are specified in a simple input language via wizard-like editors, that are based on jABC's Annotation Editor.
2. A dedicated code generator then translates the model to a Java class annotated with corresponding JML specifications.

The code generator has been realized with Genesys as an extension of the Java Class Generator 1, which performs the translation to a Java class as usual. In order to support the translation of the contracts annotated to an SLG, several additions to the generator were required. Essentially, when processing an input SLG, the generator has to check for each model constituent whether it is annotated with a contract. If a contract could be found, it has to be parsed, translated to JML and incorporated into the resulting Java code. Accordingly, the JML extension can be considered a specific code generator embedded into the Java Class Generator 1.

This extension included the addition of around 23 new SLGs that realize the tasks mentioned above. The parsing of the contracts is performed by means of a special SIB bundle from the Common SIBs library, which allows processing any content attached via the Annotation Editor (those SIBs have also been used for the Web Service Generator and for the EE Process Definition Generator presented above). The actual code generation for the contracts is realized with templates. No new SIBs had to be implemented for this extension.

Please note that the DBC concept for SLGs developed in this diploma thesis is neither restricted to JML nor to Java. In fact, by selecting or implementing another code generator, SLGs annotated with contracts can be translated for other target languages and DBC techniques.

5.4.10 iPhone Generator

The *iPhone Generator* is also the result of a diploma thesis [Spi09]. It targets *iOS* [App11], which is the operating system of Apple's well-known mobile devices such as the iPhone and the iPad. Applications for iOS are implemented in *Objective-C* [Koc09], which is an extension of ANSI C [ISO05] and which is syntactically based on Smalltalk [GR83].

Due to the fact that iOS is designed for mobiles devices, corresponding applications are subject to strong restrictions. For instance, an application can be interrupted, sent to the background or even terminated. This may, e.g., happen if the user presses specific buttons (such as "home" button), if the device is put into sleep mode for saving battery power, or if another application with a higher priority comes to the foreground, such as the telephone application in case of an incoming call. Furthermore, applications need to be aware of the devices' limited memory, as the system may terminate applications in order to free required memory. Apart from these interruptions, each application on iOS runs in a separate sandbox and thus has only very restricted access to system resources, such as the network or the hardware of the device. For accessing some resources like the Global Positioning System (GPS) hardware, the applications even have to request user permission. The code produced by the iPhone Generator has to consider such restrictions.

As depicted in Fig. 5.16, the iPhone Generator is mainly derived from the C# Generator. However, several considerable changes were necessary. Primarily, as the syntax of Objective-C is very different from the syntax of C# and Java, all templates had to be newly developed. Furthermore, classes in Objective-C are structured differently than in C# and Java, as the interface definition and the implementation of a class are divided into two separate files. In order to generate this class structure, several models had to be modified.

Spitzer [Spi09, pp. 42–49] describes further differences and characteristics which required the modification of existing and the development of new generator models. For instance, the resulting Objective-C code also contains a `main` method just like classes in C# and Java, but in contrast to the artifacts produced by the previously introduced code generators, this method is only used for simple initialization tasks.

The actual execution of the generated application is performed in a method called `applicationDidFinishLaunching`, which is additionally generated. This method is part of the *application delegate*, which is a special object that is attributed to each application and which is able to process specific events that may occur in the application's life cycle. Correspondingly, `applicationDidFinishLaunching` is called when the user starts the application on the mobile device. The application delegate also processes the interruptions mentioned above. The generated application needs to be able to handle all events received by the application delegate. Furthermore, the generated code always has to be executed in a separate thread so that it does not block the user interface of the device.

As another necessary modification, the conversion of data types had to be adjusted to Objective-C based on the converter infrastructure of the C# Generator (see Sect. 5.4.6). The required data type mappings are listed in [Spi09, p. 47]. Most simple jABC types are included, except for special sorted maps and sets, which have no equivalent counterparts in Objective-C. Unfortunately, `ContextKey` is the only complex jABC type that is supported. As another limitation, the iPhone Generator provides only limited support for control SIBs: Only macros are supported, but other control mechanisms like multi-threading and event handling are missing.

Similar to the C# Generator, the iPhone Generator emits a small runtime environment in addition to the generated class, called the *iABC* environment [Spi09, pp. 53ff]. iABC contains an Objective-C implementation of the stacked execution context as well as required data types. Furthermore, this environment manages the communication between the application delegate and the service adapters in order to process events.

As another result from restrictions of the target devices, the iABC environment also distinguishes between two different types of services. First, there are services which are realized or made accessible via service adapters, analogous to the other code generators presented previously. For the iPhone Generator, such service adapters have to be implemented in Objective-C. However, for iOS, this approach is only appropriate for functionality which does not require long computation time. In order to cope with long-running tasks, the runtime environment provides a second type of service, called the iABC services. Those services execute long-running tasks by means of specific iOS facilities, which support, among other things, the interruption by higher prioritized applications. The iPhone Generator has to consider the type of a SIB's service implementation in order to generate corresponding service calls.

In spite of these differences to existing code generators, only six new models and one new SIB (for the registration of the required data type converters) had to be implemented. The remaining functionality of the code generator could be realized by parametrization and modification of existing models.

As a further evolution step, the iPhone Generator could serve as a basis for a general code generator for Objective-C that, e.g., produces MacOS applications. Such an Objective-C Generator could be derived with little effort [Spi09].

5.4.11 Code Generators for Ruby, Perl and JavaScript

The *Ruby Generator* [Kol10] is another example of derivation from the Servlet Generator. Ruby [FM08] is an interpreted multi-paradigm programming language, which supports, e.g., procedural, object-oriented as well as functional programming. The execution of a script written in Ruby requires a corresponding runtime environment, such as the Java-based JRuby [Com11].

Similar to the case studies for C# and the iPhone, the Ruby Generator requires any service adapters to be implemented in Ruby. Service adapters are realized as *modules*, which are separate files that can be included and used in the Ruby script produced by the code generator. As for the other non-Java code generators, the stacked execution context had to be reimplemented in Ruby.

Due to Ruby's dynamic type system, the code generation for data types is much more simple, as the type information does not have to be incorporated in the resulting code. In order to take advantage of this fact, the Ruby Generator employs reductive type mappings and (more rare) type exclusions rather than introducing new data types for Ruby [Kol10]. Only for `ContextKey` the introduction of a new type was inevitable in order not to loose the contained context scoping information. In contrast to other non-Java code generators, the Ruby Generator also supports the `ContextExpression` type. In all other case studies, this type had be excluded due to the lack of a corresponding EL implementation. Ruby does not contain such an implementation out-of-the-box either, but it provides powerful constructs for processing strings. These constructs made it easy to implement a corresponding EL version in Ruby [Kol10, p. 46].

As a result of the differences mentioned above, all models of the Servlet Generator had to be adapted. Furthermore, all templates had to be rewritten in Ruby. Due to Ruby's condensed syntax and its focus on "convention over configuration", the size of most templates was considerably reduced, and some templates even could be omitted entirely. Altogether, no new models or SIBs had to be developed.

Another interpreted programming language is targeted by the *Perl Generator* [Beu10]. Just like Ruby, Perl [Wal00] supports multiple programming paradigms and uses a dynamic type system. Consequently, the derivation steps and the effort required for realizing the Perl Generator were very similar to those of the Ruby Generator. Please refer to [Beu10] for more details on the Perl Generator.

Finally, the *JavaScript Generator* [Tol11] translates SLGs into JavaScript [Fla06], which also has characteristics similar to those of Perl and Ruby (interpreted, multi-paradigm, dynamically typed). However, in comparison to the generators for Ruby and Perl, the JavaScript Generator is most recent and thus is derived from the LI variant of the Java Class Generator 2 (cf. Sect. 5.3) instead of the Servlet Generator. Apart from this difference, the required derivation steps were again very much comparable to those of the generators for Ruby and Perl (no new models required, specific data type mappings and service adapters etc.).

5.4.12 Robocode Generator

The *Robocode Generator* resulted from a diploma thesis [Sto10], that aimed at modeling strategies for the programming game *Robocode* [Lar11] in jABC. In Robocode, small virtual robots are developed that compete against each other in tournaments and challenges.

Essentially, such a robot consists of a chassis, a gun and a radar, all of them moveable independently. The radar is used to scan for other robots. Furthermore, there are several general conditions that need to be considered, such as the boundaries of the playboard, a robot's remaining energy and the temperature of the gun.

A robot is programmed as a simple Java class (further dependencies are allowed). For this purpose, Robocode offers a corresponding Java API. Storz refers to such a robot program as a *strategy* [Sto10]. Furthermore, Robocode provides a graphical environment for testing programmed robots and for letting them compete against other robots.

Apart from the obvious entertainment value, Robocode can be used for practical learning of object orientation and Java programming. As an extension to this, Storz [Sto10] proposes the use of jABC in order to formulate robot strategies as models. Accordingly, instead of a programming language, the abstract SLG notation is used for creating a strategy, which is especially easier for beginners. By means of models, they can learn abstract concepts like iteration or recursion prior to being distracted from the syntactic issues of a full programming language.

The approach underlying the diploma thesis is inspired by the project *ConnectIT* [BJM09]. This project also involves modeling game strategies in jABC, but in this case for the popular "Connect Four" game. Once modeled, those strategies can be executed with the Tracer and directly tested in a dedicated graphical game environment realized as a jABC plugin. Since several years, ConnectIT is applied successfully in projects with pupils and students. The application of Robocode as a new game for this concept is more ambitious, as the robot strategies are much more complex and challenging due to the higher degree of freedom in the game.

In Robocode, robots are developed based on events. Each robot has a standard behavior (e.g., "Move along the wall and avoid any obstacles.") which is repeated by the system until the end of the game. Furthermore, additional behavior is specified based on particular events which may occur in the game, such as "radar scanned another robot" or "wall rammed".

Due to this structure, jABC models for robot strategies differ from usual SLGs employed in other application scenarios. As a main difference, a robot strategy modeled in jABC is separated into several connected components called *sections* [Sto10, pp. 46–48]. One section describes the robot's standard behavior, and there is an additional section which specifies the behavior for each event that should cause a reaction of the robot. Each section is identified by means of a special corresponding start SIB. Furthermore, strategy models may only contain special SIBs, the *Robocode SIBs*, which reflect the Robocode API (e.g., movements of robot body/gun/radar, or firing the gun). Those strong restrictions of the models are acceptable for this application scenario, as corresponding users only work with a customized, simplified version of jABC that is not suitable for creating arbitrary models anyway.

The Robocode Generator is responsible for translating those special SLGs into corresponding Java classes, which can be executed and tested in Robocode's graphical environment. As the structure of the input models is very constrained and different from usual SLGs, the bulk of the generator had to be newly created. As depicted in Fig. 5.16, few models of the Java Class

Generator 1 (e.g., standard tasks like model loading) could be reused, which is indicated by the "partial reuse" relationship.

The generator translates every section that is contained in the strategy into a corresponding Java method. For instance, the section for the standard behavior is translated to a `run` method, and for the section specifying the event of scanning another robot, the generator produces a method called `onScannedRobot`.

In order to keep the size of the resulting Java class as small as possible[7], the Robocode SIBs relinquish the concept of service adapters. Similar to the leJOS Generator and the NXC Generator (cf. Sect. 5.4.4), the SIBs expose their own execution code as a string which is then incorporated in the generated code. Consequently, no data type conversion is required, as it is delegated to the developer who implements the Robocode SIB.

Furthermore, the restriction of the models to the Robocode SIBs prevents the creation of unstructured or erroneous models. Such SLGs are detected by a specific checker [Sto10, pp. 70f], which is always executed prior to code generation. Thus the Robocode Generator does not require any additional mechanisms for handling unstructured or erroneous input SLGs.

In addition to the 14 new models of the generator, 14 new SIBs were required. The latter covered tasks specific to the Robocode scenario, such as handling the sections of the strategy models.

By means of a dedicated jABC plugin, a modeled strategy can be executed and tested immediately. Entirely transparent to the user, the Robocode Generator translates it into a Java class, which is then automatically compiled and deployed in a temporary directory. Afterwards, the user may configure a tournament which is then directly performed within Robocode's graphical environment.

[7] For some Robocode challenges, the size of the robot's code is important.

6

Verification & Validation of Code Generators

Sect. 1.1 established verification and validation (V&V) as a central requirement of the Genesys approach (*Requirement G5 - Verification and Validation*). As the primary objective, V&V techniques should support the generator developer in constructing robust code generators, that correctly produce the desired results. First of all, this includes standard techniques of software engineering such as testing and static analysis, which are of course applicable to code generators just like to any other piece of software. Beyond that, the Genesys approach provides the decisive advantage that code generators are available as models. In consequence, they are amenable to formal methods like model checking (cf. Sect. 3.4).

Fig. 6.1 shows an overview of the V&V techniques that are employed in Genesys. As visible in the center of the figure, V&V is applied to code generator models as well as to the contained services. Above the dashed line, there are those techniques which support the generator developer while modeling a code generator. The tools that are available for this purpose require a set of local and global constraints (cf. Sect. 3.1), which are checked continuously during the modeling activity. In jABC, local constraints are checked by means of the LocalChecker and, as described in Sect. 3.2.3, specified by the SIB expert as corresponding Java code in the SIB. Furthermore, global constraints are checked by means of model checking which is provided by the jABC plugin GEAR (cf. Sect. 3.4). In order to spare the generator developer learning a temporal logic, such global constraints can also be specified in a graphical and pattern-based manner as jABC models, using the FormulaBuilder tool (which is in fact another example of a Genesys code generator).

Below the dashed line, testing techniques check whether the code generators and the contained services work as desired (or stipulated). As the services are implemented as code, they can be tested the usual way, e.g., by means of unit tests. For instance, in the case of Java, testing frameworks like JUnit [Bec04] and TestNG [BS07] are suitable for this purpose. For testing entire code generators in Genesys, a dedicated framework has been developed, and also implemented in case of the various jABC code generators (cf. Sect. 5).

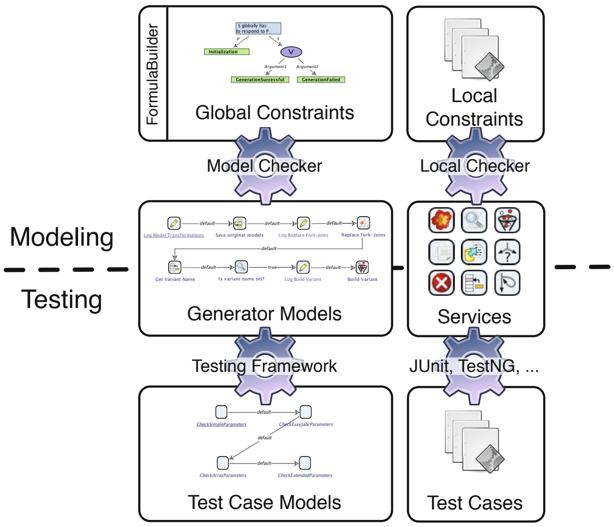


Fig. 6.1. Verification & Validation in Genesys

According to the concept underlying this framework, test cases as well as test inputs are, just like the code generators and the global constraints, specified as models in jABC.

Please note the special focus on the simple usage of the supported V&V mechanisms, especially of those working on code generator models. In order to use the model checker or Genesys' testing framework, the generator developer does not have to learn any new language or specification formalisms (*Requirement G3 - Simplicity*). The language used for all these tools is given by jABC's SLGs, which is the same language that is used for the development of the actual code generators. The only syntactic difference between those models is that different SIBs are used for their construction.

As constraints and test cases are, once created, added to the Genesys Framework, they form a continuously growing knowledge base for building robust code generators (bottom of Fig. 4.1 in Chap. 4). Consequently, each new code generator has to fulfill all suitable constraints and pass all appropriate tests from the knowledge base, which reduces the likelihood of repeating known mistakes or bugs.

With this holistic and integrated support of V&V, Genesys provides powerful mechanisms for the development and quality assurance of robust and reliable code generators, which is (to the author's knowledge) unique among existing code generation frameworks. All mechanisms have been field-tested for the jABC code generators presented in the previous chapter. They are, without further ado, applicable to other domains: In particular, the LocalChecker and the model checker GEAR are not restricted to the domain of code generation, but are intended for any models and services in jABC.

The following sections elaborate on the single V&V facets in Genesys: Sect. 6.1 is concerned with the checking of local constraints, Sect. 6.2 focuses on global constraints and model checking, and Sect. 6.3 describes the testing capabilities of Genesys.

6.1 Local Constraints for Code Generators

Sect. 3.2.3 presented the LocalChecker as an important tool for checking the correct use and configuration of available services in SLGs. This section elaborates on which local constraints are typically checked in the context of Genesys.

All Common SIBs and all dedicated Genesys services enforce the use of the built-in standard checks of the LocalChecker. These standard checks already impede the most common modeling errors, such as unconnected edges with missing source or target nodes (which resembles a breach in the execution flow) or missing branch assignments (leading to inaccessible execution paths). Another common problem which is avoided with a local constraint is missing names for context keys (cf. Sect. 3.3.2). Services that are misconfigured this way might fail reading data from or writing data to the execution context. Please note that the LocalChecker is only able to check whether the generator developer generally provided a name for a context key. However, due to its local scope, which is limited to single SIB instances, the tool is, e.g., not able to detect unused context keys or SIBs that try to access non-existing context keys. Instead, data flow analysis (e.g., via model checking [Ste91; LMS06] or dedicated tool kits [Kle+96]) can be used for spotting such problems.

Another application of local constraints, which is more specific to the domain of code generation, is given by the SIBs that make the various template engines available as services (cf. Sect. 4.1.1). For these SIBs, templates are usually specified as values of SIB parameters, which is one of the most frequent activities when developing a code generator with Genesys. Erroneous templates may cause the code generator to produce code that is not compilable, or that even shows unpredictable or undesired behavior. Consequently, apart from the trivial constraint that a template has to be specified and thus must not be empty, the generator developer is further supported by a local check that validates the syntactic correctness of the template. This check is usually realized on the basis of built-in parsers provided by the corresponding template engines. Accordingly, in case of the templates, the LocalChecker acts in the background as a simple syntax checker, which emits an error message as soon as the generator developer mistypes in a template. Fig. 3.5 in Sect. 3.2.3 contains an example of such an error message, caused by a syntax error in the Velocity template specified for the SIB `Generate Main Method Header`. In conjunction with facilities from Genesys' developer tools, such as the editor with syntax highlighting for VTL (cf. Sect. 4.3.1), this local constraint significantly curbs the danger of incorrect templates.

Furthermore, the LocalChecker can also be employed for realizing modeling conventions or best practices. For instance, Genesys uses a local constraint in order to remind generator developers of documenting the single SIB instances contained in the developed models. As the violation of this constraint is not a serious error, it is displayed to the generator developer as an information (the LocalChecker's lowest severity level).

6.2 Global Constraints for Code Generators

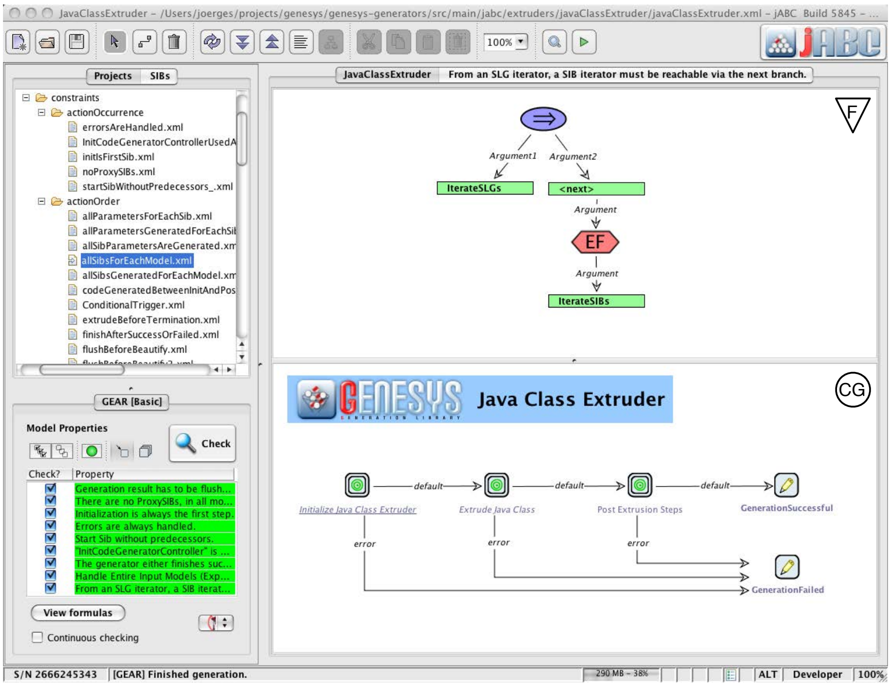
As the usage of jABC's model checking plugin GEAR has already been described in Sect. 3.4, the following sections elaborate on how global constraints are specified and organized in Genesys. First, Sect. 6.2.1 introduces the FormulaBuilder, which allows the specification of global constraints as jABC models, and which is also itself an application of Genesys. Afterwards, Sect. 6.2.2–6.2.5 present the constraint library by exemplifying typical constraints, and by motivating the creation of new constraints and patterns. The sections are mostly based on [JMS11].

6.2.1 FormulaBuilder

As already mentioned above, the FormulaBuilder [JMS06] is a tool that aims at simplifying the specification of global constraints which are, e.g., required for model checking. Instead of presupposing knowledge about property specification formalisms like temporal logics, the central idea of the tool is to allow the graphical specification of constraints as SLGs, which is the same notation that is used for modeling the actual system that is to be checked. As such graphical constraints are special jABC models, they are also called *formula graphs*. In comparison to normal SLGs, there are two restrictions on formula graphs. First, as formula graphs represent an abstract view of the syntactic structure of constraints, they usually are trees or directed acyclic graphs. Second, only specific SIBs, called *Formula Building Blocks* (FBBs) can be used to model constraints this way. Formula Building Blocks (FBBs) represent the parts of a formula such as operators or operands.

The FormulaBuilder provides a large library of FBBs for creating constraints, including logical, arithmetic, comparison and set operators, most of the specification patterns proposed by Dwyer et al. [DAC99], and other GEAR macros (cf. Sect. 3.4.1). Especially when using the specification patterns, the benefit of graphically modeling constraints becomes apparent: As the patterns virtualize from usually complex formulas, the graphical SLG representation is small and intuitive (the examples in the following sections elaborate on that). Furthermore, the use of hierarchical models allows the creation of composite constraints and new patterns.

Apart from the FBB library, the FormulaBuilder contains a retargetable code generator that translates any formula graph into a desired specification formalism such as CTL or the modal μ-calculus (cf. Sect. 3.4.1). Accordingly, the FormulaBuilder targets multiple languages, including different specification formalisms as well as different concrete syntaxes of one and the same formalism (e.g., for use with different tools).



JavaClassExtruder – /Users/joerges/projects/genesys/genesys-generators/src/main/jabc/extruders/javaClassExtruder/javaClassExtruder.xml – jABC Build 5845 – ...

JavaClassExtruder | From an SLG iterator, a SIB iterator must be reachable via the next branch.

```
constraints
  actionOccurrence
    errorsAreHandled.xml
    InitCodeGeneratorControllerUsedA
    initIsFirstSib.xml
    noProxySIBs.xml
    startSibWithoutPredecessors_.xml
  actionOrder
    allParametersForEachSib.xml
    allParametersGeneratedForEachSit
    allSibParametersAreGenerated.xm
    allSibsForEachModel.xml
    allSibsGeneratedForEachModel.xm
    codeGeneratedBetweenInitAndPos
    ConditionalTrigger.xml
    extrudeBeforeTermination.xml
    finishAfterSuccessOrFailed.xml
    flushBeforeBeautify.xml
```

Check?	Property
✓	Generation result has to be flush...
✓	There are no ProxySIBs, in all mo...
✓	Initialization is always the first step.
✓	Errors are always handled.
✓	Start Sib without predecessors.
✓	"InitCodeGeneratorController" is ...
✓	The generator either finishes suc...
✓	Handle Entire Input Models (Exp...
✓	From an SLG iterator, a SIB iterat...

S/N 2666245343 | [GEAR] Finished generation. | 290 MB – 38% | ALT | Developer | 100%

Fig. 6.2. Example: Verifying the Java Class Extruder with GEAR and the FormulaBuilder

Fig. 6.2 illustrates the interplay of the FormulaBuilder and the model checker GEAR. The top right of the figure depicts a constraint that has been specified as a formula graph, and that will be translated by the FormulaBuilder into the following formula meaning "*From an iterator that processes SLGs, an iterator that processes SIBs has to be reachable via the 'next' branch.*":

$$\mathsf{IterateSLGs} \Rightarrow \langle next \rangle\ EF(\mathsf{IterateSIBs})$$

This formula graph can be dragged into the GEAR inspector shown on the bottom left of Fig. 6.2. It is then added to the constraint library and translated on-the-fly by the FormulaBuilder, each time the constraint is required. Via the GEAR inspector and while modeling, the user is always able to check whether all selected constraints are satisfied by a given SLG and its submodels, as it is the case for the Java Class Extruder (cf. Sect. 5.1) displayed on the bottom right of Fig. 6.2.

The FormulaBuilder is also another application of Genesys. However, the very first version of the FormulaBuilder's generation mechanism (the one described in [JMS06]) was completely implemented "by hand". Later on, Genesys has been used for a complete redevelopment of the facility, as the FormulaBuilder can be considered a special, very flexible kind of code generator, which allows to configure the syntax of the generated result via an open interface (for details see [JMS06]). This configuration is performed by writing a special Java class (the so-called *target syntax*) which makes use of a dedicated API that emulates a notation close to the common BNF.

During the redevelopment of the FormulaBuilder's generation mechanism as a Genesys code generator, the bulk of the algorithm could be easily modeled by, for the most part, only resorting to the building blocks contained in jABC's Common SIBs library. In contrast to most other generators in Genesys (except for the BPEL Generator, cf. Sect. 5.4.5), this code generator is not template-based. Instead it uses a rule-based transformation (cf. Sect. 2.4.3): Along with the actual syntax description, the target syntax specification contains the transformation rules that describe how a particular FBB is translated to a part of a formula, so that the FormulaBuilder directly retrieves the necessary information from the selected target syntax specification.

In order to add this functionality to the code generator, the existing code for dealing with target syntaxes had to be divided into reasonable chunks so that suitable SIBs could be created. This resulted in around 10 new SIBs, which were required to completely model the generation facilities of the FormulaBuilder in Genesys.

6.2.2 The Constraint Library

As mentioned above, global constraints form a library which is an integral part of the Genesys Framework. This constraint library serves as a corpus of rules and guidelines for constructing code generators that, just like Genesys' repertoire of services and models, is growing continuously. There are several occasions which cause a growth of the library, e.g.:

- A new code generator is developed and raises new constraints.
- A bug traces back to a modeling mistake which has been found in a code generator. Consequently, a new constraint is added to the library in order to assure that the mistake does not reoccur.
- Existing constraints are recombined to form new, in most cases stronger and more strict, constraints.

As depicted in Fig. 6.2 at the top left side, the organization of the library is based on the semantics of the constraints. This structure is inspired by the organization of the well-known specification pattern system presented by Dwyer et al. [DAC99]. Accordingly, most constraints are currently distinguished by whether they say something about the occurrence ("`actionOccurrence`") or the relative order ("`actionOrder`") of actions.

The following sections provide example constraints from both categories. In most cases, the constraints are illustrated by means of their representation as formula graphs. For textual notations of constraints, basically the variant of CTL introduced in Sect. 3.4.1 is used, along with several non-standard additions that resemble GEAR's macros and other specifics of its input syntax. Any employed additions to the CTL variant will be explained separately in the corresponding sections.

6.2.3 Occurrence Constraints

Fig. 6.3 shows three example constraints belonging to the `actionOccurrence` category. The depicted formula graphs represent the following constraints (the numbering corresponds to Fig. 6.3):

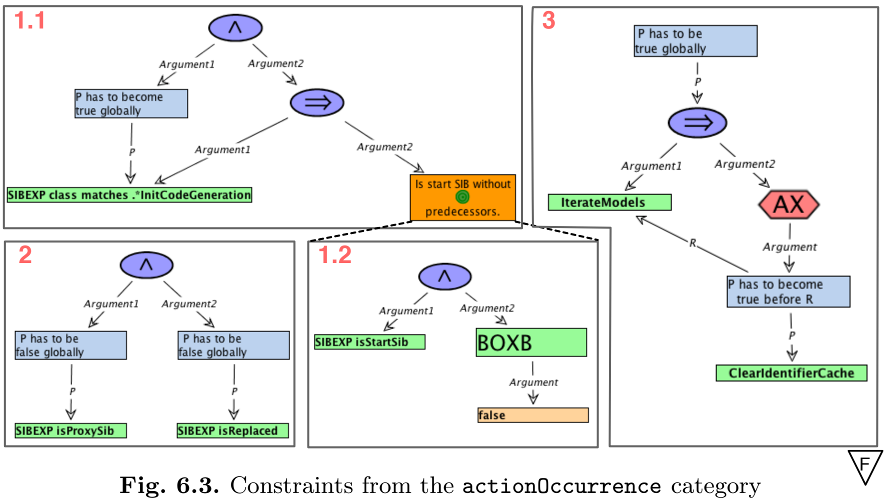


Fig. 6.3. Constraints from the `actionOccurrence` category

(1.1 & 1.2) *SIB* ***`InitCodeGeneration`*** *is used and is a start SIB without any predecessors:*

$$existence_{globally}(SIB.class == .^{*}InitCodeGeneration)$$
$$\wedge((SIB.class == .^{*}InitCodeGeneration)$$
$$\Rightarrow (SIB.isStartSib\ \wedge \overline{[.]} false))$$

(2) *No proxy SIBs:*

$$absence_{globally}(SIB.isProxySib)\ \wedge\ absence_{globally}(SIB.isReplaced)$$

(3) *After processing a model, the identifier cache has to be cleared before processing the next model*:

$$universality_{globally}(\mathsf{IterateModels} \\ \Rightarrow AX(\mathsf{ClearIdentifierCache}\ existence_{before}\ \mathsf{IterateModels}))$$

The formula graphs 1.1 and 1.2 exemplify the use of hierarchy for modeling constraints with the FormulaBuilder. Formula graph 1.2 corresponds to the formula $SIB.isStartSib \wedge \overline{[.]}false$, which is true for all start SIBs without any predecessors (usually applies to the very first SIB of an SLG hierarchy). In this formula, *SIB.isStartSib* is a GEAR macro, and the backward box operator $\overline{[.]}$ (resp. *BOXB* in formula graph 4) without any explicit actions (resp. branches) refers to all actions (i.e., it is equivalent with $\overline{[\mathcal{A}]}$). By means of hierarchy, this formula graph is embedded into formula graph 1.1 in order to produce a new constraint.

The resulting composite constraint demands the use of a SIB called `InitCodeGeneration`, which is obligatory for older code generators. In earlier versions of Genesys, this SIB was used to initialize required data structures and facilities, such as the list of reserved keywords and the identifier generation functionality (cf. Sect. 4.1.3), in one step. In current versions, the SIB is deprecated and has been separated into several SIBs in order to be more modular. However, older code generators still rely on the `InitCodeGeneration` SIB. For ensuring its presence, the constraint uses *SIB.class == .*InitCodeGeneration*, which is true for all SIBs with a class name matching the regular expression *.*InitCodeGeneration*. Furthermore, a specification pattern is used: "Existence" with scope "globally" means "P has to become true globally" and corresponds to the CTL formula $AF(\mathsf{P})$ [DAC99]. Via the embedded formula graph 1.2, the constraint additionally demands that `InitCodeGeneration` is the very first SIB of any code generator.

When comparing formula graphs 1.1 and 1.2 in Fig. 6.3 with the corresponding textual formula given above in **(1.1 & 1.2)**, the benefit of the graphical representation becomes apparent. Due to the structural reuse via the hierarchy mechanism of SLGs, formula graph representations are particularly beneficial for defining complex formulas. The example also illustrates that the high reusability already observed for the code generators also applies to constraints: Once created, formula graph 1.2 is a ready-made building block that can be reused in any constraint that includes the demand for using a specific start SIB (*Requirement G2 - Reusability and Adaptability*).

The constraint specified by formula graph 2 in Fig. 6.3 is not only applicable to code generators, but to all kinds of executable SLGs, as it checks the absence of any proxy SIBs (cf. Sect. 3.2.2). Although an SLG containing proxy SIBs can be loaded and modified without any problems, it cannot be used for execution or code generation, as proxy SIBs are only placeholders that lack the required service implementations. In order to ensure that no such placeholders are contained in an SLG, constraint 2 uses GEAR macros to check two cases:

- *SIB.isReplaced* is true if a SIB has been automatically replaced by a proxy SIB.
- *SIB.isProxySib* is true if a user misapplied a proxy SIB for modeling (which was technically possible in earlier versions of jABC).

Constraint 2 demands that these two cases do not occur in any SLGs by using the specification pattern "Absence" with the scope "globally". This pattern means "P has to be false globally" and corresponds to $AG(!\mathsf{P})$. The textual notation of the constraint is given above in **(2)**.

Finally, formula graph 3 in Fig. 6.3 represents a constraint that is required for all code generators that employ the Genesys services for identifier generation (cf. Sect. 4.1.3) along with the multiple class generation approach (cf. Sect. 5.2.2). As visible from the constraint's textual notation given above in **(3)**, it demands that if multiple models are given as an input (e.g., an SLG hierarchy), the cache for generated identifiers should be cleared after processing each model. Consequently, the generation process starts with an empty cache for each model, so that name clashes are not possible, as for each model a separate Java class file is produced. If the cache is not cleared each time, this would result in the generation of unnecessary identifiers, as the services would make the identifiers unique among all generated files, although uniqueness in a single file is sufficient. For instance, if a serial number is suffixed for unification as described in Sect. 4.1.3, this number would grow unnecessarily high, which leads to code that is less readable.

Formula graph 3 employs two specification patterns for establishing the actual constraint. First, "Existence" with scope "before" is used for specifying that the cache has to be cleared (atomic proposition ClearIdentifierCache) before starting the next iteration (atomic proposition IterateModels). The pattern means "P has to become true before R" and corresponds to $A[!\mathsf{R}\ WU\ (\mathsf{P} \wedge !\mathsf{R})]$. Second, in order to demand the validity of the constraint for the entire code generator, the formula graph contains the pattern "Universality" with scope "globally". The meaning of this pattern is "P has to be true globally", thus it corresponds to CTL's AG operator.

6.2.4 Order Constraints

Fig. 6.4 shows two examples from the `actionOrder` category. The depicted formula graphs represent the following constraints:

(1) *After it is initialized, the generator eventually either finishes successfully or fails*:

$$(\mathsf{GenerationSuccessful} \vee \mathsf{GenerationFailed})\ respondsto_{globally}\ \mathsf{Initialization}$$

(2.1 & 2.2) *When using the SIBs* `GenerateTypeName` *or* `GenerateInitializer` *in the generation phase, the type mapping system has to be initialized*

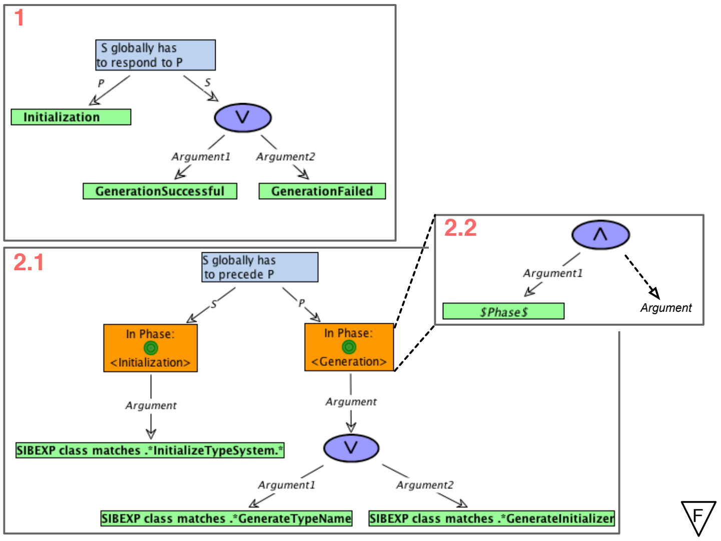

1
S globally has to respond to P
P
S
Initialization
∨
Argument1
Argument2
GenerationSuccessful
GenerationFailed
2.1
S globally has to precede P
S
P
In Phase: <Initialization>
In Phase: <Generation>
Argument
SIBEXP class matches .*InitializeTypeSystem.*
∨
SIBEXP class matches .*GenerateTypeName
SIBEXP class matches .*GenerateInitializer
2.2
∧
$Phase$
Argument
F

Fig. 6.4. Constraints from the `actionOrder` category

by previously employing the SIB `InitializeTypeSystem` *in the initialization phase*:

$$(\mathsf{Initialization}\ \wedge (SIB.class == .{*}InitializeTypeSystem.{*}))$$
$$precedes_{globally}\ (\mathsf{Generation}$$
$$\wedge\ ((SIB.class == .{*}GenerateTypeName.{*})$$
$$\vee\ (SIB.class == .{*}GenerateInitializer.{*})))$$

Constraint **(1)** (specified by formula graph 1) uses the specification pattern "Response" with scope "globally" to specify the allowed outcomes of a generation run, once the code generator has been initialized. This pattern means "S globally has to respond to P" and translates to $AG(\mathsf{P} \ \Rightarrow\ AF(\mathsf{S}))$.

The formula graphs labeled 2.1 and 2.2 in Fig. 6.4 specify a constraint that demands the proper initialization of Genesys' type mapping infrastructure (cf. Sect. 4.1.2) in the code generator's initialization phase. This is obligatory as soon as any services that access this type system (such as `GenerateTypeName` or `GenerateInitializer`) are used in the generation phase. Again, for identifying the SIBs, the constraint uses regular expressions that match against the single SIB's class name. In order to specify the desired order in which the SIBs should be employed, the constraint contains the specification pattern "Precedence" with the scope "globally". This pattern means "S globally has to precede P" and translates to $A[\neg\mathsf{P}\ WU\ \mathsf{S}]$.

The single phases of the code generator are identified by means of atomic propositions, which are true for any service employed in the corresponding phase. For instance, the atomic proposition Generation is true for any service that is contained in the generation phase. As checking whether a property holds in a particular phase is a very frequent part of constraints, formula graph 2.2 defines a simple pattern that can be conveniently used via hierarchy. Essentially, it is an incomplete formula graph defining slots for:

- an atomic proposition that identifies the desired phase, realized as a model parameter named "Phase", and
- a (sub-)formula which describes the property that is expected to hold in the particular phase, realized as a model branch called "Argument" (indicated by the dashed arrow).

Via macros, this simple pattern is embedded into formula graph 2.1 in order to identify the initialization and generation phase in the corresponding parts of the constraint. Besides being another example for the reusability of formula graphs, the comparison with the corresponding textual pendant shown above in **(2.1 & 2.2)** shows again that the use of hierarchy and patterns allows to increase the conciseness of constraint specifications. The following section elaborates on this by showing the derivation of a bigger pattern and its application in a complex composite constraint.

6.2.5 Deriving Patterns & Composing Constraints

Similar to the simple pattern presented above, many requirements for code generators lead to constraints that are very similar in terms of their basic structure. Consequently, new patterns can often be derived from structural similarities of the corresponding formula graphs (*Requirement G2 - Reusability and Adaptability*).

A fairly complex example from Genesys' library is the constraint which demands the complete inspection of all constituents of the SLGs that serve as an input for a code generator. This constraint has to check that for every input SLG, all contained SIBs are guaranteed to be processed, including all SIB parameters, branches etc. This is intended to ensure that no information is lost or forgotten when generating code from a hierarchical SLG. As all these checks are very similar in structure, a new pattern emerged, called *Handle By* [JMS11]. It is partly based upon "Precedence" and thus can be assigned to Dwyer et al.'s class of "Order patterns". Basically, the new pattern augments "Precedence" by the possibility to describe conditional cause-effect relationships. In the description style of Dwyer at. al [DAC99], the pattern's intent is:

Definition 4 (Intent of "Handle By"). *"To describe cause-effect relationships between a pair of events/states on certain conditions. Under certain conditions, an occurrence of the first, the cause, must be handled by an occurrence of the second, the effect."*

In its current version, the pattern only exists with the scope "before", as other scopes were not required yet. In textual form, this version of the pattern is

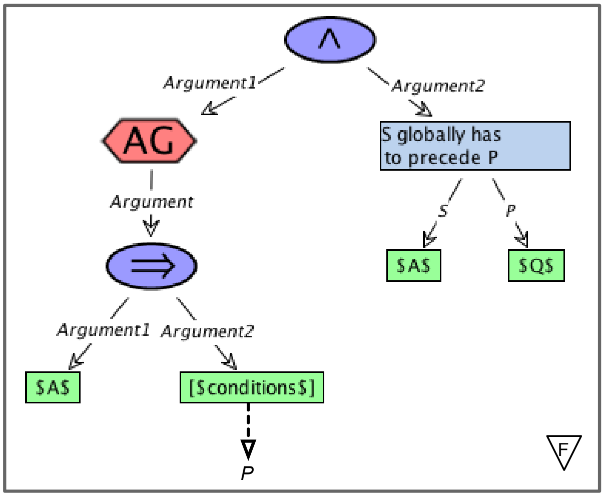


Fig. 6.5. The new "Handle By" pattern with scope "before", meaning "Handle every A by P before Q".

to be read as "Handle every A by P before Q". The formula graph in Fig. 6.5 shows the pattern, which is modeled as an incomplete formula graph with:

- one model branch "P" (indicated by the dashed arrow) as well as
- three model parameters "A", "Q" and "conditions" (surrounded by dollar signs for indicating their status as slots which are filled when the pattern is instantiated).

In Fig. 6.5, A and Q are realized as parameters of the pattern, which means that these slots can only be filled by atomic propositions, but not by entire formulas (as it is the case with P). Of course, the pattern itself is by no means restricted to this: The formula graph could be easily adjusted for supporting formulas in the slots A and Q. For instance, for the Q slot, this can be achieved by removing the node labeled "Q" and by replacing the adjacent "P" edge with a model branch. The slot called conditions provides a list of branch names that represent the conditions under which A will be handled by P.

In order to actually use the pattern, it can simply be embedded into a hierarchical formula graph, as depicted on the left hand side of Fig. 6.6. In this graph, the nodes labeled 1 to 3 use the "Handle By" pattern with the scope "before". For instance, node 1 instantiates the pattern with:

- the atomic proposition NextModel as the *cause*,
- node 2 as the effect,
- "next" as the branch condition, and
- the atomic proposition GenerationTerminated for the "before" scope.

Besides using hierarchy to simplify constraints via the use and creation of patterns, constraints can be combined with and embedded into each other

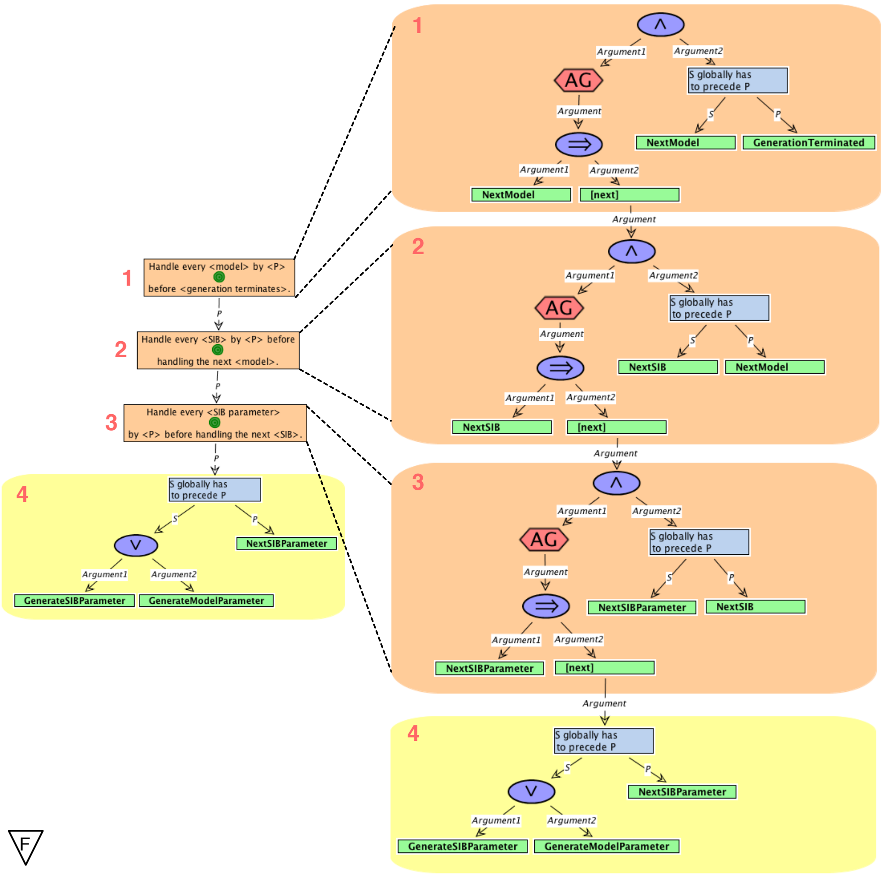

1
Handle every <model> by <P> before <generation terminates>.
2
Handle every <SIB> by <P> before handling the next <model>.
3
Handle every <SIB parameter> by <P> before handling the next <SIB>.
4
S globally has to precede P
NextSIBParameter
GenerateSIBParameter
GenerateModelParameter
AG
Argument
Argument1
Argument2
NextModel
[next]
GenerationTerminated
NextSIB
S
P
F

Fig. 6.6. A constraint checking for the complete handling of input SLGs using the "Handle By" pattern

in order to capture more complex requirements. Fig. 6.6 shows the formula graph that specifies a part of the constraint outlined above, demanding the complete processing of all input SLGs of a code generator. The depicted formula graph focuses on the generation of code for SIB parameters and thus contains the following requirements:

(1) *Handle every input SLG before the generation terminates, by*
(2) *handling every SIB before handling the next SLG, by*
(3) *handling every SIB parameter before handling the next SIB, by*
(4) *generating code for either a "normal" SIB parameter or for a model parameter before handling the next SIB parameter.*

Translating this constraint into a textual formula yields:

$$
\begin{aligned}
&AG(\mathsf{NextModel} \Rightarrow [next](\\
&\quad AG(\mathsf{NextSIB} \Rightarrow [next](\\
&\quad\quad AG(\mathsf{NextSIBParameter} \Rightarrow [next](\\
&\quad\quad\quad A[!\mathsf{NextSIBParameter}\ WU\ (\mathsf{GenerateSIBParameter}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \vee\ \mathsf{GenerateModelParameter})]\\
&\quad\quad)) \wedge A[\neg\mathsf{NextSIB}\ WU\ \mathsf{NextSIBParameter}]\\
&\quad)) \wedge A[\neg\mathsf{NextModel}\ WU\ \mathsf{NextSIB}]\\
&)) \wedge A[\neg\mathsf{GenerationTerminated}\ WU\ \mathsf{NextModel}]
\end{aligned}
$$

This formula is certainly not very intuitive, and it is rather laborious for a user to create and understand it. When modeling the constraint explicitly as a formula graph, as illustrated by Fig. 6.6, it gets more concise due to the consequent use of hierarchy and of specification patterns like "Precedence". Both the combination and embedment of constraints as well as the derivation of new patterns from structural similarities allow for a rapid growth of Genesys' constraint library for code generators. Furthermore, the constraints tend to be more and more concise and simple to use.

6.3 Testing of Code Generators

Another facet of V&V that is supported by Genesys is testing, i.e., "the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior" [Abr+04].

In the context of jABC and its precursors, there has already been a lot of research concerned with this topic. A major result is the *Integrated Test Environment* (ITE) [Nie+01b; Nie+01a; MS04], which provides a holistic approach for testing modeled systems (also called "system-level testing" [MS04]). According to this approach, test cases and entire test suites are modeled as SLGs, based on a library of test blocks (i.e., services). Similar to the graphical modeling of constraints presented in Sect. 6.2, this provides the significant advantage that test cases are specified using the same notation as the actual system under test (SUT). Consequently, no additional language has to be learned for the specification and coordination of tests (*Requirement G3 - Simplicity*). In addition, the user again profits from the general advantages of the SLG notation, such as the high potential of reusing recurrent structures, e.g., by means of hierarchically constructed test cases [MS04], or such as verifying the well-formedness of modeled test cases via model checking [Nie+01b]. Further research on testing in jABC and its precursors concerns regression testing and test suite generation via techniques like automata learning [Hag+02b; HMS03; Raf+09], e.g., for testing legacy or black box systems such as web applications [Raf+08; MNS02] or Computer Telephony Integration systems [Hag+02a].

Motivated by the need for a framework for the automated test of Genesys code generators, and based on the research experience outlined above, an extended testing approach has been developed, which is not only applicable to Genesys, but to jABC in general. Essentially, it extends the holistic ITE approach described above by the dimension of code generation. While being closely related to the notion of model-based testing, which is a little overloaded with different definitions and interpretations [UL06; Dal+99], the following sections consider this concept an instance of MDTD [Sta+07]. Fig. 6.7 shows a variant of the MDTD base model described by Stahl et al. [Sta+07, p. 258], adapted to the way it is realized for jABC using Genesys.

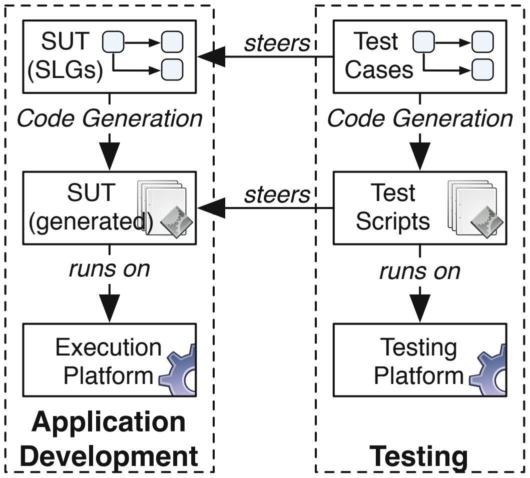


Fig. 6.7. Model-Driven Test Development (MDTD) in jABC

The left hand side of the figure depicts the standard application development in jABC: The application is modeled by means of SLGs, and an appropriate Genesys code generator translates those models into code that runs on a desired execution platform. The testing of the application is shown on the right hand side of the figure. As motivated above, the test cases are also modeled as SLGs. A test suite is a superordinate model which bundles a set of test cases by means of hierarchy. For modeling the test cases, a library of dedicated test SIBs is used. As described by Margaria and Steffen [MS04], this library consists of SIBs for steering the SUT as well as SIBs that provide the functionality of existing test tools and frameworks as services. A test case that has been modeled this way can be directly executed with the Tracer, which is in this case also responsible for steering the SLGs representing the SUT. Prior to the introduction of Genesys and code generation in jABC, the ITE exclusively performed the test execution via direct interpretation of the models.

However, through the introduction of code generation, the testing approach can also be applied on the level of the generated application. For instance, this may be desirable for testing the application's behavior under "real circumstances", i.e., on the actual target platform. Testing at this level is also beneficial if the use of a particular testing platform is mandatory for the execution of the tests or for the final reporting. Accordingly, analogous to the SUT itself, the test case models are also translated to executable code, as shown in Fig. 6.7. This is performed by another code generator created with Genesys, that produces test scripts which are executable on a desired testing platform. Upon execution, those scripts then steer the generated SUT.

This extended testing approach is consistent with the previous research on the ITE, and it particularly meets all requirements on test design, organization and coordination formulated by Niese et al. [Nie+01b]. In order to evaluate its feasibility, the approach has been exemplarily realized for the jABC code generators (cf. Sect. 5). This implementation, which is presented in the following sections, consists of

1. a strategy for testing jABC code generators, in particular for testing the expected execution behavior (Sect. 6.3.1), and
2. code generators for translating modeled test cases and test suites into code for the testing framework JUnit (Sect. 6.3.2).

6.3.1 Testing the jABC Code Generators

According to Stürmer et al., one "can assume that the code generator is working correctly if invalid test models are rejected by the code generator, [...] and valid test models are translated by the code generator and the code generated from this behaves in a 'functionally equivalent' way" [SC04]. The term "functionally equivalent" can be considered synonymous with what has been called "execution equivalence" in Sect. 5.1. In more detail, when testing a code generator, usually the following aspects are of peculiar interest:

1. **Appropriate support of the source language**: Does the code generator accept and process all valid (or desired, if the source language should not be supported entirely) inputs in the source language? Are any invalid or undesired inputs rejected?
2. **Correct translation to the target language**: Does the code generator produce syntactically valid source code in the target language? Is the execution behavior specified in the source language retained in the generated code (execution equivalence)?
3. **Parametrization of the code generator**: Do possible options of the code generator have the expected effects?

Of course, by its very nature, testing only provides incomplete answers to these questions (especially to the one concerning the execution equivalence), relative to the specified test cases. In particular, according to Stürmer et al., the traditional notions of correctness (as, e.g., employed by many of the compiler verification approaches mentioned in Sect. 2.5) cannot be directly applied to code generators, but instead "the definition of correctness has to

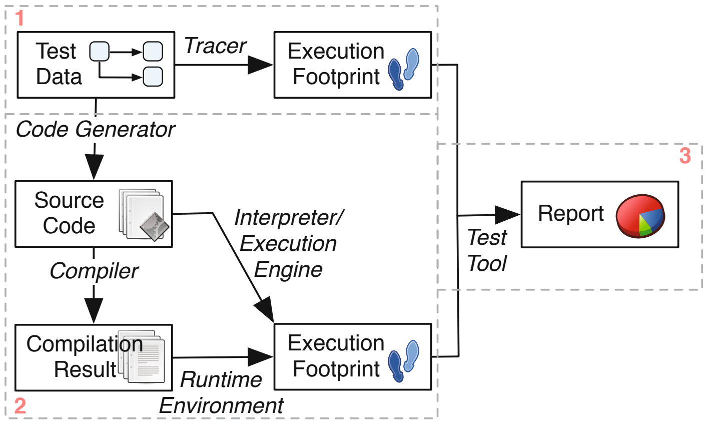


Fig. 6.8. Strategy for testing the jABC code generators

be based on a notion of sufficiently similar behavior" [Stü+07]. Consequently, testing does not aim at proving correctness in the formal mathematical sense, but pragmatically tries to answer the above questions by means of a purposeful selection of realistic and expressive test cases.

Testing Strategy:

In unit-based testing [Bec02], tests check whether a unit (e.g., a method) behaves as desired by comparing its result with some expected value. Considering a code generator such a unit, its result is the generated code. However, as Stahl et al. [Sta+07, pp. 166f] point out, testing a code generator by comparing the generated code with the expected code is rather unfeasible in practice. The main reason for this is that such an approach goes at the expense of maintainability: With any change of the code generator that concerns the syntax of its output, the corresponding tests have to be adapted, even if the semantics of the generated code did not change at all. Consequently, Stahl et al. recommend testing the generated code's *effect* (i.e., its behavior when it is executed) instead of its concrete syntax.

Fig. 6.8 shows the strategy which has been realized for testing jABC code generators (cf. Sect. 5). As the SLG notation is the source language of those code generators, the test inputs for the single test cases are again SLGs (upper left corner of Fig. 6.8). Such *test data SLGs* are constructed on the basis of a small set of dedicated SIBs, which serve a special purpose: Upon execution, each of those SIBs leaves a unique footprint in the execution context (basically a unique string). After executing a test data SLG, the concatenation of the single footprints created by all contained SIBs is the *execution footprint* of the SLG. In other words, such an execution footprint represents a particular trace through a test data SLG.

The execution footprint is used for testing the execution equivalence of the modeled application and its generated counterpart. As depicted in Fig. 6.8, this test is performed in three main steps:

1. **Direct execution:** The test data SLG is directly executed with the Tracer, resulting in a corresponding execution footprint.
2. **Execution of the generated result:** The test data SLG is translated to source code by means of the code generator under test. The resulting source code is either directly executed with an interpreter or an execution engine, or a compiler is used to translate it for a particular runtime environment, in which it can be executed afterwards. Again, both cases yield a corresponding execution footprint.
3. **Test evaluation:** A test tool compares the two execution footprints obtained in steps 1 and 2. The requirement of a "sufficiently similar behavior" mentioned above is met by a jABC code generator, if the execution footprints of the modeled application and its generated pendant are equal, i.e., if in both traces the same SIBs were executed in the exact same order. Examining those traces on the granularity of SIBs is a suitable approach, as SIBs are the atomic building blocks of SLGs. Finally, the test tool reports the test results to the user.

This testing strategy basically performs back-to-back testing [Vou90] and can be considered an instance of the code generator test approach described by Stürmer et al. [Stü+07]. In the latter, the direct execution in step 1 is called model-in-the-loop (MIL), and the execution of the generated result in step 2 is termed software-in-the-loop (SIL).

By using the testing strategy depicted in Fig. 6.8, all of the code generator aspects listed above are tested relative to the available test cases. For instance, if the code generator produces syntactically invalid code from a valid test data SLG (aspect 1), the compilation in step 2, and thus the entire test, will fail. The test for execution equivalence (aspect 2) is realized by comparing the execution footprints in step 3. Finally, the effects of the code generator's options (aspect 3) are tested by multiple executions of the testing procedure with different configurations of the generator in step 2.

Of course, a necessary precondition for this testing strategy is the predictability of the test data SLG's behavior. A repeated execution of one and the same test data SLG (given the same parametrization) should always yield the same execution footprint. Consequently, this modus operandi is not suitable for test inputs with self-adapting or randomized behavior.

Test Inputs:

As the actual test process is thus generic (due to its high configurability) and fixed (as almost all test cases follow the same strategy) at the same time, and as the different test cases almost exclusively emerge from varying test inputs, this testing strategy can also be considered data-driven testing [UL06, pp. 24f]. Similar to the constraint library (see Sect. 6.2.2), the set of test

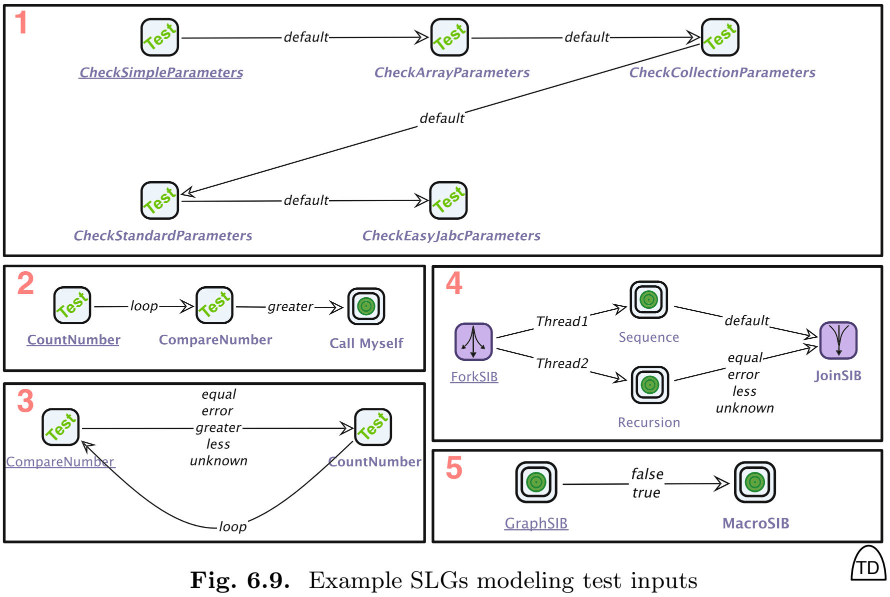


Fig. 6.9. Example SLGs modeling test inputs

data SLGs continuously grows with each new code generator and each newly identified scenario (e.g., a bug) that is not yet covered by a test.

Fig. 6.9 shows some examples of test data SLGs from Genesys' testing framework, which serve as a basis for corresponding test cases. The special SIBs mentioned above, which are designed for creating test data SLGs and which produce the execution footprints in the execution context, are marked with the word "Test" on their icon. SLG 1 is a simple sequence which tests the correct translation of different SIB parameters (except for extended SIB parameters like `ContextKey`, which are tested in a separate test data SLG). For this purpose, the SIBs contained in this SLG are equipped with corresponding SIB parameters, such as `CheckCollectionParameters`, which tests different Java collections like `ArrayList` or `HashMap`. SLGs 2–5 test different control flow mechanisms, such as recursion (2), loops (3), multi-threading (4) and hierarchy (5). As visible from SLG 4, those mechanisms are also tested in combination: Apart from the SIB for forking and joining the control flow, this model also contains macros. Currently, the test suite for the jABC code generators contains 65 of such test data SLGs, which serve as the basis of around 380 test cases, the bulk of which are proceeding according to the testing strategy described above.

6.3.2 Generation of Test Scripts for JUnit

In the context of a diploma thesis [Smo10], the effort of realizing the testing concept shown in Fig. 6.7 for the jABC code generators has been completed by means of code generators and test blocks for the well-known testing framework

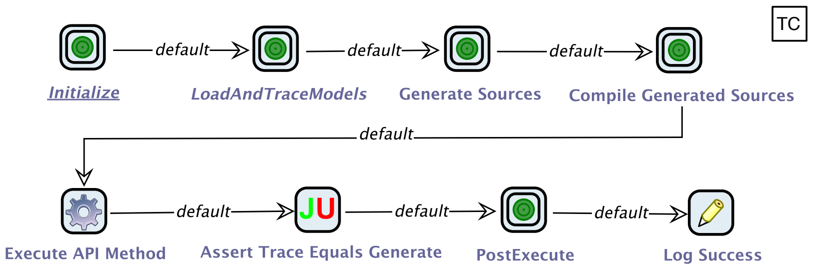


Fig. 6.10. The testing concept from Fig. 6.8, modeled in jABC

jUnit [Bec04]. jUnit is a Java instance of the general xUnit framework [Bec02] for automated testing based on so-called *units* (e.g., classes and methods). It provides an API for implementing test scripts as Java classes that are equipped with special annotations. Furthermore, most development tools such as IDEs like Eclipse or build management tools like Maven usually support testing with JUnit. In most cases, those tools also provide facilities for reporting the test results to the user. Due to this widespread use and support of the framework and based on the fact that all jABC code generators are available in Java (as they are produced by the Genesys Code Generator Generator, cf. Sect. 5.2.6), JUnit is an appropriate choice as the underlying testing platform in this scenario. However, please note that the testing concept is not particularly bound to JUnit – it could be realized for any testing platform.

Modeling Test Cases:

In order to enable the use of JUnit in modeled test cases, corresponding SIBs had to be developed [Smo10, pp. 32f]. Essentially, those SIBs provide JUnit functionality for the comparison of actual and expected values, such as the check for equality of objects (`AssertEquals`, `AssertArrayEquals`) or of object references (`AssertSame`, `AssertNotSame`), for the existence of an object (`AssertNotNull`) or for the truth of some condition (`AssertTrue`, `AssertFalse`).

Thanks to the coarse-granularity and configurability of SIBs, only those seven basic SIBs were required for representing all corresponding JUnit functionality. Apart from those JUnit services, further SIBs were developed for steering the Tracer and the code generators, and for invoking compilers (e.g., for Java). For general tasks such as the creation and deletion of required temporary directories, the existing Common SIBs (cf. Sect. 3.2.1) can be used.

Based on those SIBs, test processes can be modeled as SLGs in jABC. Fig. 6.10 shows the strategy for testing jABC code generators presented in the previous section, modeled as a test process in jABC. Most of the SIBs contained in this model are macros, hence the test process is hierarchical, and its single phases are refined by submodels. The basic steps from Fig. 6.8 are visible in this model:

- The macro `LoadAndTraceModels` obtains the first execution footprint by tracing the model, and thus corresponds to step 1.
- The macros `Generate Sources`, `Compile Generated Sources` as well as the SIB `Execute API Method` correspond to step 2, because they obtain the second execution footprint from the generated pendant.
- Finally, the SIB `Assert Trace Equals Generate` performs the comparison of the two execution footprints, as defined by step 3 in Fig. 6.8.

The macros `Initialize` and `PostExecute` perform necessary tasks before and after the actual test execution, such as the creation or deletion of required directories. The SIB `Log Success` is part of the Common SIBs and emits a simple log message once the test execution succeeded. The remaining blocks belong to the newly developed SIBs mentioned above. `Execute API Method` directly steers the SUT by executing the generated and compiled result of the previous steps via an API method (with default values for all parameters). Finally, the SIB `Assert Trace Equals Generate` uses JUnit for comparing the obtained execution footprints. Please note that the reporting of the test results is not explicitly modeled in the process, as this task is entirely performed by JUnit as the underlying testing platform.

In order to translate such modeled test processes into test scripts for JUnit, a new code generator has been developed with Genesys: the *JUnit Generator* [Smo10, pp. 35–41]. As visible in Fig. 5.16 in Sect. 5.4, the JUnit Generator was derived from the Java Class Generator 1. In order to generate JUnit-compatible classes instead of plain Java classes, only a few simple changes were required. First, instead of a `main` method, the generator produces a test method which is marked as such via a special annotation (`@Test`). Furthermore, the generation of the parameters for the resulting class had to be adapted, as JUnit also realizes the parametrization of tests with corresponding annotations (rather than with, e.g., method arguments) [Smo10, pp. 37–40]. For the latter case, one new model had to be created. Beyond that, no new models or SIBs were required for creating the JUnit Generator.

Modeling Test Suites:

Furthermore, Smolinski [Smo10] proposed the use of a so-called "suite graph" for supporting the organization of test cases as test suites. A suite graph is an SLG which only contains macros, each of them referring to a test case that belongs to the suite. Fig. 6.11 shows an excerpt from the suite graph that comprises all test cases for the Java Class Generator. The bulk of the contained macros references the test process depicted in Fig. 6.10, but each time configured with different test inputs and options.

Please note that the implementation that resulted from the diploma thesis [Smo10] does not contain support for modeling order or causalities in suite graphs, which is why all macros in Fig. 6.11 are unconnected. The implementation currently assumes the single test cases to be independent of each other and thus executes them in arbitrary order. However, future extensions will support connecting selected macros in a suite graph by edges, in order to reflect the execution order of the corresponding test cases (e.g., if the execution of one test case depends on the results of a preceding test case).

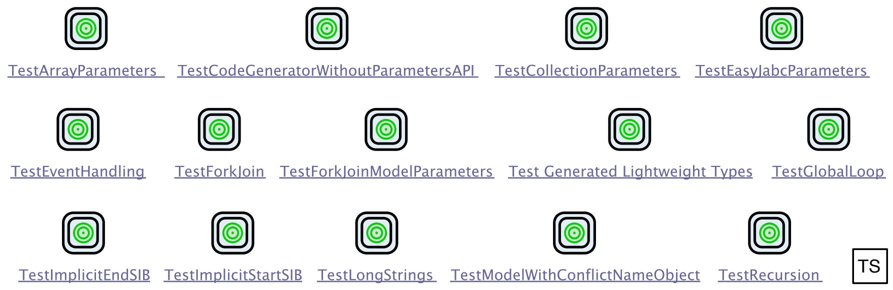


Fig. 6.11. Excerpt from the test suite graph of the Java Class Generator

For being able to execute such a modeled test suite with JUnit, another code generator, the *Test Suite Generator*, translates a suite graph into a test suite for JUnit [Smo10, pp. 41–48]. Essentially, this generator works by iterating over the macros contained in the suite graph. For each macro, the JUnit Generator is applied to the referenced test case model(s) in order to generate a corresponding JUnit test script. To this end, the entire JUnit Generator is embedded in the Test Suite Generator, and thus reused as a ready-made service. As its result, the Test Suite Generator produces a Java class, which is marked as a test suite by means of a corresponding annotation (`@Suite`). This class references all JUnit test scripts generated from the single test cases contained in the suite, so that JUnit is able to execute the tests together. For realizing the Test Suite Generator, it was sufficient to use a new parent model that introduces a new hierarchy level to the existing JUnit Generator.

A test suite or script that has been produced by the code generators described above can be run in any environment that supports JUnit, such as the corresponding plugins in Eclipse or Maven. Those environments are also responsible for the way in which the results are reported to the user (the remainder of step 3 in Fig. 6.8).

Of course, just as specified by the testing concept presented above in Fig. 6.8, and as already supported by the classical ITE [MS04], test case models can also be directly executed by means of the Tracer. In this case, a dedicated jABC plugin (also developed in the context of the diploma thesis [Smo10]) obtains the results from JUnit and displays them to the user.

7

Case Study: Domain-Specific Code Generators for EMF

The previous chapters mainly focused on the (self-)application of Genesys in the context of jABC. Although this aspect has been investigated most intensively, Genesys is not limited to the construction of code generators for jABC. Accordingly, the case study presented in this chapter is supposed to illustrate, among others, the feasibility of Genesys for other source languages and platforms (*Requirement G1 - Platform Independence*).

For this purpose, the case study is concerned with the Eclipse Modeling Framework (EMF) [Ste+09], which is part of the Eclipse Modeling Project (EMP) [Gro09], and which allows modeling based on its metamodel Ecore. With respect to their basic structure, models specified by Ecore are very similar to UML class diagrams.

EMP contains a large number of projects, frameworks and tools that deal with generating code from Ecore models and their instances. For example, for an arbitrary Ecore model, EMF itself is able to generate an Eclipse plugin which provides a tree-based graphical editor for corresponding model instances. The Graphical Modeling Framework (GMF) [Ecl11a] even generates editors for graphical notations. The template language Xpand [Ecl11f] enables the development of code generators for model instances. The specification of corresponding Xpand templates is supported by special editors in Eclipse, that provide, e.g., features such as code completion or static checking on the basis of a given Ecore model. Further examples of tools that are based on EMF are Acceleo, AndroMDA, MOFScript and Xtext (cf. Sect. 2.3.3 for the first three and Sect. 2.3.5 for the latter).

Although there are plenty of code generation solutions around EMF, none of them (to the author's knowledge) allows the construction of code generators like Genesys, in a model-driven and service-oriented way. Consequently, the case study in this chapter presents an approach for integrating EMF and Genesys. The central objective of this approach is to utilize the domain knowledge specified in a metamodel as a basis for generating domain-specific SIBs. Such generated SIBs can in turn be used in Genesys in order to create code generators for any models that conform to (i.e., are instances of) the given metamodel. By this means, a generator developer is able to resort to the specific concepts and terminology of the source language when constructing code generators, thus meeting *Requirement S1 - Domain-Specificity*. Fig. 7.1 depicts this approach.

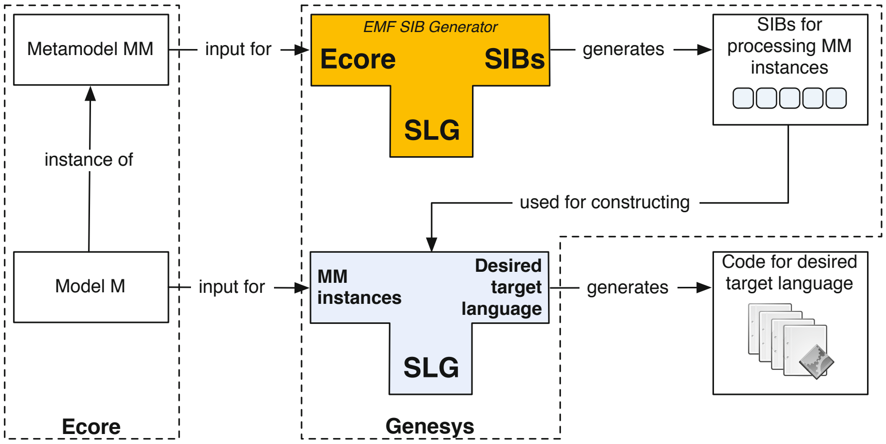


Fig. 7.1. Approach for constructing code generators for EMF with Genesys

Initially, a metamodel[1] is created with Ecore. This metamodel establishes relevant concepts and notions for corresponding models of the desired domain (cf. Sect. 2.2). The central goal is enabling generator developers to apply Genesys for constructing code generators which support any models conforming to the metamodel. Sect. 4.1.1 mentioned that the development of a code generator in Genesys requires a corresponding SIB bundle for processing the desired source language. In the case of jABC, the "Graph Model SIBs" are used for processing input SLGs. Accordingly, in order to support another source language, another specifically dedicated SIB bundle has to be employed in place of the "Graph Model SIBs".

As visible in Fig. 7.1, a special code generator, the *EMF SIB Generator*, is used to automatically generate such a SIB bundle based on the given metamodel. The EMF SIB Generator itself was also built and generated with Genesys. The resulting SIBs are able to process any models that conform to the metamodel, and thus can be considered on a par with a domain-specific "model API" for those models. Subsequently, the generated SIBs serve as a basis for constructing further Genesys code generators, that translate any instances of the metamodel into a desired target language (bottom part of Fig. 7.1). This way, a generator developer profits from the advantages of Genesys without having to relinquish EMF's strengths in domain-specificity.

[1] For the sake of simplicity, this chapter uses the notion "metamodel" synonymously with the term "domain model".

The following sections elaborate on the constituent parts of this approach. First, Sect. 7.1 and Sect. 7.2 describe the structure of EMF and the Ecore metamodel in more detail. Afterwards, Sect. 7.3 enlarges upon how the EMF SIB Generator has been constructed with Genesys, and how it generates domain-specific SIBs for an Ecore metamodel. Finally, Sect. 7.4 demonstrates the working approach by means of an example scenario, and Sect. 7.5 evaluates the results of the case study.

7.1 Eclipse Modeling Framework

Fig. 7.2 shows the basic workflow employed by EMF. An Ecore model can be created in two ways. First, it can be imported from a supported format such as UML, specifically annotated Java interfaces, XML Schema or XMI (box "Import"). Second, it can be created directly by using an Ecore editor (box "Modeling"), such as the simple tree-based Ecore editor included in EMF.

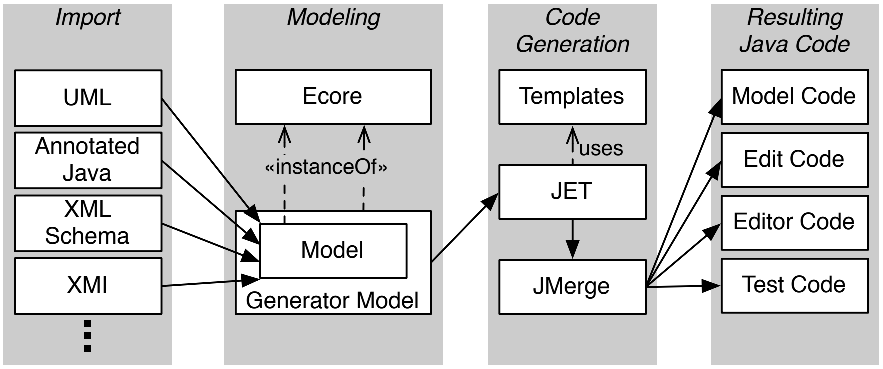


Fig. 7.2. EMF workflow

After the Ecore model is completed, the code generation needs to be prepared by creating a generator model. This generator model wraps around the actual Ecore model and decorates it with configuration information for EMF's code generation facilities. Accordingly, generator models are comparable to the generator configurations employed in the Genesys jABC plugin (cf. Sect. 4.3.2). Please note that the generator model is also an Ecore model, as visible in Fig. 7.2.

The generator model then acts as the input for EMF's integrated code generator, which basically performs two steps [Ste+09, p. 342]. First, code is generated from the Ecore model via the template engine JET (cf. Sect. 2.4.2). Second, if there are any files already existing from previous generation runs, the generated code is merged with those files by means of a tool called JMerge [Ste+09, p. 342], which is based on protected regions (cf. Sect. 2.4.4).

The second step is required due to the fact that, in EMF, generated code is allowed to be edited and extended. Otherwise, if no previously generated files exist (e.g., because it is the first generation run), corresponding new files are created.

EMF is able to generate four different kinds of Java code from a given Ecore model:

- *Model code* resembles a Java implementation of the model using the API defined by Ecore (cf. Sect. 7.2),
- *edit code* provides a UI-independent API for editing instances of the model,
- *editor code* is a complete Eclipse plugin that allows to edit model instances via a tree-based editor, and
- *test code* provides unit tests for the model.

Please note again that as mentioned above, this standard workflow is not the only way to generate code from Ecore models: There are several tools and frameworks that employ modeling with Ecore, but apply their own code generation techniques. The case study presented in this chapter is an example of such an approach, as it employs Genesys for generating code from Ecore models.

7.2 The Ecore Metamodel

Ecore [Ste+09] is the metamodel underlying EMF, i.e., the structure and concepts of any models in EMF are determined by Ecore. While being closely related to OMG's UML and MOF specifications (cf. Sect. 2.3.3), Ecore is in many respects much more simple and pragmatic. At the same time, Ecore has significantly influenced the OMG's work on those specifications. For instance, the MOF specification also defines Essential Meta-Object Facility (EMOF), which is the "lightweight core of the metamodel that quite closely resembles Ecore" [Ste+09, p. 40]. Essentially, this core contains the class diagram portion of UML and thus condenses the large specification to a very pragmatic minimum. In particular, the concepts defined by Ecore are very similar to those that can be found in Java, which shows EMF's primary focus on staying close to implementation, and on targeting developers [Ste+09, p. 11].

Fig. 7.3 provides an overview of the concepts defined by Ecore [Ecl05] as a UML class diagram. Similar to Java's `Object`, there is a root component called `EObject`, which is the basis for any other parts of Ecore. Most of the remaining concepts are also well-known from Java and UML, such as packages (`EPackage`), classes (`EClass`), data types (`EDataType`), enumerated types (`EEnum`) and annotations (`EAnnotation`). Classes may consist of attributes (`EAttribute`) and references to other classes (`EReference`). Together, both form the *structural features* of a class (`EStructuralFeature`). In order to include *behavioral features*, classes may also contain parametrized

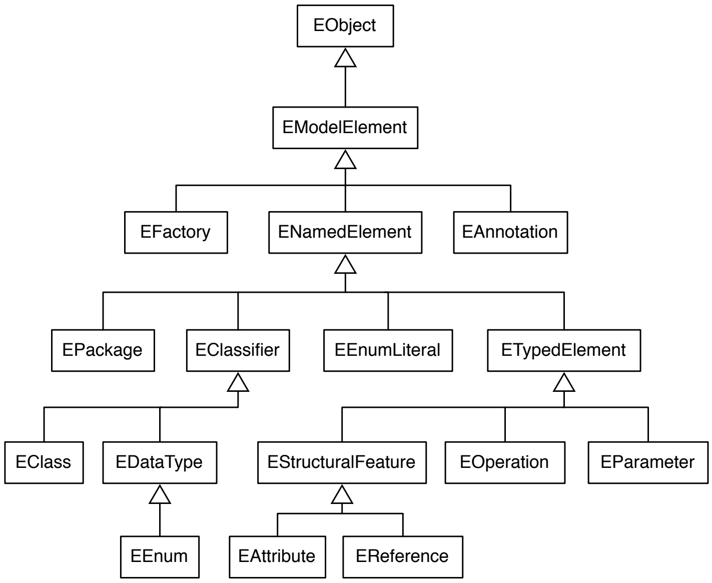


Fig. 7.3. Overview of the Ecore metamodel [Ecl05]

operations (`EOperation`), which are similar to Java methods. Furthermore, Ecore provides a set of built-in data types (not visible in Fig. 7.3), which mainly represent basic Java types such as `String` and `boolean`. For more details on the Ecore metamodel, please refer to Steinberg et al. [Ste+09, pp. 103ff].

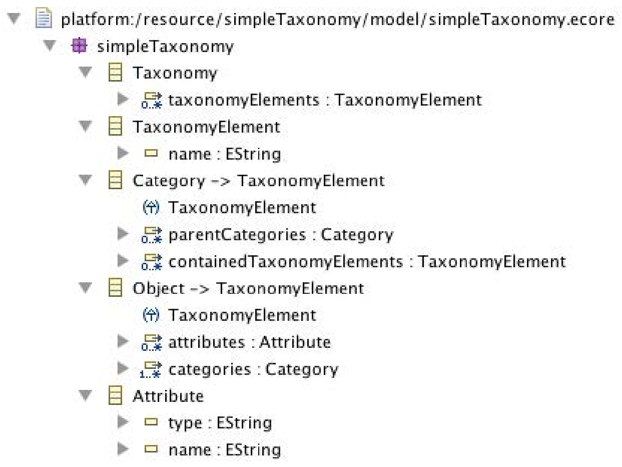
Ecore is a reflexive metamodel (cf. Sect. 2.2), i.e., it is able to describe (to model) itself. Accordingly, the Ecore metamodel is itself an Ecore model, and Ecore is just another EMF model. The reflexivity is an important reason for Ecore's flexibility, as it enables the applicability on multiple metalevels. For instance, Ecore can be used for specifying models (layer M2 in OMG's nomenclature), modeling languages like UML (M3) [Ecl11b] as well as meta-modeling languages for the creation of DSLs (M4) [Béz+05]. Sect. 7.5 further elaborates on Ecore's multi-level applicability, and on how it impacts the approach presented in this chapter.

Fig. 7.4 shows an example of an Ecore model, which is a metamodel describing a modeling language for simple taxonomies. Many of the concepts described above can be found in this model, for instance:

- `simpleTaxonomy` is an instance of `EPackage`,
- `TaxonomyElement` and all other elements on the same hierarchy level of the model are instances of `EClass`,
- `name` under `TaxonomyElement` is an `EAttribute` with the `EDataType` `EString`, and

- `taxonomyElements` under `Taxonomy` is an `EReference`, referring to an arbitrary number of `TaxonomyElement` instances.

Furthermore, it is also visible that Ecore supports inheritance of classes, as `Object` and `Category` share `TaxonomyElement` as a common super class.



```
platform:/resource/simpleTaxonomy/model/simpleTaxonomy.ecore
  simpleTaxonomy
    Taxonomy
      0..* taxonomyElements : TaxonomyElement
    TaxonomyElement
      name : EString
    Category -> TaxonomyElement
      TaxonomyElement
      0..* parentCategories : Category
      0..* containedTaxonomyElements : TaxonomyElement
    Object -> TaxonomyElement
      TaxonomyElement
      0..* attributes : Attribute
      1..* categories : Category
    Attribute
      type : EString
      name : EString
```

Fig. 7.4. Example metamodel in Ecore: Simple taxonomy

According to the metamodel in Fig. 7.4, a taxonomy consists of zero or more taxonomy elements. A taxonomy element always has a name and may be a category or an object. Objects are things that are classified in the taxonomy. Each object may contain an arbitrary number of attributes that further describe it. Additionally, an object belongs to one or more categories. A category contains an arbitrary number of taxonomy elements, i.e., objects and other categories. Furthermore, each category is associated with zero or more parent categories.

The example scenario presented in Sect. 7.4 will use this metamodel, and it will also provide an example of a corresponding model.

7.3 EMF SIB Generator

As depicted in Fig. 7.1, the EMF SIB Generator generates SIBs from an arbitrary Ecore model. These SIBs enable Genesys code generators to process models that conform to the given Ecore model. The EMF SIB Generator itself has also been developed with Genesys.

In order to construct the generator, several SIBs were required for being able to process Ecore models. As a consequence from Ecore's reflexivity that has been mentioned in the previous chapter, EMF provides a reflection API for generically accessing Ecore models [Ste+09, p. 419ff]. Based on this

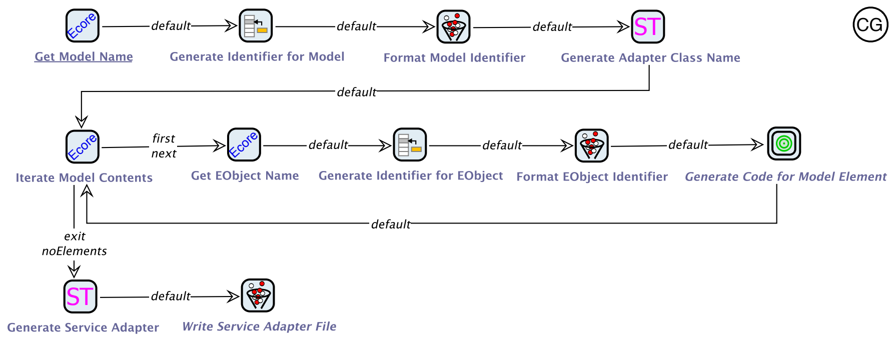


Fig. 7.5. Model `Generate EMF SIBs`

API, corresponding SIBs have been developed manually. Fig. 7.5–7.7 show examples of those SIBs, which are marked with the word "Ecore" on their icon.

For instance, the model in Fig. 7.5 contains an instance of the SIB `IterateEObjectContents` (labeled `Iterate Model Contents`), which enables iterating over the elements contained in an arbitrary Ecore model. In Fig. 7.6, the SIB `Switch EObjectType` is used for determining the type of an `EObject`. The SIBs labeled `Get Model Name` and `Get EObject Name` in Fig. 7.5 as well as `Get Feature Name` in Fig. 7.7 are instances of `GetName`, which is able to read the name of any `ENamedElement` (cf. Fig. 7.3).

In total, 14 SIBs have been developed for the EMF SIB Generator, which currently supports a subset of Ecore sufficient for the case study presented in this chapter. Sect. 7.5 reflects on the steps that are required for extending the EMF SIB Generator in order to support full Ecore.

Resulting from Ecore's simplicity, the construction of the EMF SIB Generator was straightforward. In order to illustrate the code generation process, Fig. 7.5–7.7 show an excerpt of the generator's models. The model depicted in Fig. 7.5 is the main process of the generation phase. In a preceding initialization phase not shown in the figures, the input Ecore model has been loaded from its XMI serialization and stored in the execution context for being accessible to all models of the code generator. Technically, the Ecore model is an object of type `EPackage`, as the root element of each Ecore model is always given by a package [Ste+09, p. 118]. As its template engine, the generator uses StringTemplate (see Sect. 2.4.2) via the SIB `RunStringTemplate` described in Sect. 3.2.1 – corresponding SIB instances are marked with "ST" on the icon.

The generation process in Fig. 7.5 starts with reading the name of the Ecore model (respectively, of the `EPackage` object), and then converts it to a valid identifier (cf. Sect. 4.1.3). This identifier is subsequently used to generate a name for the service adapter (step `Generate Adapter Class Name`), which is produced for the resulting SIBs. Technically, the service adapter is generated as one single class which contains one method for each resulting

SIB, implementing the corresponding SIB's execution behavior. In the following steps, the code generator iterates over the contents of the Ecore model (more precise: over all contained `EObjects`). From each element, the generator first retrieves the name and converts it into a valid identifier. Afterwards, the code corresponding to the model element is generated in a submodel. This includes code for the resulting SIB along with a suitable method for the service adapter. Finally, after all model elements have been processed, the service adapter is assembled and written to a file.

Fig. 7.6 shows the submodel which is referenced by the macro `Generate Code for Model Element`. Its main task is to determine the type of the given `EObject` and to dispatch accordingly. In Ecore, the root element of each model contains an arbitrary number of classifiers (i.e., instances of `EClassifier`, cf. Fig. 7.3) [Ste+09, p. 113]. Classifiers in Ecore are either classes (`EClass`) or data types (`EDataType`). As visible in Fig. 7.6, the EMF SIB Generator currently only supports `EClass` and `EEnum`, whereas the latter is a specialization of `EDataType`. A support of general `EDataType` instances was not required for the case study presented in this chapter. After the type of the given object has been determined, corresponding code is being generated. In the `EClass` case, first all subtypes (i.e., inheriting classes) are processed, and afterwards all structural features (cf. Sect. 7.2) of the class are translated to code.

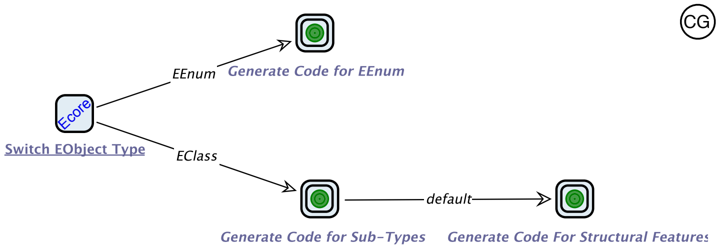


Fig. 7.6. Model `Generate Code for Model Element`

The latter is performed by the model that is depicted in Fig. 7.7. It starts with iterating over the structural features of the given `EClass` instance. Again, each feature's name is determined and converted into an identifier. Subsequently, the code generator distinguishes whether the feature is multi-valued (e.g., a list) or single-valued. If the feature is multi-valued, the generator produces a SIB that allows iterating over all values of the feature. Otherwise it generates a "getter"-SIB which enables access to the feature's single value (see Sect. 7.4 for examples of both cases). Independent of the feature's cardinality, the production of the resulting SIB consists of three steps that generate

1. a method for the service adapter,
2. a name for the SIB which is derived from the name of the feature, and
3. the SIB itself.

Finally, the generated SIB is written to a file. The produced service adapter method is not yet serialized: Instead it is stored in the execution context, as the final assembly and emission of the service adapter is performed at a higher hierarchy level, by the last two steps of the model depicted in Fig. 7.5.

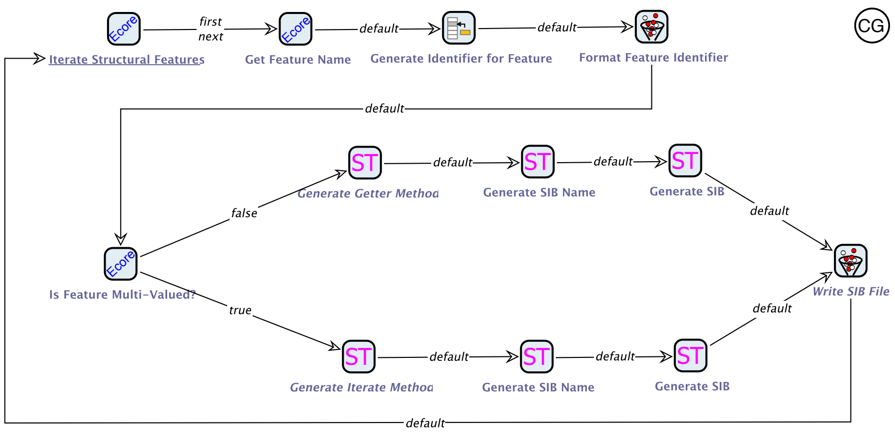


Fig. 7.7. Model `Generate Code for Structural Features`

In order to simplify the usage of the EMF SIB Generator, a corresponding plugin for Eclipse has been developed. This plugin allows to conveniently invoke the generator for any Ecore model via the right-click context menu. The plugin then creates a new Java project in the user's workspace, containing the generated source files (SIBs and service adapters). Furthermore, the plugin produces a POM for Maven (see Sect. 4.3.1), which supports the compilation and distribution of the generated SIBs.

7.4 Example: Taxonomy POJO Generator

This section illustrates the application of the approach presented in Fig. 7.1 by means of an example scenario. The scenario is based on the metamodel for simple taxonomies, which has been introduced in Sect. 7.2. As this metamodel is a complete and valid Ecore model, the EMF SIB Generator can be applied to it, e.g., via the Eclipse plugin mentioned in the previous section.

Fig. 7.8 depicts all SIBs that are produced from this metamodel by the EMF SIB Generator. The rows in the figure indicate from which metamodel element the corresponding SIBs have been generated, except for the row labeled "General", which only contains the SIB `LoadEmfModel`. This SIB is a special case, as it is responsible for loading a given model, which is a task that is always the same, independent of a particular domain or metamodel. Thus this SIB is generally emitted by the EMF SIB Generator as a default. All remaining SIBs have been derived from the metamodel for simple taxonomies, and each of them can be assigned to one of three basic categories:

1. *Getter-SIBs* (prefixed with "Get") read the value of an element (e.g., an `EAttribute`). For instance, `GetTaxonomyElementName` determines the name of a particular element (i.e., of a category or object) in the taxonomy. Another example of a Getter-SIB is `GetAttributeType`, which reads the type of an object in the taxonomy.
2. *Iterators* (prefixed with "Iterate") allow for the iteration over multi-valued data such as lists. For instance, `IterateTaxonomyTaxonomyElements` iterates over all elements contained in the taxonomy.
3. *Type Selectors* (prefixed with "Switch") are a result of inheritance and allow determining the type of an `EObject`. For example, `SwitchTaxonomy-Element` allows checking whether a taxonomy element is a category or an object.

Fig. 7.7 in the previous section shows how the EMF SIB Generator produces SIBs for categories 1 and 2.

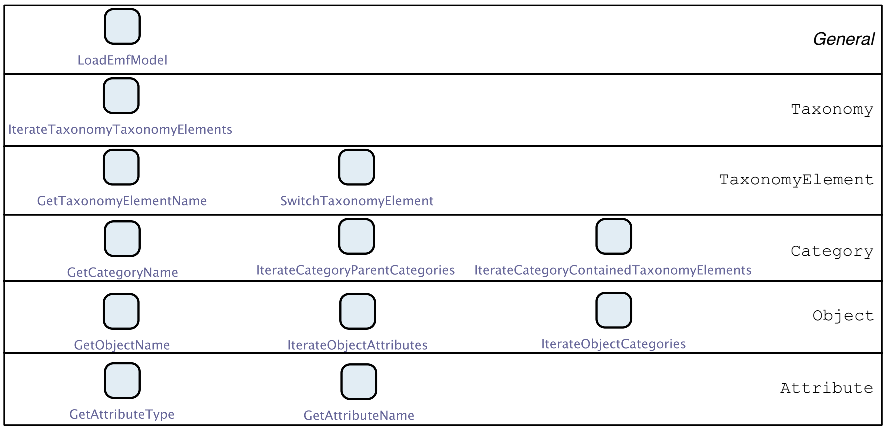


Fig. 7.8. SIBs generated from the "Simple taxonomy" metamodel

The SIBs generated by the EMF SIB Generator can now be used to develop domain-specific code generators with Genesys. As their source language, those code generators demand a taxonomy model that conforms to the simple taxonomy metamodel, which is translated into a desired target language. Fig. 7.9 and 7.10 show excerpts of such a generator, the *Taxonomy POJO Generator*. For a given taxonomy model, the Taxonomy POJO Generator produces a set of corresponding data classes as Plain Old Java Objects (POJOs) [Fow02, p. 118], or more precise as JavaBeans [Ora11d]. JavaBeans can be considered specific POJOs which contain only private attributes and corresponding public getter- and setter-methods.

As visible in Fig. 7.9, the generator first sets the output directory for the generated classes (`Get Base Outlet`), and then loads the taxonomy model prior to iterating over its elements. The submodel in Fig. 7.10 shows the part of the code generator that processes those taxonomy elements that are objects (categories are handled by another submodel). The generator produces one POJO per object in the taxonomy by first generating a private field as well as corresponding public access methods from each attribute of the object, and finally assembling and emitting the actual POJO. In contrast to this, categories are translated to interfaces (which happens in another submodel that is not depicted). If an object is contained in a category, this is reflected in the generated code by the POJO implementing the corresponding category interface. Due to this reason, the model in Fig. 7.10 determines the categories that contain the current object, and produces a comma-separated list of their names in a separate submodel, referenced by the macro `Generate Category Names List`. The resulting list is used for the `implements` statement of the POJO. Finally, the generated POJO is written to a file.

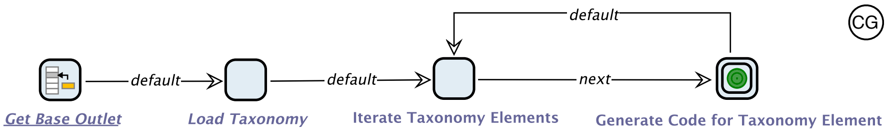


Fig. 7.9. Taxonomy POJO Generator main model

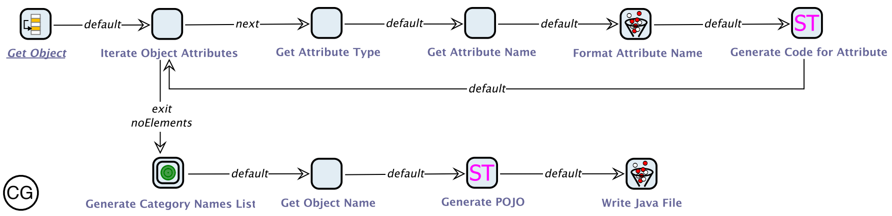
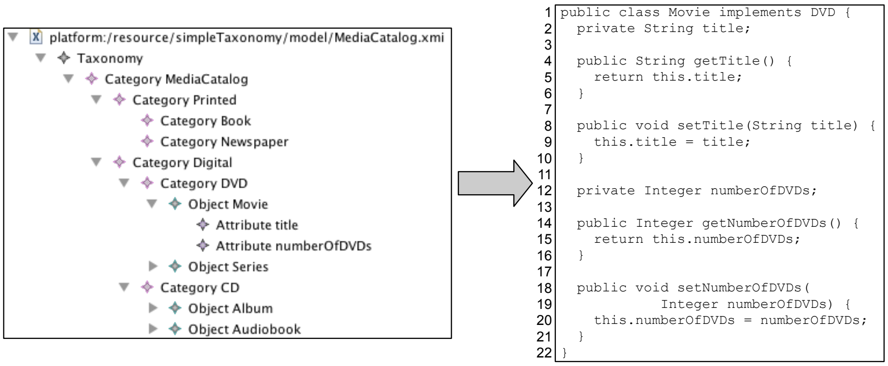


Fig. 7.10. Model `Generate Code for Object`

The left hand side of Fig. 7.11 depicts a concrete example of a taxonomy model that conforms to the simple taxonomy metamodel. It represents a catalog for classifying and archiving media, which are objects (such as "Movie" or "Audiobook") organized by means of several hierarchical categories (e.g., "Digital" or "Printed"). The right hand side of the figure shows a POJO, produced by the Taxonomy POJO Generator for the object "Movie" and its attributes.



```
platform:/resource/simpleTaxonomy/model/MediaCatalog.xmi
  Taxonomy
    Category MediaCatalog
      Category Printed
        Category Book
        Category Newspaper
      Category Digital
        Category DVD
          Object Movie
            Attribute title
            Attribute numberOfDVDs
          Object Series
        Category CD
          Object Album
          Object Audiobook
```

```
public class Movie implements DVD {
  private String title;

  public String getTitle() {
    return this.title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  private Integer numberOfDVDs;

  public Integer getNumberOfDVDs() {
    return this.numberOfDVDs;
  }

  public void setNumberOfDVDs(
            Integer numberOfDVDs) {
    this.numberOfDVDs = numberOfDVDs;
  }
}
```

Fig. 7.11. Example taxonomy model ("Media Catalog") and generated POJO

7.5 Evaluation

The case study presented in the previous sections illustrates several facets of the Genesys approach. First, it shows that the applicability and relevance of Genesys is by no means restricted to jABC (*Requirement G1 - Platform Independence*), as Ecore models and their instances were the source language in the examples. Furthermore, with the EMF SIB Generator, a Genesys-based code generator has been integrated into Eclipse via a plugin, where it of course operates completely independent of jABC.

Second, it is visible from the presented approach (cf. Fig. 7.1), that Genesys can be used for developing code generators at arbitrary metalevels, thus facilitating domain-specificity (*Requirement S1 - Domain-Specificity*).

Application at Multiple Metalevels:

Fig. 7.12 organizes the models (i.e., the taxonomy metamodel and the media catalog model) created in the case study into their respective metalevels. Additionally, for better comparability, the figure shows alternative ways of modeling the media catalog example (i.e., the concrete taxonomy model shown in Fig. 7.11). Please note that, for orientation, the metalevels are labeled M0–M4 as a reference to OMG's MDA metaarchitecture. All solid vertical arrows in the figure indicate an "instance of" relationship between two models.

The columns labeled *a)–e)* exemplify the following scenarios for modeling on different metalevels:

a) Ecore along with its general concepts is directly used as the modeling language for constructing the media catalog model ("MediaCatalog.ecore"). In other words, Ecore is the metamodel, and the media catalog model is an instance of it. M0-level instances of the media catalog model could in turn be represented by models containing concrete data (e.g., concrete

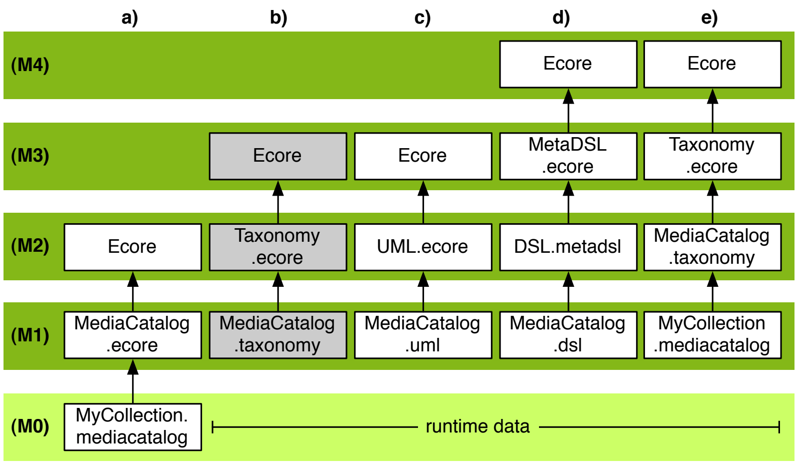


Fig. 7.12. Modeling on different metalevels[2]

DVD movies or music CDs). From the tool perspective, this example resembles the direct use of EMF's general Ecore editor (which is one way of creating Ecore models) for modeling the media catalog.

b) This example represents the modeling approach employed by the case study presented in this chapter (thus it is highlighted in the figure). First, Ecore has been used as a metametamodel in order to create a metamodel for describing taxonomies ("Taxonomy.ecore" in Fig. 7.4). Afterwards, by means of this new language, the media catalog model has been specified ("MediaCatalog.taxonomy", depicted on the left hand side of Fig. 7.11). Instead of using the general Ecore editor like in *a)*, this model was created with a domain-specific editor that was generated by EMF from the taxonomy metamodel.

c) A version of UML that is itself described with Ecore (e.g., [Ecl11b]), is used to create the media catalog example as a UML model. Please note that in terms of metalevels, this approach is similar to *b)*, as Ecore is the metametamodel in both cases, employed for describing the actual modeling language on level M2. The main difference between those two examples is the fact that *c)* uses a general-purpose modeling language (UML) for specifying the media catalog, whereas *b)* uses a domain-specific modeling language. Just like for *b)*, the editor for creating the media catalog UML model might be generated from the metamodel. For instance, this is the case for the UML2 Tools from Eclipse's Model Development Tools (MDT) project [Ecl11c].

d) This can be considered an instance of the "meta-DSL" example described in Sect. 2.2: A specific DSL is used to create the media catalog model

[2] Based on an image posted by Marco Mosconi in the EclipseZone Forums in 2008: `http://www.eclipsezone.com/eclipse/forums/t115395.html#92254259`.

("MediaCatalog.dsl"). This DSL is itself described in a language designed for describing DSLs ("DSL.metadsl"), which is in turn specified using Ecore ("MetaDSL.ecore"). In this setting, Ecore plays the role of a metametametamodel. Bézivin et al. [Béz+05] provide an example of this approach.

e) The models in this column illustrate the relativity of the metalevels, by viewing the case study presented in this chapter from the perspective of example *d)*. Analogous to *b)*, Ecore is used for describing the domain-specific taxonomy language, which is in turn used for specifying the media catalog model. However, in this example, the primary focus is not on taxonomy instances, but on instances of the media catalog taxonomy, i.e., on concrete media catalogs (such as "MyCollection.mediacatalog"). In *b)*, "MediaCatalog.taxonomy" is the end of the modeling chain, thus it is the final artifact that is further processed by the Taxonomy POJO Generator. In contrast to this, in *e)* it is treated as a specialization of the domain-specific taxonomy language, which is in turn used to specify concrete media catalog instances. From this perspective, "MediaCatalog.taxonomy" becomes a metamodel instead of a model, "Taxonomy.ecore" becomes a metametamodel instead of a metamodel, and Ecore is used as a metametametamodel.

For all examples *b)–e)*, level M0 contains concrete runtime data, instantiating the respective models on level M1.

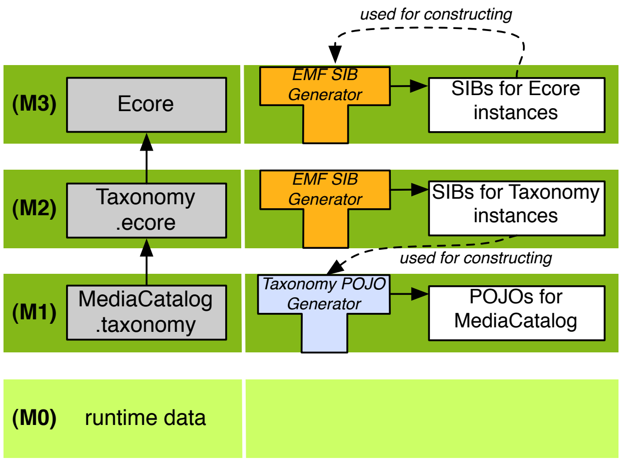


Fig. 7.13. Code generators of the case study, by metalevels

Fig. 7.13 shows the code generators created in the case study on the metalevels M1 and M2. In this context, the EMF SIB Generator has been used for generating SIBs from the taxonomy metamodel ("Taxonomy.ecore") on metalevel M2. The resulting SIBs are domain-specific, as they specifically work with model instances of the taxonomy metamodel. Subsequently, the generated SIBs were in turn used to develop the Taxonomy POJO Generator, which correspondingly is located one metalevel below the EMF SIB Generator. Finally, on this metalevel, the Taxonomy POJO Generator produces POJOs from the media catalog model ("MediaCatalog.taxonomy").

As the EMF SIB Generator can be applied to arbitrary Ecore models, this approach is not restricted to particular metalevels and thus is extremely flexible. Independent of the metalevel on which the EMF SIB Generator is used, it always produces SIBs that are suitable for building a code generator that works one metalevel below. In particular, this flexibility can be exploited for letting the EMF SIB Generator extend and complete itself.

Bootstrapping the EMF SIB Generator:

As mentioned above, the EMF SIB Generator currently only covers a subset of Ecore that was sufficient for the case study. However, an extension of the generator for complete support of Ecore can be achieved by incremental addition of new elements in several stages, i.e., by bootstrapping (cf. Sect. 2.1). This procedure is indicated in the top right corner of Fig. 7.13.

The first stage has already been finished in the case study: A first version of the EMF SIB Generator, that supports a subset of Ecore, has been created manually. In the second stage, this generator is applied to the Ecore model of Ecore itself (a file called "Ecore.ecore"). This yields around 193 SIBs that are directly derived from "Ecore.ecore", including the 14 SIBs that have been implemented manually in the first stage. The generator is now extended on the basis of the SIBs generated in this stage, so that it is able to work with additional Ecore concepts that have not been covered before, such as annotations, data types or operations. The third stage is started by applying the extended generator to "Ecore.ecore" again. This procedure is repeated until the code generation result only consists of SIBs that support Ecore concepts already covered by the code generator.

In experiments conducted after the case study, the bootstrapping has been performed up to the second stage (inclusively). Given the high number of SIBs obtained in this step, the author estimates that a third stage will suffice for extending the EMF Generator to support full Ecore.

Altogether, these results emphasize that Genesys is suitable for realizing domain-specificity independent of any metalevels.

8

Case Study: Service-Oriented Combination of Code Generation Frameworks

Service orientation is an important cornerstone of the Genesys approach. The preceding chapters already showed different facets of service orientation, that provide significant advantages for the construction of code generators, such as the high reusability and availability resulting from growing service repositories (cf. Sect. 4.1), or the flexibility and platform independence of services (cf. Sect. 5.2.1). This chapter presents a case study (published in [JS11]) that enlarges upon this topic by examining the impact of service orientation on integratability and interoperability. As described in Sect. 3.2.1, jABC's service mechanism is incarnated by SIBs, which pose no restrictions whatsoever on the granularity of service functionality. For instance, for the domain of code generation, a service may represent a small task, such as merging two strings, but it may as well incorporate an entire code generation framework. Consequently, Genesys may be seen as a code generator construction kit which allows the (re)use and combination of existing heterogeneous code generation tools, frameworks and approaches.

The case study presented in this chapter demonstrates this advantage by combining the UML code generation framework *AndroMDA* [And11] with the code generators existing in jABC (cf. Sect. 5). Both approaches highly benefit from each other when used in conjunction. AndroMDA, as a representative for most code generation solutions in the UML realm, is very powerful in generating the static aspects of an application, such as business objects, interfaces and mainly infrastructural boilerplate code. In contrast to this, the SLGs in jABC are usually (though not exclusively, cf. Sect. 3.2.3) processes which represent the dynamic behavior of an application. The combination of both approaches paves the way for full code generation (*Requirement S2 - Full Code Generation*), i.e., the automatic generation of complete application code, encompassing the static as well as the dynamic aspects. As outlined in Sect. 1.1, full code generation is highly desirable for any model-driven approach, as it removes the need for manual modification of generated code. In

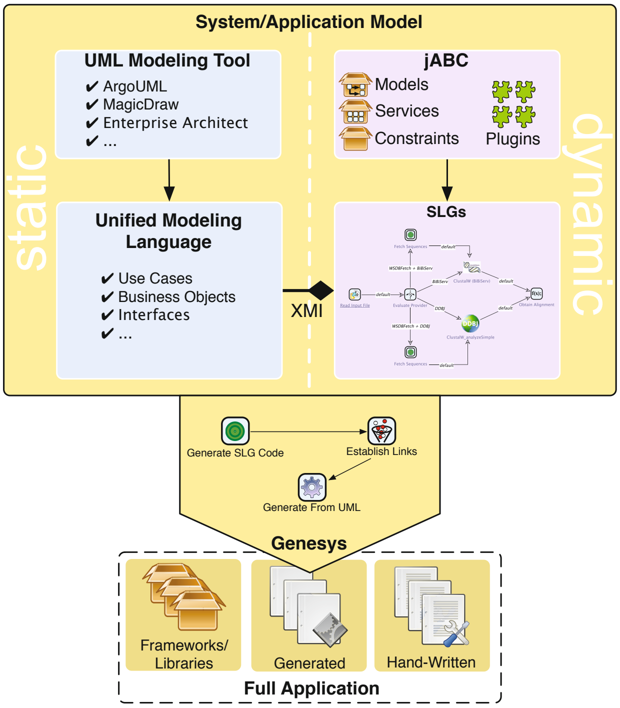


Fig. 8.1. Integration concept for combining jABC's code generators and AndroMDA

consequence, expensive tool support for synchronizing models and generated code (round-trip engineering, see Sect. 2.4.4) is no longer necessary.

Fig. 8.1 shows the concept underlying this combined approach. The realization of this concept consists of two main parts. First, it involves an integrated modeling approach (indicated by the big box at the top of Fig. 8.1), that allows to consistently describe an application by means of SLGs and UML models. Second, correspondingly integrated code generation (the middle and bottom part of Fig. 8.1) is required, which is the aspect of the case study that is most interesting for this book. Essentially, this part is enabled by integrating AndroMDA as a service into Genesys.

Before Sect. 8.3 and 8.4 elaborate on the two parts of the combined approach, Sect. 8.1 introduces AndroMDA in further detail. In order to illustrate the applicability of the proposed solution, the case study uses a practical example from the field of bioinformatics, which will be introduced in Sect. 8.2. Finally, Sect. 8.5 evaluates the results of the case study.

For the most part, the descriptions in this chapter are based on [JS11].

8.1 AndroMDA

AndroMDA is an open source code generation framework following the tenets of MDA (cf. Sect. 2.3.3). It is able to generate multi-tier code for any desired target platform from specifically annotated UML models.

Fig. 8.2 shows the code generation approach employed by AndroMDA. The UML models for which code is supposed to be generated can be created with any UML modeling tool. However, AndroMDA poses two requirements on modeling. First, the modeler has to use the AndroMDA UML profile. *Profiles* [Obj10b, pp. 669ff] are UML's mechanism for tailoring the language to particular domains. Typically, a profile introduces domain-specific terminology and modeling constructs by means of additional stereotypes, tagged values and constraints (see [Fra02, pp. 145–162] for a thorough introduction to UML profiles). By means of the elements specified in the AndroMDA UML profile, the models are annotated with additional information that is used for configuring and steering the code generation. As the second requirement, the employed UML modeling tool has to provide an export to XMI (cf. Sect. 2.2).

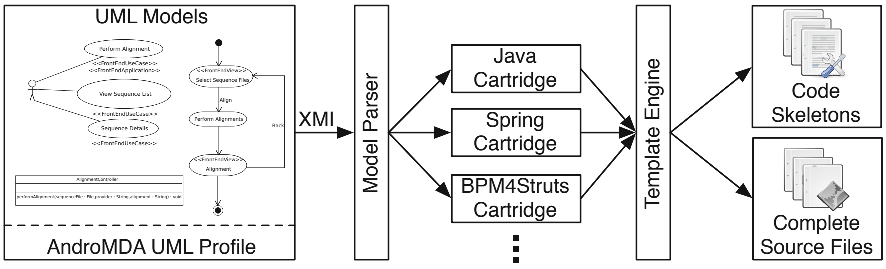


Fig. 8.2. Code generation approach in AndroMDA

As visible in Fig. 8.2, a model parser reads the input models from their XMI form and makes them available as an in-memory representation, e.g., as an Ecore model in case the UML model has been created with the EMF-based UML2 [Ecl11b]. The actual code generation in AndroMDA is template-based (cf. Sect. 2.4.2), and the single templates are specified by pluggable components called *cartridges*. An arbitrary number of such cartridges may participate in the code generation process, each of them responsible for contributing a specific part to the generation result. AndroMDA provides several ready-made cartridges for different target platforms and technologies, such as Java, Struts [Apa11d], Spring [Spr11b] or Web Services. The cartridges are configured by means of the additional information annotated to the input UML models, such as specific tagged values contained in the AndroMDA UML profile. For instance, when generating a web form, the *BPM4Struts* cartridge uses tagged values in order to determine how the single fields of the form should be rendered. If any new templates are required, they can be added by creating a new custom cartridge. Besides the cartridges, the model parser and the employed template engine are also interchangeable, which increases the flexibility of the code generation process.

During code generation, cartridges are responsible for particular model elements (e.g., marked with appropriate stereotypes recognized by a cartridge). When AndroMDA traverses the input model's in-memory representation produced by the parser, the responsible cartridges are invoked for each model element. Multiple cartridges may be responsible for one model element, and each cartridge may contain multiple templates that produce several source files.

Fig. 8.2 shows that the resulting source code is usually a mixture of complete source files and code skeletons, which have to be manually completed by a developer, thus entailing the need for round-trip engineering (cf. Sect. 2.4.4). Although UML contains several diagram types that focus on the specification of dynamic behavior, such as state diagrams, sequence diagrams or collaboration diagrams, AndroMDA's cartridges usually accept only a small subset of UML diagrams as an input. Among those are in most cases only class diagrams, use case diagrams and activity diagrams. This shows the rather weak support for the dynamic application aspects: While the framework is quite powerful in generating static parts of an application that are, e.g., concerned with infrastructural details (like XML descriptors, generic web forms, data transfer objects etc.), the code generation for the dynamic aspects containing the actual business logic is rather unsatisfactory.

8.2 Example Application: Multiple Sequence Alignment (MSA)

In bioinformatics, sequence alignments are used to find correspondences between the bases or codons of DNA, RNA or amino acid sequences [Pol07]. Such correspondences indicate similarities between species, e.g., resulting from a common ancestor. In order to assist researchers in performing such analyses, a large number of services, tools and algorithms are available. For instance, *ClustalW* [THG94] is a popular algorithm for computing sequence alignments, with several ready-to-use implementations. The Web Services provided by the Bielefeld University Bioinformatics Server (BiBiServ) [HKG07] and the DNA Data Bank of Japan (DDBJ) [JD10] are examples of such implementations.

The example process used in this case study is taken from experiments that employ these ClustalW implementations for performing sequence alignments [LMS08]. This process (in the following referred to as the MSA process) has been constructed with *Bio-jETI* [MKS08], which is a special incarnation of jABC that targets bioinformaticians, and assists them in integrating, orchestrating and providing bioinformatics services.

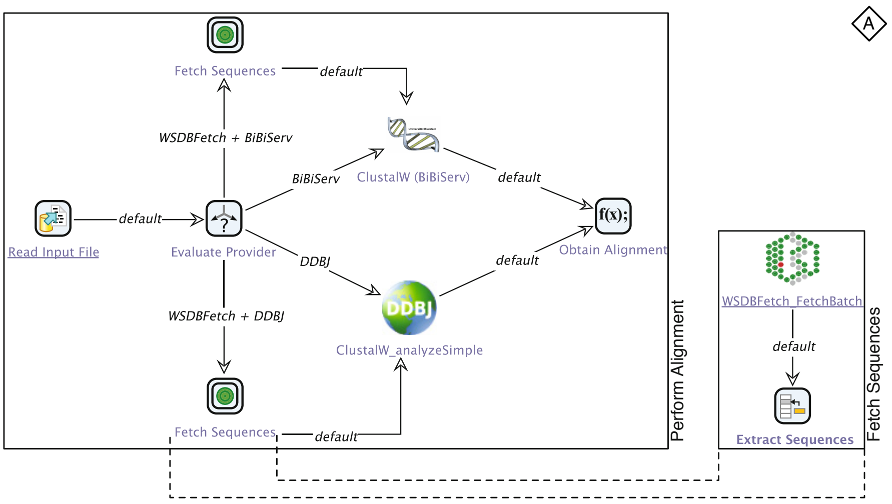


Fig. 8.3. jABC models for the Multiple Sequence Alignment (MSA) process

Fig. 8.3 shows the corresponding SLGs, which allow the user to customize the process by selecting the provider of the ClustalW implementation for the sequence alignment. The main process called "Perform Alignment" (left hand side of Fig. 8.3) starts off by reading the input data, usually the sequences that will be analyzed, from a local file (step `Read Input File`). Afterwards, the provider is determined: Depending on the user's choice, it is either BiBiServ or DDBJ that is used. The corresponding process steps `ClustalW (BibiServ)` and `ClustalW_analyzeSimple` both invoke the corresponding public Web Services. Finally, the alignment is parsed from the result returned by the executed Web Service. In addition to selecting the algorithm, the process also provides the option to read the sequences from the European Molecular Biology Laboratory (EMBL) sequence database, using the `DBFetch` [Lab+07] Web Service available from the European Bioinformatics Institute (EBI). In this case, the input file does not contain the sequences, but only IDs that are used to read them from the database. The corresponding process step is realized by a submodel called "Fetch Sequences".

The use of Genesys for the translation of those processes into executable Java classes has been examined by Lamprecht et al. [LMS09]. The combination of AndroMDA and Genesys presented in this chapter drastically improves the generation result, as it enables the generation of a fully functional web application for invoking the analysis and for viewing the results of a sequence alignment.

8.3 Integrated Modeling

In order to create the models for the static and dynamic application aspects, respective tools may be used as usual. The dynamic behavior is modeled in jABC by means of SLGs, and the UML models for the static aspects can be built with an arbitrary UML modeling tool, such as Together [Bor11] or ArgoUML [Tig11]. The UML models are then added to jABC's modeling repertoire (indicated by the composition in the middle of Fig. 8.1). For being able to perform this addition, the diagrams are expected to be available in the XMI format. Using corresponding access libraries, e.g., following the Java Metadata Interface (JMI) [Jav02], the diagrams are then available to be read and manipulated in-memory.

In the MSA example, the dynamic aspects are modeled by the MSA process presented above. The UML diagrams that model the static application aspects are depicted in Fig. 8.4. These diagrams describe everything that is required for AndroMDA in order to generate the web application for the MSA process. Diagram 1 is a use case diagram that describes, among other functionality, the use case "Perform Alignment". This use case is marked with two AndroMDA-specific stereotypes: `FrontEndUseCase` indicates that the use case should have a corresponding front-end in the resulting application, and `FrontEndApplication` marks the corresponding front-end as the application's entry point.

Via the UML modeling tool, "Perform Alignment" is associated with diagram 2, which is an activity diagram describing the flow of actions in the use case. The activity diagram starts with the activity "Select Sequence File", which allows the user to select the file containing the input data for the MSA process. Again, a specific stereotype (`FrontEndView`) tells AndroMDA that this activity should be generated as a front-end in the application. The "Align" transition leaving the activity is triggered by a so-called signal event, that carries two parameters: the input file for the MSA process and a string specifying the provider option (e.g., "DDBJ"). With this information, AndroMDA translates "Select Sequence File" to a form that allows the user to upload the input file and to set the provider option. In the subsequent steps of the diagram, the alignment is computed ("Perform Alignments") and the results are displayed for the user ("Alignment", again a `FrontEndView`).

Diagram 3 shows an excerpt of a class diagram that defines an `Alignment-Controller` responsible for mediating between the front-end functionality and the business logic on the server. It contains a method `performAlignment`, which is associated with the "Perform Alignments" activity from diagram 2, and thus is intended to realize the invocation of a sequence alignment process with the data entered by the user.

The example models might raise the question why SLGs as well as UML activity diagrams are used for describing flows of actions, and whether it would be possible to resort to only one of those notations for the entire example. In principle, the MSA process in Fig. 8.3 could be drawn as an

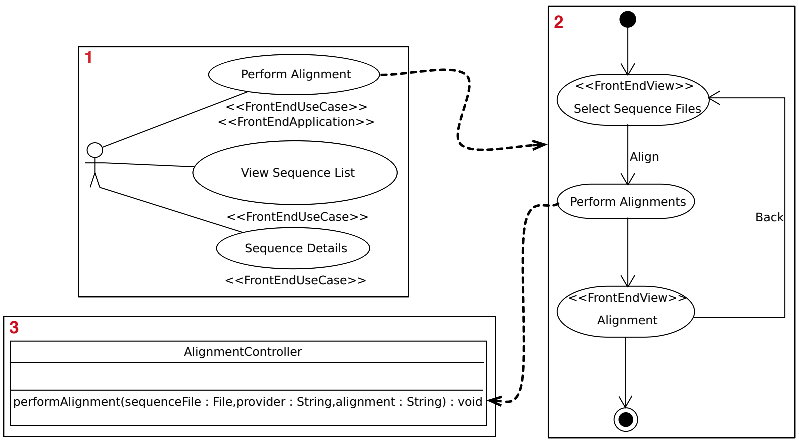
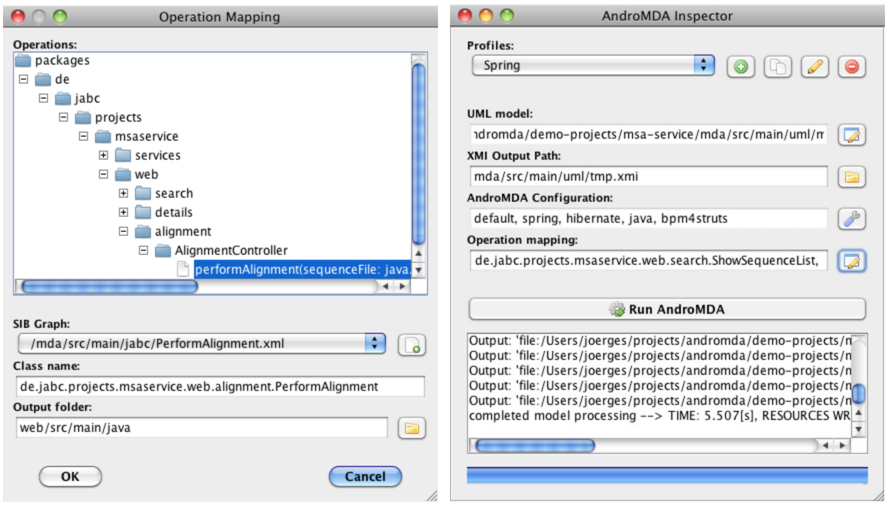


Fig. 8.4. UML diagrams for the MSA web application

activity diagram. However, activity diagrams miss a concept like jABC's SIBs, i.e., the single steps in an activity diagram do not provide an implementation or underlying service (cf. Sect. 3.5). Consequently, they are not executable and are not appropriate for full code generation. On the other hand, an SLG could easily be used for replacing diagram 2 in Fig. 8.4. However, this would imply a more complex customization of AndroMDA, for being able to extract the necessary information from SLGs instead of activity diagrams. Due to these considerations, the MSA example keeps both notations, so that the SLGs are used to describe the application's core functionality (i.e., its actual business logic) and the activity diagrams specify the technical flow of actions in the front-end.

Now that the dynamic as well as the static aspects of the application are modeled, the UML diagrams need to be associated with the SLGs. Technically, it is necessary to know which functions or methods, declared by, e.g., UML class diagrams, are realized by which SLG. This mapping can either be established automatically based on some naming convention, or it has to be specified by the modeler. For the latter case, a corresponding jABC plugin provides a GUI (developed in a diploma thesis [Len09]) that supports the modeler in establishing the links.

The left hand side of Fig. 8.5 depicts the GUI with a mapping for the MSA example. On the top of the dialog, the structure of the UML class diagram from Fig. 8.3 is displayed as a tree. The inner nodes of this tree reflect the package organization of the contained classes, and the leaves are methods defined in those classes. After selecting a method in the tree, an SLG may be associated with it, meaning that the SLG describes the flow of actions in this method. In Fig. 8.5, it is visible that the `performAlignment` method of the `AlignmentController` (cf. Fig. 8.5, diagram 3) is associated with the MSA process shown in Fig. 8.3. Subsequently, such mappings are used for



Operations:
```
packages
  de
    jabc
      projects
        msaservice
          services
          web
            search
            details
            alignment
              AlignmentController
                performAlignment(sequenceFile: java.
```

SIB Graph: /mda/src/main/jabc/PerformAlignment.xml

Class name: de.jabc.projects.msaservice.web.alignment.PerformAlignment

Output folder: web/src/main/java

Profiles: Spring

UML model: ndromda/demo-projects/msa-service/mda/src/main/uml/m

XMI Output Path: mda/src/main/uml/tmp.xmi

AndroMDA Configuration: default, spring, hibernate, java, bpm4struts

Operation mapping: de.jabc.projects.msaservice.web.search.ShowSequenceList,

```
Output: 'file:/Users/joerges/projects/andromda/demo-projects/n
Output: 'file:/Users/joerges/projects/andromda/demo-projects/n
Output: 'file:/Users/joerges/projects/andromda/demo-projects/n
Output: 'file:/Users/joerges/projects/andromda/demo-projects/n
Output: 'file:/Users/joerges/projects/andromda/demo-projects/n
completed model processing --> TIME: 5.507[s], RESOURCES WR
```

Fig. 8.5. Associating SLGs with UML diagrams (left hand side) and configuring the AndroMDA SIB (right hand side)

code generation in order to connect the artifacts generated by jABC's code generators with those produced by AndroMDA. Along with the mapping, the dialog also allows configuring the code generation for the selected SLG by specifying the name and the output location of the generated artifact.

8.4 Integrated Code Generation

In order to support code generation for the linked UML models and SLGs, Genesys' SIB library (see Sect. 4.1) has been extended by two services. The first service prepares the UML models by including the links to corresponding SLGs as tagged values, and the second service integrates AndroMDA for translating the UML models to code. Altogether, the actual code generation is divided into three main steps, which are indicated in the center of Fig. 8.1.

Step 1, "Generate SLG Code":

As the first step of the generation process, the corresponding jABC code generator produces code from the SLGs as usual. In the MSA example, the SLG depicted in Fig. 8.3 is translated into a Java class called "PerformAlignment". This step is depicted as a macro in Fig. 8.1, that references the models of the corresponding jABC code generator, which is the Java Class Generator (cf. Sect. 5.2) in case of the MSA application.

Step 2, "Establish Links":

As described above, the SLGs are linked with (parts of) the UML diagrams. When those links are established by the modeler in jABC, they are first attached as additional information to the SLGs. The "Establish Links" step of the code generation process moves this information to the UML diagrams. Technically, for each link that has been established, the affected UML element is marked with a tagged value, which in turn points to the artifact generated from the corresponding SLG in step 1. For instance, in the MSA example, the `performAlignment` method in the class `AlignmentController` (Fig. 8.4, diagram 3) is marked with the tagged value "`@genesys.class.name=PerformAlignment`", with "PerformAlignment" being the name of the Java class generated by the Java Class Generator. Step 2 is realized as a SIB that can be found in the service library of Genesys.

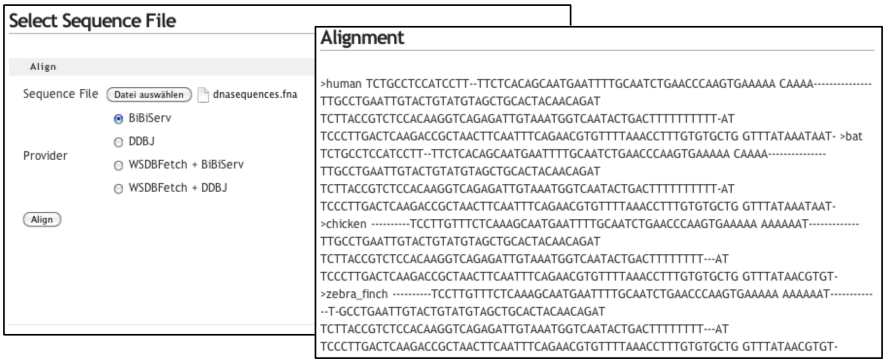
Step 3, "Generate From UML":

In this final step, AndroMDA is invoked for generating code from the UML models. For this purpose, AndroMDA is also made available as a SIB. This SIB represents a customized instance of AndroMDA, which is able to understand the tagged values that have been added to the UML diagrams in step 2, and to generate corresponding extra code. For instance, the tagged value attached to the method `performAlignment` leads to the generation of extra code that basically calls the "PerformAlignment" Java class produced in step 1. That way, the generated method is entirely implemented and executable. If we would run AndroMDA on the bare UML models without the new tagged values, this method would be generated as an empty stub.

In order to integrate AndroMDA as such a customized service, its cartridges had to be extended, so that they are able to interpret the new tagged values and to translate them into calls to classes generated by Genesys. In the context of a diploma thesis [Len09], all cartridges available in the official AndroMDA release have been adapted this way, which was possible with little effort. Please note that this customization had to be done only *once* in order to integrate AndroMDA as a SIB. Afterwards, it is a ready-made building block in Genesys' modeling repertoire, that may be reused for other applications and code generators without the need of further customization.

In consequence, any existing jABC code generator can be easily extended in order to support the combined code generation approach. For this purpose, an additional hierarchy level has to be added on top of the code generator's models. This hierarchy level contains the three steps described above, with step 1 referencing the models of the code generator that should be extended.

Just as any jABC code generator, the resulting combined code generator could be used via Genesys' jABC plugin described in Sect. 4.3.2. However, the AndroMDA service adds many configuration parameters to the code generator. Consequently, the generic GUI provided by the Genesys jABC plugin is rather inconvenient for the configuration of the code generator. Thus a special inspector has been developed in the context of a diploma thesis [Ben08], that specifically supports the usage of those code generators that contain the AndroMDA SIB. This inspector is displayed on the right side of Fig. 8.5. Besides the configuration of the code generator, it also offers the possibility to


Select Sequence File

Align

Sequence File Datei auswählen dnasequences.fna

Provider [✓ BiBiServ] DDBJ WSDBFetch + BiBiServ WSDBFetch + DDBJ

Alignment

```
>human TCTGCCTCCATCCTT--TTCTCACAGCAATGAATTTTGCAATCTGAACCCAAGTGAAAAA CAAAA--------------
TTGCCTGAATTGTACTGTATGTAGCTGCACTACAACAGAT
TCTTACCGTCTCCACAAGGTCAGAGATTGTAAATGGTCAATACTGACTTTTTTTTTT-AT
TCCCTTGACTCAAGACCGCTAACTTCAATTTCAGAACGTGTTTTAAACCTTTGTGTGCTG GTTTATAAATAAT- >bat
TCTGCCTCCATCCTT--TTCTCACAGCAATGAATTTTGCAATCTGAACCCAAGTGAAAAA CAAAA--------------
TTGCCTGAATTGTACTGTATGTAGCTGCACTACAACAGAT
TCTTACCGTCTCCACAAGGTCAGAGATTGTAAATGGTCAATACTGACTTTTTTTTTT-AT
TCCCTTGACTCAAGACCGCTAACTTCAATTTCAGAACGTGTTTTAAACCTTTGTGTGCTG GTTTATAAATAAT-
>chicken ----------TCCTTGTTTCTCAAAGCAATGAATTTTGCAATCTGAACCCAAGTGAAAAA AAAAAAT------------
TTGCCTGAATTGTACTGTATGTAGCTGCACTACAACAGAT
TCTTACCGTCTCCACAAGGTCAGAGATTGTAAATGGTCAATACTGACTTTTTTTT---AT
TCCCTTGACTCAAGACCGCTAACTTCAATTTCAGAACGTGTTTTAAACCTTTGTGTGCTG GTTTATAACGTGT-
>zebra_finch ----------TCCTTGTTTCTCAAAGCAATGAATTTTGCAATCTGAACCCAAGTGAAAAA AAAAAAT-----------
--T-GCCTGAATTGTACTGTATGTAGCTGCACTACAACAGAT
TCTTACCGTCTCCACAAGGTCAGAGATTGTAAATGGTCAATACTGACTTTTTTTT---AT
TCCCTTGACTCAAGACCGCTAACTTCAATTTCAGAACGTGTTTTAAACCTTTGTGTGCTG GTTTATAACGTGT-
```

Fig. 8.6. Screens of the Generated MSA Web Application

start the code generation. Furthermore, it provides a console for monitoring a running code generation process.

When generating code from the MSA example using the concept described above, we obtain a complete web application that is ready to be built and deployed. It runs in an appropriate container such as JBoss [Red11a] and uses the Struts web framework along with Spring. The dynamic parts of the web application are generated by jABC's Java Class Generator, which has been extended as described above, so that the static parts are handled by AndroMDA, mainly involving its *BPM4Struts* and *Spring* cartridges. Fig. 8.6 shows the main screens of the generated web application. The left hand side of the figure depicts the form for uploading the input file (containing either sequences or EMBL IDs) and for selecting the provider, and the screen on the right hand side displays an excerpt of the computed alignment.

The application along with all necessary classes, web forms and configuration files has been entirely generated without writing a single line of code, thus we have achieved full code generation in the sense described in Sect. 2.4.4 (meeting *Requirement S2 - Full Code Generation*).

8.5 Evaluation

The results of this case study demonstrate the enormous value of service orientation for the construction of code generators in Genesys. The flexibility provided by a service mechanism like jABC's SIBs allows a high-level composition of code generators on the basis of arbitrary services, ranging from small libraries like template engines to entire frameworks like AndroMDA. At the same time, the size or complexity of a service is completely invisible to the generator developer, as it is hidden behind the simple and unique interface of SIBs (*Requirement G3 - Simplicity*). The AndroMDA SIB, although representing a very complex service, is used just like any other SIB.

The case study also emphasizes that services strongly facilitate reuse (which is what they are designed for in the first place). First, AndroMDA and its cartridges have been reused by integrating the framework as a SIB into Genesys. Thanks to the SIB concept, this integration was straightforward, and no complex mediators or bridges had to be implemented. Second, once the SIB is created, it is ready to be reused for any code generator built with Genesys. Consequently, Genesys' repertoire for the construction of code generators is growing continuously, and every generator developer profits from work that has been done previously (*Requirement G2 - Reusability and Adaptability*).

Part III

Conclusions and Future Work

9 Conclusions

This monograph presented Genesys, an approach that proposes the construction of code generators by means of graphical models and services. As the current state of the art typically relies on developing code generators on the basis of textual specifications, such as templates, transformation rules or programs written in a general-purpose programming language or DSL, the Genesys approach is unique in the realm of code generation.

In order to show the feasibility of this approach, a full-fledged reference implementation, the Genesys framework, has been created. Conceptually and technically, this framework is based on the XMDD paradigm and its tool incarnation jABC, which enable model-driven and service-oriented development of software systems. The reference implementation has been used for investigating different facets of Genesys in a multitude of case studies (conducted by the author of this book and in the context of related diploma theses), which showed that the approach provides significant advantages for the construction of code generators.

9.1 Requirements of the Genesys Approach Revisited

The advantages of the Genesys approach are best described by revisiting its requirements formulated at the beginning of this book (cf. Sect. 1.1). The previous chapters showed how all those requirements are met by the Genesys framework.

Requirement G1 - Platform Independence
: As code generators in the Genesys framework are abstractly specified as jABC models, they do not depend on a particular implementation language or host machine. The case studies described in Chap. 5 resulted in a large number of code generators, that translate arbitrary SLGs into code for various platforms and languages. In particular, each of those generators could be employed for porting any code generator specified as SLGs to different host platforms.

A necessary condition for porting an SLG to a particular platform is the availability of a suitable technical grounding of the services employed in the SLG. In jABC, this grounding is provided by corresponding service adapters associated with the contained SIBs. Such service adapters can be added, modified and removed without having to change the actual SLG. This is entirely transparent to the generator developer, who only deals with the SIBs, which act as platform-independent representations of the actual underlying services.

Furthermore, the variety of different services employed in the case studies, and especially the integration of AndroMDA described in Chap. 8, shows that there are virtually no limitations on what can be (re)used as a service for building a code generator. Finally, in addition to the large diversity of target languages covered by the various code generators described in this book, the EMF case study explained in Chap. 7 also illustrates that Genesys is not restricted to any particular source language.

Altogether, code generators in Genesys are specified in a truly platform-independent way.

Requirement G2 - Reusability and Adaptability

Following the basic principles of the underlying XMDD paradigm, code generators in the Genesys framework are developed using libraries of models, services, constraints and test cases. Each time any of such artifacts has to be newly created, it is added to the corresponding library, so that it is available to be reused later. In consequence, the generator developer can resort to a continuously growing repository of those artifacts.

Chap. 5 showed this by means of the genealogy of jABC code generators developed with Genesys, and in particular by a detailed comparison of different variants created for the code generator for Java classes. The results indicate that this reuse potential significantly decreases the development effort required for each new code generator. This is because with each new generator, the generator developer has a larger basis of reusable artifacts to rely on. Furthermore, reuse and adaptation are facilitated via the variant management features of the Genesys framework (see also requirement S3 below).

Requirement G3 - Simplicity

In order to minimize the number of languages that have to be learned for building a code generator, Genesys uses SLGs as the basic specification language for all artifacts required in the different phases of development. Accordingly, the actual code generators, the constraints for model checking as well as the test cases along with their test data are all specified as SLGs in jABC.

Furthermore, the generator developer is not forced to use specific code generation techniques, as anything can be incorporated into the Genesys framework as corresponding SIBs. Thus he is, e.g., able to select a template engine he already knows, which again significantly flattens the learning curve.

Requirement G4 - Separation of Concerns

As pointed out in Sect. 3.5, jABC supports the separation of concerns via hierarchical modeling. This is particularly essential for code generators, which typically tend to grow complex. In the case studies presented in this book, hierarchical modeling was thus an indispensable feature for specifying a code generator as a hierarchy of small and manageable models, and for being able to focus only on those hierarchy levels that were interesting in a particular phase of development.

Requirement G5 - Verification and Validation

Chap. 6 elaborated on the V&V facilities provided by the Genesys framework. The verification of a code generator is supported via local checking and model checking. The former checks local constraints that are attached to the single SIBs employed in the code generator, while the latter verifies global constraints which are associated with entire SLGs. As mentioned above, both types of constraints are also managed in corresponding libraries (the local constraints are managed along with the SIBs they are attached to), so that they form a growing knowledge base of rules for building good code generators.

Validation of code generators is supported by means of a dedicated testing framework incorporated into Genesys. This framework allows the specification of test cases and corresponding test data as SLGs. It also includes an approach for back-to-back testing of code generators, which has been exemplarily implemented for the jABC code generators.

Requirement S1 - Domain-Specificity

The EMF case study described in Chap. 7 presented an approach for generating domain-specific SIBs from a given metamodel. Those SIBs can be employed for building code generators that support instances of the metamodel as their input. By means of such domain-specific SIBs, the generator developer can be provided with a domain-specific modeling language tailored to concepts of the code generator's source domain (i.e., the domain for which the source language of the generator is used).

Furthermore, jABC's customization mechanisms have been employed for creating a domain-specific version of the tool, that is optimized for the construction of code generators, e.g., by means of corresponding SIB taxonomies and plugins. (cf. Chap. 4).

Requirement S2 - Full Code Generation

All jABC code generators (i.e., those that generate code from SLGs) support full code generation, as jABC's SLGs are designed in a way that they contain all information required for producing complete code (cf. Sect. 3.5). Sect. 3.1 also pointed out that this avoidance of round-trip engineering is a basic principle of XMDD.

Furthermore, the AndroMDA case study (cf. Chap. 8) showed that full code generation can also be achieved by using Genesys for combining different source languages and corresponding code generation frameworks integrated as services, thus exploiting their individual strengths.

Requirement S3 - Variant Management and Product Lines

In order to further facilitate reuse and adaptation, the Genesys framework supports the management of variants and product lines on the basis of jABC's mechanisms for hierarchical modeling (cf. Sect. 4.1.4).

Requirement S4 - Clean Code Generator Specification

When modeling a code generator with Genesys, the generation logic is typically reflected by the SLGs of the code generator, while the output description is specified by means of, e.g., templates or transformation rules, depending on which code generation technique is employed. The output description usually incarnates as parameter values of SIBs contained in the SLGs (as for the template-based generators presented in this book, which only use confined templates, cf. Sect. 4.2.5), or it is specified in a separate place (e.g., as for the transformation rules used by the FormulaBuilder). Consequently, there is typically a clean separation between generator logic and output description, which, e.g., is the basis for being able to properly verify the generator logic via model checking.

Requirement S5 - Bootstrapping

As SLGs are executable models and as jABC's Tracer provides a full-fledged interpreter for them, bootstrapping is easily possible. For instance, in the context of Genesys, bootstrapping has been successfully employed for obtaining the first jABC code generator (cf. Sect. 5.1).

Requirement S6 - Tool-Chain Integration

The tooling provided by the Genesys framework supports the integration of code generators into development tool-chains in several ways (cf. Sect. 4.3.2). For instance, code generators for jABC can be used via the Genesys jABC plugin. Furthermore, a dedicated Maven plugin allows the incorporation of code generators built with Genesys into Maven-based development environments. Finally, a code generator can be translated to an implementation language supported by existing build management tools like Apache Ant or GNU make.

9.2 Further Applications of Genesys

The Genesys framework is actively used beyond the various applications described in the case studies of this book. This section lists further examples of projects that employ the Genesys Framework.

OCS:

The *Online Conference Service* (OCS) [NMS11;KM06] is a web-based system for the submission and review of manuscripts, which are intended for being published, e.g., in conference proceedings or journals. The system has been developed for and is operated by the international publisher Springer [Spr11a]. On its way to publication, a manuscript typically runs through various phases involving several user roles. For instance, this includes the actual submission by the author, the distribution of reviewing tasks by an editor or program committee chair, or the review and discussion of the manuscript by reviewers.

As those phases differ essentially among conferences and journals, the OCS needs to be highly adaptable in order to cope with this variability. Arising from this requirement, the system employs SLGs for specifying product lines that are tailored to the needs of particular conferences or journals. Accordingly, for each phase of the evaluation workflow, the application expert is able to define the contained actions and their order in jABC. Apart from the examples mentioned above, those actions also include notifications of the users, such as an email informing about the start of a new phase. The corresponding SLGs are then generated using Genesys' Java Class Generator, and the resulting code basically forms the domain-specific portion of a particular OCS product line.

PROPHETS:

The *PROPHETS* plugin [NLS11; Nau09] extends jABC by the idea of *loose programming* [Lam+10]. According to this approach, branches in an SLG can be marked as being loosely specified, i.e., the specification of the system is deliberately vague or incomplete at this point. Based on domain knowledge that has been specified by a domain expert (cf. Sect. 3.2), PROPHETS employs a synthesis algorithm for finding valid constellations of services, that may concretize a loosely specified branch [NLS11]. Among other things, the domain knowledge includes type definitions and domain-specific constraints. The solutions found by the synthesis algorithm are proposed to the user, who may either choose to select one of those, or to refine the constraints and synthesis parameters for restarting the synthesis algorithm. This iterative refinement is repeated until an acceptable solution is found.

In order to support agility, this workflow underlying PROPHETS is entirely modeled in jABC. By means of Genesys' Java Class Generator, the corresponding SLGs are translated into Java code, which is in turn integrated into the PROPHETS plugin.

Reengineering and Migration of Legacy Systems:

Being the topic of a PhD thesis [Wag12], this project uses the Genesys framework for two main purposes. First, the project performs automatic reverse engineering of a legacy system by transforming its source code to so-called *code models* [WMP09], which are basically very low-level SLGs. In order to validate this transformation, an appropriate Genesys code generator is used for translating those code models back into code, which can then be compared to the original source code of the legacy system. Second, the project employs the Genesys framework in the context of the migration of legacy systems. For instance, in order to migrate a C++ legacy system to Java, the C++ services contained in the corresponding SLGs (e.g., obtained by reverse engineering as described above) can be successively enriched or replaced with corresponding Java services. Consequently, in intermediate stages of this migration process, the SLGs contain C++ services as well as Java services. In order to keep the system operational in this transitional state, a corresponding Genesys code generator is able to generate code from such SLGs with mixed service groundings. Wagner [Wag12] describes such a code generator, which generates a Java application that performs calls to the contained C++ services via CORBA.

oᗡOBS:

The *Dortmunder Online Bibliographieservice* (oᗡOBS) [Bah+07] is a web application that supports the bibliographical research of publications in computer science. It has been developed in the context of a one-year student project, which is a part of the curriculum at the TU Dortmund. oᗡOBS is based on the data provided by the well-known DBLP [Ley11], and aims at providing powerful search facilities for users and an administration interface for managing the data.

The web application also includes an import feature for periodically fetching the recent data of the DBLP and updating the oᗡOBS database correspondingly. This import feature parses the DBLP data (which is publicly available as a large XML file), then searches it for new and updated bibliographic records, and finally applies all changes to the database of oᗡOBS. For being able to quickly react to future changes of the data format, and in order to enable the extension of the import to support further data sources apart from the DBLP, this import process has been modeled in jABC. From the resulting SLGs, corresponding code is generated via the Java Class Generator, thus forming the backend of the import feature in oᗡOBS.

In a sequel student project [Can+08], the import functionality has been augmented by a process for the detection and management of duplicate records.

EE Deploy Plugin:

The *EE Deploy Plugin* has been developed in the context of the jABC EE mentioned in Sect. 5.4.8. It provides a simple GUI that enables the deployment of processes modeled in jABC to a (remote) instance of the execution engine. The deployment workflow that is executed by the plugin is modeled with jABC and generated with Genesys' Java Class Generator. This workflow uses the EE Process Definition Generator (cf. Sect. 5.4.8) as a service for generating process definitions from the SLGs. Afterwards, the resulting artifacts are packaged and uploaded to the execution engine.

SHADOWS:

The goal of the *SHADOWS* project (Self-healing Approach to Designing Complex Software Systems [She08]) is equipping complex systems with mechanisms for self-healing. Such mechanisms are intended to protect the system from issues concerning performance, the system's function and concurrency, occurring at design time, testing time as well as at runtime [She08]. Bakera et al. [Bak+10] proposed an approach that aims at supporting this goal for autonomic systems described in the Autonomic System Specification Language (ASSL) [Vas08].

According to this approach, those parts of an ASSL description which cover the system's behavioral aspects are translated into corresponding SLGs. For instance, this enables the use of model checking with GEAR (cf. Sect. 3.4) as a design-time healing technique. Furthermore, the resulting SLGs can be equipped with additional abstract annotations (using jABC's Annotation Editor, cf. Sect. 3.2.3), which serve as hints as to which parts of the behavioral specification of the system should be guarded by runtime healing techniques. At this point, the Genesys framework plays a central role: Upon code generation, those annotations are processed by means of a suitable Genesys code generator. Besides translating the SLGs to executable code as usual, the code generator also enriches the resulting code on the basis of the annotations, so that corresponding tools for runtime healing are able to work with it. Technically, such enrichments may, e.g., be specific annotations in the generated code, or modified service calls that integrate the techniques for self-healing.

A big advantage of this approach is the fact that specifying the mechanisms for self-healing at the level of SLGs does not require any technical knowledge about the actual tools that are used at runtime – this knowledge is provided by the code generator.

10 Future Work

The results on the Genesys approach and its reference implementation presented in this book matured over six years. However, as Genesys is designed to explicitly support adaptability and extensibility (e.g., via services or plugins), there is plenty of potential for future research. This chapter elaborates on several aspects that deserve closer examination.

Code-to-Model Traceability:

Even when the code generation process succeeded and the applied V&V tools detected no problems, errors might still occur at runtime of the generated system. For instance, such errors might arise from exceptional situations which have not been considered before, so that they are not covered by corresponding constraints or test cases, or from unexpected effects resulting from the interplay of generated code and the target platform (e.g., a JEE application server).

In order to diagnose such errors and to find their origin, it is very useful to be able to trace them back to the original models the system has been generated from. This requires bidirectionality of the mapping between models and code: For each part of the code, it has to be possible to determine the corresponding parts of the model from which the code has been generated [Kle08, p. 164]. For instance, MetaEdit+ supports code-to-model traceability by means of a feature called "live code" [TK09], which allows to directly inspect the corresponding model elements by selecting parts of the generated code.

Currently, the Genesys framework does not provide general facilities for supporting this. However, several code generators from the jABC case studies (e.g., the various Java Class Generators, cf. Chap. 5) include mechanisms that assist the developer in finding the model elements corresponding to a part of the generated code. First, by default, those generators enrich the generated code with comments that indicate which code emanated from which model elements. For instance, service calls are always preceded by a comment that includes the label of the corresponding SIB in the original model. Second, by means of a particular option, the generators can be instructed to augment the resulting code by debug output commands. When running the generated system, those output commands produce console messages, showing the trace through the original models that corresponds to the current execution of the system. This trace is displayed as an alternating sequence of SIB labels and branch names. By default, the option for generating the debug output commands is not enabled, in order not to affect the performance of the generated system.

For the jABC code generators, this basic support of code-to-model traceability could be extended by using the Tracer plugin for *live monitoring* of the generated code. This is possible due to the fact that the Tracer plugin is able to observe and visualize remote executions [Doe06, p. 64f] of jABC models via a remote version of the *Observer* pattern [Gam+95, p. 293ff]. The Tracer provides a standard implementation of this remote observer based on communication via Remote Method Invocation (RMI) [Gro01]. For using this mechanism from languages other than Java, it could be easily extended by further communication techniques such as CORBA or Web Services.

In order to exploit this feature for realizing code-to-model traceability, the generated code could be enhanced with further code that makes it act like a subject that can be observed by the Tracer plugin. This means that while executing the actual generated system, the code simultaneously triggers a visualization of the corresponding original model in the Tracer plugin. In consequence, the user is able to observe the execution of the generated system directly in jABC, at the modeling level. This enables "live" code-to-model traceability as a means for simplifying the diagnosis of errors.

However, this approach is only suitable for, e.g., tracking down the origins of performance problems or system hangs, which usually cause the system's execution to slow down or to stop entirely. In contrast to this, typical executions are usually too fast to be observable this way.

Another solution that also allows code-to-model traceability for such typical executions is an *a posteriori replay* of the trace. The realization of this approach would require the extension of both the code generators and the Tracer. First, the above mentioned generation of debug output commands would have to be changed: Instead of displaying the trace by means of console messages at runtime of the generated system, a file containing the entire trace has to be produced using a format understood by the Tracer. Second, the Tracer plugin has to be extended so that it is able to load such trace files, and to let the user "replay" the trace via the Tracer's debugging GUI. That way, the erroneous system execution could be reproduced step-by-step, thus enabling the a detailed a posteriori diagnosis.

Of course, as a prerequisite for supporting those solutions, any preceding model-to-model transformations need to be adapted, in order to retain all information about the original models required in the generated code.

Integration of Further Verification Approaches and Tools:

Besides the existing verification facilities presented in Chap. 6, the Genesys framework could be extended by further verification approaches and tools. Sect. 6.1 already mentioned *data-flow analysis* as an example, which could be performed, e.g., via model checking, as proposed by Steffen [Ste91] and Lamprecht et al. [LMS06]. In code generator SLGs, such a mechanism could be used to detect problems like unused context keys, services that try to access non-existing context keys, or the absence of expected inputs and outputs. Typically, such an approach would require the annotation of the code generator models with specific data flow information that is used by the analysis (see, e.g., [Lam+10]). For instance, when performing the data flow analysis via model checking, such annotations could take the form of atomic propositions. The data flow information is typically part of the domain knowledge assembled by the domain expert.

Furthermore, it could be a promising perspective to integrate approaches and tools that have been applied successfully for the verification of compilers (cf. Sect. 2.5), in order to examine their feasibility for the high-level code generator models in Genesys. The integration can be performed easily by using jABC's plugin mechanism. However, when conducting such experiments, it is imperatively important to keep in mind *Requirement G3 - Simplicity.* For instance, though theorem provers may be capable of verifying the correctness of a translation, their application typically requires lots of (formal) knowledge [Fra+08]. As shown with the use of model checking in this book, the integration of any formal methods should always be performed in a way that does not corrupt the overall simplicity of the Genesys framework.

Improvement of the Testing Framework:

As already mentioned in Sect. 6.3.2, future versions of Genesys' testing framework will include the extension of the suite graphs that resemble test suites. Those suite graphs will be augmented by the possibility of connecting contained macros with edges in order to reflect interdependencies between the corresponding test cases.

In its present state, the testing strategy described in Sect. 6.3.1 is realized in a pragmatically sufficient scale, in alignment with what was actually required for thoroughly testing the jABC code generators. However, several extensions of this implementation are imaginable for the future. For instance, test data SLGs (like those exemplified in Fig. 6.9) could themselves be parametrized with varying test vectors. Stürmer et al. refer to this as "second-order test cases" [Stü+07], with the test data SLGs becoming "first-order test cases". The current test suite for the jABC code generators does not yet employ such second-order test cases: If test data SLGs are parametrizable (via model parameters), the execution is always performed with the default values of the parameters.[1] In order to improve test coverage, the test suite is planned to be extended by this dimension of second-order test cases.

Another improvement for increasing the test coverage could be the employment of approaches for automatically generating test cases and test data, such as [Hag+02b; HMS03; Raf+09; Stü+07]. Currently, all test cases in the test suite are modeled manually.

Furthermore, a test becomes more expressive if the execution of the generated code on the *real* target system is also included in the comparison of the execution footprints, as proposed in [Stü+07] (called processor-in-the-loop, PIL). Currently, all execution runs in the tests (direct execution of the test data SLGs as well as the execution of the generated pendants) are performed on the system of the developer who actually runs the tests, or on a corresponding continuous integration server. For instance, in the context of embedded systems, this demands the use of appropriate emulators in order to be able to execute the generated code. By including execution runs on the real target systems in the tests, the complexity of the overall test setup increases in favor of more expressive results, as the tests now are also able to detect, e.g., unexpected side-effects of the target system's runtime environment.

Finally, the testing framework would particularly benefit from the improvements of code-to-model traceability described above. As both the SUT and the test case are translated to code before running the tests, the reports produced by the underlying testing platform always refer to the generated code. In consequence, corresponding facilities that assist in tracing back error reports to the original models would be a significant improvement.

Improvement and Generalization of the Variant Management Features:

The facilities for variant management in the Genesys framework (cf. Sect. 4.1.4) have been implemented pragmatically, in a demand-driven fashion. However, several further improvements of those features are imaginable:

- The *graphical user interface* has to be extended in order to better support the specification of variants. Besides a visualization of variation points (e.g., with suitable overlay icons), this includes a GUI for the separate parametrization of the variants. Currently, as described in Sect. 4.1.4, the variants are restricted to providing the same model interface, in order to compensate the lack of a proper GUI for parametrization.
- The introduction of dedicated *views* will help keeping track of specified variants. The most obvious view is one that shows the SLGs of one particular variant only. In the current state, the generator developer is only able to decide locally for each variation point, which corresponding variant should be displayed. However, as described in Sect. 4.1.4, variants (or product lines) are a global concept, applied to an entire SLG hierarchy. Accordingly, a variant-specific view would allow the generator developer

[1] Remember that any parameters in SLGs and SIBs always have to be equipped with default values (cf. Sect. 3.2).

to focus on working on a particular variant, without constantly having to take the local decision. Another helpful view would be the generation of a feature model [CHE04] that provides a comprehensive overview of all specified variants. This could be realized as an extension of the existing jABC hierarchy view (Fig. 4.10 in Sect. 4.2 shows an example).
- The *model weaving* employed for the generation of variants could be extended by mechanisms that additionally support aspect orientation, using variation points as real joinpoints in the AO sense. Aspects would again be resembled by SLGs which would be weaved into the main model, as specified by the joinpoints. Such joinpoints could be defined by the generator developer (e.g., by employing markers like "before" and "after", as in "*Weave in variant X before SIB Y.*"), or they could be determined on the basis of constraints (e.g., "*Replace all SIBs satisfying the atomic proposition p with variant X.*").
- The variant management could be augmented by features that allow the *dynamic selection of variants at runtime* of the generated system. Currently, the desired variant is selected at generation time, i.e., each code generation run only yields exactly one variant of the system. However, in order to enable the flexible reaction of the resulting system to certain situations (e.g., error recovery), it might be beneficial to let the system decide on which variant should be used, based on dynamic conditions occurring at runtime. For achieving this, there has to be a way for specifying runtime variability in the models, and code generators would have to be able to produce corresponding code.

Finally, the variant management features could be generalized in order to be usable for all jABC applications, and not just for code generators. This would include moving them out of the Genesys framework to a separate project. Furthermore, plugins like the Tracer and GEAR would have to be extended for supporting variants.

Loose Specification of Code Generators:

The features for loose programming and synthesis provided by the PROPHETS plugin (described above in Sect. 9.2) could also be applied for modeling code generators. This application is imaginable at two levels. First, for a single code generator, sequential standard parts could be loosely specified and automatically completed by means of synthesis. For instance, possible parts include initialization steps or postprocessing actions, such as pretty-printing the code and writing it to files. The remaining parts of a code generator, especially the templates, are not easily synthesizable from domain knowledge. In consequence, loose programming is not suitable for those parts, which thus stay the core competence of the generator developer.

Second, at a higher level, PROPHETS could be used for the synthesis of code generators, which are in turn composed of several consecutive transformation steps. For instance, as proposed by MDA (cf. Sect. 2.3.3), a code generator may be designed as a chain of model transformations, ending with a final code generation step. Given the availability of corresponding domain knowledge and of SIBs that incorporate suitable model transformations and code generators as services, such code generators could be specified loosely: Ideally, the generator developer would only have to specify the type of the input models along with a type that classifies the resulting code. On this basis, PROPHETS would automatically synthesize possible sequences of transformations, and thus simplify the generator developer's work significantly.

Aggregated Template View:

A characteristic of template-based Genesys code generators is the fact that templates tend to be scattered over the generator models (as, e.g., visible in the example shown in Sect. 4.2). This is a direct result of Genesys' different use of templates for keeping a strict separation between generation logic and output description, as described in Sect. 2.6 and 4.2.5. However, when using template engines, generator developers usually appreciate the advantage of having only a small number of templates, that provide a comprehensive overview of the entire output description.

In order to support this in the Genesys framework, an *aggregated template view* could be computed from a given template-based code generator. This view aggregates the template fragments found in the code generator to one template, thus providing the overview familiar to many generator developers.

Technically, the generation of this view could be backed by a corresponding code generator (also developed with Genesys), that produces the template. For this purpose, the template fragments have to be connected by the control flow structures specified in the generation logic of the code generator, for which the view should be produced. Thus the code generator that generates the view will have to include an approach for identifying control flow patterns, as, e.g., described in Sect. 5.2.4.

This view could be further supplemented by editing capabilities that allow direct modifications in the view, along with the propagation of those changes to the actual code generator models.

Migration to the Eclipse Platform:

At the time of writing this book, the jABC project is undergoing a transition to the Eclipse platform in order to benefit from the rich ecosystem provided by the Eclipse community. According to the recent direction of this transition, the purely Java-based core of jABC (including SLGs and SIBs) will be redesigned on the basis of EMF, e.g., by creating an explicit metamodel for SLGs with Ecore. Although there are plenty of code generation tools for EMF (cf. Chap. 7), none of these solutions follows an approach similar to Genesys. Accordingly, a migration of the Genesys framework to the Eclipse platform could be worthwhile.

The tooling included in the Genesys framework (cf. Sect. 4.3) will most likely require significant reimplementation effort due to the manifold differences of the Eclipse platform (e.g., the usage of the Standard Widget Toolkit (SWT) [Ecl11e]). However, the migration should be comparably simple for code generators. Due to the fact the all Genesys code generators are available as models, they could be ported to the new Ecore-based SLGs by means of an appropriate model transformation. Such a model transformation could also be used to convert the entire model library contained in the Genesys framework to the new SLG format. Furthermore, an additional converter will be required for importing existing SIBs to the new platform.

Bibliography

Primary References

[ABL08] van der Aalst, W.M.P., Lassen, K.B.: Translating unstructured workflow processes to readable BPEL: Theory and implementation. Information and Software Technology 50(3), 131–159 (2008)

[Abr+04] Abran, A., et al.: Guide to the Software Engineering Body of Knowledge - SWEBOK. 2004 version. IEEE Press (2004) ISBN: 0769510000

[Aho+06] Aho, A.V., et al.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison Wesley (2006) ISBN: 0321486811

[AK02] Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation (TOMACS) 12(4), 290–321 (2002)

[AK03] Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software 20(5), 36–41 (2003)

[App98] Appel, A.W.: Modern Compiler Implementation in Java. Cambridge University Press (1998) ISBN: 0521583888

[BA04] Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley (2004) ISBN: 0321278658

[Bac+63] Backus, J.W., et al.: Revised report on the algorithm language ALGOL 60. Communications of the ACM 6(1), 1–17 (1963); Ed. by Peter Naur

[Bag02] Bagnall, B.: Core Lego Mindstorms^{TM} Programming. Prentice Hall PTR (2002) ISBN: 0130093645

[Bah+07] Bahlo, T., et al.: PG 494 L2EE: Lightweight Process Coordination & J2EE. Final Report. In German, TU Dortmund, Chair of Programming Systems (2007), `https://eldorado.tu-dortmund.de/bitstream/2003/24525/1/PG494-Endbericht.pdf`

[Bak+07] Bakera, M., et al.: Property-driven functional healing: Playing against undesired behavior. In: Proceedings of the Business Process Engineering, CONQUEST 2007, pp. 363–372, dpunkt (2007)

[Bak+09] Bakera, M., et al.: Tool-supported enhancement of diagnosis in model-driven verification. Innovations in Systems and Software Engineering 5(3), 211–228 (2009)

[Bak+10] Bakera, M., et al.: Extracting Component-Oriented Behaviour for Self-Healing Enabling. In: IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe), pp. 152–161 (2010)

[BAPM83] Ben-Ari, M., Pnueli, A., Manna, Z.: The temporal logic of branching time. Acta Informatica 20(3), 207–226 (1983)

[BBF09] Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer 42, 22–27 (2009)

[BC11] Broy, M., Cengarle, M.V.: UML formal semantics: lessons learned. In: Software and Systems Modeling, pp. 1–6 (2011)

[Bec02] Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002) ISBN: 0321146530

[Bec04] Beck, K.: JUnit Pocket Guide (2004) ISBN: 0596007434

[Ben08] Bentmann, B.: Eine UML-Perspektive für jABC mit Anbindung an Standard-UML-Tools. Diploma Thesis. TU Dortmund, Chair of Programming Systems (2008) (in German)

[Beu10] Beulshausen, B.: Automatisches Generieren von Perl Code aus grafischen Prozessmodellen. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2010) (in German)

[BG01] Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE 2001, pp. 273–280. IEEE Computer Society (2001)

[BGL05] Blech, J.O., Glesner, S., Leitner, J.: Formal Verification of Java Code Generation from UML Models. Fujaba Days (2005)

[BJM09] Bakera, M., Jörges, S., Margaria, T.: Test your Strategy: Graphical Construction of Strategies for Connect-Four. In: Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, pp. 172–181. IEEE Computer Society (2009)

[Bla01] Blackwell, A.F.: Pictorial Representation and Metaphor in Visual Language Design. Journal of Visual Languages & Computing, 223–252 (2001)

[Bla96] Blackwell, A.F.: Metacognitive Theories of Visual Programming: What do we think we are doing? In: Proceedings of the 1996 IEEE Symposium on Visual Languages, pp. 240–246. IEEE Computer Society (1996)

[BM06] Bajohr, M., Margaria, T.: MaTRICS: A servicebased management tool for remote intelligent configuration of systems. Innovations in Systems and Software Engineering 2(2), 99–111 (2006)

[BM08] Bajohr, M., Margaria, T.: High Service Availability in MaTRICS for the OCS. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 572–586. Springer, Heidelberg (2008)

[Bra+08] Bravenboer, M., et al.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming 72(1-2), 52–70 (2008)

[BS07] Beust, C., Suleiman, H.: Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley (2007) ISBN: 0321503104

[Bur+05] Burdy, L., et al.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) 7(3), 212–232 (2005)

[Béz05] BÉzivin, J.: On the unification power of models. Software and System Modeling 4(2), 171–188 (2005)

[Béz+04] Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In: Aßmann, U., Akşit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005)

[Béz+05] Bézivin, J., et al.: Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In: Proceedings of the International Workshop on Software Factories at OOPSLA 2005 (2005)

[Can+08] Can, M., et al.: PG 513 DoPAC: Dortmund Online Public Access Catalog. Final Report. TU Dortmund, Chair of Programming Systems (2008) (in German)

[CE00] Czarnecki, K., Eisenecker, U.W.: Generative programming - methods, tools and applications. Addison-Wesley (2000) ISBN: 9780201309775

[CE81] Clarke, E.M., Allen Emerson, E.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Engeler, E. (ed.) Logic of Programs 1979. LNCS, vol. 125, pp. 52–71. Springer, Heidelberg (1981)

[CGP99] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999) ISBN: 9780262032704

[CH06] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)

[Cha+05] Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)

[Cha05] Charette, R.N.: Why Software Fails. IEEE Spectrum 42(9), 42–49 (2005)

[CHE04] Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)

[CL02] Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2002), pp. 322–328. CSREA Press (2002)

[Cla+11] Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)

[CN01] Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley (2001) ISBN: 978201703320

[CNW89] Chen, M., Nunamaker Jr., J.F., Sue Weber, E.: Computer-aided software engineering: present status and future directions. SIGMIS Database 20(1), 7–13 (1989)

[Coo+07] Cook, S., et al.: Domain Specific Development with Visual Studio DSL Tools. Addison-Wesley (2007) ISBN: 9780321398208

[CV09] Cerpa, N., Verner, J.M.: Why did your project fail? Communications of the ACM 52(12), 130–134 (2009) ISSN: 0001-0782

[Cza04] Czarnecki, K.: Overview of Generative Software Development. In: Banâtre, J.-P., Fradet, P., Giavitto, J.-L., Michel, O. (eds.) UPP 2004. LNCS, vol. 3566, pp. 326–341. Springer, Heidelberg (2005)

[CØV02] Czarnecki, K., Østerbye, K., Völter, M.: Generative Programming. In: Hernández, J., Moreira, A. (eds.) ECOOP 2002 Workshops. LNCS, vol. 2548, pp. 15–29. Springer, Heidelberg (2002)

[DAC99] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 411–420. IEEE CS Press (1999)

[Dal+99] Dalal, S.R., et al.: Model-based testing in practice. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 285–294. ACM (1999)

[DF06] Denney, E., Fischer, B.: Extending Source Code Generators for Evidence-Based Software Certification. In: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), pp. 138–145. IEEE Computer Society (2006)

[DGBP11] Dumas, M., García-Bañuelos, L., Polyvyanyy, A.: Unraveling Unstructured Process Models. In: Mendling, J., Weidlich, M., Weske, M. (eds.) BPMN 2010. LNBIP, vol. 67, pp. 1–7. Springer, Heidelberg (2010)

[Doe06] Doedt, M.: Erweiterung der jABC-Framework Bibliothek um eine modular anpassbare Ausführungsschicht. Diploma Thesis. TU Dortmund, Chair of Programming Systems (2006) (in German)

[Dra06] Drazek, M.: jABC Plugin zur Konvertierung von SIB-Graph Modellen in jABC unabhängige Klassen. Diploma Thesis. TU Dortmund, Chair of Programming Systems (2006) (in German)

[DVW07] van Deursen, A., Visser, E., Warmer, J.: Model-Driven Software Evolution: A Research Agenda. In: CSMR Workshop on Model-Driven Software Evolution (MoDSE 2007), pp. 41–49 (2007), `http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2007-006.pdf`

[Eng+99] Engels, G., Hücking, R., Sauer, S., Wagner, A.: UML Collaboration Diagrams and Their Transformation to Java. In: France, R.B. (ed.) UML 1999. LNCS, vol. 1723, pp. 473–488. Springer, Heidelberg (1999)

[Eng71] Engeler, E.: Structure and Meaning of Elementary Programs. In: Symposium on Semantics of Algorithmic Languages. Lect. Notes in Mathematics, vol. 188. Springer (1971)

[Fav04] Favre, J.-M.: Towards a Basic Theory to Model Driven Engineering. In: Workshop on Software Model Engineering, WISME 2004, Joint Event with UML 2004 (2004)

[Fis09] Fischbach, A.: JML-basierte Validierung von Design-by- Contract-Beschreibungen in jABC. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2009) (in German)

[Fla06] Flanagan, D.: JavaScript: The Definitive Guide. O'Reilly (2006) ISBN: 0596101996

[FM08] Flanagan, D., Matsumoto, Y.: The Ruby Programming Language, 1st edn. O'Reilly (2008) ISBN: 9780596516178

[Fow02] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2002) ISBN: 0321127420

[Fow10] Fowler, M.: Domain Specific Languages, 1st edn. Addison-Wesley (2010) ISBN: 9780321712943

[Fra+08] Frank, S., et al.: Safety of Compilers and Translation Techniques - Status quo of Technology and Science. In: Automotive – Safety & Security 2008. Shaker Verlag (2008)

[Fra02] Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons (2002) ISBN: 0471319201

[FT96] Frakes, W., Terry, C.: Software reuse: metrics and models. ACM Computing Surveys (CSUR) 28(2), 415–435 (1996)

[FVZ03] Fehnker, A., Vaandrager, F., Zhang, M.: Modeling and Verifying a Lego Car Using Hybrid I/O Automata. In: Proceedings of the Third International Conference on Quality Software, QSIC 2003, pp. 280–289 (2003)

[Gae07] Gaeb, J.: Entwicklung eines BPEL-Plugins für das JavaABCFramework. Diploma Thesis. TU Dortmund, Chair of Programming Systems (2007) (in German)

[Gam+95] Gamma, E., et al.: Design patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995) ISBN: 9780201633610

[GB08] García-Bañuelos, L.: Pattern Identification and Classification in the Translation from BPMN to BPEL. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 436–444. Springer, Heidelberg (2008)

[Gos+05] Gosling, J., et al.: The Java Language Specification, 3rd edn. Addison-Wesley (2005) ISBN: 0321246780

[GR83] Goldberg, A., Robson, D.: Smalltalk-80, The language and its implementation. Addison-Wesley (1983) ISBN: 0201113716

[Gre+04] Greenfield, J., et al.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons (2004) ISBN: 0471202843

[Gro01] Grosso, W.: Java RMI. O'Reilly (2001) ISBN: 9781565924529

[Gro09] Gronback, R.C.: Eclipse Modeling Project: A Domain- Specific Language (DSL) Toolkit. Addison-Wesley (2009) ISBN: 9780321534071

[GSR05] Geiger, L., Schneider, C., Reckord, C.: Templateand modelbased code generation for MDA-Tools. In: 3rd International Fujaba Days (2005)

[GZ99] Goos, G., Zimmermann, W.: Verification of Compilers. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, p. 201230. Springer, Heidelberg (1999)

[Hag+02a] Hagerer, A., Hungar, H., Margaria, T., Niese, O., Steffen, B., Ide, H.-D.: Demonstration of an Operational Procedure for the Model-Based Testing of CTI Systems. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 336–340. Springer, Heidelberg (2002)

[Hag+02b] Hagerer, A., Hungar, H.: Model Generation by Moderated Regular Extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002)

[Han07] Hansen, J.C.: Lego Mindstorms^{TM} NXT^{TM} Power Programming: Robotics in C. Variant Press (2007) ISBN: 0973864923

[Hem+10] Hemel, Z., et al.: Code generation by model transformation: a case study in transformation modularity. Software and System Modeling 9(3), 375–402 (2010)

[Her03] Herrington, J.: Code Generation in Action. Manning Publications Co. (2003) ISBN: 1930110979

[HKG07] Hartmeier, S., Krüger, J., Giegerich, R.: Webservices and Workflows on the Bielefeld Bioinformatics Server: Practices and Problems. In: Proceedings of NETTAB 2007 Workshop: A Semantic Web for Bioinformatics (2007)

[HLR08] Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)

[HLT03] Heckel, R., Lohmann, M., Thöne, S.: Towards a UML Profile for Service-Oriented Architectures. In: Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003). CTIT Technical Report. University of Twente, pp. 115–120 (2003)

[HM85] Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of the ACM 32(1), 137–161 (1985)

[HMS03] Hungar, H., Margaria, T., Steffen, B.: Test- Based Model Generation For Legacy Systems. In: Proceedings of the International Test Conference, ITC 2003, pp. 971–980 (2003)

[Hoa03] Hoare, T.: The verifying compiler: A grand challenge for computing research. Journal of the ACM 50(1), 63–69 (2003)

[Hoa69] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)

[HR04] Huth, M., Ryan, M.D.: Logic in computer science - modelling and reasoning about systems, 2nd edn. Cambridge University Press (2004) ISBN: 9780521543101

[Hun90] Hunter, R.: The Design and Construction of Compilers. John Wiley & Sons (1990) ISBN: 0471280542

[Hör+08] Hörmann, M., et al.: The jABC Approach to Rigorous Collaborative Development of SCM Applications. In: Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2008, pp. 724–737 (2008)

[Hös08] Hösel, S.: Entwicklung eines Plug-ins für das jABC-Framework zur Integration von.NET mittels C#. Diploma Thesis. University of Potsdam, Chair for Service and Software Engineering (2008) (in German)

[IIT93] Telecommunication Standardization Sector of ITU (ITU-T). General Recommendations on Telephone Switching and Signalling - Intelligent Network: Introduction to Intelligent Network Capability Set 1, Recommendation Q.1211. Tech. rep. International Telecommunication Union, ITU (1993)

[IIT97] Telecommunication Standardization Sector of ITU (ITU-T). Recommendation I.329/Q.120: Intelligent Network - Global Functional Plane Architecture, 2nd edn. International Telecommunication Union, ITU (1997)

[Ive+00] Iversen, T.K., et al.: Model-checking real-time control programs: verifying Lego$^{®}$ Mindstorms^{TM} system using UPPAAL. In: Proceedings of the 12th Euromicro Conference on Real-Time Systems, Euromicro-RTS 2000, pp. 147–155. IEEE Computer Society (2000)

[JK06] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)

[JMS06] Jörges, S., Margaria, T., Steffen, B.: FormulaBuilder: A Tool for Graph-Based Modelling and Generation of Formulae. In: Proceedings of the 28th International Conference on Software engineering, ICSE 2006, pp. 815–818. ACM (2006)

[JMS08] Jörges, S., Margaria, T., Steffen, B.: Genesys: service-oriented construction of property conform code generators. Innovations in Systems and Software Engineering 4(4), 361–384 (2008)

[JMS11] Jörges, S., Margaria, T., Steffen, B.: Assuring property conformance of code generators via model checking. Formal Aspects of Computing 23(5), 589–606 (2011)

[JS04] Jürjens, J., Shabalin, P.: Automated Verification of UMLsec Models for Security Requirements. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 365–379. Springer, Heidelberg (2004)

[JS11] Jörges, S., Steffen, B.: Leveraging Service-Orientation for Combining Code Generation Frameworks. In: Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, pp. 198–207. IEEE Computer Society (2011)

[JSM10] Jörges, S., Steffen, B., Margaria, T.: Building Code Generators with Genesys: A Tutorial Introduction. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2011. LNCS, vol. 6491, pp. 364–385. Springer, Heidelberg (2011)

[Jör+07] Jörges, S., et al.: Model Driven Design of Reliable Robot Control Programs Using the jABC. In: Proceedings of the 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, EASe 2007, pp. 137–148. IEEE Computer Society (2007)

[Kah87] Kahn, G.: Natural Semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)

[KHB00] Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.J.: On Structured Workflow Modelling. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 431–445. Springer, Heidelberg (2000)

[Kic+97] Kiczales, G., et al.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)

[Kle+96] Klein, M., et al.: DFA&OPT-METAFrame: A Tool Kit for Program Analysis and Optimization. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 422–426. Springer, Heidelberg (1996)

[Kle08] Kleppe, A.: Software Language Engineering: Creating Domain- Specific Languages Using Metamodels, 1st edn. Addison-Wesley (2008) ISBN: 9780321553454

[KLR96] Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 1–21. Springer, Heidelberg (1996)

[KM06] Karusseit, M., Margaria, T.: Feature-based Modelling of a Complex, Online-Reconfigurable Decision Support Service. Electronic Notes in Theoretical Computer Science 157(2), 101–118 (2006)

[Koc09] Kochan, S.: Programming in Objective-C 2.0, 2nd edn. Addison- Wesley (2009) ISBN: 9780321566157

[Kol10] Koloch, R.: Genesys und open Architecture Ware: Ein praktischer Vergleich. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2010) (in German)

[Koz83] Kozen, D.: Results on the Propositional μ-Calculus. Theoretical Computer Science 27, 333–354 (1983)

[KP05] Kossatchev, A.S., Posypkin, M.A.: Survey of compiler testing methods. Programming and Computer Software 31(1), 10–19 (2005)

[KP09] Kelly, S., Pohjonen, R.: Worst Practices for Domain- Specific Modeling. IEEE Software 26, 22–29 (2009)

[KSV09] Klint, P., van der Storm, T., Vinju, J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 168–177. IEEE Computer Society (2009)

[KT08] Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons (2008) ISBN: 0470036664

[Kub+09] Kubczak, C., et al.: Service-oriented Mediation with jABC/- jETI. In: Semantic Web Services Challenge. Semantic Web and Beyond, vol. 8, pp. 71–99. Springer (2009)

[KV10] Kats, L.C.L., Visser, E.: The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 444–463. ACM (2010)

[Küh06] Kühne, T.: Matters of (Meta-)Modeling. Software and System Modeling 5(4), 369–385 (2006)

[Lab+07] Labarga, A., et al.: Web Services at the European Bioinformatics Institute. Nucleic Acids Research 35(Web-Server- Issue), 6–11 (2007)

[Lam+10] Lamprecht, A.-L., et al.: Synthesis-Based Loose Programming. In: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology, QUATIC, pp. 262–267. IEEE Computer Society (2010)

[Lar95] Laroussinie, F.: About the expressive power of CTL combinators. Information Processing Letters 54(6), 343–345 (1995)

[Led+01] Ledeczi, A., et al.: The Generic Modeling Environment. In: Workshop on Intelligent Signal Processing, WISP 2001, vol. 17. IEEE (2001)

[Lem+09] Lemcke, J., et al.: Advances in Solving the Mediator Scenario with jABC and jABC/GEM. In: Semantic Web Services Challenge: Proceedings of the 2008 Workshops, LG-2009-01, pp. 89–102 (2009), `http://logic.stanford.edu/reports/LG-2009-01.pdf`

[Len09] Lenzner, N.: Einbindung von UML-Modellierungs- und Codegenerierungstools in das Genesys-Framework. Bachelor Thesis. TU Dortmund, Chair of Programming Systems (2009) (in German)

[Ler06] Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, pp. 42–54. ACM (2006)

[LMB92] Levine, J., Mason, T., Brown, D.: lex & yacc, 2nd edn. (A Nutshell Handbook). O'Reilly (1992) ISBN: 1565920007

[LMM99] Latella, D., Majzik, I., Massink, M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. Formal Aspects of Computing 11(6), 637–664 (1999)

[LMS06] Lamprecht, A.-L., Margaria, T., Steffen, B.: Data-Flow Analysis as Model Checking Within the jABC. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 101–104. Springer, Heidelberg (2006)

[LMS08] Lamprecht, A.-L., Margaria, T., Steffen, B.: Seven Variations of an Alignment Workflow - An Illustration of Agile Process Design and Management in Bio-jETI. In: Măndoiu, I., Wang, S.-L., Zelikovsky, A. (eds.) ISBRA 2008. LNCS (LNBI), vol. 4983, pp. 445–456. Springer, Heidelberg (2008)

[LMS09] Lamprecht, A.-L., Margaria, T., Steffen, B.: From Bio-jETI Process Models to Native Code. In: Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, pp. 95–101. IEEE Computer Society (2009)

[LPT78] Lecarme, O., Peyrolle-Thomas, M.-C.: Self-compiling Compilers: An Appraisal of their Implementation and Portability. Software - Practice and Experience 8(2), 149–170 (1978)

[LS+08] López-Sanz, M., et al.: Modelling of Service-Oriented Architectures with UML. Electronic Notes in Theoretical Computer Science 194(4) (2008); Proceedings of the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2007, pp. 23–37 (2007)

[LY99] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley (1999) ISBN: 0201432943

[MB02] Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley (2002) ISBN: 0201748045

[McC60] McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, Part I. Communications of the ACM 3(4), 184–195 (1960)

[MER99] Medvidovic, N., Egyed, A.F., Rosenblum, D.S.: Round-Trip Software Engineering Using UML: From Architecture to Design and Back. In: Proceedings of the 2nd Workshop on Object-Oriented Reengineering, WOOR 1999 (1999)

[Mey92] Meyer, B.: Applying 'Design by Contract'. Computer 25(10), 40–51 (1992)

[Mil89] Milner, R.: Communication and Concurrency. Prentice-Hall (1989) ISBN: 9780131150072

[MKS08] Margaria, T., Kubczak, C., Steffen, B.: Bio-jETI: a service integration, design, and provisioning platform for orchestrated bioinformatics processes. BMC Bioinformatics 9(S-4), 1–17 (2008)

[MNS02] Margaria, T., Niese, O., Steffen, B.: Demonstration of an Automated Integrated Test Environment for Web-Based Applications. In: Bošnački, D., Leue, S. (eds.) SPIN 2002. LNCS, vol. 2318, pp. 250–253. Springer, Heidelberg (2002)

[MO97] Müller-Olm, M.: Modular Compiler Verification. LNCS, vol. 1283. Springer, Heidelberg (1997)

[Moo09] Moody, D.: The 'Physics' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE Transactions on Software Engineering 35(6), 756–779 (2009)

[Mor08] Morimoto, S.: A Survey of Formal Verification for Business Process Modeling. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part II. LNCS, vol. 5102, pp. 514–522. Springer, Heidelberg (2008)

[MOSS99] Müller-Olm, M., Schmidt, D.A., Steffen, B.: Model-Checking: A Tutorial Introduction. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 330–354. Springer, Heidelberg (1999)

[MOY04] Müller-Olm, M., Yoo, H.: MetaGame: An Animation Tool for Model-Checking Games. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 163–167. Springer, Heidelberg (2004)

[MRS06] Margaria, T., Rüthing, O., Steffen, B.: ViDoC - Visual Design of Optimizing Compilers. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 145–159. Springer, Heidelberg (2007)

[MS04] Margaria, T., Steffen, B.: Lightweight coarsegrained coordination: a scalable system-level approach. International Journal on Software Tools for Technology Transfer (STTT) 5(2), 107–123 (2004)

[MS06] Margaria, T., Steffen, B.: Service Engineering: Linking Business and IT. IEEE Computer 39(10), 45–55 (2006)

[MS08] Margaria, T., Steffen, B.: Agile IT: Thinking in User-Centric Models. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 490–502. Springer, Heidelberg (2008)

[MS09a] Margaria, T., Steffen, B.: Business Process Modelling in the jABC: The One-Thing Approach. In: Handbook of Research on Business Process Modeling, pp. 1–26. IGI Global (2009)

[MS09b] Margaria, T., Steffen, B.: Continuous Model- Driven Engineering. IEEE Computer 42(10), 106–109 (2009)

[MSR05] Margaria, T., Steffen, B., Reitenspiess, M.: Service-Oriented Design: The Roots. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 450–464. Springer, Heidelberg (2005)

[Nag09] Nagel, R.: Technische Herausforderungen modellgetriebener Beherrschung von Prozesslebenszyklen aus der Fachperspektive: Von der Anforderungsanalyse zur Realisierung. PhD thesis, TU Dortmund, Chair Programming Systems (2009) (in German)

[Nau09] Naujokat, S.: Automatische Generierung von Prozessen im jABC. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2009) (in German)

[Nec00] Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI 2000, pp. 83–94. ACM (2000)

[Nec97] Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 106–119. ACM (1997)

[Neu07] Neubauer, J.: LocalChecker Plugin for the jABC. Student Research Project Report. TU Dortmund, Chair of Programming Systems (2007)

[Nie+01a] Niese, O., et al.: Automated Regression Testing of CTI-Systems. In: Proceedings of the IEEE European Test Workshop, ETW 2001, pp. 51–57. IEEE Computer Society (2001)

[Nie+01b] Niese, O., Steffen, B., Margaria, T., Hagerer, A., Brune, G., Ide, H.-D.: Library-Based Design and Consistency Checking of System-Level Industrial Test Cases. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 233–248. Springer, Heidelberg (2001)

[NLS11] Naujokat, S., Lamprecht, A.-L., Steffen, B.: Tailoring Process Synthesis to Domain Characteristics. In: Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, pp. 167–175. IEEE Computer Society (2011)

[NMS11] Neubauer, J., Margaria, T., Steffen, B.: Design for Verifiability: The OCS Case Study. In: Formal Methods for Industrial Critical Systems: A Survey of Applications. John Wiley & Sons (2011) (in print) ISBN: 9780470876183

[Old+05] Oldevik, J., Neple, T., Grønmo, R., Aagedal, J.Ø., Berre, A.-J.: Toward Standardised Model to Text Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 239–253. Springer, Heidelberg (2005)

[Oul82] Oulsnam, G.: Unravelling Unstructured Programs. The Computer Journal 25(3), 379–387 (1982)

[Ouy+06] Ouyang, C., et al.: From BPMN Process Models to BPEL Web Services. In: Proceedings of the IEEE International Conference on Web Services, pp. 285–292. IEEE Computer Society (2006)

[Pap08] Papazoglou, M.P.: Web Services: Principles and Technology. Pearson, Prentice Hall (2008) ISBN: 9780321155559

[Par04] Parr, T.: Enforcing strict model-view separation in template engines. In: Proceedings of the 13th International Conference on World Wide Web, WWW 2004, pp. 224–233. ACM (2004)

[PBL05] Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005) ISBN: 3540243720

[Plo81] Plotkin, G.D.: A Structural Approach to Operational Semantics. Tech. rep. DAIMI FN-19. Computer Science Department, Aarhus University (1981)

[Pol07] Polanski, A.: Sequence Alignment. In: Bioinformatics, pp. 155-156. Springer (2007)

[PQ95] Parr, T., Quong, R.: ANTLR: A Predicated-LL(k) Parser Generator. Software - Practice and Experience 25(7), 789–810 (1995)

[PSS98] Pnueli, A., Siegel, M., Singerman, E.: Translation Validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)

[QS82] Queille, J.-P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)

[Raf+08] Raffelt, H., et al.: Hybrid test of web applications with webtest. In: Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, TAVWEB 2008, pp. 1–7. ACM (2008)

[Raf+09] Raffelt, H., et al.: Dynamic testing via automata learning. International Journal on Software Tools for Technology Transfer (STTT) 11(4), 307–324 (2009)

[Rai+04] Raistrick, C., et al.: Model Driven Architecture with Executable UML. Cambridge University Press (2004) ISBN: 0521537711

[RS09] Ryabtsev, M., Strichman, O.: Translation Validation: From Simulink to C. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 696–701. Springer, Heidelberg (2009)

[Rug+08] Rugina, A.-E., et al.: Gene-Auto: Automatic Software Code Generation for Real-Time Embedded Systems. In: Data Systems in Aerospace, DASIA 2008. European Space Agency (2008)

[SB01] Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall PTR (2001) ISBN: 9780130676344

[SC04] Stürmer, I., Conrad, M.: Code Generator Testing in Practice. In: INFORMATIK 2004, - Informatik verbindet, Band 2, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V (GI), pp. 33-37. GI (2004)

[Sch+04] Schinz, I., et al.: The Rhapsody UML Verification Environment. In: Proceedings of the Second International Conference on Software Engineering and Formal Methods, SEFM 2004, pp. 174–183. IEEE Computer Society Press (2004)

[Sch+10] Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)

[Sch06] Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39, 25–31 (2006)

[Sch07] Schulte, B.E.: Modellgetriebene Steuerung eingebetteter Systeme und ihrer Anwendung für Lego NXT. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2007) (in German)

[Sch86] Schmidt, D.A.: Denotational semantics: a methodology for language development. William C. Brown Publishers (1986) ISBN: 0697068492

[Sei03] Seidewitz, E.: What Models Mean. IEEE Software 20(5), 26–32 (2003)

[Sel03] Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20, 19–25 (2003)

[Sel09] Selic, B.: The Theory and Practice of Modeling Language Design for Model-Based Software Engineering—A Personal Perspective. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 290–321. Springer, Heidelberg (2011)

[She08] Shehory, O.: SHADOWS: Self-healing complex software systems. In: 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshop Proceedings (ASE Workshops 2008), pp. 71–76. IEEE (2008)

[SI94] Steffen, B., Ingólfsdóttir, A.: Characteristic formulae for processes with divergence. Information and Computation 110(1), 149–163 (1994)

[SK03] Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software 20(5), 42–45 (2003)

[SK97] Sztipanovits, J., Karsai, G.: Model-integrated computing. Computer 30(4), 110–111 (1997)

[SM99] Steffen, B., Margaria, T.: META*Frame* in Practice: Design of Intelligent Network Services. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 390–415. Springer, Heidelberg (1999)

[SMN05] Steffen, B., Margaria, T., Nagel, R.: Remote Integration and Coordination of Verification Tools in jETI. In: Proceedings of 12th IEEE International Conference on the Engineering of Computer Based Systems, ECBS 2005, pp. 431–436. IEEE Computer Society Press (2005)

[Smo10] Smolinski, M.: Modellbasierte Entwicklung und Generierung von Test-Suiten: Eine Fallstudie anhand von Genesys & JUnit. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2010) (in German)

[SMW10] Steffen, B., Margaria, T., Wagner, C.: Round- Trip Engineering. In: Encyclopedia of Software Engineering, pp. 1044–1055. Taylor & Francis (2010)

[SN07] Steffen, B., Narayan, P.: Full Life-Cycle Support for End-to-End Processes. IEEE Computer 40(11), 64–73 (2007)

[Spi09] Spitzer, D.: Modellbasierte Entwicklung von Code-Generatoren für stark eingeschränkte Ausführungsumgebungen am Beispiel der iPhone-Plattform. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2009) (in German)

[SS98] Schmidt, D.A., Steffen, B.: Program Analysis *as* Model Checking of Abstract Interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)

[Sta+07] Stahl, T., et al.: Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management, 2nd edn., dpunkt (2007) (in German) ISBN: 978-3-89864-448-8

[Sta73] Stachowiak, H.: Allgemeine Modelltheorie. Springer (1973) (in German) ISBN: 9783211811061

[Ste+07] Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-Driven Development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 92–108. Springer, Heidelberg (2007)

[Ste+09] Steinberg, D., et al.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley (2009) ISBN: 0321331885

[Ste+94] Steffen, B., et al.: Intelligent Software Synthesis in the Da- Capo Environment. In: Proceedings of the 6th Nordic Workshop on Programming Theory. BRICS Report N. 94/6 (December 1994)

[Ste+96] Steffen, B., et al.: Incremental Formalization: A Key to Industrial Success. Software - Concepts and Tools 17(2), 78–91 (1996)

[Ste+97] Steffen, B., et al.: Hierarchical Service Definition. Annual Review of Communication, 847–856 (1997)

[Ste89] Steffen, B.: Characteristic Formulae. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 723–732. Springer, Heidelberg (1989)

[Ste91] Steffen, B.: Data Flow Analysis as Model Checking. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 346–365. Springer, Heidelberg (1991)

[Sto10] Storz, D.: Intuitive und experimentelle Modellierung von Strategien am Beispiel von Robocode. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2010) (in German)

[Str02] Strecker, M.: Formal Verification of a Java Compiler in Isabelle. In: Voronkov, A. (ed.) CADE-18. LNCS (LNAI), vol. 2392, pp. 63–77. Springer, Heidelberg (2002)

[Stü+07] Stürmer, I., et al.: Systematic Testing of Model-Based Code Generators. IEEE Transactions on Software Engineering 33(9), 622–634 (2007)

[SWC05] Stürmer, I., Weinberg, D., Conrad, M.: Overview of existing safeguarding techniques for automatically generated code. In: Proceedings of the Second International Workshop on Software Engineering for Automotive Systems, SEAS 2005, pp. 1–6. ACM (2005)

[THG94] Thompson, J.D., Higgins, D.G., Gibson, T.J.: CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, pÆŠositionspecific gap penalties and weight matrix choice. Nucleic Acids Research 22(22), 4673–4680 (1994)

[Tho04] Thomas, D.: MDA: Revenge of the Modelers or UML Utopia? IEEE Software 21(3), 15–17 (2004)

[TK09] Tolvanen, J.-P., Kelly, S.: MetaEdit+: defining and using integrated domain-specific modeling languages. In: Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, pp. 819–820. ACM (2009)

[Tol11] Tolmatcev, O.: Erprobung und Evaluierung der Anwendbarkeit von XSLT im Code-Generator-Framework Genesys. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2011) (in German)

[UL06] Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers (2006) ISBN: 0123725011

[V03] Völter, M.: A Catalog of Patterns for Program Generation. In: Proceedings of the 8th European Conference on Pattern Languages of Programs, EuroPLoP 2003, UVK (2003)

[Vas08] Vassev, E.I.: Towards a framework for specification and code generation of automatic systems. PhD thesis. Concordia University (2008)

[VDA+03] Van Der Aalst, W.M.P., et al.: Workflow Patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)

[Vli98] Vlissides, J.: Pattern hatching: design patterns applied. Addison-Wesley (1998) ISBN: 0201432935

[Vou90] Vouk, M.A.: Back-to-back testing. Information and Software Technology 32(1), 34–45 (1990)

[VV10] Völter, M., Visser, E.: Language extension and composition with language workbenches. In: Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, pp. 301–304. ACM (2010)

[VVK08] Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 100–115. Springer, Heidelberg (2008)

[Völ09] Völter, M.: MD* Best Practices. Journal of Object Technology 8(6), 79–102 (2009)

[Wag12] Wagner, C.: Modellgetriebene Software-Migration. PhD thesis. University of Potsdam, Chair of Service and Software Engineering (2012) (in German)

[Wal00] Wall, L.: Programming Perl, 3rd edn. O'Reilly (2000) ISBN: 0596000278

[War94] Ward, M.P.: Language-Oriented Programming. Software - Concepts and Tools 15(4), 147–161 (1994)

[Wat93] Watt, D.A.: Programming Language Processors: Compilers and Interpreters. Prentice Hall (1993) ISBN: 9780137201297

[Whi97] Whitley, K.N.: Visual Programming Languages and the Empirical Evidence For and Against. Journal of Visual Languages & Computing 8(1), 109–142 (1997)

[Wil77] Howard Williams, M.: Generating Structured Flow Diagrams: The Nature of Unstructuredness. The Computer Journal 20(1), 45–50 (1977)

[Win06] Winkler, C.: Entwicklung eines jABC-Plugins zum Design von JDBC-kompatiblen Datenbankschemata. Diploma Thesis, TU Dortmund, Chair of Programming Systems (2006) (in German)

[WMP09] Wagner, C., Margaria, T., Pagendarm, H.-G.: Analysis and Code Model Extraction for C/C++ Source Code. In: Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, pp. 110–119. IEEE Computer Society (2009)

[Yoo07] Yoo, H.: Fehlerdiagnose beim Model-Checking durch animierte Strategie-Synthese. PhD thesis, TU Dortmund, Chair Programming Systems (2007) (in German)

Online References

[App11] Apple. iOS Technology Overview (2011), http://developer.apple.com/technologies/ios/

[Boo03] Boocock, P.: The Jamda Project (2003), http://jamda.sourceforge.net/

[Com11] JRuby Community. JRuby.org (2011), http://www.jruby.org

[Dmi04] Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm (2004), http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf

[Fow05] Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? (2005), http://martinfowler.com/articles/languageWorkbench.html

[JD10] DNA Data Bank of Japan (DDBJ). Web API for Biology (WABI) (2010), http://xml.nig.ac.jp/index.html

[Jet11] JetBrains. Meta Programming System (2011), http://www.jetbrains.com/mps/

[Jör10] Jörges, S.: Genesys SIBs Website (2010), http://jabc.cs.tu-dortmund.de/genesys/genesys-lib/sibdocs/index.html

[Lar11] Larsen, F.N.: Robocode Home (2011), http://robocode.sourceforge.net/

[Ley11] Ley, M.: DBLP Computer Science Bibliography (2011), http://dblp.uni-trier.de/

[Mic11] Microsoft. Microsoft .NET Framework (2011), http://www.microsoft.com/net

[Ora11a] Oracle. Enterprise JavaBeans Technology (2011), http://www.oracle.com/technetwork/java/javaee/ejb

[Ora11b] Oracle. Java Enterprise Edition Website (2011), http://www.oracle.com/technetwork/java/javaee

[Ora11c] Oracle. Java ME (2011), http://www.oracle.com/technetwork/java/javame

[Ora11d] Oracle. JavaBeans Specification (2011), http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

[TU10] TU Dortmund, Chair for Programming Systems. jABC Common SIBs Website (2010), http://www.jabc.de/sib

[Act11a] Active Endpoints. ActiveVOS BPMS (2011), http://activevos.com/

[Act11b] Activiti Team. Activiti (2011), http://www.activiti.org/

[Alt11] Altova. Altova UModel (2011), http://www.altova.com/umodel.html

[And11] AndroMDA Team. AndroMDA (2011), http://www.andromda.org/

[Apa10] Apache Foundation. Apache Velocity (2010), http://velocity.apache.org

[Apa11a] Apache Foundation. Apache Ant (2011), http://ant.apache.org/

[Apa11b] Apache Foundation. Apache Maven (2011), http://maven.apache.org

[Apa11c] Apache Foundation. Apache ODE (2011), `http://ode.apache.org/`

[Apa11d] Apache Software Foundation. Apache Struts (2011), `http://struts.apache.org/`

[Apa11e] Apache Software Foundation. Apache Subversion (2011), `http://subversion.apache.org/`

[Bor11] Borland. Together (2011), `http://www.borland.com/us/products/together/`

[Car11] Schwennicke, C., Gallert, S.: XCoder (2011), `http://xcoder.sf.net`

[Ecl05] Eclipse Foundation. EMF Developer Guide (2005), `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html`

[Ecl11a] Eclipse Foundation. Eclipse Graphical Modeling Framework (GMF) (2011), `http://www.eclipse.org/gmf/`

[Ecl11b] Eclipse Foundation. Eclipse Model Development Tools (MDT): UML2 (2011),`http://www.eclipse.org/modeling/mdt/?project=uml2`

[Ecl11c] Eclipse Foundation. Eclipse Model Development Tools (MDT): UML2 Tools (2011), `http://www.eclipse.org/modeling/mdt/?project=uml2tools`

[Ecl11d] Eclipse Foundation. JET (2011), `http://www.eclipse.org/modeling/m2t/?project=jet#jet`

[Ecl11e] Eclipse Foundation. SWT: The Standard Widget Toolkit (2011), `http://www.eclipse.org/swt/`

[Ecl11f] Eclipse Foundation. Xpand (2011), `http://www.eclipse.org/modeling/m2t/?project=xpand`

[Ecl11g] Eclipse Foundation. Xtend 2 (2011), `http://www.eclipse.org/Xtext/#xtend2`

[Ecl11h] Eclipse Foundation. Xtext (2011), `http://www.eclipse.org/Xtext/`

[Fre11a] Free Software Foundation. GNU 'make' (2011), `http://www.gnu.org/software/make/manual/make.html`

[Fre11b] FreeMarker Project. FreeMarker: Java Template Engine Library (2011), `http://freemarker.org/`

[ISO05] ISO. The ANSI C standard (C99). Tech. rep. WG14 N1124. ISO/IEC (2005), `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`

[Jav02] Java Community Process. JSR 40: The tadata Interface (JMI) Specification (2002), `http://www.jcp.org/en/jsr/detail?id=40`

[Jav06] Java Community Process. JSR 52: A Standard Tag Library for JavaServer Pages (2006), `http://www.jcp.org/en/jsr/detail?id=52`

[Jav07] Java Community Process. JSR 139: Connected Limited Device Configuration 1.1 (2007), `http://www.jcp.org/en/jsr/detail?id=139`

[Jav09a] Java Community Process. JSR 222: Java Architecture for XML Binding (JAXB) 2.0 (2009), `http://www.jcp.org/en/jsr/detail?id=222`

[Jav09b] Java Community Process. JSR 245: JavaServer Pages 2.1 (2009), `http://www.jcp.org/en/jsr/detail?id=245`

[Jav09c] Java Community Process. JSR 271: Mobile Information Device Profile 3 (2009), `http://www.jcp.org/en/jsr/detail?id=271`

[Jav11a] Java Community Process. JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.0 (2011), `http://jcp.org/en/jsr/detail?id=224`

[Jav11b] Java Community Process. JSR 315: Java Servlet 3.0 Specification (2011), `http://www.jcp.org/en/jsr/detail?id=315`

[Ken03] Kennedy-Carter. UML ASL Reference Guide, ASL Language Level 2.5 (2003), `http://www.ooatool.com/docs/ASL03.pdf`

[Mic11] Microsoft. Official Microsoft ASP.NET Site (2011), `http://www.asp.net/`

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`

[Obe11] Obeo. Acceleo - MDA generator (2011), `http://www.acceleo.org`

[Obj03a] Object Management Group. Common Warehouse Metamodel (CWM) Specification, Version 1.1. formal/03-03-02 (2003), `http://www.omg.org/spec/CWM/1.1/`

[Obj03b] Object Management Group. Model Driven Architecture (MDA) Guide Version 1.0.1. omg/2003-06-01 (2003), `http://www.omg.org/cgi-bin/doc?omg/03-06-01`

[Obj04] Object Management Group. Human-Usable Textual Notation (HUTN) Specification. formal/2004-08-01, `http://www.omg.org/spec/HUTN/1.0/`

[Obj07] Object Management Group. MOF 2.0/XMI Mapping, Version 2.1.1. formal/2007-12-01 (2007), `http://www.omg.org/spec/XMI/2.1.1/`

[Obj08] Object Management Group. MOF Model to Text Transformation Language, Version 1.0. formal/2008-01-16 (2008), `http://www.omg.org/spec/MOFM2T/1.0/`

[Obj10a] Object Management Group. OMG Unified Modeling Language$^{\mathrm{TM}}$ (OMG UML), Infrastructure, Version 2.3. formal/2010-05-03 (2010), `http://www.omg.org/spec/UML/2.3/`

[Obj10b] Object Management Group. OMG Unified Modeling Language^{TM} (OMG UML), Superstructure, Version 2.3. formal/2010-05-05 (2010), `http://www.omg.org/spec/UML/2.3/`

[Obj11a] Object Management Group. OMG Unified Modeling Language^{TM} (OMG UML), Superstructure, Version 2.3. formal/2010-05-05 (2010), `http://www.omg.org/spec/UML/2.3/`

[Obj11b] Object Management Group. Common Object Request Broker Architecture, CORBA (2011), `http://www.omg.org/spec/CORBA/`

[Obj11c] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification, Version 1.1. formal/2011-01-01 (2011), `http://www.omg.org/spec/QVT/1.1/`

[Obj11d] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification, Version 1.1. formal/2011-01-01 (2011), `http://www.omg.org/spec/QVT/1.1/`

[Red11a] Red Hat. JBoss Application Server (2011), `http://www.jboss.org/jbossas/`

[Red11b] Red Hat. jBPM (2011), `http://www.jboss.org/jbpm`

[Spr11a] Springer Science+Business Media. Springer - International Publisher Science, Technology, Medicine (2011), `http://www.springer.com/`

[Spr11b] Spring Source. Spring Framework (2011), `http://www.springsource.org/`

[The11] The MathWorks. Simulink - Simulation and Model-Based Design (2011), `http://www.mathworks.co.uk/products/simulink/`

[Tig11] The MathWorks. Simulink - Simulation and Model-Based Design (2011), `http://www.mathworks.co.uk/products/simulink/`

[W3C07] The MathWorks. Simulink - Simulation and Model-Based Design (2011), `http://www.mathworks.co.uk/products/simulink/`

[W3C09] W3C. W3C Document Object Model (2009), `http://www.w3.org/DOM/`

[W3C11] W3C. W3C XML Schema (2011), `http://www.w3.org/XML/Schema`

Index

A

abstract form target 27, 31, 37, 143
Annotation Editor 56, 140, 147, 148, 213
application expert 45, 46, 77, 101
aspect orientation 85, 219
atomic proposition 50, 68, 164, 217

B

benchmark framework 97, 133
bootstrapping 8, 12, 69, 106, 191, 210
BPM 26, 35, 70, 117
business expert *see* application expert
Business Process Modeling *see* BPM

C

CASE 4, 19–20, 37, 53
code generation
  full 8, 24, 33–34, 41, 69, 107, 193, 202, 209
  template-based 28–31, 36, 96, 195, 210, 220
  visitor-based 29
code-to-model traceability 215–216, 218
Common SIBs 46, 78–79, 128, 129, 131, 143, 148, 157, 160, 174
compiler 12, 171
  cross-compiler 12
  self-compiling 12, 107
  self-hosting 12
Computer-Aided Software Engineering, *see* CASE
concrete form target 27
constraints 42, 63, 155, 219
  composite 158, 162
  global 42, 54, 63–66, 77, 158–168, 209
  local 42, 54, 55, 157–158, 209
context key 58, 80, 110, 113, 157, 217
ContextExpression 59, 128, 145, 151
ContextKey 59, 80, 113, 144, 145, 149, 151, 173
control flow pattern 61, 78, 115–117, 120, 146, 220
  fork-join 61, 116, 118, 173

D

domain 14
domain expert 45, 53, 55, 56, 67, 77, 211, 217
domain-specific language *see* DSL
Domain-Specific Modeling *see* DSM
DSL 15–17, 77
  external 16, 54
  internal 16, 29, 54
DSM 24, 34, 37, 71

E

Eclipse Modeling Framework *see* EMF
Ecore 18, 80, 177, 180–182, 188–190, 220
embedded systems 108, 140, 144

EMF 18, 76, 177, 179–180, 220
event handling 62, 123, 150
Executable UML 34
execution
  context 47, 58–60, 78, 110, 112, 122, 124, 157, 171
    flat 60, 61, 89, 94
    scope 59, 113, 128, 151
    stacked 58, 61, 114, 122, 128, 144, 150
  environment 57, 146
  equivalence 102, 108, 170–172
  footprint 171
  semantics 52, 57–58, 60, 76, 103, 114
Extreme Model-Driven Development, *see* XMDD
Extruder 57, 103–109, 121, 128, 141, 146, 147

F

FBB 158
flow charts 115, 117
ForkSIB 61
Formula Building Block *see* FBB
formula graph 77, 158, 161–168
FormulaBuilder 64, 77, 143, 155, 158–160, 162, 210
forward engineering 32

G

GEAR 63–69, 155, 158, 159, 213, 219
  macro 64, 158, 161–163
generation gap 33
generation logic 8, 27, 28, 30, 36, 95, 210, 220
Generative Programming *see* GP
generator developer 77
generator user 77
Genesys
  approach 4–9
  framework 75–77
Genesys Code Generator Generator 76, 95, 97, 99, 100, 125, 129, 174
GP 20–21, 38
GraphSIB 61, 122, 131

I

identifier generation 83–85, 119, 162, 163
incremental formalization 42, 63
inspector 44
interpreter 26, 57, 70, 106, 121–125, 134, 136, 146, 171, 210
IT expert 45, 47, 55, 57, 114, 145, 155

J

jABC 38, 43–46, 75
  plugins 48, 53–56, 62, 66, 69, 77, 96, 99, 152, 153, 176, 199, 209–211, 217
jABC EE 146, 212
jABC Execution Engine *see* jABC EE
jETI 56, 196
JoinSIB 61

K

Kripke Transition System 50, 64

L

language workbench 19, 25–26, 37, 53, 71
LocalChecker 55, 63, 69, 155, 157–158
loose programming 211, 219

M

macro 51, 52, 58, 61, 76, 115, 118, 122, 131, 165, 173, 175
MacroSIB 61, 89, 131
Maven 9, 77, 99, 185, 210
MD* 3
MDA 17, 22–23, 33, 69, 188
Meta-Object Facility *see* MOF
metalevel 17–19, 188–191
metametamodel 17
metamodel 7, 15, 53, 180, 220
model 3, 14, 17
  executable 5, 42, 70, 107, 210
  hierarchical 4, 41, 51, 52, 61, 69, 85, 162, 168, 209, 210

reflexive 18, 181
model branch 52, 123
model checking 7, 38, 41, 63, 155, 168, 209, 210, 213, 217
Model Driven Architecture *see* MDA
model parameter, 52, 104, 217
model-based development 3
model-driven development 3
modeling language *see* DSL (domain-specific language)
MOF 17, 180
multi-threading 61, 150, 173

O

One-Thing-Approach *see* OTA
OTA 41
output description 8, 27, 28, 30, 36, 95, 210, 220

P

product lines 8, 20, 85, 120, 210, 211, 218–219
property specification patterns 64, 158, 160, 162–168
protected regions 32, 179
Pure Generator 109

R

recursion 58, 115, 128, 131, 133, 173
reverse engineering 32, 211
round-trip engineering 8, 23, 32–33, 41, 101, 194, 196, 209
  full 33
  partial 32

S

self-generating generator 107
semantics 16
  static 7, 15, 54, 55
  translational 16
service adapter 48, 50, 57, 68, 70, 110–112, 114, 122, 123, 139, 141, 144, 150, 153, 183, 208
  descriptor 110, 111, 122
Service Independent Building Block *see* SIB
Service Logic Graph *see* SLG
SIB 43, 46–50
  branch, 47, 48, 51, 122
  control SIBs 54, 60–62, 108, 122, 123, 131, 133, 142, 150
  data types 48
  parameter 47, 48
  proxy SIB 53, 58, 162
  start SIB 57, 152, 162
SIB expert *see* IT expert
SIBGraphModel 53, 103, 108, 121, 129, 133, 135, 141
SLG 43, 50–54
  branch *see* model branch
  parameter *see* model parameter
  recursive 59
structured code 103, 114–121, 126, 128, 131, 134, 136, 142, 143
syntax
  abstract 7, 15, 53
  concrete 15, 53, 159, 171
synthesis 211, 219

T

T-diagram 13, 95, 106
taxonomy 44, 53, 56, 69, 209
template 28, 29, 83, 91, 93, 95, 131, 157, 210, 219, 220
  engine 28, 29, 36, 60, 79, 131, 157, 179, 183, 209, 220
  language 30, 38, 60, 95, 177
test
  case 156, 168–176, 209, 217
  script 170, 175
  suite 133, 168, 169, 173, 175–176, 217
testing 7, 76, 155, 168–176, 209, 217–218
  back-to-back 172, 209
  data-driven 172
ThreadSIB 61, 122
Tracer 57–63, 69, 70, 76, 94, 102–110, 114, 129, 131–133, 139, 141, 146, 152, 169, 171, 174, 176, 210, 216, 219
transformation 23
  engine 31

hard-coded 27, 28, 31
model-to-code 19
model-to-model 31, 87, 119, 125, 136, 142, 216, 220, 221
model-to-text 19, 31
rule-based 27, 31, 37, 160, 210
source-driven 27, 36, 96
target-driven 27, 31, 36, 96
type mapping 79–83, 103, 113–114, 119, 142, 144, 145, 149, 164

U

UML 17, 30, 33, 34, 37, 54, 69, 180, 193
profile 195
Unified Modeling Language *see* UML
user objects 53, 54, 97

V

V&V 7, 34–35, 69, 155–176, 209
variants 45, 85–87, 120, 125, 128, 130, 135, 208, 210, 218–219
variation point 85, 120, 218, 219
views 42, 218, 220
visitor 28, 29

W

weaving 87, 219

X

XMDD 38, 40–43, 75