

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

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

© Springer-Verlag Berlin Heidelberg 2013

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

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

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

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.

VIII Acknowledgments

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 DocuGenerator main model (topmost hierarchy level) ...	89
4.5	The Initialize DocuGenerator 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 <code>actionOccurrence</code> category	161
6.4	Constraints from the <code>actionOrder</code> 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