Lecture Notes in Computer Science 10001

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, Lancaster, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Zurich, Switzerland
John C. Mitchell
Stanford University, Stanford, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Dortmund, Germany
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbrücken, Germany

More information about this series at http://www.springer.com/series/7408

Wolfgang Ahrendt · Bernhard Beckert
Richard Bubel · Reiner Hähnle
Peter H. Schmitt · Mattias Ulbrich (Eds.)

Deductive Software Verification – The KeY Book

From Theory to Practice

Springer

Editors

Wolfgang Ahrendt
Chalmers University of Technology
Gothenburg
Sweden

Reiner Hähnle
Technische Universität Darmstadt
Darmstadt
Germany

Bernhard Beckert
Karlsruher Institut für Technologie (KIT)
Karlsruhe
Germany

Peter H. Schmitt
Karlsruher Institut für Technologie (KIT)
Karlsruhe
Germany

Richard Bubel
Technische Universität Darmstadt
Darmstadt
Germany

Mattias Ulbrich
Karlsruher Institut für Technologie (KIT)
Karlsruhe
Germany

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-49811-9 ISBN 978-3-319-49812-6 (eBook)
DOI 10.1007/978-3-319-49812-6

Library of Congress Control Number: 2016957483

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer International Publishing AG 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, 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.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Authors' Affiliations

Wolfgang Ahrendt (Chalmers University of Technology, Gothenburg, Sweden, email: ahrendt@chalmers.se)

Bernhard Beckert (Karlsruher Institut für Technologie (KIT), Germany, email: beckert@kit.edu)

Richard Bubel (Technische Universität Darmstadt, Germany, email: bubel@cs.tu-darmstadt.de)

Frank S. de Boer (Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, email: f.s.de.boer@cwi.nl)

Stijn de Gouw (Open University, Heerlen, The Netherlands, email: stijn.degouw@ou.nl)

Christoph Gladisch (Karlsruher Institut für Technologie (KIT), Germany, email: gladisch@ira.uka.de)

Daniel Grahl (Karlsruher Institut für Technologie (KIT), Germany, email: daniel.grahl@alumni.kit.edu)

Sarah Grebing (Karlsruher Institut für Technologie (KIT), Germany, email: grebing@ira.uka.de)

Simon Greiner (Karlsruher Institut für Technologie (KIT), Germany, email: simon.greiner@kit.edu)

Reiner Hähnle (Technische Universität Darmstadt, Germany, email: haehnle@cs.tu-darmstadt.de)

Martin Hentschel (Technische Universität Darmstadt, Germany, email: hentschel@cs.tu-darmstadt.de)

Mihai Herda (Karlsruher Institut für Technologie (KIT), Germany, email: herda@kit.edu)

Marieke Huisman (University of Twente, The Netherlands, email: m.huisman@utwente.nl)

Ran Ji (Technische Universität Darmstadt, Germany, email: rj82cnus@cs.cmu.edu)

Vladimir Klebanov (Karlsruher Institut für Technologie (KIT), Germany,
email: klebanov@kit.edu)

Wojciech Mostowski (Halmstad University, Sweden,
email: wojciech.mostowski@hh.se)

Jurriaan Rot (Radboud University, Nijmegen, The Netherlands,
email: jrot@cs.ru.nl)

Philipp Rümmer (Uppsala University, Sweden,
email: philipp.ruemmer@it.uu.se)

Christoph Scheben (Karlsruher Institut für Technologie (KIT), Germany,
email: scheben@ira.uka.de)

Peter H. Schmitt (Karlsruher Institut für Technologie (KIT), Germany,
email: pschmitt@ira.uka.de)

Mattias Ulbrich (Karlsruher Institut für Technologie (KIT), Germany,
email: ulbrich@kit.edu)

Nathan Wasser (Technische Universität Darmstadt, Germany,
email: nate@sharpmind.de)

Benjamin Weiß (Karlsruher Institut für Technologie (KIT), Germany,
email: benjamin.weiss@alumni.uni-karlsruhe.de)

Foreword

Program verification has a long and distinguished history in computer science. As early as 1949, Alan Turing, in a technical report titled *On Checking a Large Routine*, raised the question of how to verify computer programs. Ever since that time, this problem has been investigated by several researchers. Some of them earned the ACM Turing Award for their work on the subject.

With this concerted effort of several scientists in mind, one would think that the "problem" of software correctness has been "solved" by now, whatever the qualification "solved" is supposed to mean. Nothing is further from the truth.

In books on algorithms, for example, those covering the standard material on algorithms on data structures, the correctness of the presented algorithms is ignored or glossed over. At best a justification of the selected algorithms is given by presenting a mathematical argument without a rigorous explanation of why this argument can be applied to the program in question. The point is that a program is just a piece of text, while the presented mathematical argument refers to the program execution. The reasoning then tacitly presupposes an implementation that conforms to the informal description of the program meaning given in English. This subtle gap in reasoning is especially acute if one deals with recursive programs or programs involving dynamic data structures.

An alternative is to rely on a large body of literature on program verification and use one of the formal systems developed to provide rigorous correctness proofs using axioms and proof rules. Unfortunately, formal correctness proofs are very tedious and hence error prone. This raises a natural question: If we do not trust the correctness of a program, why should we trust its correctness proof?

A possible answer lies in relying on *mechanized verification* that provides a programming environment allowing us to check the correctness proof of a program in the sense that each step of the proof is mechanically verified. Such a programming environment, when properly built, allows us to treat the underlying proof system as a parameter, just as the program that is to be verified.

Mechanized verification has a distinguished history as well, and this is not the place to trace it. It suffices to say that the KeY project, the subject of this book, is possibly the most ambitious endeavor in this area. It started in 1998 and gradually evolved into, what the authors call, the *KeY framework*.

This framework goes beyond mechanized verification by also providing a means of program specification, a test case generation, a teaching tool for a number of courses on software engineering, and a debugging tool. The current book is a substantial revision and extension of the previous edition that takes into account this evolution of the KeY system. It systematically explains several facets of the KeY framework, starting with the theoretical underpinning and ending with a presentation of nontrivial case studies.

I would like to congratulate the authors not only for the outcome but also for their persistence and their vision. The current scientific climate aiming at maximizing your number of publications and your H-index does not bode well with long-term projects. It is refreshing to see that not everybody has yielded to it.

October 2016 Krzysztof R. Apt

Preface

Wer hat an der Uhr gedreht? Ist es wirklich schon so spät?—Every child growing up in the 1970s in Germany (like the author of this text) is familiar with these lines. They are from a theme song played during the closing credits of the German version of the *Pink Panther Show*. The ditty conveys the utter bafflement ("Who advanced the clock?") of the listener about 30 minutes of entertainment having passed in what seemed to be mere seconds. And it is exactly this feeling that we editors have now: What? Ten years since the first book about KeY [Beckert et al., 2007] was published? Impossible! But of course it is possible, and there are good reasons for a new book about KeY, this time simply called *The KeY Book*.

What Is New in The KeY Book?

In short: almost everything! This is not merely an overhaul of the previous book, but a completely different volume: only eight of 15 chapters from the previous edition are still around with a roughly similar function, while there are 11 completely new chapters. But even most of the chapters retained were rewritten from scratch. Only three chapters more or less kept their old structure. What happened?

First of all, there were some major technical developments in the KeY system that required coverage as well as changes in the theoretical foundations:

- With KeY 2.x we moved from a memory model with an implicit heap to one with explicit heaps, i.e., heaps have a type in our logic, can be quantified over, etc. As a consequence, it was possible to:
- Implement better support for reasoning about programs with heaps [Weiß, 2011, Schmitt et al., 2010], essentially with a variant of dynamic frames [Kassios, 2011].
- We dropped support for specification with the Object Constraint Language (OCL) and drastically improved support for the Java Modeling Language (JML) [Leavens et al., 2013].
- Rules and automation heuristics for a number of logical theories were added, including finite sequences, strings [Bubel et al., 2011], and bitvectors.
- Abstract interpretation was tightly integrated with logic-based symbolic execution [Bubel et al., 2009].

In addition, the functionality of the KeY was considerably extended. This concerns not merely the kind of analyses that are possible, but also usability.

- Functional verification is now only one of many analyses that can be performed with the KeY system. In addition, there is support for debugging and visualization [Hentschel et al., 2014a,b], test case generation [Engel and Hähnle, 2007, Gladisch, 2008], information flow analysis [Darvas et al., 2005, Grahl, 2015, Do et al., 2016], program transformation [Ji et al., 2013], and compilation [Ji, 2014].

- There are IDEs for KeY, including an Eclipse extension, that make it easy to keep track of proof obligations in larger projects [Hentschel et al., 2014c].
- A stripped down version of KeY, specifically developed for classroom exercises with Hoare logic, was provided [Hähnle and Bubel, 2008].

Finally, the increased maturity and coverage of the KeY system permitted us to include much more substantial case studies than in the first edition.

Inevitably, some of the research strands documented in the first edition did not reach the maturity or importance we hoped for and, therefore, were dropped. This is the case for specification patterns, specification in natural language, induction, and proof reuse. There is also no longer a dedicated chapter about Java integers: The relevant material is now, in much condensed form, part of the 'chapters on "First-Order Logic" and "Theories."'

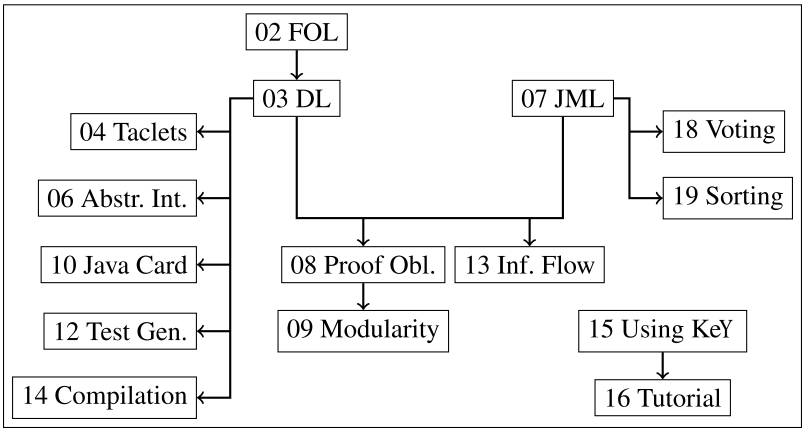
A number of topics that are actively researched have not yet reached sufficient maturity to be included: merging nodes in symbolic execution proof trees, KeY for Java bytecode, certification, runtime verification, regression verification, to name just a few. Also left out is a variant of the KeY system for the concurrent modeling language ABS called KeY-ABS, which allows one to prove complex properties about unbounded programs and data structures. We are excited about all these developments, but we feel that they have not yet reached the maturity to be documented in the KeY book. We refer the interested reader to the research articles available from the KeY website at www.key-project.org.

Also not covered in this book is KeYmaera, a formal verification tool for hybrid, cyber-physical systems developed in André Platzer's research group at CMU and that has KeY as an ancestor. It deserves a book in its own right, which in fact has been written [Platzer, 2010].

The Concept Behind This Book

Most books on foundations of formal specification and verification orient their presentation along traditional lines in logic. This results in a gap between the foundations of verification and its application that is too wide for most readers. The main presentation principles of the KeY book is something that has *not* been changed between the first book about KeY and this volume:

- The material is presented on an advanced level suitable for graduate (MSc level) courses, and, of course, active researchers with an interest in verification.
- The dependency graph on the chapters in the book is not deep, such that the reader does not have to read many chapters before the one (s)he is most interested in. Moreover, the dependencies are not all too strong. More advanced readers may not have to strictly follow the graph, and less advanced readers may decide to follow up prerequisites on demand. The graph shows that the chapters on First-Order Logic, the Java Modeling Language, and Using the KeY Prover are entirely self-contained. The same holds for each chapter not appearing in the following graph.



- The underlying verification paradigm is deductive verification in an expressive program logic.
- As a rule, the proofs of theoretical results are not contained here, but we give pointers on where to find them.
- The logic used for reasoning about programs is not a minimalist version suitable for theoretical investigations, but an industrial-strength version. The first-order part is equipped with a type system for modeling of object hierarchies, with underspecification, and with various built-in theories. The program logic covers full Java Card and substantial parts of Java. The main omissions are: generics (a transformation tool is available), floating-point types, threads, lambda expressions.
- Much emphasis is on specification, including the widely used JML. The generation of proof obligations from annotated source code is discussed at length.
- Two substantial case studies are included and presented in detail.

Nevertheless, we cannot and do not claim to have fully covered formal reasoning about (object-oriented) software in this book. One reason is that the choice of topics is dependent on our research agenda. As a consequence, important topics in formal verification, such as specification refinement or model checking, are out of our scope.

Typographic Conventions

We use a number of typesetting conventions to give the text a clearer structure. Occasionally, we felt that a historical remark, a digression, or a reference to material outside the scope of this book is required. In order to not interrupt the text flow we use gray boxes, such as the one on page 40, whenever this is the case.

In this book a considerable number of specification and programming languages are referred to and used for illustration. To avoid confusion we usually typeset multiline expressions from concrete languages in a special environment that is set apart from the main text with horizontal lines and that specifies the source language as, for example, on page 14.

Expressions from concrete languages are written in typewriter font with keywords highlighted in boldface, the exception being UML class and feature names. These are set in sans serif, unless class names correspond to Java types. Mathematical meta symbols are set in math font and the rule names of logical calculi in sans serif.

Companion Website

This book has its own website at www.key-project.org/thebook2, where additional material is provided: most importantly, the version of the KeY tool that was used to run all the examples in the book (except for Chaps. 10,19) including all source files, for example, programs and specifications (unless excluded for copyright reasons), various teaching materials such as slides and exercises, and the electronic versions of papers on KeY.

Acknowledgments We are very grateful to all researchers and students who contributed with their time and expertise to the KeY project. We would like to acknowledge in particular the editorial help of Dr. Daniel Grahl and Dr. Vladimir Klebanov during the production of this book.

Many current and former KeY project members made valuable contributions to the KeY project, even though they are not directly involved as chapter authors: Prof. Thomas Baar (HTW Berlin), Dr. Thorsten Bormer, Dr. Ádám Darvas, Dr. Crystal Din, Huy Qouc Do, Dr. Christian Engel, Dr. Tobias Gedell, Dr. Elmar Habermalz, Dr. Kristofer Johannisson, Michael Kirsten, Daniel Larsson, Prof. Wolfram Menzel (KIT; emeritus, co-founder of the KeY project), Gabriele Paganelli, Prof. Aarne Ranta (Gothenburg University), Dr. Andreas Roth, Prof. Ina Schaefer (TU Braunschweig), Dominic Scheurer, Prof. Steffen Schlager (HS Offenburg), Prof. Shmuel Tyszberowicz (The Academic College Tel Aviv-Yaffo), Dr. Bart van Delft, Dr. Isabel Tonin, Angela Wallenburg, and Dr. Dennis Walter.

Besides the current and former project members and the chapter authors of this book, many students helped with implementing the KeY system, to whom we extend our sincere thanks: Gustav Andersson, Dr. Marcus Baum, Hans-Joachim Daniels, Marco Drebing, Marius Hillenbrand, Eduard Kamburjan, Stefan Käsdorf, Dr. Bastian Katz, Uwe Keller, Stephan Könn, Achim Kuwertz, Denis Lohner, Moritz von Looz, Martin Möller, Dr. Ola Olsson, Jing Pan, Sonja Pieper, Prof. André Platzer (CMU Pittsburgh), Friedemann Rößler, Bettina Sasse, Dr. Ralf Sasse, Prof. Gabi Schmithüsen (Universität des Saarlandes), Max Schröder, Muhammad Ali Shah, Alex Sinner, Hubert Schmid, Holger Stenger, Kai Wallisch, Claus Wonnemann, and Zhan Zengrong.

The authors acknowledge support by the Deutsche Forschungsgemeinschaft (DFG) under projects “Program-level Specification and Deductive Verification of Security Properties” (DeduSec) and “ Fully Automatic Logic-Based Information Flow” within Priority Programme 1496 “Reliably Secure Software Systems – RS^3” and under project “Deductive Regression Verification for Evolving Object-Oriented Software” within Priority Programme 1593 “Design for Future: Managed Software Evolution,” as well as support by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) for project “Formale Informationsflussspezifikation und -analyse in komponentenbasierten Systemen” as part of Software Campus. We also acknowledge support by the European Commission (EC) under FP7 Integrated Project “Highly Adaptable & Trustworthy Software using Formal Methods” and STREP “Engineering Virtualized Services.”

The authors have done their job. The success of the result of their toil will now be judged by the readers. We hope they will find the text easily accessible, illuminating, stimulating and, yes, fun to read. What can be more fun than gaining insight into what was nebulous before or solving a problem at hand that defied a solution so far? We will not fail to admit that besides all the labor, writing this book was fun. Putting in words often helped us reach a better understanding of what we were writing about. Thus, it is not easy to say who profits more, the authors in writing the book or the readers from reading it. In the end this may be irrelevant, authors and readers together constitute the scientific community and together they advance the state of the art. It would be the greatest reward for us to see this happening and perhaps after another ten years, or even earlier, a return of the pink panther.

August 2016

Wolfgang Ahrendt
Bernhard Beckert
Richard Bubel
Reiner Hähnle
Peter H. Schmitt
Mattias Ulbrich

Contents

List of Figures . . . XXV

List of Tables . . . XXIX

List of Listings . . . XXXI

1 Quo Vadis Formal Verification? . . . 1
- 1.1 What KeY Is . . . 1
- 1.2 Challenges To Formal Verification . . . 2
- 1.3 Roles of Deductive Verification . . . 4
  - 1.3.1 Avoid the Need for Formal Specification . . . 4
  - 1.3.2 Restricted Target Language . . . 5
  - 1.3.3 Formal Verification in Teaching . . . 6
  - 1.3.4 Avoid the Need for Fully Specified Formal Semantics . . . 8
  - 1.3.5 Where Are We Now? . . . 8
- 1.4 The Architecture of KeY . . . 10
  - 1.4.1 Prover Core . . . 10
  - 1.4.2 Reasoning About Programs . . . 12
  - 1.4.3 Proof Obligations . . . 14
  - 1.4.4 The Frontends . . . 15
- 1.5 The Next Ten Years . . . 16
  - 1.5.1 Modular Architecture . . . 16
  - 1.5.2 Relational Properties Are Everywhere . . . 17
  - 1.5.3 Learning from Others . . . 18
  - 1.5.4 Code Reviews . . . 18
  - 1.5.5 Integration . . . 18

Part I Foundations

2 First-Order Logic . . . 23
2.1 Introduction . . . 23
2.2 Basic First-Order Logic . . . 23
2.2.1 Syntax . . . 23
2.2.2 Calculus . . . 27
2.2.3 Semantics . . . 31
2.3 Extended First-Order Logic . . . 35
2.3.1 Variable Binders . . . 36
2.3.2 Undefinedness . . . 37
2.4 First-Order Logic for Java . . . 37
2.4.1 Type Hierarchy and Signature . . . 38
2.4.2 Axioms for Integers . . . 39
2.4.3 Axioms for Heap . . . 40
2.4.4 Axioms for Location Sets . . . 43
2.4.5 Semantics . . . 44

3 Dynamic Logic for Java . . . 49
3.1 Introduction . . . 49
3.2 Syntax of JavaDL . . . 50
3.2.1 Type Hierarchies . . . 51
3.2.2 Signatures . . . 51
3.2.3 Syntax of JavaDL Program Fragments . . . 52
3.2.4 Syntax of JavaDL Terms and Formulas . . . 54
3.3 Semantics . . . 55
3.3.1 Kripke Structures . . . 55
3.3.2 Semantics of JavaDL Terms and Formulas . . . 56
3.4 Describing Transitions between States: Updates . . . 57
3.4.1 Syntax and Semantics of JavaDL Updates . . . 57
3.4.2 Update Simplification Rules . . . 59
3.5 The Calculus for JavaDL . . . 60
3.5.1 JavaDL Rule Schemata and First-Order Rules . . . 60
3.5.2 Nonprogram Rules for Modalities . . . 63
3.5.3 Soundness and Completeness of the Calculus . . . 63
3.5.4 Schema Variables for Program Constructs . . . 66
3.5.5 The Active Statement in a Modality . . . 67
3.5.6 The Essence of Symbolic Execution . . . 67
3.5.7 Components of the Calculus . . . 68
3.6 Rules for Symbolic Execution of Java Programs . . . 69
3.6.1 The Basic Assignment Rule . . . 69
3.6.2 Rules for Handling General Assignments . . . 70
3.6.3 Rules for Conditionals . . . 75
3.6.4 Unwinding Loops . . . 76
3.6.5 Replacing Method Calls by their Implementation . . . 77
3.6.6 Instance Creation and Initialization . . . 87
3.6.7 Handling Abrupt Termination . . . 93

3.7 Abstraction and Modularization Rules 97
3.7.1 Replacing Method Calls by Specifications: Contracts . 98
3.7.2 Reasoning about Unbounded Loops: Loop Invariants . 101

4 Proof Search with Taclets 107
4.1 Introduction 107
4.1.1 Purpose and Organization of this Chapter 108
4.2 A Taclet Tutorial 109
4.2.1 A Basic Theory of Lists 109
4.2.2 The List Theory in Concrete Syntax 111
4.2.3 Definitional Extension of the List Theory 114
4.2.4 Derivation of Lemmas 117
4.3 A Reference Manual of Taclets 121
4.3.1 The Taclet Language 121
4.3.2 Schema Variables 130
4.3.3 Application of Taclets 136
4.4 Reflection and Reasoning about Soundness of Taclets 138
4.4.1 Soundness in Sequent Calculi 139
4.4.2 Meaning Formulas of Sequent Taclets 140
4.4.3 Meaning Formulas for Rewriting Taclets 142
4.4.4 Elimination of Schema Variables 143

5 Theories 149
5.1 Introduction 149
5.2 Finite Sequences 149
5.3 Strings 156
5.3.1 Sequences of Characters 156
5.3.2 Regular Expressions for Sequences 158
5.3.3 Relating Java String Objects to Sequences of Characters 159
5.3.4 String Literals and the String Pool 160
5.3.5 Specification of the Java String API 161
5.4 Integers 161
5.4.1 Core Integer Theory 162
5.4.2 Variable Binding Integer Operations 163
5.4.3 Symbolic Execution of Integer Expressions in Programs 164

6 Abstract Interpretation 167
6.1 Introduction 167
6.2 Integrating Abstract Interpretation 168
6.2.1 Abstract Domains 168
6.2.2 Abstractions for Integers 170
6.2.3 Abstracting States 170
6.3 Loop Invariant Generation 171
6.4 Abstract Domains for Heaps 174
6.5 Abstract Domains for Objects 178
6.5.1 Null/Not-null Abstract Domain 179

6.5.2 Length Abstract Domain 179
6.6 Extensions 184
6.6.1 Abstractions for Arrays 184
6.6.2 Loop Invariant Rule with Value and Array Abstraction . 184
6.6.3 Computation of the Abstract Update and Invariants . . 186
6.6.4 Symbolic Pivots 187
6.7 Conclusions 189

Part II Specification and Verification

7 Formal Specification with the Java Modeling Language 193
7.1 Introduction to Method Contracts 196
7.1.1 Clauses of a Contract 196
7.1.2 Defensive Versus Offensive Method Implementations . 199
7.1.3 Specifications and Implementations 200
7.2 Expressions 201
7.2.1 Quantified Boolean Expressions 201
7.2.2 Numerical Comprehensions 203
7.2.3 Evaluation in the Prestate 204
7.3 Method Contracts in Detail 205
7.3.1 Visibility of Specifications 205
7.3.2 Specification Cases 206
7.3.3 Semantics of Normal Behavior Specification Cases . . 208
7.3.4 Specifications for Constructors 209
7.3.5 Notions of Purity 209
7.4 Class Level Specifications 210
7.4.1 Invariants 211
7.4.2 Initially Clauses 216
7.4.3 History Constraints 217
7.4.4 Initially Clauses and History Constraints: Static vs. Instance 218
7.4.5 Inheritance of Specifications 218
7.5 Nonnull Versus Nullable Object References 220
7.6 Exceptional Behavior 222
7.7 Specification-Only Class Members 225
7.7.1 Model Fields and Model Methods 225
7.7.2 Ghost Variables 229
7.7.3 Ghost Variables Versus Model Fields 229
7.8 Integer Semantics 230
7.9 Auxiliary Specification for Verification 233
7.9.1 Framing 233
7.9.2 Loop Invariants 234
7.9.3 Assertions and Block Contracts 238

7.10 Conclusion 239
7.10.1 Tool Support for JML 239
7.10.2 Comparison to Other Program Annotation Languages . 240

8 From Specification to Proof Obligations 243
8.1 Formal Semantics of JML Expressions 244
8.1.1 Types in JML 245
8.1.2 Translating JML Expressions to JavaDL 246
8.1.3 Abstract Data Types in JML 252
8.1.4 Well-Definedness of Expressions 253
8.2 From JML Contract Annotations to JavaDL Contracts 254
8.2.1 Normalizing JML Contracts 255
8.2.2 Constructor Contracts 264
8.2.3 Model Methods and Model Fields 265
8.2.4 JavaDL Contracts 268
8.2.5 Loop Specifications 271
8.3 Proof Obligations for JavaDL Contracts 272
8.3.1 Proof Obligations for Functional Correctness 272
8.3.2 Dependency Proof Obligations 278
8.3.3 Well-Definedness Proof Obligations 279

9 Modular Specification and Verification 289
9.1 Modular Verification with Contracts 291
9.1.1 Historical and Conceptual Background 291
9.1.2 Example: Implementing a List 294
9.1.3 Modular Program Correctness 296
9.1.4 Verification of Recursive Methods 298
9.2 Abstract Specification 300
9.2.1 Model Fields 302
9.2.2 Model Methods 311
9.3 The Frame Problem 319
9.3.1 Motivation 320
9.3.2 Dynamic Frames 322
9.3.3 Proof Obligations for Dynamic Frames Specifications . 324
9.3.4 Example Specification with Dynamic Frames 325
9.4 Calculus Rules for Modular Reasoning 328
9.4.1 Anonymizing Updates 329
9.4.2 An Improved Loop Invariant Rule 330
9.4.3 A Rule for Method Contracts 336
9.4.4 A Rule for Dependency Contracts 339
9.4.5 Rules for Class Invariants 341
9.5 Verifying the List Example 343
9.6 Related Methodologies for Modular Verification 347
9.7 Conclusion 351

10 Verifying Java Card Programs . . . 353
10.1 Introduction . . . 353
10.2 Java Card Technology . . . 354
10.3 Java Card Transactions on Explicit Heaps . . . 357
10.3.1 Basic Transaction Roll-Back . . . 358
10.3.2 Transaction Marking and Balancing . . . 359
10.3.3 Object Creation and Deletion . . . 360
10.3.4 Persistent and Transient Arrays . . . 361
10.3.5 Nonatomic Updates . . . 363
10.4 Taclets for the New Rules . . . 364
10.5 Modular Reasoning with Multiple Heaps . . . 367
10.6 Java Card Verification Samples . . . 368
10.6.1 Conditional Balance Updating . . . 369
10.6.2 Reference Implementation of a Library Method . . . 370
10.6.3 Transaction Resistant PIN Try Counter . . . 374
10.7 Summary and Discussion . . . 376
10.7.1 Related Work . . . 377
10.7.2 On-going and Future Research . . . 377
10.7.3 Multiple Heaps for Concurrent Reasoning . . . 377

Part III From Verification to Analysis

11 Debugging and Visualization . . . 383
11.1 Introduction . . . 383
11.2 Symbolic Execution . . . 385
11.3 Symbolic Execution Debugger . . . 390
11.3.1 Installation . . . 390
11.3.2 Basic Usage . . . 391
11.3.3 Debugging with Symbolic Execution Trees . . . 395
11.3.4 Debugging with Memory Layouts . . . 397
11.3.5 Help Program and Specification Understanding . . . 398
11.3.6 Debugging Meets Verification . . . 399
11.3.7 Architecture . . . 401
11.4 A Symbolic Execution Engine based on KeY . . . 403
11.4.1 Symbolic Execution Tree Generation . . . 403
11.4.2 Branch and Path Conditions . . . 407
11.4.3 Hiding the Execution of Query Methods . . . 408
11.4.4 Symbolic Call Stack . . . 409
11.4.5 Method Return Values . . . 409
11.4.6 Current State . . . 410
11.4.7 Controlled Execution . . . 411
11.4.8 Memory Layouts . . . 411
11.5 Conclusion And Future Work . . . 412

12 Proof-based Test Case Generation 415
12.1 Introduction 415
12.2 A Quick Tutorial 416
12.2.1 Setup 417
12.2.2 Usage 417
12.2.3 Options and Settings 419
12.3 Test Cases, Test Suites, and Test Criteria 421
12.4 Application Scenarios and Variations of the Test Generator . . 426
12.4.1 KeYTestGen for Test Case Generation 426
12.4.2 KeYTestGen for Formal Verification 427
12.5 Architecture of KeYTestGen 428
12.6 Proof-based Constraint Construction for Test Input Data 430
12.6.1 Symbolic Execution for Test Constraint Generation . . 430
12.6.2 Implicit Case Distinctions 431
12.6.3 Infeasible Path Filtering 432
12.6.4 Using Loop Unwinding and Method Inlining 433
12.6.5 Using Loop Invariants and Method Contracts 434
12.7 From Constraints to Test Input Data 437
12.7.1 The Type System 439
12.7.2 Preserving the Semantics of Interpreted Functions . . . 440
12.7.3 Preventing Integer Overflows 441
12.7.4 Model Extraction 442
12.8 Specification-based Test Oracle Generation 442
12.8.1 Generating a Test Oracle from the Postcondition 442
12.8.2 Using a Runtime Assertion Checker 445
12.9 Synthesizing Executable Test Cases 445
12.10 Perspectives and Related Work 448
12.11 Summary and Conclusion 450

13 Information Flow Analysis 453
13.1 Introduction 453
13.2 Specification and the Attacker Model 455
13.3 Formal Definition of Secure Information Flow 457
13.4 Specifying Information Flow in JML 458
13.5 Information Flow Verification with KeY 460
13.5.1 Efficient Double Symbolic Execution 461
13.5.2 Using Efficient Double Symbolic Execution in KeY . . 469
13.6 Summary and Conclusion 470

14 Program Transformation and Compilation 473
14.1 Interleaving Symbolic Execution and Partial Evaluation 474
14.1.1 General Idea 474
14.1.2 The Program Specialization Operator 478
14.1.3 Specific Specialization Actions 479
14.1.4 Example 481
14.2 Verified Correct Compilation 482

14.3 Implementation and Evaluation 491
14.4 Conclusion 492

Part IV The KeY System in Action

15 Using the KeY Prover 495
15.1 Introduction 495
15.2 Exploring KeY Artifacts and Prover Simultaneously 498
15.2.1 Exploring Basic Notions And Usage 499
15.2.2 Exploring Terms, Quantification, and Instantiation . . . 511
15.2.3 Exploring Programs in Formulas 515
15.3 Understanding Proof Situations 532
15.4 Further Features 537
15.5 What Next? 539

16 Formal Verification with KeY: A Tutorial 541
16.1 Introduction 541
16.2 A Program without Loops 543
16.3 A Brief Primer on Loop Invariants 546
16.3.1 Introduction 546
16.3.2 Why Are Loop Invariants Needed? 547
16.3.3 What Is A Loop Invariant? 547
16.3.4 Goal-Oriented Derivation of Loop Invariants 549
16.3.5 Generalization 550
16.3.6 Recovering the Context 551
16.3.7 Proving Termination 553
16.3.8 A More Complex Example 555
16.3.9 Invariants: Concluding Remarks 557
16.4 A Program with Loops 558
16.5 Data Type Properties of Programs 562
16.6 KeY and Eclipse 565
16.6.1 Installation 566
16.6.2 Proof Management with KeY Projects 566
16.6.3 Proof Results via Marker 567
16.6.4 The Overall Verification Status 568

17 KeY-Hoare 571
17.1 Introduction 571
17.2 The Programming Language 572
17.3 Background 573
17.3.1 First-Order Logic 573
17.3.2 Hoare Calculus 574
17.4 Hoare Logic with Updates 575
17.4.1 State Updates 576
17.4.2 Hoare Triples with Update 577

17.4.3 Hoare Style Calculus with Updates 577
17.4.4 Rules for Updates 579
17.5 Using KeY-Hoare 580
17.6 Variants of the Hoare Logic with Updates 582
17.6.1 Total Correctness 582
17.6.2 Worst-Case Execution Time 582
17.7 A Brief Reference Manual for KeY-Hoare 584
17.7.1 Installation 584
17.7.2 Formula Syntax 585
17.7.3 Input File Format 585
17.7.4 Loading and Saving Problems and Proofs 587
17.7.5 Proving 588
17.7.6 Automation 589

Part V Case Studies

18 Verification of an Electronic Voting System 593
18.1 Electronic Voting 593
18.2 Overview 594
18.2.1 Verification of Cryptographic Software 595
18.2.2 Verification Approach 595
18.2.3 System Description 596
18.3 Specification and Verification 599
18.3.1 Specification 599
18.3.2 Verification 603
18.4 Discussion 605
18.4.1 A Hybrid Approach to Information Flow Analysis . . . 605
18.4.2 Related Work 606
18.4.3 Conclusion 607

19 Verification of Counting Sort and Radix Sort 609
19.1 Counting Sort and Radix Sort Implementation 609
19.2 High-Level Correctness Proof 612
19.2.1 General Auxiliary Functions 613
19.2.2 Counting Sort Proof 614
19.2.3 Radix Sort Proof 615
19.3 Experience Report 617

Part VI Appendices

A Java Modeling Language Reference 621
A.1 JML Syntax 621
A.2 JML Expression Semantics 625
A.3 JML Expression Well-Definedness 628

B KeY File Reference 631
B.1 Predefined Operators in JavaDL 631
B.1.1 Arithmetic Functions 631
B.1.2 Arithmetic Functions with Modulo Semantics 632
B.1.3 Predicates for Arithmetics and Equality 634
B.1.4 Integer Semantics Dependent Functions 634
B.1.5 Integer Semantics Dependent Predicate Symbols 636
B.1.6 Heap Related Function and Predicate Symbols 636
B.1.7 Location Sets Related Function and Predicate Symbols 637
B.1.8 Finite Sequence Related Function and Predicate Symbols 638
B.1.9 Map Related Function and Predicate Symbols 639
B.2 The KeY Syntax 640
B.2.1 Notation, Keywords, Identifiers, Numbers, Strings . . . 641
B.2.2 Terms and Formulas 643
B.2.3 Rule Files 651
B.2.4 User Problem and Proof Files 659
B.2.5 Schematic Java Syntax 662

References 667

Index 691

List of Figures

1.1 Architecture of the KeY tool set 11
1.2 GUI of the KeY system with a loaded proof obligation 15

2.1 First-order rules for the logic FOL 28
2.2 Equality rules for the logic FOL 30
2.3 Mandatory JFOL type hierarchy 38
2.4 Mandatory JFOL vocabulary 39
2.5 Integer axioms and rules 39
2.6 Rules for the theory of arrays. 41
2.7 Rules for the predicate *wellFormed* 42
2.8 Axioms for functions related to Java types 43
2.9 Rules for data type *LocSet* 44
2.10 Semantics on type domains 45
2.11 Semantics for the mandatory JFOL vocabulary 46
2.12 Semantics for the predicate *wellFormed* 47

3.1 An example program with method overriding 82
3.2 Initialization part in a schematic class 88
3.3 Mapping of a class declaration to initialization schema 89
3.4 Implicit method `<prepare>()` 91
3.5 Example for constructor normal form 92
3.6 Building the constructor normal form 93
3.7 Method contract rule 100
3.8 Basic loop invariant rule 102
3.9 Invariant rule for loops with side effects in the loop condition 104
3.10 Invariant rule for loops with abrupt termination 106

4.1 Vocabulary and induction axiom for a simplified theory of lists 111
4.2 Inductive example proof 114
4.3 Vocabulary and axioms for defined list functions 115

4.4 Derived taclets for the list theory . . . 118
4.5 The taclet syntax . . . 122

5.1 The vocabulary of finite sequences . . . 150
5.2 Axioms of the core theory of finite sequences . . . 150
5.3 Definition of *seqDepth* . . . 152
5.4 Definition for noncore vocabulary in mathematical notation . 153
5.5 Some derived rules for finite sequences . . . 154
5.6 Some derived rules for permutations . . . 155
5.7 Axioms for *clIndexOfChar* and *clContains* . . . 158

6.1 Example abstract domain for vehicles . . . 168
6.2 Sign Domain: An abstract domain for integers . . . 170
6.3 Instantiation of the heap abstract domain for fields . . . 177
6.4 Instantiation of the heap abstract domain for arrays . . . 177
6.5 Abstract domain for objects . . . 179
6.6 Length-based abstract domain for objects . . . 180
6.7 Family of abstract domains for object based on exact type . . 181
6.8 Galois connection for object abstract domain based on types . 181
6.9 Example instantiation of abstract domain for objects . . . 182
6.10 Galois connection for combining object abstract domains . . . 183
6.11 Combined object abstract domain . . . 183

8.1 Flattening a nested JML specification . . . 256
8.2 The type hierarchy of exceptions in Java . . . 261

9.1 A list interface and its implementations . . . 294
9.2 Listings of `Cell`/`Recell` example . . . 315
9.3 Listings of `Cell`/`Recell` example annotated with model methods 317
9.4 Proof structure for `Client` method `m` . . . 345

11.1 Symbolic execution tree of method `min` . . . 386
11.2 Symbolic execution tree of method `sum` . . . 387
11.3 Symbolic execution tree of method `sum` using a loop invariant 388
11.4 Symbolic execution tree of method `run` . . . 389
11.5 Symbolic execution tree of method `average` using contracts . 391
11.6 SED: Interactive symbolic execution . . . 393
11.7 SED: Collapsed frame . . . 394
11.8 SED: Possible memory layouts of a symbolic state . . . 395
11.9 Symbolic execution tree of mergesort . . . 396
11.10 Initial symbolic object diagram: AVL rotate left . . . 398
11.11 Current symbolic object diagram: AVL rotate left . . . 399
11.12 Symbolic execution tree of method `indexOf` . . . 401
11.13 Architecture of the Symbolic Execution Debugger (SED) . . . 402

12.1 The Test Suite Generation window 419
12.2 TestGen macro and Finish symbolic execution macro 420
12.3 Use-cases of KeYTestGen 427
12.4 Components and workflow of KeYTestGen 428
12.5 SMT solver options 438
12.6 OpenJML output for array copy 446

13.1 Program declassifying the sum of an array 459
13.2 Program with multiple information flow contracts 459
13.3 Example of a secure program 461
13.4 Control flow graphs: Original vs. self-composition 462
13.5 Reducing verification overhead by compositional reasoning . 463
13.6 Proof tree of information flow example 468

14.1 Simple control circuit: Source and control flow graph 475
14.2 Symbolic execution tree of the control circuit program 475
14.3 Partial evaluation of a simple control circuit program 476
14.4 Symbolic execution with interleaved partial evaluation 477
14.5 Type hierarchy for the GUI example 482
14.6 Work flow of synthesizing loop 487
14.7 Symbolic execution tree until conditional 488
14.8 Symbolic execution tree of then branch 488
14.9 Symbolic execution tree of else branch 489
14.10 Source code of the `Power` example 492

15.1 Verification process using the KeY system 496

16.1 The KeY verification workflow 542
16.2 Mapping loop execution into a well-founded order 554
16.3 Condensed finished proof tree 561
16.4 Strategy macros 564
16.5 Sequent after application of rule `seqPermFromSwap` 565
16.6 KeY project with example content 567
16.7 Closed and open proof marker 568
16.8 Recursive specification marker 569
16.9 The overall verification status 569

17.1 Rules of standard Hoare calculus. 575
17.2 Rules of Hoare calculus with updates. 578
17.3 Rewrite rules for update computation. 579
17.4 Loop invariant rule for total correctness. 582
17.5 Loop invariant rule for execution time reasoning 583
17.6 Worst-case execution time specification 583
17.7 Input file for the `searchMax` example. 586
17.8 Input file grammar 587
17.9 Screen shot of KeY-Hoare system 588

18.1 UML class diagram of the e-voting system 597
18.2 The overall protocol of the e-voting system 598

19.1 Iterations of the last loop in Counting sort 611
19.2 Successive iterations of the Radix sort loop 611

List of Tables

3.1 Simplification rules for updates 59
3.2 Names of schema variables and their kinds 66
3.3 Components of rule assignmentSaveLocation for fields 72
3.4 Components of rule assignmentSaveLocation for arrays . . . 72
3.5 Implicit object repository and status fields 87
3.6 Implicit methods for object creation and initialization 88

4.1 Matrix of different taclet modes and different find patterns . . 126
4.2 Most important rule sets in KeY 131
4.3 Kinds of schema variables in the context of a type hierarchy . 132
4.4 Examples of schematic expressions and their instantiations . . 135

5.1 The additional vocabulary for the theory of Strings 157
5.2 Pattern expressions 158
5.3 Core and extension functions for the `int` data type 162
5.4 Functions for Java arithmetics 162

8.1 Defined operations on the `\seq` in JML 253
8.2 Default values for absent clauses and operators 260
8.3 Clauses in JML loop specifications 271

11.1 Symbolic execution tree nodes 392
11.2 Views of perspective Symbolic Debug 393
11.3 Classification of proof nodes for SE trees 405

12.1 MC/DC coverage example 425

A.1 Additional class members in JML 622
A.2 Contract grammar in JML 622
A.3 Modifiers in JML 623
A.4 JML annotation grammar 623

A.5 JML expression grammar 624
A.6 Translation of Boolean JML operators 625
A.7 Translation of special Boolean JML operators 625
A.8 Predicates on location sets 625
A.9 Translation of Java integer expressions 626
A.10 Translation of comprehension expressions 627
A.11 Restrictions on JavaDL types to match JML types 627
A.12 Translation of special JML operators of arbitrary type 628
A.13 Reference expressions 628
A.14 JML location set expressions 629
A.15 JML sequence expressions 629
A.16 Definition of the well-definedness operator 630

List of Listings

6.1 Type declarations 182
6.2 Example program for array abstraction 185

7.1 First JML example specification 198
7.2 Class `CStudent` with `spec_public` variables 206
7.3 Specifying limited size integer set 207
7.4 Interface `Student` with class level specifications 212
7.5 Interface `CallBack` 213
7.6 Invariant broken during callback 214
7.7 Checking history constraints 218
7.8 Class `Average` 223
7.9 Class `Average2` 224
7.10 Relational represents clause 227
7.11 Interface `Student` with model fields and an implementation. 228
7.12 Using a ghost field to track recursion depth 230
7.13 Full specification of `Student` with assignable clauses 235
7.14 Loop invariant example to search an element in an array 236
7.15 Loop invariant example to compute the sum of an array 237
7.16 Usage of a block contract to swap two values 238

8.1 JML functional method contract specification case template 256
8.2 JML method dependency contract specification case template . . . 256
8.3 Example of a JML method contract prior to desugaring 264
8.4 Example of desugared JML method contract 265
8.5 The two semantics of **`assignable`** clauses 276

9.1 Linked list implementation of the `List` interface 295
9.2 Nonempty lists is a behavioral subtype to lists 296
9.3 Java interface `List` specified using pure methods 301
9.4 Interface specification using a model field 304
9.5 `ArrayList` implementation of the `List` interface 305

9.6 Cell/Recell annotated with footprint specifications 318
9.7 Client code using two instances of the List interface 321
9.8 Client specification exposing implementation by using framing . . 321
9.9 List specification using pure methods and dynamic frames 326
9.10 List client specified with dynamic frames 327
9.11 ArrayList implementation of the List interface 329

11.1 Minimum of two integers 385
11.2 Sum of all array elements 387
11.3 Wrong and weak loop invariants 388
11.4 Method call with inheritance 389
11.5 Method contract of method sum 390
11.6 Average of all array elements 390
11.7 Defective part of a mergesort implementation 397
11.8 Exception thrown by mergesort 398
11.9 A defective and only partially specified implementation 400

12.1 Method arrCopy violates its contract 416
12.2 First example where branch coverage is difficult to achieve 435
12.3 Second example where branch coverage is difficult to achieve . . . 435
12.4 Generated test oracle for array copy 443
12.5 Generated test case for array copy 447

13.1 Example for an explicit leak 453
13.2 Example for an implicit leak 454

14.1 Generated program: Simple example 490
14.2 Generated specialized program 491

16.1 First example: Postincrement 543
16.2 Second example: Sorting an array 559
16.3 Third example: Permutations 563

18.1 Implementation and functional contract for method main 599
18.2 Information flow contract for the main() method 600
18.3 Loop invariant for the loop in method main 601
18.4 Information flow contract of publishResult. 602
18.5 Declaration of the interface Environment. 603
18.6 Contract of method onSendBallot 604

19.1 Counting sort 610
19.2 Radix sort 611
19.3 Generic sorting contract 612
19.4 Contract specifying stability 613