

Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

For further volumes:

www.springer.com/series/7592

José Bacelar Almeida • Maria João Frade •
Jorge Sousa Pinto • Simão Melo de Sousa

Rigorous Software Development

An Introduction to
Program Verification

Dr. José Bacelar Almeida
Depto. Informática
Universidade do Minho
Campus de Gualtar
Braga 4710-057
Portugal
jba@di.uminho.pt

Dr. Jorge Sousa Pinto
Depto. Informática
Universidade do Minho
Campus de Gualtar
Braga 4710-057
Portugal
jsp@di.uminho.pt

Dr. Maria João Frade
Depto. Informática
Universidade do Minho
Campus de Gualtar
Braga 4710-057
Portugal
mjf@di.uminho.pt

Dr. Simão Melo de Sousa
Depto. Informática
Universidade Beira Interior
rua Marques d'Avila e Bolama
Covilhã 6201-001
Portugal
desousa@di.ubi.pt

Series editor
Ian Mackie

Advisory board
Samson Abramsky, University of Oxford, Oxford, UK
Chris Hankin, Imperial College London, London, UK
Dexter Kozen, Cornell University, Ithaca, USA
Andrew Pitts, University of Cambridge, Cambridge, UK
Hanne Riis Nielson, Technical University of Denmark, Lyngby, Denmark
Steven Skiena, Stony Brook University, Stony Brooks, USA
Iain Stewart, University of Durham, Durham, UK

ISSN 1863-7310

ISBN 978-0-85729-017-5

e-ISBN 978-0-85729-018-2

DOI 10.1007/978-0-85729-018-2

Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

© Springer-Verlag London Limited 2011

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

The use of 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 laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Printed on acid-free paper

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

Preface

It has become common sense that problems in software may have catastrophic consequences, ranging from economic to human safety aspects. This has led to the development of software norms that prescribe the use of particular development methods. These norms advise the use of *validation* techniques, and at the highest levels of certification, of *mathematical* validation.

Program verification is the area of computer science that studies mathematical methods for checking that a program conforms to its specification. It is part of the broader area of *formal methods*, which groups together very heterogeneous rigorous approaches to systems and software development. Program verification is concerned with properties of code, which can be studied in more than one way. From methods based on logic, in particular the combined use of a *program logic* and *first-order theories*, to other approaches like *software model checking*; *abstract interpretation*; and *symbolic execution*.

This book is a self-contained introduction to program verification using logic-based methods, assuming only basic knowledge of standard mathematical concepts that should be familiar to any computer science student. It includes a self-contained introduction to propositional logic and first-order reasoning with theories, followed by a study of program verification that combines theoretical and practical aspects—from a program logic to the use of a realistic tool for the verification of C programs, through the generation of verification conditions and the treatment of errors.

More specifically, we start with an overview of propositional (Chap. 3) and first-order (Chap. 4) logic, and then go on (Chap. 5) to establish a setting for the verification of sequential programs, building on the axiomatic semantics (based on Hoare logic) of a generic *While* language, whose behaviour is specified using *preconditions* and *postconditions*.

It is then shown how a set of first-order proof obligations, usually known as *verification conditions*, can be mechanically generated from a program and a specification that the program is required to satisfy (Chap. 6). Concrete programming languages can be obtained by instantiating the language of program expressions, which we illustrate with a language of integer expressions and then a language of integer-type arrays.

This setting is then adapted to cover the treatment of *safety properties* of programs, by explicitly incorporating in the semantics the possibility of runtime errors occurring. The set of generated verification conditions is extended to guard against that possibility (Chap. 7). This is illustrated in the concrete languages with arithmetic errors and out-of-bounds array accesses.

Finally the setting is extended to allow for the specification and verification of programs consisting of mutually-recursive procedures, based on annotations included in the code, usually known as *contracts* (Chap. 8). The idea here is that each procedure has its own public specification, called a contract in this context, and when reasoning individually about the correctness of an individual procedure one can assume that all procedures are correct with respect to their respective specifications.

The last part of the book illustrates the specification (Chap. 9) and verification (Chap. 10) of programs of a real-world programming language. We have chosen to use the ACSL specification language for C programs, and the Frama-C/Jessie tool for their verification. It is important to understand that these are being actively developed as this book is written, but we do believe that the core of the specification language and verification tools will remain unchanged.

Our emphasis is on the integrated presentation, and on making explicit the bits of the story that may be harder to grasp. Program verification already has a long history but remains a very active research field, and this book contains some material that as far as we can tell can only be found in research papers. The approach followed has logic at its core, and the theories that support the reasoning (which may include user-supplied parts) are always explicitly referred. The generation of verification conditions is performed by an algorithm that is synthesized from Hoare logic in a step-by-step fashion. The important topic of *specification adaptation* is also covered, since it is essential to the treatment of procedures. Finally, we develop a general framework for the verification of contract-based mutually-recursive procedures, which is an important step towards understanding the use of modern verification tools based on contracts.

The book will prepare readers to use verification methods in practice, which we find cannot be done correctly with only a superficial understanding of what is going on. Although verification methods tend to be progressively easier to use, software engineers can greatly benefit from an understanding of why the verification methods and tools are sound, as well as what their limitations are, and how they are implemented. Readers will also be capable of constructing their own verification platforms for specific languages—but it must be said at this point that one of the most challenging aspects, the treatment of heap-based dynamic data structures based on a memory model—is completely left out.

As mentioned before, program verification belongs to what are usually described as *formal methods* for the development of software. Chapter 1 motivates the importance of program verification, and explains how formal approaches to software development and validation are more and more important in the software industry. Chapter 2 offers an overview of formal methods that will hopefully give the reader a more general context for understanding the program verification techniques covered in the rest of the book.

Notes to the Instructor The book assumes knowledge of the basic mathematical concepts (like sets, functions, and relations) that are usually taught in introductory courses on mathematics for computer science students. No knowledge of logical concepts is assumed.

The accompanying website for this book is <http://www.di.uminho.pt/rsd-book>, where teaching material, solutions to selected exercises, source code, and links to useful online resources can be found.

Acknowledgments The authors are indebted to their students and colleagues at the departments of Informatics of the universities of Minho and Beira Interior, who have provided feedback and comments on drafts of several chapters. The idea of writing this book came up when the third author was on leave at the Department of Computer Science of the University of Porto (Faculty of Science), which kindly provided a welcoming environment for writing the first draft chapters.

Contents

- 1 Introduction 1**
 - 1.1 A Formal Approach to Software Engineering 1
 - 1.1.1 Test and Simulation-Based Reliability 2
 - 1.1.2 An Alternative Approach: Formal Methods 3
 - 1.1.3 Requirements: Functional, Security, and Safety 4
 - 1.2 Formal Methods and Industrial Norms 5
 - 1.3 From Classic Software Engineering to Formal Software Engineering 7
 - 1.4 This Book 11
 - References 12
- 2 An Overview of Formal Methods Tools and Techniques 15**
 - 2.1 The Central Problem 16
 - 2.1.1 Some Existing Formal Methods Taxonomies 16
 - 2.1.2 This Overview 17
 - 2.2 Specifying and Analysing 17
 - 2.2.1 Model-Based Specification 18
 - 2.2.2 Algebraic Specification 24
 - 2.2.3 Declarative Modelling 25
 - 2.3 Specifying and Proving 26
 - 2.3.1 Logic in a Nutshell 27
 - 2.3.2 Proof Tools 30
 - 2.3.3 Model Checking 31
 - 2.3.4 Program Logics and Program Annotation 32
 - 2.4 Specifying and Deriving 33
 - 2.4.1 Refinement 34
 - 2.4.2 Extraction 36
 - 2.4.3 Execution 36
 - 2.5 Specifying and Transforming 36
 - 2.6 Conclusions 37
 - 2.6.1 Are Formal Methods Tools Ready for Industry? 38
 - 2.6.2 Is Industry Ready to Use Formal Methods? 39

2.7	To Learn More	39
	References	40
3	Propositional Logic	45
3.1	Syntax	46
3.2	Semantics	47
3.3	Proof System	54
3.4	Soundness and Completeness	61
3.5	Validity Checking: Semantic Methods	64
	3.5.1 Normal Forms in Propositional Logic	65
	3.5.2 Validity of CNF Formulas	68
	3.5.3 Satisfiability of CNF Formulas	69
3.6	Validity Checking: Deductive Methods	71
3.7	To Learn More	74
3.8	Exercises	76
	References	78
4	First-Order Logic	81
4.1	Syntax	82
4.2	Semantics	87
4.3	Proof System	95
4.4	Soundness and Completeness	100
4.5	Validity Checking	102
	4.5.1 Negation and Prenex Normal Forms	102
	4.5.2 Herbrand/Skolem Normal Forms and Semi-Decidability . .	104
	4.5.3 Decidable Fragments	108
4.6	Variations and Extensions	109
	4.6.1 First-Order Logic with Equality	109
	4.6.2 Many-Sorted First-Order Logic	110
	4.6.3 Second-Order Logic	113
4.7	First-Order Theories	115
	4.7.1 Equality	117
	4.7.2 Natural Numbers	118
	4.7.3 Integers	119
	4.7.4 Arrays	120
	4.7.5 Other Theories	121
	4.7.6 Combining Theories	121
4.8	To Learn More	122
4.9	Exercises	124
	References	127
5	Hoare Logic	129
5.1	Annotated <i>While</i> Programs	130
	5.1.1 Program Semantics	131
	5.1.2 The $\text{While}^{\text{int}}$ Programming Language	134
5.2	Specifications and Hoare Triples	136

5.3	Loop Invariants	140
5.4	Hoare Calculus	143
5.5	The $\text{While}^{\text{array}}$ Programming Language	148
5.5.1	A Rule of Hoare Logic for Array Assignment	150
5.6	Loop Termination and Total Correctness	151
5.7	Adaptation	151
5.8	To Learn More	154
5.9	Exercises	155
	References	157
6	Generating Verification Conditions	159
6.1	Mechanising Hoare Logic	159
6.2	The Weakest Precondition Strategy	162
6.3	An Architecture for Program Verification	167
6.4	A VCGen Algorithm	168
6.4.1	Calculating the Weakest Precondition	168
6.4.2	Calculating Verification Conditions	170
6.4.3	Putting It All Together	170
6.5	Verification Conditions for $\text{While}^{\text{array}}$ Programs	172
6.6	To Learn More	177
6.7	Exercises	178
	References	179
7	Safety Properties	181
7.1	Error Semantics and Safe Programs	181
7.1.1	$\text{While}^{\text{int}}$ with Errors	184
7.2	Safety-Sensitive Calculus and VCGen	185
7.2.1	Safe $\text{While}^{\text{int}}$ Programs	187
7.3	Bounded Arrays: The $\text{While}^{\text{array}[N]}$ Language	188
7.3.1	Safe $\text{While}^{\text{array}[N]}$ Programs	190
7.3.2	An Alternative Formalisation of Bounded Arrays	192
7.4	To Learn More	193
7.5	Exercises	193
	References	194
8	Procedures and Contracts	195
8.1	Procedures and Recursion	196
8.1.1	The $\sim$ Notation	197
8.1.2	Recursive Procedures	198
8.1.3	Procedure Calls in System $\mathcal{H}_g$	199
8.2	Contracts and Mutual Recursion	200
8.2.1	Programming with Contracts	202
8.2.2	Inference System for Parameterless Procedures	203
8.2.3	Verification Conditions for Parameterless Procedures	204
8.3	Frame Conditions	206
8.4	Procedures with Parameters	208

8.4.1	Parameters Passed by Value	211
8.4.2	Parameters Passed by Reference	215
8.4.3	Aliasing	218
8.5	Return Values and Pure Functions	220
8.6	To Learn More	226
8.7	Exercises	226
	References	227
9	Specifying C Programs	229
9.1	An Introduction to ACSL	230
9.1.1	Array-Based Programs	230
9.1.2	Using Axiomatics	231
9.1.3	Function Calls	233
9.1.4	State Labels and Behaviours	234
9.2	To Learn More	238
9.3	Exercises	239
	References	239
10	Verifying C Programs	241
10.1	Safety Verification	242
10.1.1	Arithmetic Overflow Safety	243
10.1.2	Safety of Array Access	244
10.1.3	Adding Loop Invariants	245
10.1.4	Termination Checking and Loop Variants	246
10.1.5	Safety of Function Calls	247
10.2	Functional Correctness: Array Partitioning	248
10.3	Functional Correctness: Multiset Preservation	250
10.4	A Word of Caution	251
10.5	Pointer Variables and Parameters Passed by Reference	253
10.6	To Learn More	254
10.7	Exercises	255
	References	255
	Index	258