
Undergraduate Topics in Computer Science

Series editor

Ian Mackie

Advisory Board

Samson Abramsky, University of Oxford, Oxford, UK

Karin Breitman, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil

Chris Hankin, Imperial College London, London, UK

Dexter C. Kozen, Cornell University, Ithaca, USA

Andrew Pitts, University of Cambridge, Cambridge, UK

Hanne Riis Nielson, Technical University of Denmark, Kongens Lyngby, Denmark

Steven S. Skiena, Stony Brook University, Stony Brook, USA

Iain Stewart, University of Durham, Durham, UK

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.

More information about this series at <http://www.springer.com/series/7592>

Peter Csaba Ölveczky

Designing Reliable Distributed Systems

A Formal Methods Approach Based
on Executable Modeling in Maude

Peter Csaba Ölveczky
University of Oslo
Oslo
Norway

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-1-4471-6686-3 ISBN 978-1-4471-6687-0 (eBook)
DOI 10.1007/978-1-4471-6687-0

Library of Congress Control Number: 2017947868

© Springer-Verlag London 2017

The author(s) has/have asserted their right(s) to be identified as the author(s) of this work in accordance with the Copyright, Designs and Patents Act 1988.

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. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer-Verlag London Ltd.
The registered company address is: The Campus, 4 Crinan Street, London, N1 9XW, United Kingdom

To Cecilia, Roland, and Robert

Foreword

De facto, both individually and socially, all of us rely more and more on software-mediated systems and devices. However, as software disasters and successful cyber-attacks keep piling up, the crucial importance of software quality and reliability, and the sobering realization of how vulnerable our systems are, loom larger and larger. In areas such as avionics, railway systems, microprocessor design, and security protocols, the obvious consequence, namely, the need for *mathematical* methods providing high assurance beyond the insufficient assurance made possible by testing alone is well understood, so that formal methods are applied in practice in such areas. But this is far from being the case in general. In particular, since most systems nowadays are *distributed* systems, which are very hard to test and can have very subtle bugs, the necessary but insufficient role of testing is painfully felt; but the obvious need for stronger verification methods beyond testing is still not fully understood or appreciated in practice.

An important question is why this highly problematic state of affairs remains largely unresolved. It is certainly true that, although big advances in both scalability and automation of formal methods have been made and very important successful formal verification efforts have been carried out, *scalability* is still an important challenge. However, in my view two closely related problems, quite orthogonal to scalability, present a serious obstacle, namely: (1) verifying *designs*, as opposed to verifying *code*, is hindered in practice by the lack of suitable mathematical models for system designs; and (2) there is considerable ignorance about the *mathematical modeling* nature of programming made possible by *declarative languages*. The importance of solving problem (1) is one of effectiveness: design errors can be orders of magnitude more expensive than coding errors and in fact account for most of the critical errors in system development. This does not mean that verifying code is unimportant; however, correct-by-construction code generation from verified designs is a promising alternative to standard code verification and can be a considerably more cost-effective way of achieving code correctness. Problem (2) is quite serious and is self-inflicted. In many prestigious universities worldwide most

undergraduates now only learn to program in imperative languages like C, C++, or Java, and often do not even *know* that it is possible for a program to also be a *mathematical model* of the problem it solves.

The point is that problems (1) and (2) are closely related. A declarative program, that is, a program written in a computational logic and specified as a *theory* in such a logic has two key advantages: (i) it defines a *mathematical model* of the system it executes, which means that the distinction between design and code either evaporates or becomes reduced to one of refining and optimizing a high-level declarative program into a more efficient, yet equivalent, program; and (ii) since a system design specified as a declarative program is already a mathematical object, verifying its properties is typically much easier than verifying them for a program written in an imperative language. This all means that understanding the crucial role of declarative programs as formal executable specifications can greatly help in solving problems (1) and (2) at the same time.

An important distinction to be made is that between what I call *system specification* and *property specification and verification*. A computational system can obviously be programmed. By programming it in a declarative language, we obtain a *mathematical model* of the *system* thus programmed. But only by having a mathematical model of a system is it meaningful at all to *verify* its mathematical *properties*. Such properties need not be expressed in the computational logic of the declarative language in which the system in question has been specified. Indeed, many properties, for example, temporal logic properties or inductive theorems, need not be *executable* at all. This means that system properties to be verified *about* a system design may be specified in various logics in which such properties have a natural and easy expression. This also means that formal verification can be seen as the task of proving that the model defined by a formal, executable specification \mathcal{S} —that is, by a declarative program \mathcal{S} —*satisfies* a set $\{\varphi_1, \dots, \varphi_n\}$ of desired properties expressed as formulas $\varphi_1, \dots, \varphi_n$ in a suitable property specification logic.

All this brings us to the present book, that addresses the above problems (1) and (2) in an excellent and eminently practical way. One of its key contributions to undergraduate CS education is how well it shows students that programming as mathematical modeling in a declarative language such as Maude is: (i) quite easy, (ii) fairly intuitive, and (iii) actually fun to do. Once this is done through many well-chosen examples and exercises, students come to realize, almost as an afterthought, that they have been doing *mathematical modeling* all along. This happens just as for the man who suddenly realized that he had been speaking in prose all his life. This “aha moment” opens the door for discussing issues of *formal correctness* and *formal specification and verification* of system *properties*, so that property logics and their associated verification methods can be naturally introduced and explained.

In the first part of this book, all this is done for *deterministic systems* specified in equational logic as functional programs in Maude. Since the mathematical model defined by an equational program is the *initial algebra* of such a program as an equational theory, students are then introduced to the specification and verification of *inductive properties* satisfied by such initial algebras, and are shown how Maude itself can be used as a simple inductive theorem prover to verify such properties.

Since equational logic is a sublogic of rewriting logic, which is a natural and simple logic in which to specify *distributed systems*, the book then moves in a natural and seamless way from its first part focused on deterministic systems into its second and main part, focused on the executable specification of distributed systems as rewrite theories in Maude. Properties of distributed systems and their specification and verification are then explained. The same gentle and gradual approach is followed in this second part. This is achieved so well and with such a wealth of examples, that the book can also be used as a first introduction to distributed systems, their modeling, and their verification at the undergraduate level. The same gradual method of approach is also followed for the specification and verification of properties. First, the simplest of such properties, namely, *invariants*, are introduced, and explicit-state *reachability analysis* supported by Maude's `search` command is used to automatically verify such invariants, or to do so up to a given depth bound if the system is infinite-state. After this, a gentle, yet quite thorough, introduction to linear-time temporal logic (LTL) and its semantics is given, and many examples are given showing how Maude's LTL model checker can be used to automatically verify LTL properties of a distributed system formally specified as a rewrite theory in Maude. Finally, broader perspectives are opened up by explaining how additional topics such as the specification and verification of real-time and of probabilistic systems can be treated by corresponding extensions of rewriting logic by means of real-time rewrite theories and probabilistic rewrite theories; and at the property level by suitable real-time and probabilistic extensions of temporal logic. Each notion is again illustrated by means of well-chosen examples and exercises.

In summary, this book addresses an important and serious need in undergraduate CS education and, at the same time, the broader need of training a next generation of computer scientists who are well acquainted with both distributed systems and with the mathematical modeling and verification of such systems. Given the present state of affairs, both in the vulnerability of our systems and the serious gaps in mathematical modeling abilities in undergraduate CS education, the appearance of this book could not be more timely. I have been using earlier drafts of this book in a program verification course at the University of Illinois at Urbana-Champaign and plan to recommend the present book to my students as reading material for such a course in the years to come. I am sure that it will be of great help to many other persons teaching programming languages, formal methods, and distributed systems at the undergraduate level and, above all, to the students themselves.

Cabo Palos
June 2017

José Meseguer

Preface

The two main goals of this book are to:

1. provide an introduction to formal modeling and analysis of both data types and, in particular, distributed systems; and
2. provide an introduction to distributed computer systems and the challenges of designing and analyzing such systems.

The book is meant to be a first introduction to formal methods and therefore does not assume any previous knowledge about formal methods or distributed systems; it is based on a third-year course at the University of Oslo, but can equally well be taught at the second-year level. Some previous exposure to programming could be useful; likewise, experience with simple recursive functions is helpful but not necessary. There are no prerequisites on the mathematical side.

A distinguishing feature of this book is the significant use of the rewriting-logic-based Maude language and simulation and model checking tool for formally modeling both data types and distributed systems. Data types are specified using a functional programming style that students tend to like. Indeed, a valuable side effect of studying this book is training in writing recursive programs. For formally modeling distributed systems, Maude provides a simple yet intuitive and expressive modeling formalism that is particularly suitable for modeling distributed systems in an object-oriented way. Maude is by now a mature and well-established tool that is increasingly used around the world.

About the Content

As mentioned above, one main goal of this book is to gently introduce students to a wide range of concepts in formal methods, including:

- *verifying* properties about programs and (models of) systems; e.g., proving that a specification/program terminates for all possible inputs, and using equational logic to prove semantic properties;
- logics and inference systems; and
- automated model checking techniques to analyze properties for some—but not all—possible inputs/system configurations.

This book is divided into two parts. The first part deals with specifying the data types needed to model complex distributed systems. This part introduces classical algebraic specification and term rewriting theory, including reasoning about termination, confluence, and inductive equational properties.

The second part deals with formally modeling and analyzing distributed systems in rewriting logic using Maude. This part introduces rewriting logic and object-oriented modeling of distributed systems. It also introduces temporal logic to specify requirements that a system should satisfy. Such models are analyzed using Maude simulations, reachability analysis, and temporal logic model checking, thereby also giving the students a hands-on experience of the state-space explosion problem for distributed systems. As mentioned above, the second main goal of this book is to introduce the students to the problems of designing and analyzing distributed systems. Instead of giving theoretical explanations of these issues, the book tries to convey intuition about distributed systems and their design challenges through a range of examples/case studies in different domains, including: the dining philosophers problem, transport protocols like the alternating bit protocol and the sliding window protocol, classic distributed algorithms such as the distributed two-phase protocol for distributed database systems, distributed mutual exclusion and leader election algorithms, and the NSPK cryptographic protocol. Finally, the book briefly introduces two extensions of standard distributed systems: real-time systems and probabilistic systems.

The book is based on a course that has been given at the University of Oslo for more than 10 years, which implies that the book contains a wealth of exercises, both smaller ones and larger ones suitable for course projects, etc. Most of the executable code presented in this book, as well as other supplementary material, can be found at <http://peterol.at.ifi.uio.no/BOOK>.

I would like to thank José Meseguer, Dorel Lucanu, Narciso Martí-Oliet, and Ralf Sasse for many insightful and very helpful comments on earlier versions of this book, Indranil Gupta for discussions on distributed systems, Jon Grov for providing the figures used in this book, Si Liu for performing the statistical model checking experiments, Lars Kristiansen for discussions on logic, and Shiji Bijo, Antonio Gonzalez Burgueño, Benjamin Oliver, and Olaf Owe for pointing out mistakes in those earlier drafts. I also thank Hanne Riis Nielson and Ian Mackie for encouraging me to publish this book with Springer, and Simon Rees and Wayne Wheeler for their patience in waiting for it to be finished.

Contents

1	Introduction	1
1.1	Modeling	2
1.1.1	Models of Distributed Systems	2
1.1.2	From Model to System	3
1.2	The Maude Modeling Language and Analysis Tool	4
1.3	Why Maude?	5
1.4	Contents of the Book	6
1.4.1	Part I: Algebraic Specification and Term Rewriting	6
1.4.2	Part II: Dynamic Systems	7
1.4.3	Appendix: Mathematical Background	8
 Part I Equational Specifications and Their Analysis		
2	Equational Specification in Maude	11
2.1	Hello World: Our First Maude Specifications	12
2.1.1	Natural Numbers with Addition	13
2.1.2	The Boolean Values and Functions	14
2.1.3	Module Importation	15
2.2	Many-Sorted Equational Specifications	16
2.3	Requirements of Equational Specifications	19
2.3.1	One-to-one Constructor Basis	19
2.3.2	Termination: No Infinite Computations	20
2.3.3	Uniqueness of the “Result”	21
2.3.4	Definedness: The Result Should be a Constructor Term	21
2.3.5	Maude and the Requirements	22
2.4	Many-Sorted Specification of Data Types	22
2.4.1	Defining Functions: Getting Started	23
2.4.2	Expressiveness of Many-Sorted Equational Specifications	23
2.4.3	Maude Specifications of Some Data Types	24
2.5	Order-Sorted Equational Specifications	29

2.5.1	Examples of Order-Sorted Equational Specifications	30
2.6	Membership Equational Logic Specifications	33
2.7	Built-in Data Types	35
2.7.1	Booleans	35
2.7.2	Natural Numbers	36
2.7.3	Integers	38
2.7.4	Rational Numbers	38
2.7.5	Floating-Point Numbers	39
2.7.6	Strings	39
2.7.7	Random Numbers	40
2.8	Associativity and Commutativity: Lists and Multisets	41
2.8.1	Commutativity, Associativity, and Identity	41
2.8.2	Associativity and Identity: Lists	43
2.8.3	Associativity, Commutativity, and Identity: Multisets and Sets	44
2.9	Examples	47
2.9.1	Two Sorting Algorithms	47
2.9.2	Some NP-Complete Problems	49
2.10	* Some Other Maude Features	54
2.10.1	Parameterized Modules	54
2.10.2	Telling Maude how to Evaluate an Expression	56
2.10.3	Other Features	57
3	Operational Semantics of Equational Specifications	59
3.1	The Reduction Relation	59
3.1.1	Basic Definitions	60
3.1.2	The Reduction Relation	62
3.1.3	Some Derived Relations	62
3.2	Operational Properties	63
3.3	Conditional Equations and Matching with <i>assoc/comm</i>	64
3.3.1	Conditional Equations	64
3.3.2	* A-, C-, and AC-matching is NP-hard	65
4	Termination	67
4.1	Undecidability of Termination	68
4.2	Nontermination	72
4.3	Proving Termination Using “Weight Functions”	73
4.4	Simplification Orders	76
4.4.1	The Lexicographic Path Order	79
4.4.2	The Multiset Path Order and Other Variations of lpo	80
4.4.3	Comparing Weight Functions and Simplification Orders	81

5	Confluence	85
5.1	Unification	87
5.2	Checking Local Confluence	89
6	Equational Logic	93
6.1	Equational Logic.	94
6.1.1	* Knuth-Bendix Completion	99
6.2	Inductive Theorems	101
6.2.1	Proving Inductive Theorems for Nat	103
6.2.2	Inductive Theorems for Other Data Types.	105
7	Models of Equational Specifications	109
7.1	Many-Sorted Σ -Algebras	110
7.1.1	Homomorphisms and Isomorphisms	112
7.1.2	Term Algebras.	115
7.2	(Σ, E) -Models: (Σ, E) -Algebras	116
7.2.1	Quotient Algebras	117
7.2.2	The Algebra $\mathcal{T}_{\Sigma, E}$	117
7.2.3	The Normal Form Algebra	118
7.3	Soundness and Completeness of Equational Logic	118
7.4	Intended Models: Initial Algebras.	120
7.5	Empty Sorts and Many-Sorted Equational Logic	124

Part II Specification and Analysis of Distributed Systems in Maude

8	Modeling Distributed Systems in Rewriting Logic	127
8.1	Dynamic Systems	127
8.1.1	Properties of Dynamic and Distributed Systems	128
8.1.2	Behaviors of Distributed Systems	128
8.2	Modeling Dynamic Systems in Rewriting Logic.	129
8.2.1	Rewrite Rules	130
8.2.2	Rewriting Logic Specifications	131
8.2.3	Examples.	132
8.3	Concurrency	135
8.3.1	Sideways Concurrency	136
8.3.2	Nested Concurrency	138
8.4	Deduction in Rewriting Logic.	139
8.4.1	Concurrent Steps	140
8.4.2	Termination and Confluence	142
8.5	* Frozen Operators	143
8.6	* Denotational Semantics	144
9	Executing Rewriting Logic Specifications in Maude	145
9.1	Executing One Sequential Rewrite Step	145
9.2	Simulating Single Behaviors.	147
9.3	Search.	150

10	Concurrent Objects in Maude	155
10.1	Modeling Concurrent Objects in Maude	155
10.1.1	Rewrite Rules for Objects	156
10.2	Concurrent Objects in Full Maude	162
10.2.1	Using Full Maude	162
10.2.2	Object-Oriented Modules in Full Maude	163
10.2.3	Subclasses	165
10.2.4	Search in Full Maude	168
10.2.5	Using Full Maude: Repetition	169
10.3	Example: The Dining Philosophers.	170
10.3.1	Problem Description	170
10.3.2	Modeling the Dining Philosophers	171
10.3.3	Deadlock and Livelock	173
10.3.4	Fairness Issues.	173
10.3.5	Version 2: A Deadlock-Free Solution	173
10.3.6	Version 3: A Deadlock-Free and Livelock-Free Solution	173
10.4	Randomized Simulations: Winning in Vegas	176
10.4.1	Blackjack.	176
10.4.2	Modeling Blackjack Rounds	177
10.4.3	Further Guarantees	182
11	Modeling Communication in Maude	183
11.1	Synchronous Communication	184
11.2	Unordered Asynchronous Communication by Message Passing	185
11.2.1	Unordered Unicast.	185
11.2.2	Multicast	188
11.2.3	Broadcast.	189
11.2.4	Wireless Broadcast	190
11.2.5	Modeling Unreliable Communication	190
11.3	Ordered Asynchronous Communication using Links	193
11.3.1	Unreliable Links	195
11.3.2	Links with Limited Capacity	196
11.4	Asynchronous Communication Using Shared Variables	197
12	Modeling and Analyzing Transport Protocols.	199
12.1	Reliable Communication Using Sequence Numbers	199
12.1.1	Maude Modeling	200
12.1.2	Formal Analysis	203
12.2	The Alternating Bit Protocol.	204
12.3	The Sliding Window Protocol	206
12.3.1	Sliding Window with Links.	209

13	Distributed Algorithms	211
13.1	Atomicity of Distributed Transactions: Two-Phase Commit	211
13.1.1	The Two-Phase Commit Protocol	212
13.1.2	Abstraction	213
13.1.3	Assumptions	214
13.1.4	Specification and Analysis of 2PC in Maude	214
13.2	Distributed Mutual Exclusion	221
13.2.1	Modeling the Central Server Algorithm	222
13.2.2	Analyzing the Central Server Algorithm	224
13.3	Distributed Leader Election	226
13.3.1	A Ring-based Leader Election Algorithm	226
13.3.2	A Spanning-Tree-based Algorithm for Wireless Networks	227
13.4	Consensus Algorithms	231
14	Analyzing a Cryptographic Protocol	233
14.1	Public-Key Cryptography	233
14.1.1	Digital Signatures	234
14.1.2	Symmetric-Key Cryptography	235
14.2	The Needham-Schroeder Public-Key (NSPK) Protocol	235
14.3	Modeling NSPK in Maude	236
14.3.1	Executing the NSPK Specification	240
14.4	Modeling Intruders	241
14.5	Analyzing NSPK with Intruders	244
14.6	Discussion	247
14.7	The Corrected Protocol	248
15	System Requirements	249
15.1	State-based and Action-based Properties	250
15.1.1	Actions/Events	251
15.1.2	State Propositions	252
15.2	Temporal Properties	252
15.2.1	Invariance: “Nothing Bad Will Happen”	253
15.2.2	Guarantee: “Something Good Must Eventually Happen”	254
15.2.3	Reachability: “Something Bad Could Happen”	255
15.2.4	Response: “A Request Will Always be Answered”	256
15.2.5	Stability	256
15.2.6	Other Requirements	257
15.3	Analyzing Invariants	260
16	Formalizing and Checking Requirements	263
16.1	Linear Temporal Logic	263
16.1.1	Behaviors	264

16.1.2	The Syntax of LTL	264
16.1.3	The Semantics of LTL	266
16.1.4	* Kripke Structures	268
16.2	Some LTL Formulas	269
16.2.1	Formalizing Classes of Requirements	269
16.2.2	Fairness Assumptions	271
16.3	Model Checking in Maude	273
16.3.1	Getting Started	273
16.3.2	Defining Atomic Propositions	274
16.3.3	Defining LTL Formulas	274
16.3.4	Performing Model Checking	275
16.3.5	Example: Analyzing Mutual Exclusion	277
16.4	* Some More Temporal Logic	281
17	Real-Time and Probabilistic Systems	283
17.1	Real-Time Systems	283
17.1.1	Specifying Real-Time Systems in Rewriting Logic	284
17.1.2	Timed Temporal Logics	291
17.1.3	Real-Time Maude	292
17.2	Probabilistic Systems	293
17.2.1	Probabilistic Rewrite Theories	294
17.2.2	Probabilistic Temporal Logics	296
17.2.3	PV _E STA Analysis	297
	Appendix A: Mathematical Preliminaries	299
	References	303
	Index	309