

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

Faron Moller • Georg Struth

Modelling Computing Systems

Mathematics for Computer Science

 Springer

Faron Moller
Department of Computer Science
Swansea University
Swansea, UK

Georg Struth
Dept. Computer Science
University of Sheffield
Sheffield, UK

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 Kozen, Cornell University, Ithaca, USA
Andrew Pitts, University of Cambridge, Cambridge, UK
Hanne Riis Nielson, Technical University of Denmark, Kongens Lyngby, Denmark
Steven Skiena, Stony Brook University, Stony Brook, USA
Iain Stewart, University of Durham, Durham, UK

ISSN 1863-7310 Undergraduate Topics in Computer Science
ISBN 978-1-84800-321-7 ISBN 978-1-84800-322-4 (eBook)
DOI 10.1007/978-1-84800-322-4
Springer London Heidelberg New York Dordrecht

Library of Congress Control Number: 2013943907

© Springer-Verlag London 2013

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. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

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.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

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

Contents

Starred sections are optional and often represent advanced material.

Preface	xiii
0 Introduction	1
0.1 Examples of System Failures	2
0.1.1 Clayton Tunnel Accident	2
0.1.2 USS Scorpion	4
0.1.3 Therac 25 Radiotherapy Machine	4
0.1.4 London Ambulance Service	5
0.1.5 Intel Pentium	6
0.1.6 Ariane 5	7
0.1.7 Needham-Schroeder Protocol	7
0.2 System, Model, Abstraction and Notation	9
0.3 Specification, Implementation and Verification	13
Part I: Mathematics for Computer Science	15
1 Propositional Logic	17
1.1 Propositions and Deductions	18
1.2 The Language of Propositional Logic	21
1.2.1 Propositional Variables	22
1.2.2 Negation	22
1.2.3 Disjunction	23
1.2.4 Conjunction	25
1.2.5 Implication	25
1.2.6 Equivalence	27
1.2.7 The Syntax of Propositional Logic	27
1.2.8 Parentheses and Precedences	28
1.2.9 Syntax Trees	30
1.3 Modelling with Propositional Logic	32
1.4 Ambiguities of Natural Languages	35
1.5 Truth Tables	40
1.6 Equivalences and Valid Arguments	45
★ 1.7 Algebraic Laws for Logical Equivalences	47
1.8 Additional Exercises	50
2 Sets	57
2.1 Set Notation	57
2.2 Membership, Equality and Inclusion	59
2.3 Sets and Properties	63
2.3.1 Russell's Paradox	64

	2.4	Operations on Sets	65
	2.4.1	Union	65
	2.4.2	Intersection	66
	2.4.3	Difference	67
	2.4.4	Complement	68
	2.4.5	Powerset	69
★	2.4.6	Generalised Union and Intersection	72
	2.5	Ordered Pairs and Cartesian Products	73
	2.6	Modelling with Sets	76
★	2.7	Algebraic Laws for Set Identities	79
★	2.8	Logical Equivalences versus Set Identities	81
	2.9	Additional Exercises	83
★	3	Boolean Algebras and Circuits	87
	3.1	Boolean Algebras	87
	3.2	Deriving Identities in Boolean Algebras	90
	3.3	The Duality Principle	93
	3.4	Logic Gates and Digital Circuits	95
	3.5	Making Computers Add	100
	3.5.1	Binary Numbers	100
	3.5.2	Adding Binary Numbers	102
	3.5.3	Building Half Adders	103
	3.5.4	Building Full Adders	104
	3.5.5	Putting It All Together	105
	3.6	Additional Exercises	106
	4	Predicate Logic	109
	4.1	Predicates and Free Variables	109
	4.2	Quantifiers and Bound Variables	111
	4.2.1	Universal Quantification	113
	4.2.2	Existential Quantification	115
	4.2.3	Bounded Quantifications	118
	4.3	Rules for Quantification	120
	4.4	Modelling in Predicate Logic	124
	4.5	Additional Exercises	127
★	5	Proof Strategies	131
	5.1	A First Example	132
	5.2	Proof Strategies for Implication	134
	5.3	Proof Strategies for Negation	138
	5.4	Proof Strategies for Conjunction and Equivalence	142
	5.5	Proof Strategies for Disjunction	144
	5.6	Proof Strategies for Quantifiers	147
	5.6.1	Universal Quantification	147

	5.6.2	Existential Quantification	149
	5.6.3	Uniqueness	152
	5.7	Additional Exercises	153
	6	Functions	155
	6.1	Basic Definitions	155
	6.2	One-To-One and Onto Functions	160
	6.3	Composing Functions	163
★	6.4	Comparing the Sizes of Sets	166
★	6.5	The Knaster-Tarski Theorem	173
	6.6	Additional Exercises	176
	7	Relations	179
	7.1	Basic Definitions	179
	7.2	Binary Relations	181
	7.2.1	Functions as Binary Relations	185
	7.3	Operations on Binary Relations	186
	7.3.1	Boolean Operations	186
	7.3.2	Inverting Relations	187
	7.3.3	Composing Relations	188
	7.3.4	The Domain and Range of a Relation	189
	7.4	Properties of Binary Relations	190
	7.4.1	Reflexive and Irreflexive Relations	190
	7.4.2	Symmetric and Antisymmetric Relations	191
	7.4.3	Transitive Relations	191
	7.4.4	Orderings Relations	192
	7.4.5	Equivalence Relations	193
	7.4.6	Equivalence Classes and Partitions	195
	7.5	Additional Exercises	197
	8	Inductive and Recursive Definitions	201
	8.1	Inductively-Defined Sets	201
	8.2	Inductively-Defined Syntactic Sets	205
	8.3	Backus-Naur Form	207
	8.4	Inductively-Defined Data Types	210
	8.5	Inductively-Defined Functions	212
	8.6	Recursive Functions	216
	8.7	Recursive Procedures	218
	8.8	Additional Exercises	220
	9	Proofs by Induction	223
	9.1	Convincing but Inconclusive Evidence	223
	9.2	A Primary School Induction Argument	227
	9.3	The Induction Argument	228

	9.4	Strong Induction	234
	9.5	Induction Proofs from Inductive Definitions	235
★	9.6	Fun with Fibonacci Numbers	237
	9.6.1	A Fibonacci Number Test	237
	9.6.2	A Carrollean Paradox	239
	9.6.3	Fibonacci Decompositions	240
	9.7	When Inductions Go Wrong	241
	9.8	Examples of Induction in Computer Science	244
	9.9	Additional Exercises	246
	10	Games and Strategies	251
	10.1	Strategies for Games-of-No-Chance	252
	10.2	Nim	260
★	10.3	Fibonacci Nim	262
	10.4	Chomp	264
	10.5	Hex	266
	10.6	Bridg-It	269
	10.7	Additional Exercises	271
	Part II:	Modelling Computing Systems	277
	11	Modelling Processes	279
	11.1	Labelled Transition Systems	281
	11.2	Computations and Processes	287
	11.3	A Language for Describing Processes	292
	11.3.1	The Nil Process 0	292
	11.3.2	Action Prefix	293
	11.3.3	Process Definitions	294
	11.3.4	Choice	295
	11.4	Distinguishing Between Behaviours	299
	11.5	Equality Between Processes	302
	11.6	Additional Exercises	303
	12	Distinguishing Between Processes	309
	12.1	The Bisimulation Game	309
	12.2	Properties of Game Equivalence	313
	12.3	Bisimulation Relations	315
	12.4	Bisimulation Colourings	319
★	12.5	The Bisimulation Game Revisited: To Infinity and Beyond!	322
	12.5.1	Ordinal Numbers	323
	12.5.2	Ordinal Bisimulation Games	324
	12.6	Additional Exercises	328

13	Logical Properties of Processes	333
	13.1 The Mays and Musts of Processes	334
	13.2 A Modal Logic for Properties	336
	13.3 Negation Is Definable	341
	13.4 The Vending Machines Revisited	344
	13.5 Modal Properties and Bisimulation	346
★	13.6 Characteristic Formulæ	350
★	13.7 Global Semantics	352
	13.8 Additional Exercises	353
14	Concurrent Processes	357
	14.1 Synchronisation Merge	357
	14.2 Counters	360
	14.3 Railway Level Crossing	362
	14.4 Mutual Exclusion	365
	14.4.1 Dining Philosophers	365
	14.4.2 Peterson's Algorithm	368
	14.5 A Message Delivery System	371
	14.6 Alternating Bit Protocol	373
	14.7 Additional Exercises	377
★	15 Temporal Properties	381
	15.1 Three Standard Temporal Operators	382
	15.1.1 Always: $\Box P$	382
	15.1.2 Possibly: $\Diamond P$	383
	15.1.3 Until: $P U Q$	384
	15.2 Recursive Properties	385
	15.2.1 Solving Recursive Equations	387
	15.2.2 Fixed Point Solutions	388
	15.3 The Modal Mu-Calculus	390
	15.4 Least versus Greatest Fixed Points	392
	15.4.1 Approximating Fixed Points	393
	15.5 Expressing Standard Temporal Operators	397
	15.5.1 Always: $\Box P$	398
	15.5.2 Possibly: $\Diamond P$	398
	15.5.3 Until: $P U Q$	398
	15.6 Further Fixed Point Properties	399
	15.7 Additional Exercises	401
	Solutions to Exercises	405
	Index	493

List of Figures

1.1	The formulæ of propositional logic.	28
1.2	$B =$ “the symbol is black”, $C =$ “the symbol is a circle”.	34
2.1	An example Venn diagram.	62
3.1	The Laws of Boolean Algebra.	88
4.1	A Sudoku puzzle.	126
6.1	A function $f : A \rightarrow B$ from A to B	156
6.2	The graph of the function $f(x) = x^3 - x$	159
6.3	Enumerating the positive rationals.	172
6.4	Enumerating the rational numbers	178
7.1	James Bond Films.	181
8.1	Towers of Hanoi with five disks.	220
8.2	A family tree of male (σ) and female (φ) bees.	222
9.1	Counting regions in circles.	226
9.2	Dominos inductively falling down.	229
9.3	Induction as a boot-strapping operation.	230
10.1	The game tree for a simple coin game.	257
10.2	An analysis of 3×4 CHOMP.	266
10.3	The HEX board.	267
10.4	HEX never ends in a draw.	268
10.5	The BRIDG-IT board.	269
10.6	BRIDG-IT never ends in a Draw.	270
10.7	The winning strategy for BRIDG-IT.	271
11.1	A simple calculator.	280
11.2	An example labelled transition system.	283
11.3	The LTS of the Man-Wolf-Goat-Cabbage Riddle.	286
11.4	A process computing the GCD of 246 and 174.	288
11.5	A simple lamp control process.	289
11.6	Syntax and semantics of the process language PROC.	297
11.7	Clock processes.	298

11.8	Three implementations of a vending machine.	299
11.9	Transition systems for the three vending machines.	300
11.10A	car seat belt safety system.	306
12.1	Example transition systems.	310
12.2	Example transition systems.	317
13.1	Syntax and semantics of the modal logic HML	338
13.2	Global semantics of the modal logic HML	352
14.1	The railway level crossing.	362
14.2	Dining Philosophers.	366
14.3	The message-passing system.	374
15.1	Global semantics of the modal mu-calculus.	390
15.2	Venn diagram for Exercise 2.15.	417
15.3	Venn diagram for Exercise 2.27.	420
15.4	The LTS of the Missionaries and Cannibals Riddle.	460
15.5	The LTS of the change-making machine.	468
15.6	The LTS for Exercise 11.16.	469
15.7	The enhanced message-passing system.	487

Preface

The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.

- Ted Nelson.

Computer Science is a relatively young discipline. University Computer Science Departments are rarely more than a few decades old. They will typically have emerged either from a Mathematics Department or an Engineering Department, and until recently a Computer Science degree was predominantly about writing computer programs (the mathematical software) and building computers (the engineering hardware). Textbooks typically referred to programming as an “art” or a “craft” with little scientific basis compared to traditional engineering subjects, and many computer programmers still like to see themselves as part of a pop culture of geeks and hackers rather than as academically-trained professionals.

However, the nature of Computer Science is changing rapidly, reflecting the increasing ubiquity and importance of its subject matter. In the last decades, computational methods and tools have revolutionised the sciences, engineering and technology. Computational concepts and techniques are starting to influence the way we think, reason and tackle problems; and computing systems have become an integral part of our professional, economic and social lives. The more we depend on these systems – particularly for safety-critical or economically-critical applications – the more we must ensure that they are safe, reliable and well designed, and the less forgiving we can be of failures, delays or inconveniences caused by the notorious “computer glitch.”

Unlike traditional engineering disciplines which are solidly rooted on centuries-old mathematical theories, the mathematical foundations underlying Computer Science are younger, and Computer Scientists have yet to agree on how best to approach the fundamental concepts and tasks in the design of computing systems. The Civil Engineer knows exactly how to define and analyse a mathematical model of the components of a bridge design so that it can be relied on not to fall down, and the Aeronautical Engineer knows exactly how to define and analyse a mathematical model of an aeroplane wing for the same purpose. However, Software Engineers have few universally-accepted mathematical modelling tools at their disposal. In the words of the eminent Computer Scientist Alan Kay, “most undergrad-

uate degrees in computer science these days are basically Java vocational training.” But computing systems can be at least as complex as bridges or aeroplanes, and a canon of mathematical methods for modelling computing systems is therefore very much needed. “Software’s Chronic Crisis” was the title of a popular and widely-cited *Scientific American* article from 1994, and, unfortunately, its message remains valid today.

University Computer Science Departments face a sociological challenge posed by the fact that computers have become everyday, deceptively easy-to-use objects. A single generation ago, new Computer Science students typically had teenage backgrounds spent writing Basic and/or Assembly Language programs for their early hobbyist computers. A passion for this activity is what drove these students into University Computer Science programmes, and they were not disappointed with the education they received. Their modern-day successors on the other hand – born directly into the heart of the computer era – have grown up with the internet, a billion dollar computer games industry, and mobile phones with more computing power than the space shuttle. They often choose to study Computer Science on the basis of having a passion for using computing devices throughout their everyday lives, for everything from socialising with their friends to downloading the latest films, and they often have less regard than they might to the considerations of what a University Computer Science programme entails, that it is far more than just using computers.

There is a universal trend of large numbers of first-year students transferring out of Computer Science programmes and into related programmes such as Media Studies or Information Studies. This trend, we feel, is often unjustified, and can be reversed by a more considered approach to modelling and the mathematical foundations of system design, one which the students can connect and feel at home with right from the beginning of their University education. This has been our motivation in writing this textbook aimed at teaching first-year undergraduate students the essential mathematics and modelling techniques for computing systems in a novel and relatively light-weight way.

The book is divided into two parts. Part I, subtitled *Mathematics for Computer Science*, introduces concepts from Discrete Mathematics which are in the curriculum of any University Computer Science programme, as well as much which often is not. This material is typically taught in service modules by mathematicians, and new Computer Science students often find it difficult to engage with the material presented in a purely mathematical context. We attempt here to present the material in an engaging and motivating fashion as the basis of computational thinking.

Part II of the book – *Modelling Computing Systems* – develops a par-

ticular approach to modelling based on state transition systems. State transition systems have always featured in the Computer Science curriculum, but traditionally (and increasingly historically) only within the study of formal languages. Here we introduce them as general modelling devices, and explore languages and techniques for expressing and reasoning about system specifications and (concurrent) implementations. Although Part I covers twice as many pages as Part II, the title of the book is nonetheless justified: much of the Mathematics presented in Part I itself is used directly for modelling systems, and forms the basis on which the approach developed in Part II is based.

The main benefit of mathematical formalisation is that systems can be modelled and analysed in precise and unambiguous ways; but formal precision can also be a major pitfall in modelling since it can compromise simplicity and intuition. In this book, therefore, we always try to start from intuition and examples, and we aim at developing precise concepts from that basis. How and when to be precise is certainly not less important to learn than precision itself: the ability to give mathematical proofs often does not depend on knowing precise formal definitions and foundations. One can, for example, write down recursive functions without having a precise formal concept in mind.

There is a long standing tradition in disciplines like Physics to teach modelling through little artifacts. The fundamental ideas of computational modelling and thinking as well can better be learned from idealised examples and exercises than from many real world computer applications. This book builds on a large collection of logical puzzles and mathematical games that require no prior knowledge about computers and computing systems; these can be much more fun and sometimes much more challenging than analysing a device driver or a criminal record database. Also, computational modelling and thinking is about much more than just computers!

In fact, games play a far more important role in the book: they provide a novel approach to understanding computer software and systems which is proving to be very successful both in theory and practice. When a computer runs a program, for example, it is in a sense playing a game against the user who is providing the input to the program. The program represents a strategy which the computer is using in this game, and the computer wins the game if it correctly computes the result. In this game, the user is the adversary of the computer and is naturally trying to confound the computer, which itself is attempting to defend its claim that it is computing correctly, that is, that the program it is running is a winning strategy. (In Software Engineering, this game appears in the guise of *testing*.) Similarly, the controller of a software system that interacts with its environment plays

a game against the environment: the controller tries to maintain the system's properties, while the environment tries to confound them.

This view suggests an approach to addressing three basic problems in the design of computing systems:

1. *Specification* refers to the problem of precisely identifying the task to be solved, as well as what exactly constitutes a solution. This problem corresponds to the problem of defining a winning strategy.
2. *Implementation* or *Synthesis* refers to the problem of devising a solution to the task which respects the specification. This problem corresponds to the problem of implementing a winning strategy.
3. *Verification* refers to the problem of demonstrating that the devised solution does indeed respect the specification. This problem corresponds to the problem of proving that a given strategy is in fact a winning strategy.

This analogy between the fundamental concepts in Software Engineering on the one hand, and games and strategies on the other, provides a mode of computational thinking which comes naturally to the human mind, and can be readily exploited to explain and understand Software Engineering concepts and their applications. It also motivates our thesis that Game Theory provides a paradigm for understanding the nature of computation.

There are over 200 exercises presented throughout each chapter, all of which have complete solutions at the back of the book, as well as over 200 further exercises at the end of each chapter whose solutions are not provided. The exercises within the chapters are often used to explore subtleties or side-issues, or simply to put lengthy arguments into an appendix, and as such should all be attempted; their solutions at the back of the book should be looked at as well, as they often explain the issues which the exercises are attempting to highlight.

Most of the material in this book has been used successfully for over a decade in first-year Discrete Mathematics and Systems Modelling modules. Countless eyes have passed over the text, and a thousand students have solved its exercises. Nonetheless there will inevitably be a (hopefully small) flurry of errors in the text for which we accept full responsibility and offer our sincere apologies.

Faron Moller
Swansea

Georg Struth
Sheffield