Computation Engineering
Applied Automata Theory and Logic

Computation Engineering
Applied Automata Theory and Logic

Ganesh Gopalakrishnan
University of Utah

Springer

Ganesh Gopalakrishnan
University of Utah
Department of Computer Science
50 S. Central Campus Drive
Salt Lake City, UT 84112-9205

Computation Engineering: Applied Automata Theory and Logic

Library of Congress Control Number: 2006920602

ISBN 0-387-24418-2 e-ISBN 0-387-32520-4
ISBN 9780387244181

Printed on acid-free paper.

© 2006 Springer Science+Business Media LLC
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now know or hereafter developed is forbidden.

The use in this publication of trade names, trademarks, service marks and similar terms, even if the are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

springer.com

Cover: Original cartoon art on the front and back covers by Geoff Draper.

Whatever you do will be insignificant,
but it is very important that you do it.
– Mohandas K. Gandhi

Contents

Foreword XXV

Preface XXVII

1 Introduction 1
Computation Science and Computation Engineering 1
What is 'Computation?' 2
A Minimalist Approach 3
How to Measure the Power of Computers? 5
Complexity Theory 5
Automata Theory and Computing 6
Why "Mix-up" Automata and Mathematical Logic? 7
Why Verify? Aren't Computers "Mostly Okay?" 7
Verifying Computing Systems Using Automaton Models 9
Automaton/Logic Connection 10
Avoid Attempting the Impossible 11
Solving One Implies Solving All 11
Automata Theory Demands a Lot From You! 12
A Brief Foray Into History 12
Disappearing Formal Methods 13

Exercises 14

2 Mathematical Preliminaries 15
2.1 Numbers 15
2.2 Boolean Concepts, Propositions, and Quantifiers 15
2.3 Sets 16
2.3.1 Defining sets 16
2.3.2 Avoid contradictions 17

2.3.3 Ensuring uniqueness of definitions 18
2.4 Cartesian Product and Powerset 21
2.4.1 Powersets and characteristic sequences 22
2.5 Functions and Signature 22
2.6 The λ Notation 23
2.7 Total, Partial, 1-1, and Onto Functions 25
2.8 Computable Functions 27
2.9 Algorithm versus Procedure 27
2.10 Relations 28
2.11 Functions as Relations 30
2.11.1 More λ syntax 30

Exercises 31

3 Cardinalities and Diagonalization 37
3.1 Cardinality Basics 37
3.1.1 Countable sets 38
3.1.2 Cardinal numbers 38
3.1.3 Cardinality "trap" 39
3.2 The Diagonalization Method 39
3.2.1 Simplify the set in question 40
3.2.2 Avoid dual representations for numbers 40
3.2.3 Claiming a bijection, and refuting it 41
3.2.4 'Fixing' the proof a little bit 41
3.2.5 Cardinality of 2^{Nat} and $Nat \to Bool$ 43
3.3 The Schröder-Bernstein Theorem 43
3.3.1 Application: cardinality of all C Programs 44
3.3.2 Application: functions in $Nat \to Bool$ 44
3.3.3 Proof of the Schröder-Bernstein Theorem 46

Exercises 50

4 Binary Relations 53
4.1 Binary Relation Basics 53
4.1.1 Types of binary relations 54
4.1.2 Preorder (reflexive plus transitive) 57
4.1.3 Partial order (preorder plus antisymmetric) 57
4.1.4 Total order, and related notions 58
4.2 Equivalence (Preorder plus Symmetry) 58
4.2.1 Intersecting a preorder and its inverse 59
4.2.2 Identity relation 59
4.2.3 Universal relation 59

4.2.4 Equivalence class 60
4.2.5 Reflexive and transitive closure 60
4.3 The *Power* Relation between Machines 61
4.3.1 The equivalence relation over machine types 62
4.4 Lattice of All Binary Relations over S 64
4.5 Equality, Equivalence, and Congruence 64
4.5.1 Congruence relation 64

Exercises 67

5 Mathematical Logic, Induction, Proofs 73
5.1 To Prove or Not to Prove! 73
5.2 Proof Methods 75
5.2.1 The implication operation 75
5.2.2 'If,' or 'Definitional Equality' 76
5.2.3 Proof by contradiction 77
5.2.4 Quantification operators $\forall$ and $\exists$ 78
5.2.5 Generalized DeMorgan's Law Relating $\forall$ And $\exists$ . . 80
5.2.6 Inductive definitions of sets and functions 80
5.3 Induction Principles 82
5.3.1 Induction over natural numbers 82
5.3.2 Noetherian induction 84
5.3.3 Structural 85
5.4 Putting it All Together: the Pigeon-hole Principle 85

Exercises 86

6 Dealing with Recursion 93
6.1 Recursive Definitions 93
6.1.1 Recursion viewed as solving for a function 94
6.1.2 Fixed-point equations 95
6.1.3 The `Y` operator 96
6.1.4 Illustration of reduction 97
6.2 Recursive Definitions as Solutions of Equations 97
6.2.1 The *least* fixed-point 100
6.3 Fixed-points in Automata Theory 101

Exercises 103

7 Strings and Languages 105
7.1 Strings 106
7.1.1 The *empty* string ε 106
7.1.2 Length, character at index, and substring of a string 107
7.1.3 Concatenation of strings 107
7.2 Languages 107
7.2.1 How many languages are there? 108
7.2.2 Orders for Strings 108
7.2.3 Operations on languages 110
7.2.4 Concatenation and exponentiation 110
7.2.5 Kleene Star, '$*$' 112
7.2.6 Complementation 112
7.2.7 Reversal 113
7.2.8 Homomorphism 113
7.2.9 Ensuring homomorphisms 113
7.2.10 Inverse homomorphism 114
7.2.11 An Illustration of homomorphisms 115
7.2.12 Prefix-closure 115

Exercises 116

8 Machines, Languages, DFA 119
8.1 Machines 119
8.1.1 The DFA 121
8.1.2 The "power" of DFAs 126
8.1.3 Limitations of DFAs 126
8.1.4 Machine types that accept non-regular languages . 127
8.1.5 Drawing DFAs neatly 129

Exercises 130

9 NFA and Regular Expressions 133
9.1 What is Nondeterminism? 135
9.1.1 How nondeterminism affects automaton operations 136
9.1.2 How nondeterminism affects the power of machines 137
9.2 Regular Expressions 137
9.3 Nondeterministic Finite Automata 141
9.3.1 Nondeterministic Behavior Without ε 141
9.3.2 Nondeterministic behavior with ε 143
9.3.3 *Eclosure* (also known as ε-closure) 145
9.3.4 Language of an NFA 147

9.3.5 A detailed example: telephone numbers 149
9.3.6 Tool-assisted study of NFAs, DFAs, and REs 151

Exercises 157

10 Operations on Regular Machinery 159
10.1 NFA to DFA Conversion 159
10.2 Operations on Machines 161
10.2.1 Union 162
10.2.2 Intersection 165
10.2.3 Complementation 165
10.2.4 Concatenation 165
10.2.5 Star 166
10.2.6 Reversal 167
10.2.7 Homomorphism 168
10.2.8 Inverse Homomorphism 168
10.2.9 Prefix-closure 169
10.3 More Conversions 169
10.3.1 RE to NFA 169
10.3.2 NFA to RE 170
10.3.3 Minimizing DFA 174
10.4 Error-correcting DFAs 176
10.4.1 DFA constructed using error strata 177
10.4.2 DFA constructed through regular expressions 177
10.5 Ultimate Periodicity and DFAs 179

Exercises 181

11 The Automaton/Logic Connection, Symbolic Techniques 185
11.1 The Automaton/Logic Connection 186
11.1.1 DFA can 'scan' and also 'do logic' 186
11.2 Binary Decision Diagrams (BDDs) 187
11.3 Basic Operations on BDDs 191
11.3.1 Representing state transition systems 192
11.3.2 Forward reachability 193
11.3.3 Fixed-point iteration to compute the least fixed-point 194
11.3.4 An example with multiple fixed-points 197
11.3.5 Playing tic-tac-toe using BDDs 198

Exercises 200

12 The 'Pumping' Lemma 205
12.1 Pumping Lemmas for Regular Languages 205
12.1.1 A stronger incomplete Pumping Lemma 209
12.2 An adversarial argument 210
12.3 Closure Properties Ameliorate Pumping 211
12.4 Complete Pumping Lemmas 212
12.4.1 Jaffe's complete Pumping Lemma 212
12.4.2 Stanat and Weiss' complete Pumping Lemma 213

Exercises 213

13 Context-free Languages 217
13.1 The Language of a CFG 218
13.2 Consistency, Completeness, and Redundancy 220
13.2.1 More consistency proofs 223
13.2.2 Fixed-points again! 224
13.3 Ambiguous *Grammars* 225
13.3.1 If-then-else ambiguity 226
13.3.2 Ambiguity, inherent ambiguity 227
13.4 A Taxonomy of Formal Languages and Machines 228
13.4.1 Non-closure of CFLs under complementation 230
13.4.2 Simplifying CFGs 232
13.5 Push-down Automata 234
13.5.1 DPDA versus NPDA 235
13.5.2 Deterministic context-free languages (DCFL) 235
13.5.3 Some Factoids 236
13.6 Right- and Left-Linear CFGs 237
13.7 Developing CFGs 239
13.8 A Pumping Lemma for CFLs 239

Exercises 242

14 Push-down Automata and Context-free Grammars . . . 245
14.1 Push-down Automata 245
14.1.1 Conventions for describing PDAs 247
14.1.2 Acceptance by final state 247
14.1.3 Acceptance by empty stack 249
14.1.4 Conversion of P_1 to P_2 ensuring $L(P_1) = N(P_2)$ . . 251
14.1.5 Conversion of P_1 to P_2 ensuring $N(P_1) = L(P_2)$ . . 251
14.2 Proving PDAs Correct Using Floyd's Inductive Assertions 253
14.3 Direct Conversion of CFGs to PDAs 254
14.4 Direct Conversion of PDAs to CFGs 257

14.4.1 Name non-terminals to match stack-emptying possibilities . . . 258
14.4.2 Let start symbol S set up all stack-draining options 258
14.4.3 Capture how each PDA transition helps drain the stack . . . 259
14.4.4 Final result from Figure 14.6 . . . 260
14.5 The Chomsky Normal Form . . . 262
14.5.1 Cocke-Kasami-Younger (CKY) parsing algorithm . 262
14.6 Closure and Decidability . . . 264
14.7 Some Important Points Visited . . . 264
14.7.1 Chapter Summary – Lost Venus Probe . . . 267

Exercises . . . 268

15 Turing Machines . . . 271
15.1 *Computation:* Church/Turing Thesis . . . 272
15.1.1 "Turing machines" according to Turing . . . 273
15.2 Formal Definition of a Turing machine . . . 273
15.2.1 Singly- or doubly-infinite tape? . . . 275
15.2.2 Two stacks+control = Turing machine . . . 276
15.2.3 Linear bounded automata . . . 277
15.2.4 Tape vs. random access memory . . . 278
15.3 Acceptance, Halting, Rejection . . . 278
15.3.1 "Acceptance" of a TM closely examined . . . 278
15.3.2 Instantaneous descriptions . . . 279
15.4 Examples . . . 279
15.4.1 Examples illustrating TM concepts and conventions 280
15.4.2 A DTM for $w\#w$ . . . 282
15.5 NDTMs . . . 282
15.5.1 Guess and check . . . 283
15.5.2 An NDTM for ww . . . 286
15.6 Simulations . . . 286
15.6.1 Multi-tape vs. single-tape Turing machines . . . 286
15.6.2 Nondeterministic Turing machines . . . 286
15.6.3 The Simulation itself . . . 287

Exercises . . . 288

16 Basic Undecidability Proofs . . . 291
16.1 Some Decidable and Undecidable Problems . . . 292
16.1.1 An assortment of decidable problems . . . 292
16.1.2 Assorted undecidable problems . . . 294

16.2 Undecidability Proofs 295
16.2.1 Turing recognizable (or recursively enumerable) sets 295
16.2.2 Recursive (or decidable) languages 297
16.2.3 Acceptance (A_{TM}) is undecidable (important!) . . . 298
16.2.4 Halting ($Halt_{TM}$) is undecidable (important!) 299
16.2.5 Mapping reductions 301
16.2.6 Undecidable problems are "A_{TM} in disguise" 305

Exercises 306

17 Advanced Undecidability Proofs 309
17.1 Rice's Theorem 309
17.2 Failing proof attempt 310
17.2.1 Corrected proof 311
17.2.2 Greibach's Theorem 312
17.3 The Computation History Method 312
17.3.1 Decidability of LBA acceptance 313
17.3.2 Undecidability of LBA language emptiness 313
17.3.3 Undecidability of PDA language universality 313
17.3.4 Post's correspondence problem (PCP) 315
17.3.5 PCP is undecidable 316
17.3.6 Proof sketch of the undecidability of PCP 317

Exercises 320

18 Basic Notions in Logic including SAT 323
18.1 Axiomatization of Propositional Logic 324
18.2 First-order Logic (FOL) and Validity 326
18.2.1 A warm-up exercise 326
18.2.2 Examples of interpretations 327
18.2.3 Validity of first-order logic is undecidable 329
18.2.4 Valid FOL formulas are enumerable 331
18.3 Properties of Boolean Formulas 331
18.3.1 Boolean satisfiability: an overview 331
18.3.2 Normal forms 332
18.3.3 Overview of direct DNF to CNF conversion 333
18.3.4 CNF-conversion using gates 336
18.3.5 DIMACS file encoding 337
18.3.6 Unsatisfiable CNF instances 339
18.3.7 3-CNF, $\neq$-satisfiability, and general CNF 340
18.3.8 2-CNF satisfiability 341

Exercises 342

19 Complexity Theory and NP-Completeness 345
19.1 Examples and Overview 346
19.1.1 The traveling salesperson problem 346
19.1.2 P-time deciders, robustness, and 2 vs. 3 346
19.1.3 A note on complexity measurement 347
19.1.4 The robustness of the Turing machine model 348
19.1.5 Going from "2 to 3" changes complexity 348
19.2 Formal Definitions 348
19.2.1 NP viewed in terms of verifiers 348
19.2.2 Some problems are outside NP 349
19.2.3 NP-complete and NP-hard 350
19.2.4 NP viewed in terms of deciders 350
19.2.5 An example of an NP decider 351
19.2.6 Minimal input encodings 353
19.3 NPC Theorems and proofs 353
19.3.1 NP-Completeness of 3-SAT 354
19.3.2 Practical approaches to show NPC 358
19.4 NP-Hard Problems can be Undecidable (Pitfall) 360
19.4.1 Proof that Diophantine Equations are NPH 360
19.4.2 "Certificates" of Diophantine Equations 361
19.4.3 What other complexity measures exist? 362
19.5 NP, CoNP, etc. 362

Exercises 365

20 DFA for Presburger Arithmetic 369
20.1 Presburger Formulas and DFAs 371
20.1.1 Presburger formulas 371
20.1.2 Encoding conventions 373
20.1.3 Example 1 — representing $x \leq 2$ 373
20.1.4 Example 2 — $\forall x.\exists y.(x + y) = 1$ 375
20.1.5 Conversion algorithm: Presburger formulas to automata 376
20.2 Pitfalls to Avoid 378
20.2.1 The restriction of equal bit-vector lengths 379

Exercises 380

21 Model Checking: Basics 381
21.1 An Introduction to Model Checking 381
21.2 What Are Reactive Computing Systems? 384
21.2.1 Why model checking? 385
21.2.2 Model checking vs. testing 387
21.3 Büchi automata, and Verifying Safety and Liveness 389
21.4 Example: Dining Philosophers 390
21.4.1 Model (proctype) and property (never) automata . 394

Exercises 396

22 Model Checking: Temporal Logics 399
22.1 Temporal Logics 399
22.1.1 Kripke structures 399
22.1.2 Computations vs. computation trees 401
22.1.3 Temporal formulas are Kripke structure classifiers! 403
22.1.4 LTL vs. CTL through an example 405
22.1.5 LTL syntax 406
22.1.6 LTL semantics 406
22.1.7 CTL syntax 407
22.1.8 CTL semantics 408

Exercises 414

23 Model Checking: Algorithms 419
23.1 Enumerative CTL Model Checking 419
23.2 Symbolic Model Checking for CTL 421
23.2.1 EG p through fixed-point iteration 421
23.2.2 Calculating EX and AX 424
23.2.3 LFP and GFP for 'Until' 425
23.2.4 LFP for 'Until' 426
23.2.5 GFP for Until 427
23.3 Büchi Automata and LTL Model Checking 428
23.3.1 Comparing expressiveness 428
23.3.2 Operations on Büchi automata 430
23.3.3 Nondeterminism in Büchi automata 431
23.4 Enumerative Model Checking for LTL 432
23.4.1 Reducing verification to Büchi automaton emptiness 433

Exercises 436

24 Conclusions 439

Book web site and tool information 443
A.1 Web site and e-mail address 443
A.2 Software tool usage per chapter 443
A.3 Possible Syllabi 444

BED Solution to the tic-tac-toe problem 447

References 453

Index 461

List of Figures

1.1 The power of various machines, and how to realize them 4

3.1 Proof of the Schröder-Bernstein Theorem 46
3.2 Proof approach 47

4.1 Some example binary relations 54
4.2 The binary relation *Power* is shown. The dotted edges are *some* of the edges implied by transitivity. Undotted and dotted means the same in this diagram. Therefore, *Power* actually contains: (i) the pairs corresponding to the solid edges, (ii) the pairs indicated by the dotted edges, (iii) and those pairs indicated by those dotted transitive edges not shown. 61
4.3 The equivalence relation $Power \cap Power^{-1}$ 63
4.4 The identity relation over MT 64

8.1 A DFA that recognizes strings over $\{0, 1\}$ ending with 1 . 121
8.2 Pseudocode for the DFA of Figure 8.1 122
8.3 Multiple symbols labeling a transition in lieu of multiple transitions 122
8.4 A DFA with unreachable (redundant) states 122
8.5 Another example DFA 123
8.6 Drawing DFAs using `dot` 129
8.7 Processing .ps files in Latex 129

9.1 Minimal DFAs for L_k for $k = 2, 3, 4, 5$, with 2^{k+1} states . 140
9.2 An NFA for $(0 + 1)^* \; 0 \; (0 + 1)^k$, for $k = 2$ 141
9.3 An NFA for $((0 + 1)^* \; 0 \; (0 + 1)^k)^+$, for $k = 2$ 143
9.4 An example NFA 145

9.5 DFA obtained using `grail` 145
9.6 Legal and illegal phone numbers in our example 149
9.7 An NFA formally specifying allowed telephone numbers, and the RE describing it 150
9.8 A minimal DFA for the NFA of Figure 9.7 151
9.9 Skeleton of an encoding of the telephone number syntax in `flex` 152

10.1 NFA to DFA conversion illustrated on the NFA of Figure 9.3 160
10.2 $\text{DFA}_{(a)} \cup \text{DFA}_{(b)} = \text{DFA}_{(c)}$; $\text{DFA}_{(a)} \cap \text{DFA}_{(b)} = \text{DFA}_{(d)}$; $\overline{\text{DFA}_{(a)}} = \text{DFA}_{(e)}$ 163
10.3 $\text{NFA}_{(c)} = \text{NFA}_{(a)} \cup \text{NFA}_{(b)}$; $\text{NFA}_{(d)} = \text{NFA}_{(a)} \circ \text{NFA}_{(b)}$. 164
10.4 The result of Starring Figure 10.3(c) 166
10.5 The result of reversing the NFA of Figure 10.4 167
10.6 An example NFA to be converted to a regular expression 171
10.7 Result of the preprocessing step 171
10.8 Result of eliminating state A1 172
10.9 Result of Eliminating B0 and A2 172
10.10 Result of Eliminating FA 173
10.11 (i) Example for DFA minimization, (ii) initial table for DFA minimization, (iii) steps in DFA minimization, and (iv) the final minimized DFA 175
10.12 A DFA that has two error strata implementing all strings that are a Hamming distance of 2 away from the language `(0101)+` 177
10.13 Minimal DFAs for the same language are isomorphic 178
10.14 A regular expression for all 0-, 1-, and 2-bit errors 179

11.1 Minimal DFAs for $d \vee (a \wedge b \wedge c)$ for (a) variable ordering *abcd*, and (b) *dabc*. The edges show transitions for inputs arriving according to this order. 188
11.2 Minimal DFAs where the variable ordering matters 189
11.3 BDD for $a = b \wedge c = d \wedge e = f$ for variable order *acebdf* . 190
11.4 Simple state transition system (example SimpleTR) 192
11.5 BED commands for reachability analysis on SimpleTR, and the fixed-point iteration leading up to the least fixed-point that denotes the set of reachable states starting from `I` 196

11.6 Example where multiple fixed-points exist. This figure shows attainment of a fixed-point $a \vee b$ which is between the least fixed-point of $a \wedge b$ and the greatest fixed-point of 1. The figure shows the initial approximant P0 and the next approximant P1 198

13.1 (a) The parse tree for string 110 with respect to CFG G_2; (b) and (c) are parse trees for 110 with respect to G_4.219
13.2 A string that does not cause zero-crossings. The numbers below the string indicate the running difference between the number of a's and the number of b's at any point along the string 221
13.3 A string that causes zero-crossings 221
13.4 Parse tree for $4 + 5 * 6$ with respect to G_8. 226
13.5 The Chomsky hierarchy and allied notions 228
13.6 A PDA for the language $L_0 = \{ww^R \mid w \in \{0,1\}^*\}$ 235
13.7 Depiction of a parse tree for the CFL Pumping Lemma. The upper drawing shows a very long path that repeats a non-terminal, with the lowest two repetitions occurring at V_2 and V_1 (similar to `Occurrence-1` and `Occurrence-2` as in the text). With respect to this drawing: (i) the middle drawing indicates what happens if the derivation for V_2 is applied in lieu of that of V_1, and (ii) the bottom drawing depicts what happens if the derivation for V_2 is replaced by that for V_1, which, in turn, contains a derivation for V_2 241

14.1 Transition table and transition graph of a PDA for the language $L_0 = \{ww^R \mid w \in \{0,1\}^*\}$, and an illustration of the $\vdash$ relation on input 001100 248
14.2 The PDA of Figure 13.6 converted to one that accepts by empty stack. There are some redundancies in this PDA owing to our following a standard construction procedure. 250
14.3 A PDA for $L_{a^m b^n c^k}$ of Illustration 13.2.1 252
14.4 A PDA whose language is being proved correct using Floyd's method 253
14.5 CFG to PDA conversion for the CFG of Illustration 13.2.1256
14.6 PDA to CFG conversion. Note that `e` means the same as ε. 257
14.7 Steps of the CKY parsing algorithm on input 001 263
14.8 Steps of the CKY parsing algorithm on input aabbab . . . 266

15.1 A TM for $\{a^n b^n c^n \mid n \geq 0\}$, with start state q0, final state q6, and moves occurring from the row-states to column-states . . . 281
15.2 A Turing machine for $w\#w$ . . . 281
15.3 A Nondeterministic Turing machine for ww. Letter 'S' means that the head stays where it is . . . 285

16.1 A_{TM} to $Halt_{TM}$ reduction. Notice that *if* we assume that the inner components – namely the OR-gate, the ability to run M on w, and $D_{Halt_{TM}}$ exist, then D_{ATM} can be constructed; and hence, $D_{Halt_{TM}}$ cannot exist! . . . 299
16.2 Illustration of mapping reduction $A \leq_M B$ . . . 301
16.3 How the mapping reduction from A_{TM} to $Halt_{TM}$ works 302
16.4 Mapping reduction from A_{TM} to E_{TM} . . . 303
16.5 Mapping reduction from A_{TM} to $Regular_{TM}$ . . . 303

17.1 Machine M' in the proof of Rice's Theorem . . . 310
17.2 An accepting computation history and its encoding as per the PCP encoding rules . . . 318

18.1 Min/Max terms for *nand*, whose DNF form is $m_0 + m_1 + m_2$ and CNF form is M_3 (the only max term where the function is 0) . . . 332
18.2 A CNF generator, Part-1 (continued in Figure 18.3) . . . 334
18.3 A CNF generator, Part-2 (continued from Figure 18.2) . . 335
18.4 An unsat CNF instance . . . 339

19.1 Venn diagram of the language families P, NP, and NPC; these set inclusions are *proper* if $P \neq NP$ — which is an open question . . . 346
19.2 Diagram illustrating how NPC proofs are accomplished. Definition 19.2 is illustrated by the reduction shown on the left while Definition 19.7 is illustrated on the right. . . 354
19.3 Proof of the Cook-Levin Theorem . . . 355
19.4 The Proof that *Clique* is NPH using an example formula $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (x_1 \vee x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_2)$ . . . 359
19.5 The language families P, NP, and NPC. All these set inclusions are likely to be proper . . . 362

20.1 Prenexing code in Ocaml . . . 372
20.2 Presburger Formula $x \leq 2$ and its DFA . . . 373

20.3 DFA for $x + y = 1$. The transition names are referred to in Section 20.1.5. 374
20.4 NFA for $\exists y.x + y = 1$ and its equivalent DFA 375
20.5 Obtaining the DFA for $\forall x.\exists y.(x + y) = 1$ from the DFA in Figure 20.4 through complementation, hiding, determinization, and complementation 377
20.6 Study of $\forall x.\exists y.\ x - y = 1$ versus $\forall y.\exists x.\ x - y = 1$ 379

21.1 Three dining philosophers 390
21.2 Promela model for three dining philosophers 391
21.3 Message sequence chart showing liveness violation bug . . 395

22.1 Two Kripke structures and some of their computations. In the Kripke structure on the left, the assertion 'Henceforth $(a \vee b)$' is true. 400
22.2 The basic motivation for computation trees 402
22.3 Computation trees for the two Kripke structures of Figure 22.2, as well as the left-hand side Kripke structure of Figure 22.1 402
22.4 AG (EF x) is true, yet there is a computation where x is permanently false 405
22.5 Illustration of basic notions relating to computation trees 409
22.6 Nine diagrams, some of which are Kripke Structures 415

23.1 A Kripke structure for critical sections 420
23.2 Formula and its Subformulas 421
23.3 Algorithm used in CTL model checking illustrated on the "AF" operator 422
23.4 Table for our CTL model checking example 423
23.5 Kripke structures that help understand CTL 423
23.6 BDD for `EX (a != b)` 424
23.7 The Strong Until (U) iteration that reaches a fixed-point 426
23.8 Transition graphs read as an NFA or as an NBA 428
23.9 An NBA accepting finitely many a's (left) and a DBA accepting infinitely many a's (right) 431
23.10 A Kripke structure (left), its corresponding Büchi automata (middle), and a property automaton expressing GFx (right) 432
23.11 System automaton (left), complemented property automaton (middle), and product automaton (right) 432

23.12 Searching of a product graph for cycles: Example product graph (left), group of stacks for the outer DFS (middle), and group of stacks for the inner DFS (right) . . 434

Foreword

It takes more effort to verify that digital system designs are correct than it does to design them, and as systems get more complex the proportion of cost spent on verification is increasing (one estimate is that verification complexity rises as the square of design complexity).

Although this verification crisis was predicted decades ago, it is only recently that powerful methods based on mathematical logic and automata theory have come to the designers' rescue. The first such method was equivalence checking, which automates Boolean algebra calculations. Next came model checking, which can automatically verify that designs have – or don't have – behaviours of interest specified in temporal logic. Both these methods are available today in tools sold by all the major design automation vendors.

It is an amazing fact that ideas like Boolean algebra and modal logic, originating from mathematicians and philosophers before modern computers were invented, have come to underlie computer aided tools for creating hardware designs.

The recent success of 'formal' approaches to hardware verification has lead to the creation of a new methodology: assertion based design, in which formal properties are incorporated into designs and are then validated by a combination of dynamic simulation and static model checking. Two industrial strength property languages based on temporal logic are undergoing IEEE standardisation.

It is not only hardware design and verification that is changing: new mathematical approaches to software verification are starting to be deployed. Microsoft provides windows driver developers with verification tools based on symbolic methods.

Discrete mathematics, logic, automata, and the theory of computability are the basis for these new formal approaches. Although they

have long been standard topics in computer science, the uses made of them in modern verification are quite different to their traditional roles, and need different mathematical techniques. The way they are taught often puts more emphasis on cultivating 'paper and pencil' proof skills, and less on their practical applications and implementation in tools. Topics in logic are often omitted, or taught without emphasizing connections with automata, and without explaining the algorithms (e.g., fixed-point computation) used in verification.

This classroom-tested undergraduate textbook is unique in presenting logic and automata theory as a single subject. Public domain software is used to animate the theory, and to provide a hands-on taste of the algorithms underlying commercial tools. It is clearly written and charmingly illustrated. The author is a distinguished contributor to both theory and to new tool implementation methods.

I highly recommend this book to you as the best route I know into the concepts underlying modern industrial formal verification.

Dr. Michael J.C. Gordon FRS
Professor of Computer Assisted Reasoning
The University of Cambridge Computer Laboratory

Preface

Computation Engineering, Applied Automata Theory and Logic:

With the rapidly evolving nature of Computing, and with multiple new topics vying for slots in today's undergraduate and graduate curricula, "classical" topics such as Automata Theory, Computability, Logic, and Formal Specification cannot fill the vast expanses they used to fill in both the undergraduate and the graduate syllabi. This move is also necessary considering the fact that many of today's students prefer learning *theory* as a tool rather than *theory for theory's sake.* This book keeps the above facts in mind and takes the following fresh approach:

- approaches automata theory and logic as the underlying *engineering mathematics* for *Computation Engineering*,
- attempts to restore the Automaton-Logic connection missing in most undergraduate books on automata theory,
- employs many interactive tools to illustrate key concepts,
- employs humor and directly appeals to intuitions to drive points home,
- covers classical topics such as the Rice's Theorem, as well as modern topics such as Model Checking, Büchi Automata, and Temporal Logics.

We now elaborate a bit further on these points, and then provide a chapter-by-chapter description of the book.

Teach Automata Theory and Logic as if they were Engineering Math:

The computer hardware and software industry is committed to using mathematically based (formal) methods, and realizes that the biggest impediment to the large scale adoption of these methods would be

the lack of adequately trained manpower. In this context, it is crucial that students who go through automata theory and logic courses *retain* what they have learned, and *know how to use* their knowledge. Today's undergraduate textbooks in the area of this book typically emphasize automata theory, and not logic. This runs the risk of imparting a skewed perspective, as *both* these perspectives are needed in practice. Also most of today's books do not include tool-based experiments. Experience shows that tool based experimentation can greatly enhance one's retention.

Restoring the Missing Automaton/Logic Connection:

Automata theory and logic evolved hand-in-hand - and yet, this connection was severed in the 1970's when the luxury of *separate* automata theory and logic courses became possible. Now, the crowded syllabi that once again forces these topics to co-exist may actually be doing a huge favor: providing the opportunity to bring these topics back together! For example, Binary Decision Diagrams (BDD) - central data structures for representing Boolean functions - can be viewed as minimized DFA. One can introduce this connection and then show how finite state machines can be represented and manipulated either using *explicit* state graphs or *implicitly* using BDD based Boolean transition relations. Another example I've employed with great success is that of the formulation and solution of the logical validity of simple sentences from Presburger arithmetic – such as "$\forall xyz : x + y = z$" – using DFA. Here, the use of automaton operations (such as intersection, complementation, and projection) and corresponding operations in logic (such as conjunction, negation, and existential quantification) in the *same setting* helps illustrate the interplay between two intimately related areas, and additionally helps build strong intuitions.

Teaching Through Interactive Tools:

To the best of my knowledge, none of the present-day undergraduate books in automata theory employ interactive tools in any significant manner. This approach tends to give the false impression to students that these are topics largely of theoretical interest, with the only practical examples coming from the area of compiler parsers. We encourage tool usage in the following ways:

- we illustrate the use of the Grail tools, originally from the University of Western Ontario, to illustrate the application of operations on automata in an interactive manner,

- we illustrate the use of the JFLAP tool kit written by Professor Susan Rodger's group at Duke when discussing nondeterministic Turing machines,
- we employ the BED Binary Decision Diagram package of Andersson and Hulgaard in illustrating several aspects of BDD manipulation and fixed-point computation,
- we illustrate the use of Boolean satisfiability tools such as Zchaff and Minisat in leading up to the concept of NP-completeness,
- we illustrate DNF to CNF conversion and obtaining the Prenex Normal Form using simple programs written in the functional language Ocaml, and finally
- we present simple examples using the model checking tool SPIN.

On page 443, we provides the address of the book website where tool-specific instructions will be maintained.

Letting Intuitions be the Guide:

I have found that introducing diagonalization proofs early on can help students tie together many later ideas more effectively. I employ gentle intuitive introductions (e.g., "the US map has more points than hair on an infinite-sized dog").

Many topics become quite clear if demonstrated in multiple domains. For example, I illustrate fixed-point theory by discussing how context-free grammars are recursive language equations. I also introduce fixed-points by pointing out that if one repeatedly photocopies[1] a document, it will often tend towards one of the fixed points of the image transformation function of the photocopy machine(!).

My example to introduce the fact that there are a countably infinite number of C programs consists of pointing out that the following are legal C programs: main(){}, main(){{}}, main(){{{}}}, ... and then I use the Schröder-Bernstein theorem relating this sequence to the sequence of even numbers ≥ 8 (which can then be bijectively mapped to Natural numbers). In another example, I show that the set of C programs are not regular by applying the Pumping Lemma to main(){{...}} and getting a syntax error when the Pumping Lemma mangles the bracketing structure of such C programs! In my opinion, these examples do not diminish rigor, and may succeed in grabbing the attention of many students.

[1] Xerox is still a trademark of Xerox Inc.

Model Checking, Büchi Automata, and Temporal Logics:

From a practical point of view, automata theory and logic play a central role in modeling and verifying concurrent reactive systems. The hardware and software industry employs *model checking* tools to find deep seated bugs in hardware and software. This book closes off with an introduction to the important topic of model checking.

A Tour Through the Book:

Chapter 1 motivates the material in the book, presenting in great detail why the topics it covers are important both in terms of theory and practice.

Chapter 2 begins with the quintessentially important topics of *sets*, *functions*, and *relations*. After going through details such as expressing function signatures, the difference between partial and total functions, we briefly examine the topic of *computable* functions - functions that computers can hope to realize within them. We point out important differences between the terms *procedure* and *algorithm*, briefly touching on the $3x + 1$ problem - a four-line program that confounds scientists despite decades of intense research. In order to permit you to discuss functions concretely, we introduce the *lambda* notation. A side benefit of our introduction to Lambda calculus is that you will be able to study another formal model of computation besides Turing machines (that we shall study later).

Chapter 3 goes through the concept of *cardinality* of sets, which in itself is extremely mentally rewarding, and also reinforces the technique of proof by contradiction. It also sets the stage for defining fine distinctions such as 'all languages,' of which there are "uncountably many" members, and 'all Turing recognizable languages,' of which there are only "countably many" members.

Chapter 4 discusses important classes of binary relations, such as reflexive, transitive, preorder, symmetric, anti-symmetric, partial order, equivalence, identity, universal, equivalence, and congruence (modulo operators).

Chapter 5 provides an intuitive introduction to mathematical logic. We have written this chapter with an eye towards helping you read definitions involving the operators *if*, *if and only if (iff)*, and quantifiers *for all*, and *there exists*. The full import of definitions laden with *if*s and *if and only if*s, as well as quantifiers, is by no means readily apparent - and so it is essential to cultivate sufficient practice. You will see proof by contradiction discussed at a 'gut level' - we encourage you

to play the game of Mastermind, where you can apply this technique quite effectively to win pretty much every time. Nested quantification is reinforced here, as well as while discussing the pumping lemma in Chapter 12. Our message is: *write proofs clearly and lucidly - that way, in case you go wrong, others can spot the mistake and help you.*

Chapter 6 studies the topic of recursion at some depth. Given that a book on automata theory is 'ridden with definitions,' one has to understand carefully when these definitions make sense, and when they end up being 'total nonsense,' owing, say, to being circular or not uniquely defining something. Lambda calculus provides basic notation with which to reason about recursion. We present to you the "friendliest foray into fixed-point theory' that we can muster." We will use fixed-points as a 'one-stop shopping' conceptual tool for understanding a diverse array of topics, including context-free productions and the reachable states of finite-state machines.

In Chapter 7, we begin discussing the notions of *strings* and *languages*. Please, however, pay special attention[2] to "the five most confused objects of automata theory," namely ε, $\emptyset$, $\{\varepsilon\}$, $\{\emptyset\}$, and the equation $\emptyset^* = \{\varepsilon\}$. We discuss the notion of concatenation, exponentiation, 'starring,' complementation, reversal, homomorphism, and prefix-closure applied to languages.

In Chapter 8, we discuss machines, languages, and deterministic finite automata. We construct Deterministic Finite Automata (DFA) for several example languages. One problem asks you to build a DFA that scans a number presented in any number-base b, either most significant digit-first, or least significant digit-first, and shine its green light exactly at those moments when the number scanned so far equals 0 in some modulus k. The latter would be equivalent to division carried out least significant digit-first, with only the modulus to be retained. We will have more occasions to examine the true "power" of DFAs in Chapters 12 and 20. Chapter 8 closes off with a brief study of the limitations of DFA.

Chapter 9 continues these discussions, now examining the crucially important concept of *non-determinism*. It also discusses regular expressions - a syntactic means for describing regular languages. The important topic of Non-deterministic Finite Automaton (NFA) to DFA conversion is also examined.

Automata theory is a readily usable branch of computer science theory. While it is important to obtain a firm grasp of its basic principles

[2] Once you understand the fine distinctions between these objects, you will detect a faint glow around your head.

using paper, pencil, and the human brain, it is quite important that one use automated tools, especially while designing and debugging automata. To paraphrase Professor Dana Scott, *computer-assisted tools are most eminently used as the* telescopes *and* microscopes *of learning - to see farther, to see closer, and to see clearer than the human mind alone can discern.* We demonstrate the use of `grail` tools to generate and verify automata.

In Chapter 10, we discuss operations that combine regular languages, most often yielding new regular languages as a result. We discuss the conversion of NFA to DFA - an important algorithm both theoretically and in practice. We also discuss the notion of *ultimate periodicity* which crystallizes the true power of DFAs in terms of language acceptance, and also provide a tool-based demonstration of this idea.

In Chapter 11, we will begin discussing binary decision diagrams (BDD), which are nothing but minimized DFA for the language of satisfying assignments (viewed as sequences, as we will show) for given Boolean expressions. The nice thing about studying BDDs is that it helps reinforce not only automata-theoretic concepts but also concepts from the area of formal logic. It teaches you a technique widely used in industrial practice, and also paves the way to your later study of the theory of NP-completeness.

In Chapter 12, we discuss the Pumping Lemma. We define the lemma in first-order logic, so that the reader can avoid common confusions, and grasp how the lemma is employed. We also discuss *complete* Pumping Lemmas (regular if and only if certain conditions are met).

In Chapter 13, we present the idea of *context-free languages.* Context-free languages are generated by a *context-free grammar* that consists of *production rules.* By reading production rules as recursive equations, we can actually solve for the context-free language being defined. We will also have occasion to *prove*, via induction, that context-free productions are *sound* and *complete* - that they do not generate a string outside of the language, but do generate all strings within the language. The important notions of *ambiguity* and *inherent ambiguity* will also be studied.

Chapter 14 will introduce our first *infinite state* automaton variety - the *push-down automaton (PDA)* - a device that has a finite-control and exactly one unbounded stack. We will study a method to convert PDA to CFG and vice versa. We will also show how to prove PDAs correct using Floyd's Inductive Assertions method.

In Chapter 15, we study Turing machines (TM). Important notions, such as *instantaneous descriptions* (ID) are introduced in this chapter.

Several Turing machine simulators are available on the web - you are encouraged to download and experiment with them. In this chapter, we will also introduce *linear bounded automata* (LBA) which are TMs where one may deposit new values only in that region of the tape where the original input was presented.

Chapter 16 discusses some of the most profound topics in computer science: the *halting* problem, the notion of semi-decision procedures, and the notion of algorithms. With the introduction of unbounded store, many decision problems will become formally undecidable. In a large number of cases, it will no longer be possible to predict what the machine will do. For example, it will become harder or impossible to tell whether a machine will halt when started on a certain input, whether two machines are equivalent, etc. In this chapter, we will formally state and prove many of these undecidability results. We will present three proof methods: (i) through contradiction, (ii) through reductions from languages not known to be decidable, and (iii) through mapping reductions.

Chapter 17 continues with the notion of undecidability, discussing two additional proof techniques: (i) through the computational history method, and (ii) by employing Rice's theorem. These are advanced proof methods of undecidability that may be skipped during the initial pass through the textbook material. One can proceed to Chapter 18 after finishing Chapter 16 without much loss of continuity.

Chapter 18 sets the stage to discuss the theory of NP completeness. It touches on a number of topics in mathematical logic that will help you better appreciate all the nuances. We briefly discuss a "Hilbert style" axiomatization of propositional logic, once again touching on soundness and completeness. We discuss basic definitions including satisfiability, validity, tautology, and contradiction. We discuss the various "orders" of logic including zeroth-order, first-order, and higher-order. We briefly illustrate that the validity problem of first-order logic is only semi-decidable by reduction from the Post Correspondence problem. We illustrate how to experiment with modern SAT tools. We also cover related ideas such as $\neq$-sat, 2-sat, and satisfiability-preserving transformations.

Chapter 19 discusses the notion of polynomial-time algorithms, computational complexity, and the notion of NP-completeness. We reiterate the importance of showing that the problem belongs to NP, in addition to demonstrating NP-hardness.

In Chapter 20, we introduce a small subset of first-order logic called Presburger arithmetic that enjoys the following remarkable property:

given a formula in this logic, we can build a finite automaton such that the automaton has an accepting run if and only if the formula is satisfiable. The neat thing about this technique is that it reinforces the automaton/logic connection introduced in the previous chapter.

Chapters 21 through 23 introduce temporal logic and model checking. These topics are ripe for introduction to undergraduates and graduates in all areas of computer science, but without all the generality found in specialized books. To this end, in Chapter 21, we provide a history of model checking and also a detailed example. Chapter 22 introduces linear-time temporal logic and Computational Tree Logic, contrasting their expressive power, and exactly why *computation trees* matter. Finally, Chapter 23 presents an enumerative as well as a symbolic algorithm for CTL model checking. We also present an enumerative algorithm for LTL model checking through an example. We introduce Büchi automata, discussing how Boolean operations on Büchi automata are performed, and that non-deterministic Büchi automata are not equivalent to deterministic Büchi automata.

Chapter 24 reviews the material presented in the book. Chapter A lists the book website, software tool related information, and possible syllabi based on this book.

Acknowledgements:

This book was written over three years, with numerous interruptions. I have classroom-tested parts of this book in an undergraduate class entitled Models of Computation (CPSC 3100, Spring 2002), a graduate class entitled Foundations of Computing (CPSC 6100, Spring 2005), and a graduate class entitled Model Checking (CPSC 6964, Fall 2005). Many thanks to students who took these classes (with due apologies for having subjected them to half-done material). It was also offered as one of the references by Professor Konrad Slind when he taught CPSC 3100 during Fall 2005, for which I am grateful.

I am very grateful for many friends, colleagues, and students who have proof-read significant parts of this book and provided very valuable comments. As best as I recall, here is a list of people who deserve special mention: Steven Johnson, John Harrison, Rajeev Alur, Michael J.C. Gordon, Mark Aagaard, Paliath Narendran, Ching-tsun Chou, Panagiotis (Pete) Manolios, Konrad Slind, Joe Hurd, Kathi Fisler, Salman Pervez, Xiaofang Chen, Michael Freedman, Geoff Draper, Vijay Durairaj, Shrish Jain, Vamshi Kadaru, Seungkeol Choe, and Sonjong Hwang (and apologies to those I am forgetting). I would also like to

thank the authors of the computer-aided design tools I have used in this book (listed on page 443).

Thanks also to the support team at Springer, especially Frank Holzwarth and Deborah Doherty, for their timely help. Many thanks to Michael Hackett, Editor, for helping me launch this book project. Special thanks to Carl W. Harris, Senior Editor, Springer, for patiently leading me through all the major stages of this book, and for very patiently working with the reviewers of this book. Thanks, Carl!

I especially thank Geoff Draper, a student in my CPSC 6100 class (Spring 2005) who provided extensive comments, drew the cartoons you see in this book, as well as kindly gave me permission to employ his cartoons in this book. A reader expecting to read "the usual theory book" is in for a surprise: they will find me joking about at various junctures, as I normally do when I lecture. I keep books such as *The Cartoon Guide to Genetics* by Larry Gonick and Mark Wheelis in mind when I engage in these diversions. Such books are immensely important contributions, in that they make an otherwise dull and dry subject come alive, especially for newcomers. In my humble opinion, many more such books are badly needed to describe computer science theory in a plain and accessible manner to students who might otherwise swallow theory as if it were a bitter pill. I have attempted to slant this book in that direction, without compromising the seriousness of the topic. If I have failed in any way to do so, I offer my advance apologies.[3]

I have benefited immensely by reading many contemporary books - notably one by Michael Sipser and one by Dexter Kozen. I have also drawn from Zohar Manna's book *Mathematical Theory of Computation.* I also like to thank the creators of all the computer based tools employed in this book.

Last but not least, my sincere thanks to my wife Kalpana and daughters Kamala and Kajal for all their love, caring, and cheer. Without their understanding and accommodation, I could not have worked on this book. I also thank my parents, whose names are my middle and last names, for their venerable position of bringing me into this world and seeing to it that I lacked nothing.

Salt Lake City, Utah, USA *Ganesh Gopalakrishnan*
April 2006

[3] Even Patch Adams might occasionally have overdone his antics!