SAT-Based Scalable Formal Verification Solutions

Series on Integrated Circuits and Systems

Series Editor: Anantha Chandrakasan
Massachusetts Institute of Technology
Cambridge, Massachusetts

SAT-Based Scalable Formal Verification Solutions
Malay Ganai and Aarti Gupta
ISBN 978-0-387-69166-4, 2007

Ultra-Low Voltage Nano-Scale Memories
Kiyoo Itoh, Masashi Horiguchi and Hitoshi Tanaka
ISBN 978-0-387-33398-4, 2007

Routing Congestion in VLSI Circuits: Estimation and Optimization
Prashant Saxena, Rupesh S. Shelar, Sachin Sapatnekar
ISBN 978-0-387-30037-5, 2007

Ultra-Low Power Wireless Technologies for Sensor Networks
Brian Otis and Jan Rabaey
ISBN 978-0-387-30930-9, 2007

Sub-Threshold Design for Ultra Low-Power Systems
Alice Wang, Benton H. Calhoun and Anantha Chandrakasan
ISBN 978-0-387-33515-5, 2006

High Performance Energy Efficient Microprocessor Design
Vojin Oklibdzija and Ram Krishnamurthy (Eds.)
ISBN 978-0-387-28594-8, 2006

Abstraction Refinement for Large Scale Model Checking
Chao Wang, Gary D. Hachtel, and Fabio Somenzi
ISBN 978-0-387-28594-2, 2006

A Practical Introduction to PSL
Cindy Eisner and Dana Fisman
ISBN 978-0-387-35313-5, 2006

Thermal and Power Management of Integrated Systems
Arman Vassighi and Manoj Sachdev
ISBN 978-0-387-25762-4, 2006

Leakage in Nanometer CMOS Technologies
Siva G. Narendra and Anantha Chandrakasan
ISBN 978-0-387-25737-2, 2005

Statistical Analysis and Optimization for VLSI: Timing and Power
Ashish Srivastava, Dennis Sylvester, and David Blaauw
ISBN 978-0-387-26049-9, 2005

Malay Ganai
Aarti Gupta

SAT-Based Scalable Formal Verification Solutions

Springer

Malay Ganai
NEC Labs America
4 Independence Way
Princeton, NJ 08540
USA

Aarti Gupta
NEC Labs America
4 Independence Way
Princeton, NJ 08540
USA

Series Editor:
Anantha Chandrakasan
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
USA

Library of Congress Control Number: 20077922183

ISBN 0-387-69166-9 e-ISBN 0-387-69167-7
ISBN 978-0-387-69166-4 e-ISBN 978-0-387-69167-1

Printed on acid-free paper.

© 2007 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 they 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.

9 8 7 6 5 4 3 2 1

springer.com

Dedication

This book is dedicated to all
those who continuously strive to
produce better algorithmic and
engineering solutions to
complex verification problems.

Preface

"Engineering is the profession in which a knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind" —Engineers Council for Professional Development (1961/1979)

Functional verification has become an important aspect of the IC (Integrated Chip) design process. Significant resources, both in industry and academia, are devoted to bridge the gap between design complexity and verification efforts. SAT-based verification techniques have attracted both industry and academia equally. This book discusses in detail several latest and interesting SAT-based techniques that have been shown to be scalable in an industry context. Unlike other books on formal methods that emphasize theoretical aspects with dense mathematical notation, this book provides algorithmic and engineering insights into devising scalable approaches for an effective and robust realization of verification solution. We also describe specific strengths of the various approaches in regards to their applicability. This books nicely complements other excellent books on introductory or advanced formal verification primarily in two aspects:

First, with growing interest in SAT-based approaches for formal verification, this book attempts to bring various emerging SAT-based scalable verification techniques and trends under one hood. In the last few years, several new SAT-based techniques have emerged. Not all of these are covered by other books: Hybrid SAT Solver, Efficient Problem Representation, Customized SAT-based Bounded Model Checking, Verification using Efficient Memory Modeling, Distributed SAT and SAT-Bounded

Model Checking, Proof-based Iterative Abstraction, High-level Bounded Model Checking, SAT-based Unbounded Model Checking, and Synthesis for Verification Paradigm.

Second, and more importantly, due to the practical significance of these techniques, they are appropriate for direct implementation in industry settings. In this book, we describe how these techniques have been architected into a verification platform called *VeriSol* (formerly *DiVer*) which has been used successfully in the industry for the last four years. We also share our practical experiences and insights in verifying large industry designs using this platform.

We strongly believe that the techniques described in this book will continue to gain importance in the verification area, given that the verification complexity is growing at an alarming rate with the design complexity. We also believe that that this book will provide useful information about foundation work for future verification technologies.

The book expects the reader to have a basic understanding of formal verification, model checking and issues inherent in model checking. The book primarily targets researchers, scientists and verification engineers who would like to get an in-depth understanding of scalable SAT-based verification techniques that can be further improved. The book also targets CAD tool developers who would like to incorporate various SAT-based advanced techniques in their products. Currently, colleges do not emphasize adequately the algorithmic and engineering aspects of designing a verification tool. Such practices should be encouraged, as a good infrastructure is required to produce quality research. We strongly believe that this book will motivate such activities in the future.

Here is the outline of the book: With an introduction and background on current design verification challenges for model checking techniques in Chapters 1 and 2 respectively, we divide the rest of the book into five parts, each with 1-4 chapters. Part I describes the underlying infrastructure — efficient problem representation and SAT-solvers — to realize scalable verification algorithms. Parts II-IV describe SAT-based model checking algorithms for various verification tasks such as accelerated falsification, robust proof methods, and iterative abstraction/refinement, respectively. Part V gives detail of an industry tool *VeriSol* and several industry cases studies. It also covers future trends in SAT-based model checking such as, synthesis for verification paradigm, and high-level model checkers, to further improve the scalability.

We would like to express our deep gratitude to NEC Laboratories America, Princeton, NJ for providing the opportunities and the infrastructure to carry out the research, and Central Research Laboratories, Tokyo Japan for packaging and deploying our technology to the end-users. Individually we would like to thank Dr. Pranav Ashar for his numerous and valuable insights; Dr. Kazutoshi Wakabayashi and Akira Mukaiyama for stimulating us with the verification challenges that are encountered in an industry setting. We also acknowledge the support of other team members in building the tool: Dr. James Yang, Dr. Lintao Zhang, Dr. Chao Wang, and Dr. Pankaj Chauhan.

We also thank all those who are involved in the publication of the book, especially Carl Harris and his colleagues of Springer Publishing Company. Last, but not the least, we thank our families for their patience during the project.

Dr. MALAY K. GANAI
Dr. AARTI GUPTA

Princeton, New Jersey 2006

Contents

LIST OF FIGURES XIX

LIST OF TABLES XXVII

1 DESIGN VERIFICATION CHALLENGES 1

1.1 INTRODUCTION 1
1.2 SIMULATION-BASED VERIFICATION 1
1.3 FORMAL VERIFICATION 2
1.3.1 Model Checking 3
1.4 OVERVIEW 5
1.5 VERIFICATION TASKS 6
1.6 VERIFICATION CHALLENGES 8
1.6.1 Design Features 8
1.6.2 Verification Techniques 9
1.6.3 Verification Methodology 11
1.7 ORGANIZATION OF BOOK 13

2 BACKGROUND 17

2.1 MODEL CHECKING 17
2.1.1 Correctness Properties 18
2.1.2 Explicit Model Checking 19
2.1.3 Symbolic Model Checking 19
2.2 NOTATIONS 20
2.3 BINARY DECISION DIAGRAMS 22
2.4 BOOLEAN SATISFIABILITY PROBLEM 23

2.4.1 Decision Engine 25
2.4.2 Deduction Engine 26
2.4.3 Diagnosis Engine 28
2.4.4 Proof of Unsatisfiability 29
2.4.5 Further Improvements 30
2.5 SAT-BASED BOUNDED MODEL CHECKING (BMC) 32
2.5.1 BMC formulation: Safety and Liveness Properties 33
2.5.2 Clocked LTL Specifications 36
2.6 SAT-BASED UNBOUNDED MODEL CHECKING 37
2.7 SMT-BASED BMC 39
2.8 NOTES 40

PART I: BASIC INFRASTRUCTURE 41

3 EFFICIENT BOOLEAN REPRESENTATION 43

3.1 INTRODUCTION 43
3.2 BRIEF SURVEY OF BOOLEAN REPRESENTATIONS 45
3.2.1 Extended Boolean Decision Diagrams (XBDDs) 45
3.2.2 Boolean Expression Diagrams (BEDs) 45
3.2.3 AND/INVERTER Graph (AIG) 46
3.3 FUNCTIONAL HASHING (REDUCED AIG) 49
3.3.1 Three-Input Case 50
3.3.2 Four-Input Case 52
3.3.3 Example 54
3.4 EXPERIMENTS 57
3.5 SIMPLIFICATION USING EXTERNAL CONSTRAINTS 60
3.6 COMPARING FUNCTIONAL HASHING WITH BDD/SAT SWEEPING 61
3.7 SUMMARY 62
3.8 NOTES 62

4 HYBRID DPLL-STYLE SAT SOLVER 63

4.1 INTRODUCTION 63
4.2 BCP ON CIRCUIT 65
4.2.1 Comparing CNF- and Circuit-based BCP Algorithms 67
4.3 HYBRID SAT SOLVER 68
4.3.1 Proof of Unsatisfiability 69
4.3.2 Comparison with Chaff 69
4.4 APPLYING CIRCUIT-BASED HEURISTICS 71
4.4.1 Justification Frontier Heuristics 71
4.4.2 Implication Order 72
4.4.3 Gate Fanout Count 73
4.4.4 Learning XOR/MUX Gates 74
4.5 VERIFICATION APPLICATIONS OF HYBRID SAT SOLVER 75

4.6 SUMMARY 75
4.7 NOTES 76

PART II: FALSIFICATION 77

5 SAT-BASED BOUNDED MODEL CHECKING 79

5.1 INTRODUCTION 79
5.2 DYNAMIC CIRCUIT SIMPLIFICATION 81
5.2.1 Notation 82
5.2.2 Procedure Unroll 83
5.2.3 Comparing Implicit with Explicit Unrolling 84
5.3 SAT-BASED INCREMENTAL LEARNING AND SIMPLIFICATION 86
5.4 BDD-BASED LEARNING 90
5.4.1 Basic Idea 90
5.4.2 Procedure: BDD_learning_engine 91
5.4.3 Seed Selection 92
5.4.4 Creation of BDDs 93
5.4.5 Generation of Learned Clauses 94
5.4.6 Integrating BDD Learning with a Hybrid SAT Solver 95
5.4.7 Adding Clauses Dynamically to a SAT Solver 95
5.4.8 Heuristics for Adding Learned Clauses 96
5.4.9 Application of BDD-based Learning 97
5.5 CUSTOMIZED PROPERTY TRANSLATION 98
5.5.1 Customized Translation for *F(p)* 100
5.5.2 Customized Translation of *G(q)* 102
5.5.3 Customized Translation of $F(p \wedge G(q))$ 103
5.6 EXPERIMENTS 104
5.6.1 Comparative Study of Various Techniques 105
5.6.2 Effect of Customized Translation and Incremental Learning 108
5.6.3 Effect of BDD-based Learning on BMC 109
5.6.4 Static BDD Learning 109
5.6.5 Dynamic BDD Learning 110
5.7 SUMMARY 112
5.8 NOTES 112

6 DISTRIBUTED SAT-BASED BMC 113

6.1 INTRODUCTION 113
6.2 DISTRIBUTED SAT-BASED BMC PROCEDURE 114
6.3 TOPOLOGY-COGNIZANT DISTRIBUTED-BCP 116
6.3.1 Causal-effect Order 117
6.4 DISTRIBUTED-SAT 118
6.4.1 Tasks of the Master 119
6.4.2 Tasks of a Client C_i 120

6.5 SAT-BASED DISTRIBUTED-BMC 120
6.6 OPTIMIZATIONS 121
6.6.1 Memory Optimizations in Distributed-SAT 121
6.6.2 Tight Estimation of Communication Overhead 121
6.6.3 Performance Optimizations in Distributed-SAT 123
6.6.4 Performance Optimization in SAT-based Distributed-BMC 124
6.7 EXPERIMENTS 124
6.8 RELATED WORK 128
6.9 SUMMARY 129
6.10 NOTES 129

7 EFFICIENT MEMORY MODELING IN BMC 131

7.1 INTRODUCTION 131
7.2 BASIC IDEA 132
7.3 MEMORY SEMANTICS 134
7.4 EMM APPROACH 135
7.4.1 Efficient Representation of Memory Modeling Constraints 136
7.4.2 Comparison with ITE Representation 139
7.4.3 Non-uniform Initialization of Memory 140
7.4.4 EMM for Multiple Memories, Read, and Write Ports 141
7.4.5 Arbitrary Initial Memory State 143
7.5 EXPERIMENTS ON A SINGLE READ/WRITE PORT MEMORY 144
7.6 EXPERIMENTS ON MULTI-PORT MEMORIES 149
7.6.1 Case Study on Quick Sort 150
7.6.2 Case Study on Industry Design (Low Pass Filter) 151
7.7 RELATED WORK 151
7.8 SUMMARY 152
7.9 NOTES 153

8 BMC FOR MULTI-CLOCK SYSTEMS 155

8.1 INTRODUCTION 155
8.1.1 Nested Clock Specifications 155
8.1.2 Verification Model for Multi-clock Systems 156
8.1.3 Simplification of Verification Model 156
8.1.4 Clock Specification on Latches 157
8.2 EFFICIENT MODELING OF MULTI-CLOCK SYSTEMS 158
8.3 REDUCING UNROLLING IN BMC 160
8.4 REDUCING LOOP-CHECKS IN BMC 161
8.5 DYNAMIC SIMPLIFICATION IN BMC 162
8.6 CUSTOMIZATION OF CLOCKED SPECIFICATIONS IN BMC 163
8.7 EXPERIMENTS 166
8.7.1 VGA/LCD Controller 167
8.7.2 Tri-mode Ethernet MAC Controller 168

8.8 RELATED WORK 169
8.9 SUMMARY 170
8.10 NOTES 171

PART III: PROOF METHODS 173

9 PROOF BY INDUCTION 175

9.1 INTRODUCTION 175
9.2 BMC PROCEDURE FOR PROOF BY INDUCTION 176
9.3 INDUCTIVE INVARIANTS: REACHABILITY CONSTRAINTS 177
9.4 PROOF OF INDUCTION WITH EMM 179
9.5 EXPERIMENTS 180
9.5.1 Use of Reachability Invaraints 180
9.5.2 Case Study: Use of Induction proof with EMM 181
9.6 SUMMARY 182
9.7 NOTES 183

10 UNBOUNDED MODEL CHECKING 185

10.1 INTRODUCTION 185
10.2 MOTIVATION 187
10.3 CIRCUIT COFACTORING APPROACH 188
10.3.1 Basic Idea 188
10.3.2 The Procedure 189
10.3.3 Comparing circuit cofactoring with cube-wise enumeration 190
10.4 COFACTOR REPRESENTATION 191
10.5 ENUMERATION USING HYBRID SAT 192
10.5.1 Heuristics to Enlarge the Satisfying State Set 193
10.6 SAT-BASED UMC 197
10.6.1 SAT-based Existential Quantification using Circuit Cofactor 198
10.6.2 SAT-based UMC for *F(p)* 198
10.6.3 SAT-based UMC for *G(q)* 199
10.6.4 SAT-based UMC for *F(p∧G(q))* 202
10.7 EXPERIMENTS FOR SAFETY PROPERTIES 203
10.7.1 Industry Benchmarks 203
10.7.2 Public Verification Benchmarks 206
10.8 EXPERIMENTS FOR LIVENESS PROPERTIES 207
10.9 RELATED WORK 209
10.10 SUMMARY 211
10.11 NOTES 212

PART IV: ABSTRACTION/REFINEMENT 213

11 PROOF-BASED ITERATIVE ABSTRACTION 215

11.1 INTRODUCTION 215
11.2 PROOF-BASED ABSTRACTION (PBA): OVERVIEW 218
11.3 LATCH-BASED ABSTRACTION 219
11.4 PRUNING IN LATCH INTERFACE ABSTRACTION 222
11.4.1 Environmental Constraints 223
11.4.2 Latch Interface Propagation Constraints 224
11.5 ABSTRACT MODELS 225
11.6 IMPROVING ABSTRACTION USING LAZY CONSTRAINTS 226
11.6.1 Making Eager Constraints Lazy 227
11.7 ITERATIVE ABSTRACTION FRAMEWORK 228
11.7.1 Inner Loop of the Framework 228
11.7.2 Handling Counterexamples 229
11.7.3 Lazy Constraints in Iterative Framework 230
11.8 APPLICATION OF PROOF-BASED ITERATIVE ABSTRACTION 231
11.9 EMM WITH PROOF-BASED ABSTRACTION 232
11.10 EXPERIMENTAL RESULTS OF LATCH-BASED ABSTRACTION 233
11.10.1 Results for Iterative Abstraction 233
11.10.2 Results for Verification of Abstract Models 235
11.11 EXPERIMENTAL RESULTS USING LAZY CONSTRAINTS 236
11.11.1 Results for Use of Lazy Constraints 236
11.11.2 Proofs on Final Abstract Models 239
11.12 CASE STUDY: EMM WITH PBIA 240
11.13 RELATED WORK 242
11.14 SUMMARY 243
11.15 NOTES 243

PART V: VERIFICATION PROCEDURE 245

12 SAT-BASED VERIFICATION FRAMEWORK 247

12.1 INTRODUCTION 247
12.2 VERIFICATION MODEL AND PROPERTIES 248
12.3 VERIFICATION ENGINES 250
12.4 VERIFICATION ENGINE ANALYSIS 254
12.5 VERIFICATION STRATEGIES: CASE STUDIES 256
12.6 SUMMARY 261
12.7 NOTES 261

13 SYNTHESIS FOR VERIFICATION 263

13.1 INTRODUCTION 263
13.2 CURRENT METHODOLOGY 265
13.3 SYNTHESIS FOR VERIFICATION PARADIGM 267
13.4 HIGH-LEVEL VERIFICATION MODELS 269
13.4.1 High-level Synthesis (HLS) 269

13.4.2 Extended Finite State Machine (EFSM) Model 269
13.4.3 Flow Graphs 271
13.5 "BMC-FRIENDLY" MODELING ISSUES 272
13.6 SYNTHESIZING "BMC-FRIENDLY" MODELS 273
13.7 EFSM LEARNING 274
13.7.1 Extraction: Control State Reachability (CSR)...............................274
13.7.2 On-the-Fly Simplification 275
13.7.3 Unreachablility of Control States 277
13.8 EFSM TRANSFORMATIONS 277
13.8.1 Property-based EFSM Reduction...278
13.8.2 Balancing Re-convergence..278
13.8.3 Balancing Re-convergence without Loops....................................280
13.8.4 Balancing Re-convergence with Loops...282
13.9 HIGH-LEVEL BMC ON EFSM..285
13.9.1 Expression Simplifier..286
13.9.2 Incremental Learning in High-level BMC 287
13.10 EXPERIMENTS 287
13.10.1 Controlled Case Study 287
13.10.2 Experiments on Industry Software bc-1.06 289
13.10.3 Experiments on Industry Embedded System Software.............292
13.10.4 Experiments on System-level Model..293
13.11 SUMMARY AND FUTURE WORK 294
13.12 NOTES 295

REFERENCES 297

GLOSSARY 309

INDEX 317

ABOUT THE AUTHORS 325

List of Figures

Fig. 2.1: Symbolic forward traversal algorithm for safety property.........................20

Fig. 2.2: Effect of variable ordering on BDD size: (a) with ordering $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$, and (b) with ordering $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$. A solid line — denotes a then-branch and a dashed line — denotes an else-branch. 23

Fig. 2.3: DPLL-style SAT Solver...24

Fig. 2.4: Lazy update of a CNF clause during BCP 27

Fig. 2.5: Example for Conflict Analysis...28

Fig. 2.6: BMC check (a) No loop (b) Loop 33

Fig. 2.7: Debug trace for $\boldsymbol{G(p \rightarrow F(\neg q))}$ with fairness f 34

Fig. 2.8: Example: Timing diagram for clocked specification 37

Fig. 2.9: SAT-based existential quantification using cube enumeration 38

Fig. 2.10: Least fixed-point computation using SAT 39

Fig. 2.11: SMT-based BMC...40

Fig. 3.1: Example for graph comparison: (a) a netlist with output g=$(a \vee b) \wedge abc$, (b) equivalent BED representation. The symbol $\neg$ denotes negation 46

Fig. 3.2: Pseudo-code for 2-input structural hashing....47

Fig. 3.3: Example for circuit graph construction: (a) a functionally redundant Miter structure generated to check functional equivalence of outputs x and y, (b) circuit graph using structural hashing constructed according to [135] with two-input vertices, (c) circuit graph with four-input vertices. 48

Fig. 3.4: Pseudo-code for *And* using functional hashing....50

Fig. 3.5: Pseudo-code for *3_input_signature* and *And_3* 51

Fig. 3.6: A 3-input example: (a) pre-lookup structure $sig_f = \langle 10100110 \rangle$ and $sig_h = \langle 11100001 \rangle$ (b) post-lookup structure $f=b$ and $h=\neg b$....52

Fig. 3.7: Pseudo-code for *4_input_signature* and *And_4* 53

Fig. 3.8: Local reduction of output of the netlist in Figure 3.1 using procedure *And_4* of Figure 3.7, (a) reduction of using case 144, (b) reduction of using case 245 and then recursively using case 144 54

Fig. 3.9: Example of Figure 3.3 for circuit graph construction using multi-level hashing: (a) vertices 1–5 are identical to Figure 3.3(b), (b) vertices 6 and 7 yield more compact structures, (c) vertices 8–10 yield no compaction, (d) vertices 5', 1''', and 7' are merged with vertices 5, 1, and 7, respectively....55

Fig. 3.10: Miter structure of Industry Example (a) without hashing (b) with two-input (or 2-level) hashing scheme (c) with multi-level hashing scheme....56

Fig. 3.11: Distribution of circuit sizes....57

Fig. 3.12: AIG size comparison using multi-level vs 2-input hashing: (a) Compression ratio (ratio of the size of circuit built by multi-level to that by 2-input), (b) Time overhead (ratio of the time taken for circuit building by multi-level hashing to that by 2-input). 58

Fig. 3.13: Graph size comparison AIG with BEDs: (a) Reduction ratio (ratio of the size of circuit built by multi-level hashing to that by BEDs), (b) Time overhead (ratio of the time taken for circuit building by multi-level hashing to that by BEDs) 59

Fig. 3.14: Fraction of Miters solved: (a) comparison between multi-level and 2-input hashing (for the example indicated, two-input hashing solves 31% while multi-level hashing solves 86% Miters), (b) comparison between multi-level and BEDs (for the example indicated, BED solves 29% while multi-level hashing solves 87% Miters) 60

Fig. 3.15: Pseudo-code for structural merging 61

Fig. 4.1: a) Circuit-based SAT, b) Circuit-based learning 65

Fig. 4.2: Circuit-based BCP 66

Fig. 4.3: 2-input AND Lookup Table for Fast Implication Procedure 67

Fig. 4.4: Comparison of BCP time per million implications of gate clauses between CNF-based using *lazy update* and Circuit-based Solver using *fast table lookup*. 68

Fig. 4.5: Comparison of BCP time per million implications of gate and learned clauses between CNF-based solver, Chaff and hybrid solver 70

Fig. 4.6: Comparison of total solver time of Chaff and hybrid solver (both using same decision heuristics) 70

Fig. 4.7: Comparison of SAT time of Chaff and hybrid solver using justification frontier heuristics (jft) 72

Fig. 4.8: Example of counting literals in Clauses 74

Fig. 4.9: Learning of CNF-clauses in Hybrid SAT (a) XOR gate, (b) MUX gate 74

Fig. 5.1: Iterative array model of synchronous circuit 80

Fig. 5.2: A small design example. The initial values for X_1 and X_2 are both 0. 83

Fig. 5.3: Pseudo-code for *Unroll* Procedure 84

Fig. 5.4: Comparison of unrolled graph sizes between *explicit* and *implicit* approach on following examples: (a) *m_ciu*, (b) *ifpf*, (c) *larx*, (d) *b17* 85

Fig. 5.5: Constraint sharing of SAT instances S_1 and S_2 87

Fig. 5.6: *L3* Learning: Merging unsatisfiable F^k to *const_0* for $0 \le k \le d$ 88

Fig. 5.7: Use of Incremental SAT techniques on an example formula $\Phi = \varphi_1 \vee (\neg\varphi_2 \wedge \varphi_3)$ 89

Fig. 5.8: Example for BDD Learning 91

Fig. 5.9: BDD-based Learning Technique 92

Fig. 5.10: BDD from seed a) Fanin Cone, b) Region around the seed 94

Fig. 5.11: Unrolling of property tree node *p* 100

Fig. 5.12: BMC Customization for ***F****(p)* 101

Fig. 5.13: BMC Customization for *G(q)* with fairness constraints *B* 102

Fig. 5.14: BMC Customization for ***F****(p∧****G****(q))* with fairness constraints 104

Fig. 5.15: Cumulative improvements of BMC using various techniques on HD3 108

Fig. 5.16: Static Learning a) with time limit 3hrs for D1, D2, and D4, b) with BMC bound limit and normalized time for D3, D5, and D6 110

Fig. 5.17: Dynamic Learning: a) Depth limit with normalized time for D1-D6, b) Time limit on D1, D2 and D4 111

Fig. 6.1: Partitioning of Unrolled Circuit 114

Fig. 6.2: Distribtued DPLL-style SAT Solver 115

Fig. 6.3: Distributed-SAT and SAT-based Distributed-BMC 119

Fig. 6.4: Timeline for a tight estimation of communication overhead 122

Fig. 7.1: Design with embedded memory 132

Fig. 7.2: Unrolled design with embedded memory 133

Fig. 7.3: Typical memory access timing diagram 134

Fig. 7.4: Conventional way of adding constraints using ITE operator 135

Fig. 7.5: BMC augmented with EMM constraints 137

Fig. 7.6: Efficient Modeling of Memory Constraint 138

Fig. 7.7: Quadratic growth of EMM with BMC depth k 140

Fig. 7.8: Stack implementation using RAM 144

Fig. 7.9: Software programs with embedded stack used in experiments 145

Fig. 8.1: (a) Multi-clock system, (b) Single-clock model 159

Fig. 8.2: Enable clock signal circuit 159

Fig. 8.3: Modeling complex clock constructs 160

Fig. 8.4: Ticks of global clock *gclk* 161

Fig. 8.5: Procedure to get next clock tick depth 164

Fig. 8.6: Unrolling of clocked property tree node p 165

Fig. 8.7: BMC Customization for clocked Property (***F**(p))@clk)* 165

Fig. 9.1: BMC-based inductive proof steps (a) Base (b) Induction on $\neg$***F**(p)* 175

Fig. 9.2: Customized BMC with Induction Proof 176

Fig. 9.3: BDD traversal: (a) backward from the bad states B_0, (b) forward from Initial states F_0 178

Fig. 9.4: BDD to circuit structure 179

Fig. 9.5: Induction Proof using reachability constraints 179

Fig. 10.1: An example circuit 187

Fig. 10.2: Cofactor-based quantification with minterm (a) $\{u_1=1, u_2=1\}$ (b) $\{u_1=1, u_2=0\}$ 190

Fig. 10.3: Example of circuit-graph reuse in circuit-based cofactor quantification 192

Fig. 10.4: Circuit-based cofactor quantification using heuristic *H1* 194

Fig. 10.5: Procedure *Get_more_frontiers* 195

Fig. 10.6: Circuit-based cofactor quantification using heuristic *H2* 196

Fig. 10.7: Circuit-based cofactor quantification using heuristic *H3* 196

Fig. 10.8: SAT-based existential quantification using circuit cofactor 198

Fig. 10.9: Least fixed-point computation using circuit cofactor 199

Fig. 10.10: Least fixed-point computation using unrolling 199

Fig. 10.11: Pre-image computation using unrolling 200

Fig. 10.12: Greatest fixed-point computation using circuit cofactoring 201

Fig. 10.13: Greatest fixed-point computation using unrolling 202

Fig. 10.14: Customized translation for $\boldsymbol{F}(p \wedge \boldsymbol{G}(q))$ 203

Fig. 10.15: a) CC vs BDD on VVB, b) CC vs BC on VVB, c) CC vs BMC on VVB, d) CC vs BC vs BDD on swap 207

Fig. 11.1: Customized BMC with PBA for $\boldsymbol{F}(p)$ 219

Fig. 11.2: Latch-based abstraction 221

Fig. 11.3: Example for pruning the set *R(k)* 223

Fig. 11.4: Example of PI logic 225

Fig. 11.5: Iterative abstraction framework 228

Fig. 11.6: Handling counterexamples 229

Fig. 12.1: Verification Framework: *VeriSol* 248

Fig. 12.2: Combinational-Loop Breaker 249

Fig. 12.3: Modeling multi-clock system 250

Fig. 12.4: Wheel of verification engine 251

Fig. 12.5: Verification flow without (a) / with (b) embedded memory 257

Fig. 12.6: Verification flow for example *Industry I* 258

Fig. 12.7: Verification flow on example *Industry II* 259

Fig. 12.8: Verification flow on design *Industry III* 260

Fig. 12.9: Verification flow on example *Quicksort* 260

Fig. 13.1: Current Methodology: HLS with Verification 267

Fig. 13.2: Another dimension to HLS: Verification 267

Fig. 13.3: Synthesis for Verification Paradigm 268

Fig. 13.4: STG of EFSM M 271

Fig. 13.5: STG of transformed EFSM M' 280

Fig. 13.6: Pseudo-code of *Balance_path* 282

Fig. 13.7: Execution steps of Balancing Re-convergence on an example: a) Reducible Flow graph $G(V,E,v_1)$ where *i* represents the node v_i, b) DAG $G(V,E^f,v_1)$ with edge weights (=1 if not shown) after executing *Balance_path* procedure, c) weights on the back-edges after balancing loops, d) final balanced flow graph after inserting *n-1* NOP states for edge with weight *n* 285

Fig. 13.8: Accelerated High-level BMC (HMC) 286

Fig. 13.9: Integrating HLS with verification at high-level 295

List of Tables

Table 5.1: Customized BMC Verification Results 106

Table 5.2: Effect of Incremental Learning and BMC Customization 108

Table 5.3: Effect of Incremental Learning and BMC Customization 109

Table 6.1: Memory Utilization in distributed SAT-based BMC............................125

Table 6.2: Performance evauation in distributed SAT-based BMC.......................126

Table 6.3: Comparison of non-distributed (mono) and distributed BMC on Industry designs..128

Table 7.1: Comparison of # clauses and gates in EMM constraints 140

Table 7.2: Comparison of memory modeling on *Fibonacci* model (AW=12, DW=32)...147

Table 7.3: Comparison of memory modeling on *3n+1* model (AW=12, DW=2)..148

Table 7.4: Comparison of memory modeling on *Towers-of-Hanoi* (AW=12, DW=22)...148

Table 7.5: Comparison of memory modeling on industrial design (AW=12, DW=12)..149

Table 7.6: Comparison of memory modeling (for *3n+1*) with DW=12 and varying AW.......149

Table 7.7: Comparision of EMM vs Explicit memory modeling on *Quick Sort*.......150

Table 8.1: Unrolled circuit nodes for the multi-clock Example 8.1(a) 163

Table 8.2: Comparative evaluation of BMC for clocked properties on VGA_LCD.......168

Table 8.3: Comparative evaluation of BMC for clocked properties on Ethernet MAC 169

Table 9.1: Results of proof by induction using BDD-based reachability invariant.......181

Table 10.1: Performance summary of SAT-based UMC methods 203

Table 10.2: Performance comparison of UMC methods: CC vs/ BC 205

Table 10.3: Comparison of heuristics in CC on D2 and proof.......206

Table 10.4: Completeness Bounds for $\boldsymbol{F(p \wedge G(q)}$ 208

Table 10.5: BMC with Completeness Bounds.......209

Table 11.1: Results for iterative abstraction 234

Table 11.2: Results for verification on abstract models.......235

Table 11.3: Abstraction using lazy constraints (first iteration, i=1).......238

Table 11.4: Abstraction using lazy constraints (subsequent iteration, i>0) 239

Table 11.5: Verification on final abstraction model using lazy constraints.......240

Table 11.6: Results of abstraction using EMM with PBA 241

Table 12.1: Description of memory interface signals 249

Table 13.1: Control State Reachability on EFSM (a) M and (b) M’ 280

Table 13.2: Impact of HLS parameters on Verification 288

Table 13.3: Comparison of high-level BMC accelerators 291

Table 13.4: Evaluating the effectiveness of EFSM transformation in
SMT-based BMC 293

1 DESIGN VERIFICATION CHALLENGES

1.1 Introduction

Verification ensures that the design meets the specification and has become an indispensable part of a product development cycle of a digital hardware design. Cost of chip failure is enormous due to high cost of res-pins and delayed tape-out, resulting in loss of opportunity to launch product on time in a highly competitive market. With the increasing design complexity of digital hardware, functional verification has become increasingly on the critical path of the cycle [1], requiring expensive and time-consuming efforts, as much as 70% of the product development cycle. As per the 2002/2004 functional verification study conducted by *Collett International Research* (as reported by *EETimes.com* [2]), functional/logic flaws account for 75% causes for respins of more than two-thirds of IC/ASIC designs to reach volume. Of these 75% flaws, more than 80% are due to design errors and remaining are due to incorrect/incomplete specification, internal and external IPs. Market forces mandate scalable verification solutions and radical shifts in design methodology to overcome the difficulty in verifying complex designs. Not surprisingly, traditional “black-box” verification methodology is giving way to “white-box” verification methodology, where more than half of the engineers in the design team are verification engineers who are getting involved in the early phase of design and specification.

1.2 Simulation-based Verification

Conventionally, designs are verified using extensive simulation. A model of the design is built in software, to which small monitors are added. These monitors check for failures of the design assertions. Large numbers of input sequences, called *tests*, are applied to this model; these tests are generated by (possibly biased) random test pattern generators, or by hand. If for a given test the assertion is violated, the corresponding monitor enters a "violation" state, flagging the failure. The effectiveness of simulation sequences, i.e., the *test-bench,* is assessed using several coverage metrics: *code coverage* [3, 4], *tag coverage* [5], *event coverage* [6], and *state machine coverage* [7-10]. *Code-based coverage* includes statement, branch, sub-expression, and path coverage. *Tag coverage* evaluates the observability of possible incorrect evaluation represented as tags at the circuit outputs. *Event coverage* is measured by activating the coverage models on the event trace. *State machine coverage* is based on the number of distinct states visited and transitions occurred in an abstracted design.

A simulation-based verification approach is simple, scales well with design size and has traditionally been the *de facto* workhorse for functional verification. However, it cannot guarantee completeness of coverage and hence, design correctness. More disturbingly, for practical designs, the fraction of design space which can be covered by simulation is vanishing small; resulting in a significant probability of *respin severity bugs* undetected in the design even after substantial simulation efforts [11]. Besides diminished coverage, development and debugging of test-benches is a non-trivial time-consuming process, often mandating the verification team members to understand the design behaviors, features to be tested, and interpreting the simulation results.

1.3 Formal Verification

Formal Verification (FV) refers to mathematical analysis of proving or disproving the correctness of a hardware or software system with respect to a certain unambiguous specification or property. The methods for analysis are known as *formal verification methods*, and unambiguous specifications are referred as *formal specifications*. Formal verification complements simulation but with higher complexity of analysis, where mathematical analysis is done on an abstract model of the system, modeled using finite state machines or labeled transition systems[1]. Formal verification can provide complete coverage and can therefore, ensure design correctness with respect to the properties specified, without requiring any test-bench. However, one should not

[1] Other mathematical models also used are Petri nets, timed automata, hybrid automata, process algebra, operational semantics, denotation semantics, and Hoare's Logic.

construe the formal verification to produce a "defect-free" design as it is impossible to formally specify what defect-free really means. To reiterate, *formal verification can ensure the correctness of a design only with respect to certain properties that it is able to prove*.

Formal verification can be broadly classified into two methods: *Model Checking* and *Theorem Proving*. Model Checking [12] consists of a systematic exhaustive exploration of all states and transitions in a model. It is implemented using explicit or implicit state enumeration techniques on a suitably abstracted model, and proving or disproving the existence of "defect" states in the model. Theorem Provers [13, 14], on the other hand, use mathematical reasoning and logical inference to prove the correctness of the systems and often, require a "theorem prover guru" with substantial understanding of the system-under-verification.

1.3.1 Model Checking

Model checking is an automated technique and hence, more popular in the industry as an alternate verification strategy to simulation. The applications of model checking are typically classified as *equivalence checking* and *property checking.* In *equivalence checking*, a "golden model" is used as a reference model to check if the given implementation model has a "defect". In *property checking*, the correctness properties describing the desirable/undesirable features of the design are specified using some formal logic (e.g. temporal logic) and verification is performed by proving or disproving that the property is satisfied by the model. In this book, we will focus our discussion on efficient property checking techniques that make formal verification practical and realizable.

Model checking techniques, in practice, are inherently limited by the state-explosion problem, i.e., the fact that the number of states is exponential in the number of state elements (e.g. registers, latches) in the design. Model checking approaches are broadly classified into two, based on state enumerations techniques employed: *explicit* and *implicit* (or symbolic). Explicit model checking techniques [15, 16] store the explored states in a large hash table, where each entry corresponds to a single system state. A system with as few as a hundred state elements amounts to a state space with $\sim 10^{11}$ states. Understandably, model checkers need to pay special attention to scalability of the techniques used. Symbolic model checking techniques [17] store sets of explored states symbolically by using characteristic functions represented by canonical/semi-canonical structures, and traverse the state space symbolically by exploring a set of states in a single operation. Canonical structures such as Binary Decision Diagrams (BDDs) [18, 19] allow constant time satisfiability checks of Boolean expressions, and are used to perform symbolic Boolean function manipulations [17, 20-22]. Though these BDD-based methods have greatly improved scalability in comparison to explicit state enumeration techniques, by and large, they are limited to designs with a few hundred state holding elements, which is not even at the level of an individual designer subsystem. This is mainly due to frequent space-outs and severe performance degradation [23] as BDDs constructed in the course of symbolic traversal grow extremely large, and BDD size is critically dependent on variable ordering. Though several variable ordering heuristics to reduce BDD sizes have been proposed [24-27], in many cases BDDs are hard to optimize [28, 29]. Several variations of BDDs such as Free BDDs [30], zBDDs [31], partitioned-BDDs [32] and subset-BDDs [33] have also been proposed to target domain-specific application; however, in practice, they have not scaled adequately for industry applications.

In a quest for robust and scalable approaches, research has been heavily directed toward separating Boolean reasoning and representation. Boolean Satisfiability (SAT), which has been studied over several decades, has emerged [34] as a workhorse for Boolean reasoning primarily due to many recent advances in DPLL-style [35] SAT-solvers [35-41]. Efficient Boolean representation [42, 43] such as semi-canonical representations, that are simple and reduced, are also emerging [44] as a *de facto* structure due to their less sensitivity to variable ordering and compact representation compared to BDDs. SAT, together with efficient representation, have become a viable alternative to BDDs for model checking applications. This helped to make SAT-based symbolic model checking techniques both realizable and practical.

With emerging power of Boolean reasoning, various robust and scalable SAT-based techniques are simultaneously developed to

- *target* specific verification tasks such as falsification, proofs, and abstraction/refinements;
- *address* current design features such as embedded memories and multiple clocks domains;
- *address* complex specifications due to the presence of nested clocks;
- *overcome* limitation of computation resources of a single workstation.

We see a clear preference in verification communities: *BDD-based Methodists are becoming SAT-based Methodists.*

1.4 Overview

In this book, we discuss various SAT-based formal verification methods that we have developed and applied in an industry setting, especially to address the scalability and performance issues that have been major limitations in BDD-based methods. These techniques comprise robust Boolean reasoning [41], efficient problem representation [42, 45], accelerated bug finding techniques such as bounded model checking (BMC) [45, 46], proof techniques such as induction and unbounded model checking (UMC) [47], and improved abstraction and refinement approaches [48, 49]. These methods efficiently handle designs with complex features such as embedded memories [50, 51], and multiple clock-domains [52], with complex clocked specifications. These methods are implemented in our verification platform *VeriSol* (formerly *DiVer*) [53], and have been applied successfully in industry for the last four years (as of 2006) to verify large hardware designs. Specifically, *VeriSo*l drives the *Property Checker* in NEC's *CyberWorkBench* (CWB) [54, 55], a high level design and synthesis environment that automatically generates RTL (Register Transfer Level) designs and properties from high-level behavioral descriptions. Based on this verification platform, we share our practical experiences and insights in verifying large industry designs. We also use *VeriSol* as the primary model checking workhorse in our software verification platform, *F-Soft* [56].

Scalability of formal verification tools will always remain an open research problem, as design complexity continues to grow. In this book, we discuss the following two trends that have shown some potential in mitigating this problem: Synthesis-For-Verification, and High-level Model Checkers.

Synthesis-For-Verification (SFV) paradigm [57, 58] addresses generation of "verification aware" models to improve the effectiveness of verification techniques. In particular, we discuss how a high-level synthesis (HLS) tool can be guided within its existing infrastructure to obtain "verification-friendly" models that are relatively easy to model check. Such an approach also leverages off the various advancements in verification techniques as discussed in this book.

High-level model checkers [59-62] are applied at word-level models to cope with inherent limitations of formal techniques at the bit level — due to requirement of finite datapaths, inefficient translations into SAT, and loss of high-level design information. Thus, high-level model checkers have potential to scale up to industry designs. We discuss later how the performance of high-level model checkers can be improved using high-level information extracted from the high-level models, on-the-fly simplification and model transformations.