Mingsong Chen · Xiaoke Qin
Heon-Mo Koo · Prabhat Mishra

System-Level Validation

High-Level Modeling and Directed Test Generation Techniques

System-Level Validation

Mingsong Chen · Xiaoke Qin
Heon-Mo Koo · Prabhat Mishra

System-Level Validation

High-Level Modeling and Directed Test Generation Techniques

Springer

Mingsong Chen
Software Engineering Institute
East China Normal University
Shanghai
People's Republic of China

Xiaoke Qin
Department of Computer and Information Science and Engineering
University of Florida
Gainsville, FL
USA

Heon-Mo Koo
Intel Corporation
Santa Clara, CA
USA

Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida
Gainsville, FL
USA

ISBN 978-1-4614-1358-5 ISBN 978-1-4614-1359-2 (eBook)
DOI 10.1007/978-1-4614-1359-2
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2012943624

© Springer Science+Business Media New York 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)

To our families

Preface

We are immersed in the world of computing in our everyday lives. We are aware of computing systems when we use desktops, laptops, or servers. In many scenarios, we interact with systems that have embedded computing in them, such as cell phones, gadgets, and monitoring systems. When we fly on an airplane or even drive a car, there are many computing devices working together to ensure that our journey is pleasant and safe. Can we assume that these embedded computing systems are correct by construction and therefore we can safely rely on them? The simple answer is that no one can prove their absolute infallibility. Yet, since we do not have a choice, we can sit back, relax, and enjoy life!

Consider a simple *adder* to understand the difficulty in verifying today's systems. An adder is one of the simplest computations in a calculator. It adds two input values and produces the result. Typically, the input values are 32-bit integers. Therefore, to verify this adder we have to simulate several trillions ($2^{32} \times 2^{32}$) of test vectors. Clearly, it is too expensive to apply trillions of input vectors to verify an adder. In fact, it is infeasible to simulate an adder using all possible input sequences when the input values are 64-bit integers. If we cannot completely verify a simple adder, what is the hope that we can verify today's embedded and hybrid systems that consist of complex software and hardware including multicore/manycore processors, memories, buses, controllers, interfaces, and so on? Functional validation is widely acknowledged as a major bottleneck in System-on-Chip (SoC) design methodology—up to 70 % of the overall design time and resources are spent on functional validation. In spite of such extensive efforts, many SoC designs fail at the very first time (silicon failures) due to functional errors. The functional verification complexity is expected to increase further due to the combined effects of increasing design complexity and reduced time-to-market constraints.

Existing validation approaches use a combination of simulation-based techniques and formal verification methods. Simulation is the most widely used form of validation using random and constrained-random tests. For instance, in the adder example, instead of trying all possible input combinations, validation engineers can create tests for interesting scenarios based on coverage criteria and corner cases. Since it is impractical to generate and apply all possible tests,

simulation-based validation does not guarantee correctness. On the other hand, formal verification methods (such as model checking) do not use input vectors but explore the state space to ensure correctness. Unfortunately, this exploration can lead to state space explosion for large designs. Today's state-of-the-art verification methodologies use an efficient combination of both approaches. For example, the complete SoC design is simulated using billions of tests whereas very critical components such as controllers or specific protocols are fully verified by formal methods.

There are various promising directions to significantly reduce the overall validation effort by exploiting the synergies between simulation and formal verification. It is beneficial to verify as much as possible at the specification level (which is orders-of-magnitude simpler than implementation but has all the functional details) and automatically reuse high-level validation efforts for verifying lower level implementation. Similarly, it is promising to use directed tests for design validation since directed tests can achieve the same coverage goal using orders-of-magnitude less tests compared to random or constrained-random tests. As a result, simulation using efficient directed tests can drastically reduce the overall validation effort. However, directed test generation is mostly performed by human intervention. Hand-written tests entail laborious and time-consuming effort of verification engineers who have deep knowledge of the design under verification. For a complex design, it is infeasible to manually generate all directed tests to achieve a comprehensive coverage goal. Therefore, it is necessary to develop tools and techniques for automated generation of directed tests.

This book describes innovative ways of combining simulation and formal methods for verifying complex systems. It provides a comprehensive coverage of system-level modeling and validation techniques to both reduce overall validation effort and improve the product quality. It describes challenges associated with verifying complex systems and presents efficient techniques to address these challenges. It studies various system-level modeling approaches using SystemC, UML, and transaction level models. It presents state-of-the-art techniques for automated generation of directed tests focusing on reduction of test generation time and validation effort through clustering, decomposition, and learning techniques. It also describes innovative ways of reusing high-level validation effort across different abstraction levels.

The reader of this book will get a comprehensive understanding of design verification challenges and how system-level validation can significantly improve design quality and reduce overall cost. Chapter 1 highlights the benefits of system-level validation. Chapter 2 describes how to specify complex systems using high-level models. Chapter 3 provides the basic framework for automated generation of directed tests. The next chapter presents how to significantly reduce the number of tests without sacrificing the coverage goal. Chapter 5–9 describe efficient techniques for generation of directed tests. The next two chapters present test generation techniques focusing on validation of multicore/manycore architectures. Chapter 12 describes innovative ways of reusing high-level tests and assertions for validation of implementation. Finally, Chapter 13 concludes the book with a short

discussion of future research directions. We hope you enjoy reading this book and find the information useful for your purpose.

Audience

This book is designed for senior undergraduate students, graduate students, researchers, CAD tool developers, designers, and managers interested in development of efficient tools and techniques for system-level design and verification, directed test generation, and functional validation of heterogeneous SoC designs.

Acknowledgments

This book is the result of a decade long academic research and industrial collaborations. The book includes the system-level validation techniques and insights that resulted from Ph.D. dissertations of Prof. Mingsong Chen, Dr. Heon-Mo Koo, and Dr. Xiaoke Qin. We would like to acknowledge our sponsors for providing the financial support to enable this research work. This work was partially supported by National Science Foundation (CAREER Award 0746261, CCF-0903430, and CNS 0905308), Semiconductor Research Corporation (2009-HJ-1979), Intel Corporation, Doctoral Fund of Ministry of Education of China 20110076120025, and State High-Tech Development Plan of China 2011AA010101.

This book has the footprints of many collaborations. We would like to acknowledge the contributions of Dr. Magdy Abadir (Freescale), Dr. Jayanta Bhadra (Freescale), Prof. Nikil Dutt (UC Irvine), Prof. Masahiro Fujita (University of Tokyo), Dr. Zhuo Huang, Dr. Dhrubajyoti Kalita (Intel), Prof. Tulika Mitra (NUS Singapore), and Prof. Abhik Roychoudhury (NUS Singapore). We are also thankful to all the members of the *CISE Embedded Systems Lab* at University of Florida. The list of members include Dr. Kanad Basu, Hadi Hajimiri, Kamran Rahmani, Kartik Shrivastava, Chetan Murthy, Seok-Won Seong, and Dr. Weixun Wang.

Contents

1 Introduction 1
1.1 Growing SoC Validation Complexity 1
1.2 System-Level Validation: Opportunities and Challenges 3
1.2.1 Top-Down Design and Validation Flow 4
1.2.2 SoC Validation Approaches 6
1.2.3 Opportunities in System-Level Validation 10
1.2.4 System-Level Validation Challenges 12
1.3 A Comprehensive Approach for System-Level Validation 14
1.4 Book Organization 14
References 15

2 Modeling and Specification of SoC Designs 19
2.1 Introduction 19
2.2 Modeling of Complex Systems 19
2.2.1 Graph-Based Modeling 20
2.2.2 FSM-Based Behavior Modeling 20
2.3 Specification Using SystemC TLMs 22
2.3.1 Modeling of SystemC TLM Designs 22
2.3.2 Transformation from SystemC TLM to SMV 24
2.3.3 Case Study: A Router Example 28
2.4 Specification Using UML Activity Diagrams 29
2.4.1 Graphic Notations 30
2.4.2 Formal Modeling of UML Activity Diagrams 32
2.4.3 Transformation from UML Activity Diagrams to SMV 36
2.4.4 Case Study: A Stock Exchange System 40
2.5 Chapter Summary 40
References 40

3 Automated Generation of Directed Tests . . . 43
3.1 Introduction . . . 43
3.2 Related Work . . . 44
3.3 The Workflow of Model Checking Based Test Generation . . . 45
3.4 Coverage-Driven Property Generation . . . 46
3.4.1 Safety Property and Its Negation . . . 46
3.4.2 Testing Adequacy Using Model Checking . . . 48
3.4.3 Fault Models . . . 48
3.4.4 Functional Coverage Based on Fault Models . . . 51
3.5 Test Generation Using Model Checking Techniques . . . 51
3.5.1 Test Generation Using Unbounded Model Checking . . . 51
3.5.2 Test Generation Using Bounded Model Checking . . . 52
3.6 Case Studies . . . 54
3.6.1 A Control System . . . 55
3.6.2 A Stock Exchange System . . . 56
3.7 Chapter Summary . . . 57
References . . . 58

4 Functional Test Compaction . . . 61
4.1 Introduction . . . 61
4.2 Manufacturing Test Reduction Techniques . . . 61
4.2.1 Test Compression . . . 63
4.2.2 Test Compaction . . . 63
4.2.3 Applicability and Limitations . . . 65
4.3 Functional Test Compaction . . . 67
4.3.1 Binary Format of FSM Models . . . 68
4.3.2 Number of FSM States and Transitions . . . 70
4.3.3 Property Compaction of FSM States and Transitions . . . 71
4.3.4 FSM Coverage-Driven Test Selection and Generation . . . 73
4.3.5 A Case Study . . . 74
4.4 Chapter Summary . . . 76
References . . . 76

5 Property Clustering and Learning Techniques . . . 79
5.1 Introduction . . . 79
5.2 Related Work . . . 80
5.3 Background: SAT Solver Implementation . . . 81
5.3.1 DPLL Algorithm . . . 81
5.3.2 Conflict Clause . . . 82
5.4 Property Clustering . . . 84
5.4.1 Similarity Based on Structural Overlap . . . 85
5.4.2 Similarity Based on Textual Overlap . . . 86
5.4.3 Similarity Based on Influence . . . 87
5.4.4 Similarity Based on CNF Intersection . . . 88
5.4.5 Determination of Base Property . . . 88

5.5 Conflict Clause Based Test Generation 89
5.5.1 Conflict Clause Forwarding Techniques. 89
5.5.2 Name Substitution for Computation of Intersections 91
5.5.3 Identification and Reuse of Common Conflict Clauses . . . 92
5.6 Case Studies 93
5.6.1 A MIPS Processor. 94
5.6.2 A Stock Exchange System 102
5.7 Chapter Summary 104
References 105

6 Decision Ordering Based Learning Techniques 107
6.1 Introduction 107
6.2 Related Work 108
6.3 Decision Ordering Based Learnings. 108
6.3.1 Motivation 109
6.3.2 Bit Value Ordering 110
6.3.3 Variable Ordering 111
6.3.4 Hybrid Learning from Conflict Clauses and Decision Ordering 112
6.4 Test Generation Using Decision Ordering Techniques 113
6.4.1 Test Generation for a Single Property 114
6.4.2 Test Generation for Similar Properties. 116
6.5 Case Studies 119
6.5.1 Intra-Property Learning 119
6.5.2 Inter-Property Learning 123
6.6 Chapter Summary 127
References 127

7 Synchronized Generation of Directed Tests. 129
7.1 Introduction 129
7.2 Related Work 130
7.3 Synchronized Test Generation. 130
7.3.1 Correctness of STG. 135
7.3.2 Implementation Details 136
7.4 Case Studies 137
7.4.1 A Stock Exchange System 137
7.4.2 A MIPS Processor. 140
7.5 Chapter Summary 142
References 142

8 Test Generation Using Design and Property Decompositions 145
8.1 Introduction 145
8.2 Related Work 147
8.3 Decomposition of Design and Property 148
8.3.1 Design Decomposition 149
8.3.2 Property Decomposition 150
8.4 Decompositional Test Generation 155
8.4.1 Test Generation Using Module-Level Partitioning 158
8.4.2 Test Generation Using Path-Level Partitioning 160
8.5 Merging Partial Counterexamples 161
8.6 A Case Study 162
8.6.1 Module-Level Decomposition 163
8.6.2 Group-Level Decomposition Based on Time Step 164
8.6.3 Discussion: Applicability and Limitations 166
8.7 Chapter Summary 166
References 166

9 Learning-Oriented Property Decomposition Approaches 169
9.1 Introduction 169
9.2 Related Work 170
9.3 Learning-Oriented Property Decomposition 171
9.3.1 Spatial Property Decomposition 171
9.3.2 Temporal Property Decomposition 174
9.4 Decision Ordering Based Learning Techniques 176
9.5 Test Generation Using Decomposition and Learning Techniques 178
9.6 An Illustrative Example 179
9.6.1 Spatial Decomposition 179
9.6.2 Temporal Decomposition 180
9.7 Case Studies 181
9.7.1 A MIPS Processor 181
9.7.2 A Stock Exchange System 182
9.8 Chapter Summary 183
References 184

10 Directed Test Generation for Multicore Architectures 185
10.1 Introduction 185
10.2 Related Work 186
10.3 Test Generation for Multicore Architectures 187
10.3.1 Correctness of TGMA 191
10.3.2 Implementation Details 193
10.3.3 Heterogeneous Multicore Architectures 194

10.4 Case Studies . . . 195
10.4.1 Experimental Setup . . . 195
10.4.2 Results. . . . 196
10.5 Chapter Summary . . . 199
References . . . 199

11 Test Generation for Cache Coherence Validation . . . 201
11.1 Introduction . . . 201
11.2 Related Work . . . 202
11.3 Background and Motivation . . . 203
11.4 Test Generation for Transition Coverage . . . 204
11.4.1 SI Protocol . . . 205
11.4.2 MSI Protocol . . . 206
11.4.3 MESI Protocol . . . 208
11.4.4 MOSI Protocol . . . 209
11.5 Case Studies . . . 210
11.6 Chapter Summary . . . 213
References . . . 213

12 Reuse of System-Level Validation Efforts . . . 215
12.1 Introduction . . . 215
12.2 Related Work . . . 216
12.3 RTL Test Generation from TLM Specifications . . . 217
12.3.1 Automatic TLM Test Generation . . . 217
12.3.2 Translation from TLM Tests to RTL Tests. . . . 219
12.3.3 A Prototype Tool for TLM-to-RTL Validation Refinement. . . . 222
12.4 Case Studies . . . 224
12.4.1 A Router Example . . . 224
12.4.2 A Pipelined Processor Example . . . 228
12.5 Chapter Summary . . . 233
References . . . 233

13 Conclusions . . . 235
13.1 Summary . . . 235
13.2 Future Directions. . . . 237

Appendix A: Acknowledgments of Copyrighted Materials . . . 239

Index . . . 243

About the Authors

Mingsong Chen is an Associate Professor in the Software Engineering Institute at the East China Normal University, China. His research focuses on design automation of embedded systems, cyber-physical systems, model checking techniques and software engineering. He received his B.S. and M.E. from Nanjing University in 2003 and 2006, respectively, and Ph.D. from University of Florida in 2010—all in Computer Science. He has published many research papers in premier international journals and conferences. One of his papers was nominated for best paper award in International Conference on VLSI Design, 2009. Currently, his research is supported by the National Nature Science Foundation of China, Ministry of Education of China, and State High-Tech Development Plan. He has served as a program/organizing committee member of several ACM and IEEE conferences including SAC and ICFEM. He is a member of both ACM and IEEE.

Xiaoke Qin received his B.S. and M.S. degrees from the Department of Automation, Tsinghua University, Beijing, China, in 2004 and 2007, respectively. He received his Ph.D. from the Department of Computer and Information Science and Engineering, University of Florida, USA, in 2012. His research interests are in the area of code compression, model checking, system-level validation of multicore architectures, and real-time scheduling of embedded systems. He has published many research articles in premier international journals and conferences. His work on cache coherence validation has been nominated for best paper award in Design Automation and Test in Europe (DATE), 2012. Dr. Qin has served as a reviewer of several ACM and IEEE conferences and journals.

Heon-Mo Koo is a Media Architect in Visual and Parallel Computing Group at Intel Corporation. His research interests are design and verification of embedded systems for media technologies including video codec, video processing, and computer vision. He received his B.S. and M.S. degrees at the Department of Electronic and Electrical Engineering from Kyungpook National University in Korea in 1993 and 1995, respectively. Prior to joining his Ph.D. study, he had worked at LG Electronics Research Center for 8 years. He received his Ph.D. degree in Computer Engineering at the University of Florida in 2007.

Prabhat Mishra is an Associate Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include design automation of embedded systems, energy-aware computing, hardware/software verification, and design of trustworthy systems. He received his B.E. from Jadavpur University, Kolkata in 1994, M.Tech. from the Indian Institute of Technology, Kharagpur in 1996, and Ph.D. from the University of California, Irvine in 2004—all in Computer Science. Prior to joining University of Florida, he spent several years in various semiconductor and design automation companies, including Intel, Motorola, Synopsys, and Texas Instruments. He has published four books, ten book chapters and more than 100 research articles in premier journals and conferences. His research has been recognized by several awards including the NSF CAREER Award from the National Science Foundation, two best paper awards (VLSI Design 2011 and CODES+ISSS 2003), several best paper award nominations, and 2004 EDAA Outstanding Dissertation Award from the European Design Automation Association. He has also received the 2007 International Educator of the Year Award from the UF College of Engineering for his significant international research and teaching contributions. He currently serves as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems (TODAES), IEEE Design & Test of Computers (D&T), and Journal of Electronic Testing (JETTA), Guest Editor of IEEE Transactions on Computers, and as a program/organizing committee member of several ACM and IEEE conferences, including ICCAD, DATE, ASPDAC, CODES+ISSS, and VLSI Design. He has also served as General Chair of IEEE High-Level Design Validation and Test (HLDVT) 2010, Program Chair of HLDVT 2009, Information Director of ACM TODAES, and Guest Editor of IEEE D&T, Springer JETTA and IJPP. He is a senior member of both ACM and IEEE.

Acronyms

ABV	Assertion-Based Validation
ADL	Architecture Description Language
ATE	Automatic Test Equipment
ATM	Automated Teller Machine
ATPG	Automatic Test Pattern Generator
BCP	Boolean Constraint Propagation
BDD	Binary Decision Diagrams
BIST	Built In Self Test
BMC	Bounded Model Checking
CNF	Conjunctive Normal Form
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
DSE	Design Space Exploration
DUV	Design Under Validation
ESL	Electronic System Level
FSM	Finite State Machine
IC	Integrated Circuit
LTL	Linear Temporal Logic
MoC	Model of Computation
MPSoC	Multiprocessor SoC
PSL	Property Specification Language
RTL	Register Transfer Level
SAT	Satisfiability
SoC	System-on-Chip
SVA	SystemVerilog Assertion
TLM	Transaction Level Modeling
TRS	Test Refinement Specification
UMC	Unbounded Model Checking
UML	Unified Modeling Language
VSIDS	Variable State Independent Decaying Sum
XMI	XML Metadata Interchange

Chapter 1
Introduction

1.1 Growing SoC Validation Complexity

System-on-Chip (SoC) integrates all necessary hardware components (e.g., microprocessors, memory blocks and peripherals) and software modules (e.g., real-time operating systems and control programs) into a single integrated circuit to fulfill a required functionality. It may perform a variety of computations including digital, analog, and mixed-signal functions. Figure 1.1 shows the block diagram of a typical multiprocessor SoC (MPSoC) architecture [1]. The SoC design example consists of four RISC processors and various general input–output devices, which are connected by advanced high-performance bus (AHB) and advanced peripheral bus (APB). SoC is widely used in the field of embedded systems, such as automotive electronics, avionics, telecommunication, smart buildings, handheld devices, and so on.

Due to increasing complexity of both software and hardware components, the design complexity of modern SoCs is increasing at an alarming rate. To accommodate the emerging application and computation requirements, today's SoCs employ many complicated architectural features to support synchronization, communication, dynamic frequency/voltage scaling, cache coherence, interconnect pipelining, and so on. Increasing design complexity has been translated to the growth in verification complexity. The rapid technology evolution together with fierce market competition make SoC development confront a dilemma: how to quickly guarantee the functional correctness of SoC designs? Functional verification of SoC designs is widely acknowledged as a major bottleneck. It is infeasible to simulate even a simple 2-input adder using all possible input sequences since it will require $2^{64} \times 2^{64}$ input (test) vectors when each input is 64 bits. Verifying a complex SoC is analogous to verifying millions to billions of complex adders interacting together in the form of software and/or hardware. It is a major challenge to detect and fix all the functional errors as well as electrical faults due to the combined effects of increasing complexity and shrinking time-to-market constraints.

According to the functional verification studies by Ron Collett International (in 2002 and 2004) and FarWest Research (in 2007), functional errors are becoming

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_1,
© Springer Science+Business Media New York 2013

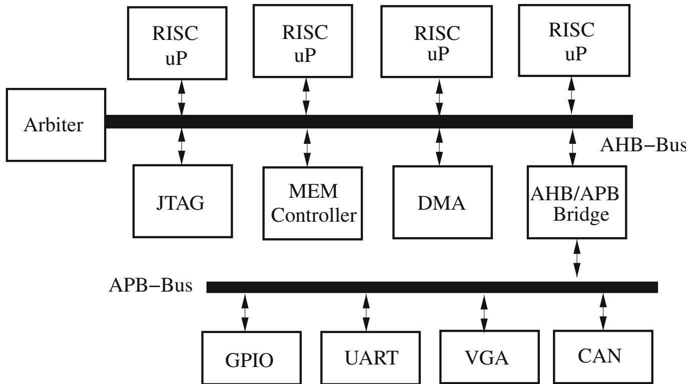


Fig. 1.1 An MPSoC design example

the leading causes of SoC failures. Around three quarters of errors in SoC design are due to functional bugs. The statistics shows that such errors are introduced due to various factors including: careless coding, misinterpretation of the specification, microarchitectural design complexity, corner cases, and so on. It is significantly more expensive to detect and fix errors in lower level implementation compared to performing these activities at higher abstraction levels. For example, debugging an error is easier during pre-silicon validation compared to post-silicon validation since the observability of internal signals is limited in a fabricated chip. Once the root cause of a bug is analyzed, the chip may need to be fabricated again, which is very expensive, unless that bug can be fixed or masked using existing mechanisms. If buggy software/hardware products are released to consumers, it can lead to various unpleasant scenarios depending on application domains—significant financial loss for commodity products or even catastrophic consequences in safety–critical systems. For example, in 1994, Intel's Pentium processor had a functional error called FDIV bug[1] and the company had to spend a staggering cost (around half a billion dollars) to replace the flawed processors. Similarly, Ariane 5 explosion[2] was caused due to a data conversion error in its control software. Therefore, it is important to guarantee the functional correctness before delivering the products.

Functional validation (or verification[3]) can expose logic/functional errors in software/hardware implementations which are not consistent with its specification.

[1] The Pentium FDIV bug was the most infamous one of the Intel microprocessor bugs. Due to an error in a lookup table, certain floating point division operations would produce incorrect results.

[2] On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded a few seconds after its liftoff.

[3] "Validation" is performed at different abstraction levels to uncover two types of faults: specification flaws and implementation bugs. Formal verification refers to the use of formal methods to ensure that the implementation satisfies the specification. In this book, the term "verification" refers to the use of both simulation-based validation and formal methods to detect implementation bugs (errors). Please note that the terms "validation" and "verification" have different usage and interpretation in different domains.

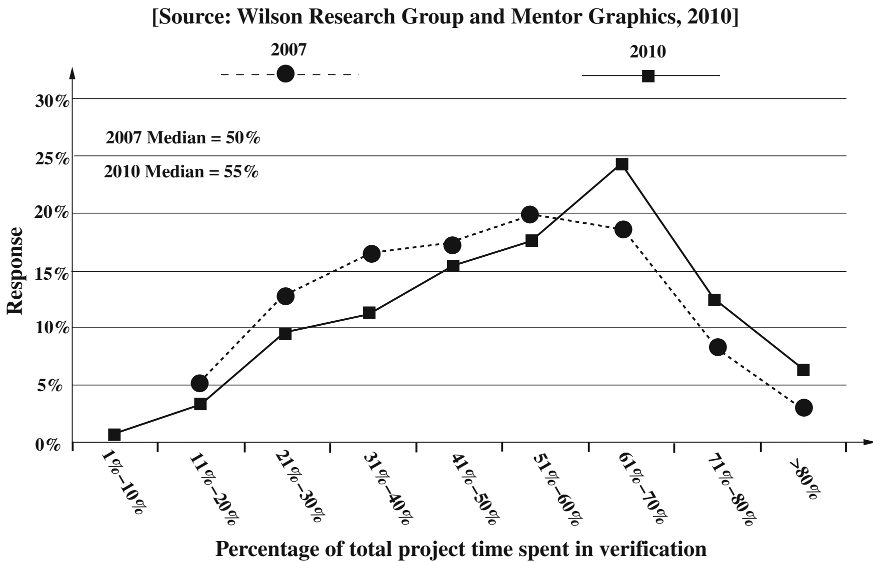


Fig. 1.2 Percentage of total project time spent in verification

Figure 1.2 shows the importance of verification in various industrial projects in terms of the percentage of total project time spent in verification by comparing the 2007 Far West Research study (in dotted line) with the 2010 Wilson Research Group study (in solid line). It can be observed that from the year 2007 to 2010, the percentage of verification in overall project time has increased in more projects. The study also indicates that the percentage of time spent in verification is still increasing. Therefore, functional validation is expected to remain as a major bottleneck in SoC design methodology in foreseeable future.

In spite of the extensive functional validation efforts, majority of the SoC designs fail at the very first time (silicon failure). According to the functional verification studies from the Collett International Research 2002 and 2004, and FarWest Research 2007 shown in Fig. 1.3, you can see a continual upward trend in respin numbers. In other words, respins (re-design and re-fabrication) are becoming more frequent. Obviously, such respins are very expensive and strongly affect the time-to-market. Therefore, it is imperative to develop/employ efficient functional validation techniques to reduce overall validation effort and improve the product quality.

1.2 System-Level Validation: Opportunities and Challenges

This section first describes the importance of system-level validation and introduces the traditional top–down design and validation flow. Next, it surveys existing functional validation methods. Finally, it discusses the research challenges and associated system-level validation opportunities.

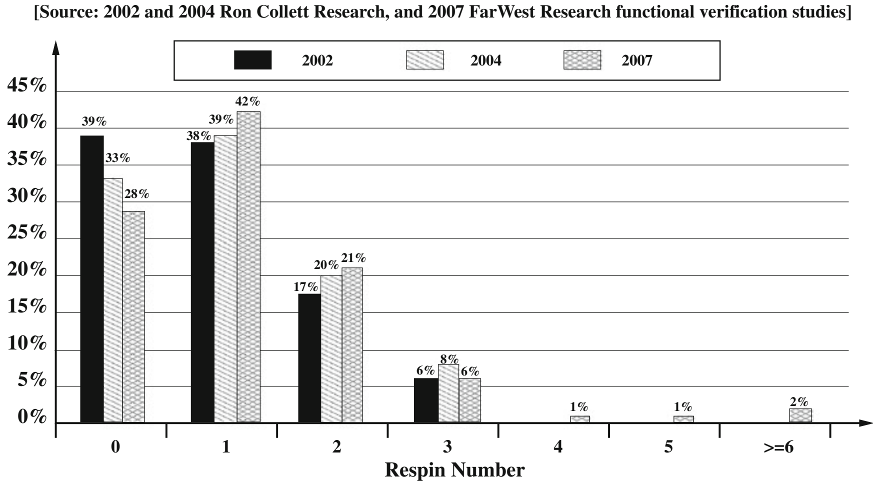


Fig. 1.3 Industry trends in achieving first silicon success

1.2.1 Top-Down Design and Validation Flow

Early design space exploration (DSE) is required to find the best possible SoC architecture under various design constraints such as cost, area, power, energy, performance, and so on. Automated exploration is beneficial given the time-to-market constraints and the trend of adding more and more complex features into new SoC applications. Although register transfer level (RTL) designs contain more details, the RTL simulation is too slow for exploration of large and complex SoC designs. To improve the productivity with faster simulation speed, recent innovations in SoC design shift the design flow to a higher level—electronic system level (ESL[4]). ESL focuses on designing various system-level specifications, which consists of high-level languages (e.g., SystemC [3], MATLAB [4], Esterel [5]) and models (e.g., Simulink [6], SysML [7], Petri-net [8]) in an abstraction level that is above the familiar RTL in hardware and the source code in software.[5] Such high-level specifications are promising in describing system-level behaviors and have been widely used in architecture exploration, hardware/software co-design and co-verification.

Traditionally, SoC design starts with system-level exploration to find a suitable architecture specification, and then the specification is transformed into lower level implementation. Various electronic design automation (EDA) softwares (tools) have been developed to enable rapid SoC design, which brings greater efficiency and

---

[4] In the ESL Design and Verification book [2], the term ESL is defined as: "the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner.".

[5] Depending on application domains and abstraction levels, different types of specification languages may be applicable. For example, architecture description languages (ADL) [9] are very popular in early exploration and validation of processor-based SoCs.

productivity. Typically, the top–down design methodology for hardware design comprises of the following abstraction levels:

System/architectural level. A system-level layer focusing on the overall system behavior specification and architectural exploration.
Register transfer level. Implementing the functionalities derived from system-level specifications at register transfer level using a hardware description language (HDL).
Gate/logic level. Implementing functional designs into gate-level logic using logic synthesis EDA software tools.
Transistor/physical level. Implementing logic designs using physical components (i.e., transistors, wires, etc.) with placement and routing tools.

Software design also has a similar top–down flow. For both HW/SW designs, sufficient validation work is needed across each level of the SoC design to guarantee the correctness of the implementation. Since this book focuses on system-level validation, we only consider the highest two levels of SoC design for both software and hardware.

Figure 1.4 presents a simplified design and validation flow for the top two (out of four discussed above) levels. Various hardware and software modeling paradigms are used for SoC specification. Two of the most widely used specifications are transaction level modeling (TLM) [10, 11] and unified modeling language (UML) [12]. They establish a standard to enable fast simulation speed and easy model interoperability for hardware/software co-design. Generally, TLM is promising for hardware modeling and UML focuses on software modeling. TLM mainly allows modeling of transactions between different hardware components of a system and data processing in each component. UML models can capture both structural and behavioral information of a software system. Validated specification can be used as a golden reference model for validation of software and hardware implementations. Depending on modeling style, specifications can capture different functional and timing details. For example, TLM provides two kinds of modeling styles: loosely timed models can be used to model the system behavior with less timing information, and approximately timed models can enable timing analysis of system behavior. Although TLM is promising for system-level modeling and simulation, it is still hard to accurately describe various implementation and timing details. Once early exploration and system-level simulation are done, RTL is suitable to model the implementation-level behavior. In Fig. 1.4, the hardware part is implemented using an RTL language such as VHDL or Verilog, and the software is implemented using a programming language such as C or C++.

While system-level specification is beneficial for early exploration and validation, the introduction of a new abstraction level also brings potential sources of errors. According to [13], currently there are two key contributors to the SoC failures (silicon respin): implementation errors and specification errors. It is reported that 82 % of the designs with respins resulting from functional flaws had implementation errors. Interestingly, almost 47 % of the designs with respins resulting from functional flaws had also incorrect or incomplete specifications [13]. During the stepwise refinement

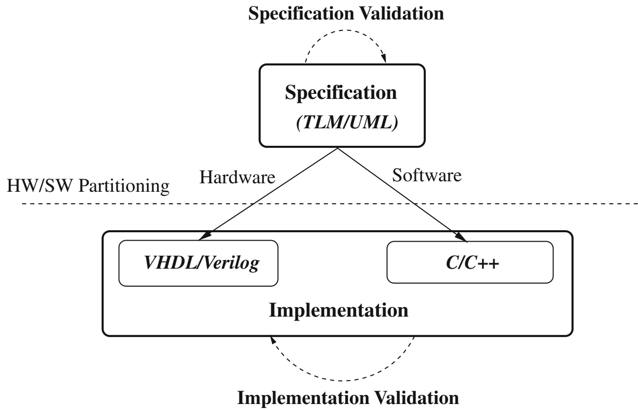


Fig. 1.4 A top–down SoC design and validation flow

from system-level down to transistor-level, system-level errors also gets refined, which become harder to discover. Furthermore, it is very costly to identify and fix an error in later stages of the design cycle compared to doing it in the early stages. Therefore, it is important to ensure the correctness of system-level specifications. In the last decade, there has been a tremendous amount of research effort both in industry and academia in developing system-level validation methodologies [14–16]. System-level specifications allow fast simulation speed and simplified communication interface. By performing as much validation as possible in the early stages of the design, the overall validation cost can be reduced while the quality of the implementation can be improved. Existing research efforts range from simulation-based techniques to formal methods. In this book, we are focusing on simulation-based validation of both specification and implementation using efficient directed tests. Specifically, this book studies various high-level modeling and directed test generation techniques to reduce overall validation effort. Therefore, the cost of test generation and the quality of generated tests are two major concerns that will be covered in the following chapters.

1.2.2 SoC Validation Approaches

SoC validation techniques can be broadly divided into two categories: formal verification and simulation-based approaches. Figure 1.5 shows some of the widely used techniques in these two categories. The remainder of this section provides an overview of these design validation techniques.

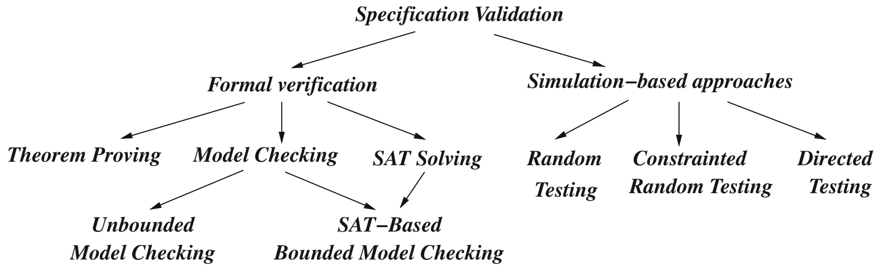


Fig. 1.5 SoC validation approaches

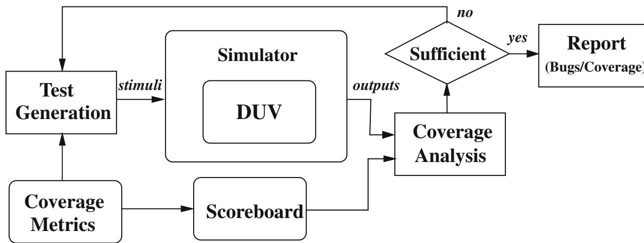


Fig. 1.6 Simulation-based validation flow

1.2.2.1 Simulation-Based Validation

Simulation-based validation [17] discovers design errors using test vectors consisting of input stimuli and expected outputs [18–21]. Figure 1.6 shows the traditional flow of simulation-based validation. The validation flow consists of three basic procedures: i) generation of test vectors, ii) simulation of SoC designs using the test vectors, and iii) comparison of the generated outputs with the expected results. In the validation flow, the coverage metrics provide a way to see which part of the design has not been verified and what tests should be added. Many coverage metrics have been proposed for different types of design errors at different design abstraction levels. In coverage-driven test generation, tests are created to activate a target coverage point and it can effectively reduce the number of tests compared to the random tests. Through simulation, the coverage is analyzed by examining whether target functionalities have been covered or not, thereby we can measure the validation progress. If coverage holes are found (i.e., the testing is not sufficient), additional tests are generated to exercise them. If higher degree of confidence is required, we can improve the coverage metric or make use of additional coverage measures. Verification engineers can change the scope or depth of coverage during the validation process. For example, they can start from simple coverage metrics in the early verification stage and use more complex coverage metrics later on.

Test generation plays an important role in simulation-based validation. It not only determines the testing quality, but also strongly affects the simulation time.

There are three types of test generation techniques: random, constrained-random, and directed. In the current industrial practice [22, 23], random and constrained-random test generation techniques are most widely used because test vectors can be produced automatically and most design errors can be uncovered early in the design cycle. However, a huge number of tests are required to achieve high confidence of the design correctness, and corner cases (e.g., pipeline interactions, timing relation) are easily missed. Compared to random testing methods which use billions to trillions of random and pseudorandom tests in the traditional design flow, directed testing uses fewer tests to obtain the required functional coverage goal since it exploits the design structure information [24, 25]. By applying the directed tests, the overall validation effort can be reduced without sacrificing the coverage goal. Most directed test generation methods assume the expert knowledge of the *Design Under Validation* (DUV). Typically, directed test generation methods are laborious and error prone. Due to the inevitable human intervention, it is infeasible to generate all directed tests to achieve a comprehensive coverage goal in a reasonable time. This book describes several efficient techniques to perform automated generation of directed tests.

Simulation-based methods scale well for complex designs. It can easily verify whether a functional scenario is activated by a given test on a multimillion line designs. However, it is difficult to ensure correctness of the design using simulation-based validation. For example, to ensure correctness of an SoC, all possible input stimuli are required. However, it may not be possible to generate and simulate all possible input sequences since the number of sequences can be prohibitively large (potentially infinite!). Typically, billions to trillions of random or constrained-random tests are used during simulation due to time-to-market and other constraints. Since only partial stimuli are applied (instead of all possible input sequences), it does not provide the assurance of the design correctness. We cannot assume that the uncovered scenarios (functionalities) are either correct or not reachable. Moreover, there is a lack of good coverage metrics to quantify the degree of confidence and to qualify a test set. Therefore, it is hard to answer the question, *"When is verification done?"*, due to difficulty in measuring verification progress and test effectiveness.

1.2.2.2 Formal Verification

Formal verification techniques [26, 27] provide the completeness of verification by proving mathematically the correctness of a design. Some of the widely used formal methods include model checking [28, 29], theorem proving [30, 31], equivalence checking [32] and *satisfiability solving* (SAT) [33, 34]. Model checking based approaches are widely adopted in SoC validation. Figure 1.7 shows the basic idea of the model checking process. First, it translates the DUV into a formal model and a validation target (VT) into a property in the form of temporal logic [29]. The property describes the functionality of the specified scenario (i.e., a functional coverage instance). The model checker tries to find whether the DUV satisfies the property by exploring all the reachable states of the DUV.

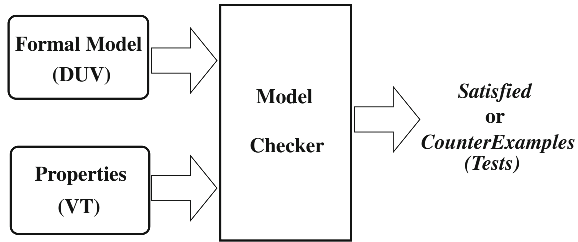


Fig. 1.7 Validation flow using model checking

If the property is not satisfied (i.e., there exists some state which violates the property), the model checker will report a counterexample for the property. As described in Chap. 3, property falsification can be used to derive counterexamples which can be used as directed tests [35]. Note that, the complexity of verifying a property is significantly higher than simulating a design using a test. This is due to the fact that simulation validates the effect of a test in a specific timeframe, whereas model (property) checking needs to ensure the correctness (implementation satisfies the property) for entire lifetime of the system. This search complexity of formal methods can lead to state space explosion[6] in case of large and complex designs. Due to state space limitations, formal methods are typically used to verify small but critical components, whereas simulation-based methods are more advantageous in validation of large designs though sacrifices validation completeness.

1.2.2.3 Semiformal Validation

Formal verification approaches can provide correctness guarantee but can lead to state space explosion when dealing with complex designs. On the other hand, simulation-based techniques can handle large complex designs but cannot prove the correctness of systems due to input space complexity.[7] To combine the benefits of both simulation and formal verification, more and more SoC validation approaches adopt the semiformal validation techniques, which make a tradeoff between the simulation-based approaches and formal verification methods. For example, assertion based validation (ABV) [36] is a popular semiformal technique that extends the notion of assertion in simulation-based method with the linear temporal logic semantics. Figure 1.8 shows that the adoption of semiformal assertion/property checking has grown by 53 % from year 2007 to 2010. By incorporating such assertions, ABV not only increases the design observability in simulation, but also takes advantage of formal techniques for improving the overall verification quality and results.

Currently, there are two most popular ABV languages: property specification language (PSL) [37] and SystemVerilog assertion (SVA) [38]. PSL is platform-independent and can be used in multilayer design. SVA is similar to PSL, nevertheless it is customized for SystemVerilog designs. Although the semiformal validation

[6] The size of the state space grows exponentially with the number of state variables of the system.

[7] The input space (all possible input sequences) can be prohibitively large for complex designs with large number of inputs.

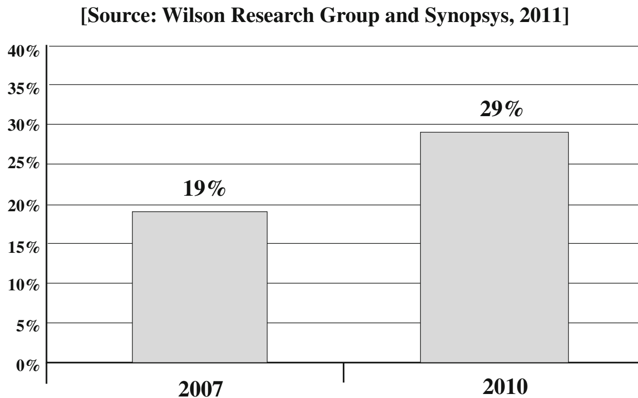


Fig. 1.8 Increasing adoption of semiformal assertion/property checking

Table 1.1 Comparison of three methods

Approaches	Simulation	Semiformal	Formal
Completeness	Low	Middle	High
Coverage Convergence	Slow	Slow	Fast
Directed Testing Automation	Low	Middle	High
Scalability	High	Middle	Low

extends the temporal capabilities of assertions, it is still mainly based on the simulation. Similar to the simulation-based methods, the completeness of validation still cannot be guaranteed. The verification engineers needs to address various challenges including, how many assertions are enough, how to activate a given assertion, how often the assertion should be exercised, and so on.

Table 1.1 presents a comparison of above three methods. It can be found that both simulation-based approaches and formal methods have their pros and cons. Therefore, at the beginning of the project, designers need to determine a proper validation plan which is suitable for their project. Generally, the most suitable method depends on the project itself as well as the capacity and completeness of validation methods.

1.2.3 Opportunities in System-Level Validation

Since system-level specification is used as a golden reference model in the top–down SoC design flow, a functional error in the specification will lead to buggy implementation. Finding an error in implementation is more time-consuming since implementations are more complex than system-level specifications. Therefore, it is beneficial to check as many functional scenarios as possible during validation of

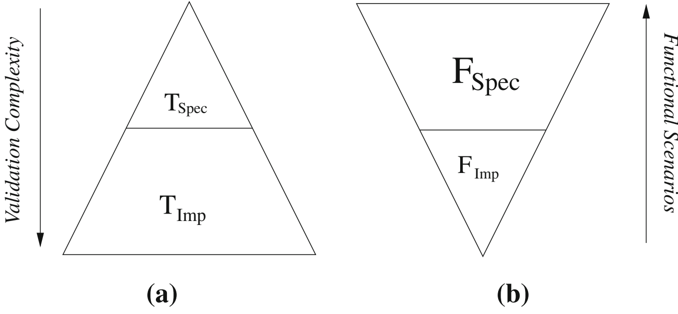


Fig. 1.9 Functional validation of specification and implementation

system-level specification. Interestingly, the same set of functional scenarios needs to be validated at both specification and implementation levels. If the validation efforts between different abstraction levels can be reused, the overall validation effort can be significantly reduced.

Figure 1.9 compares various validation issues related to specification and implementation levels. Assume that a design D has a total number of F_{Total} functional scenarios that need to be checked. Assume that there are F_{Spec} functional scenarios that can be activated at the specification level. Also assume that the test generation of each specification-level functional scenario needs an average time of T_{Spec}. In addition to F_{Spec} functional scenarios, there are F_{Imp} functional scenarios that can be activated in the implementation only. Assume that activation of each implementation scenario (i.e., test generation time) needs an average time of T_{Imp}. Figure 1.9a indicates that when checking a functional scenario, implementation validation is significantly more expensive than specification validation. Figure 1.9b shows that majority of the overall system functional scenarios can be activated at the specification level, and the implementation inherits all such scenarios with its own new additional functional scenarios due to the introduction of implementation details. In this book, the complexity of validating a functional scenario refers to validation of SoCs using directed tests. So the validation effort for a functional scenario comprises of test generation time, simulation time as well as associated debug time (if any).

$$\begin{aligned} \text{Minimize}: \quad & F_{\text{Spec}} \times T_{\text{Spec}} + \{F_{\text{Spec}} + F_{\text{Imp}}\} \times T_{\text{Imp}} \\ \text{Subject to}: \quad & \begin{cases} F_{\text{Spec}} + F_{\text{Imp}} = F_{\text{Total}} \\ T_{\text{Spec}} << T_{\text{Imp}} \\ F_{\text{Spec}} > F_{\text{Imp}} \end{cases} \end{aligned} \tag{1.1}$$

In order to achieve the functional coverage goal as well as to minimize the overall validation effort, it is necessary to find a method to optimize the Eq. 1.1. For directed test generation, there are four feasible options.

No optimization. Specification-level test generation and implementation-level test generation are independent, and in each level there are no optimizations.

Specification-level optimization. Specification-level test generation and implementation-level test generation are independent. The overall specification-level test generation time can be reduced by certain optimization methods.

Reuse between specification and implementation. No optimization for specification and implementation-level test generation, but the specification-level tests can be reused for implementation-level validation.

Specification-level optimization+reuse between specification and implementation. Optimizations reduce the overall specification-level test generation time, and the specification level tests can be reused for implementation-level validation.

Table 1.2 compares these approaches. Assume that during system validation we can find a specification-level test generation optimization that can produce α times ($\alpha > 1$) speedup compared to traditional test generation techniques. Also, assume that we can obtain β times ($\beta > 1$) speedup due to validation reuse. According to the comparison shown in Table 1.2, the last option is most beneficial. This book describes efficient test generation and reuse techniques to reduce the overall validation effort using the fourth (last) option.

1.2.4 System-Level Validation Challenges

Top–down design and validation flow is promising since the specification is significantly simpler to analyze, explore, optimize, and validate compared to complex implementation. Due to ever increasing consumer demand for complex applications coupled with other design constraints, the complexity of system-level specification is also increasing drastically. During system-level validation, each of the components (such as IP cores, processors and memories) can be verified using existing validation approaches. However, the validation of the overall system is extremely complex due to exponentially large number of possible interactions that are extremely hard to model, analyze and validate. More and more heterogeneous components (e.g., heterogeneous cores and GPUs) make the system-level SoC integration more difficult. Significant validation effort is needed to check the specified functional scenarios in system-level specification as well as to check the consistency between specification and implementation. To enable specification-driven validation, the following challenges need to be addressed:

Modeling and specification of complex SoCs. System-level specifications with formal unambiguous semantics can enable automated verification starting early in the design process. The fully validated specification can be used as a golden reference model for subsequent refinements (e.g., generation of RTL models) as well as automated reuse of high-level validation effort across abstraction levels. Two specific challenges are outlined below:

Table 1.2 Comparison of four options

Optimization	Time
None	$F_{\mathrm{Spec}} \times T_{\mathrm{Spec}} + F_{\mathrm{Total}} \times T_{\mathrm{Imp}}$
Specification-level optimization	$F_{\mathrm{Spec}} \times T_{\mathrm{Spec}}/\alpha + F_{\mathrm{Total}} \times T_{\mathrm{Imp}}$
Reuse	$F_{\mathrm{Spec}} \times T_{\mathrm{Spec}} + F_{\mathrm{Spec}} \times T_{\mathrm{Imp}}/\beta + F_{\mathrm{Imp}} \times T_{\mathrm{Imp}}$
Specification-level optimization + Reuse	$F_{\mathrm{Spec}} \times T_{\mathrm{Spec}}/\alpha + F_{\mathrm{Spec}} \times T_{\mathrm{Imp}}/\beta + F_{\mathrm{Imp}} \times T_{\mathrm{Imp}}$

- *Semiformal semantics:* Many SoC specification languages are semiformal, which hinders the automated validation process. Therefore, a challenging step is to extract formal models from high-level specifications to enable top–down validation of SoCs.
- *Lack of comprehensive coverage metrics:* Traditional validation approaches relies on code coverage, branch coverage etc. to evaluate the validation adequacy. Unfortunately, such coverage metrics are not enough to indicate the validation progress. It is a major challenge to identify a set of comprehensive functional fault models and associated coverage metrics for system-level validation.

Validation using Directed Tests. Due to time-to-market constraints, it is absolutely necessary to reduce the overall validation effort by using efficient directed tests. In this context, two specific challenges are outlined below:

- *Efficient generation of directed tests:* Directed tests are promising since it requires only a smaller test set compared to random tests for the same functional coverage goal. Currently, most directed test generation approaches assume the expert-knowledge which is time-consuming and error prone. Model checking based counterexample (directed test) generation can be fully automated. However, due to state space explosion problem, the cost of deriving a large set of tests is very costly and infeasible in many scenarios. It is important to significantly reduce the overall test generation complexity to make it applicable for complex SoC designs including multicore architectures with cache coherence protocols.
- *Compaction of directed tests:* Same coverage goal can be achieved using several-orders-of-magnitude less directed tests compared to random tests. However, the number of directed tests can still be extremely large (in the order of millions) for today's complex SoCs. An important challenge is how to reduce the redundancy in the generated tests without sacrificing the functional coverage goal.

Consistency between specification and implementation. A typical top–down design flow involves multiple reference models at different abstraction levels. The presence of multiple reference models raises an important question—how do we maintain consistency between so many reference models? When transforming system-level specification to lower abstraction levels, it is important to ensure consistency between different abstraction levels. One promising direction is to apply the same set of tests, assertions and properties on both specification and implementation. A major challenge is how to efficiently reuse the specification-level properties and tests for validation of SoC implementation.

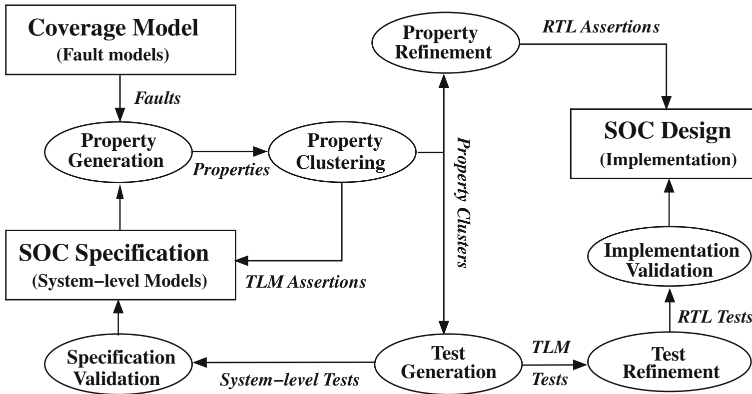


Fig. 1.10 Top–down validation of SoC designs

1.3 A Comprehensive Approach for System-Level Validation

The focus of this book is to present system-level validation techniques to reduce overall SoC validation effort. Figure 1.10 presents an overview of system-level validation framework. It consists of four major components as follows:

Modeling and specification of SoC designs. Modeling plays an important role in system-level validation. Chapter 2 describes various widely used SoC specification languages and how they can be translated to formal models to enable automated analysis and validation.

Coverage-driven property generation. Functional coverage is important to determine the adequacy of functional validation. Chapter 3 defines various fault models for SoC specifications. Based on these fault models, it describes techniques to automatically generate properties to validate the specified functional scenarios.

Directed test generation. The primary focus of this book is to discuss many efficient techniques for automated generation of directed tests for a wide variety of SoC designs. To alleviate the state explosion problem in the existing model checking based approaches, Chap. 4–11 describe efficient test generation techniques to reduce both test generation time and memory requirement.

Automated reuse of validation efforts. Chapter 12 presents a framework to automatically reuse high-level tests and assertions in implementation validation.

1.4 Book Organization

This book is organized as follows.

Chapter 2 [*Modeling and Specification of SoC Designs*]. This chapter presents how to extract formal models from high-level specification and fault models based on functional coverage metrics to enable high-level validation.

Chapter 3 [*Automated Generation of Directed Tests*]. This chapter describes how to automatically derive tests using various model checking based approaches, including unbounded model checking and SAT-based bounded model checking (BMC) techniques.
Chapter 4 [*Functional Test Compaction*]. This chapter proposes a test compaction approach that can significantly reduce the number of directed tests without sacrificing functional coverage goal.
Chapter 5 [*Property Clustering and Learning Techniques*]. This chapter presents various clustering and conflict clause based learning techniques to improve the overall test generation time.
Chapter 6 [*Decision Ordering based Learning Techniques*]. This chapter presents decision ordering based learning techniques that can efficiently reduce the test generation time for a single property as well as for a cluster of similar properties.
Chapter 7 [*Synchronized Generation of Directed Tests*]. This chapter presents an efficient BMC-based test generation technique that can simultaneously solve multiple properties to enable the maximum utilization of learned knowledge.
Chapter 8 [*Test Generation using Design and Property Decompositions*]. To alleviate the state space explosion problem, this chapter presents promising design and property decomposition techniques to reduce the test generation time.
Chapter 9 [*Learning-Oriented Property Decomposition Approaches*]. To simplify the subtest composition process in Chap. 8, this chapter presents a fully automated test generation approach based on the property decomposition and decision ordering based learning techniques.
Chapter 10 [*Directed Test Generation for Multicore Architectures*]. By reusing the learned knowledge from one core to other remaining cores, this chapter presents a novel BMC-based test generation technique for multicore architectures.
Chapter 11 [*Test Generation for Cache Coherence Validation*]. This chapter proposes an on-the-fly test generation approach for cache coherence protocols by efficiently traversing the state space structure of their corresponding global FSMs.
Chapter 12 [*Reuse of System-Level Validation Efforts*]. This chapter presents a framework that can reuse the generated tests and properties for implementation validation.
Chapter 13 [*Conclusions*]. Concludes the book and proposes interesting directions for further investigation.

References

1. Wolf W, Jerraya A, Martin G (2008) Multiprocessor system-on-chip (MPSoC) technology. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 27(10):1701–1713
2. Bailey B, Martin G, Piziali A (2007) ESL design and verification: a prescription for electronic system level methodology. Morgan Kaufmann/Elsevier, Burlington
3. Open SystemC Initiative (OSCI) (2006) SystemC. http://www.systemc.org
4. MATLAB. http://www.mathworks.com/products/matlab/
5. Berry G, Gonthier G (1992) The esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152

6. Simulink. http://www.mathworks.com/products/simulink/
7. SysML. http://www.sysml.org/
8. Peterson J (1981) Petri nets theory and the modeling of systems. Prentice-Hall, Englewood Cliffs
9. Mishra P, Dutt N (2008) Processor description languages: applications and methodlogies. Morgan Kaufmann, San Fransisco
10. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of international conference on hardware/software codesign and system, synthesis (CODES+ISSS), pp 19–24
11. Rose A, Swan S, Pierce J, Fernandez J (2005) Transaction level modeling in systemC. http://www.systemc.org
12. Object Management Group (2007) UML superstructure V2.1.2. http://www.omg.org/docs/formal/07-11-02.pdf. Accessed Nov 2007
13. Schutten R, Fitzpatrick T (2003) Design for verification methodology allows silicon success. EETIMES, Number: 16500856
14. Bentley B (2002) High level validation of next generation microprocessors. In: Proceedings of high level design validation and test (HLDVT), pp 31–35
15. Schubert T (2003) High level formal verification of next generation microprocessors. In: Proceedings of design automation conference (DAC), pp 1–6
16. Fine S, Ziv A (2003) Coverage directed test generation for functional verification using Bayesian networks. In: Proceedings of design automation conference (DAC), pp 286–291
17. Bryant R (1991) A methodology for hardware verification based on logic simulation. J ACM 38(2):299–328
18. Adir A, Asaf S, Fournier L, Jaeger I, Peled O (2007) A framework for the validation of processor architecture compliance. In: Proceedings of design automation conference (DAC), pp 902–905
19. Ezer S, Johnson S (2005) Smart diagnostics for configurable processor verification. In: Proceedings of design automation conference (DAC), pp 789–794
20. Puig-Medina M, Ezer G, Konas P (2000) Verification of configurable processor cores. In: Proceedings of design automation conference (DAC), pp 426–431
21. Roy A, Panda S, Kumar R, Chakrabarti P (2005) A framework for systematic validation and debugging of pipeline simulators. ACM Trans Des Autom Electron Syst (TODAES) 10(3):462–491
22. Adir A, Almog E, Fournier L, Marcus E, Rimon M, Vinov M, Ziv A (2004) Genesys-Pro: innovations in test program generation for functional processor verification. IEEE Des Test Comput 21(2):84–93
23. Shimizu K, Gupta S, Koyama T, Omizo T, Abdulhafiz J, McConville L, Swanson T (2006) Verification of the cell broadband engine processor. In: Proceedings of design automation conference (DAC), pp 338–343
24. Koo H, Mishra P (2006) Functional coverage-driven test generation for microprocessor verification. In: Proceedings of US-Korea conference (UKC), pp 19–24
25. Mishra P, Dutt N (2005) Functional verification of programmable embedded architectures: a top-down approach. Springer, Berlin
26. Camurati P, Prinetto P (1988) Formal verification of hardware correctness: introduction and survey of current research. IEEE Comput 21(7):8–19
27. Kern C, Greenstreet M (1999) Formal verification in hardware design: a survey. ACM Trans Des Autom Electron Syst (TODAES) 4(2):123–193
28. McMillan K (1993) Symbolic model checking: an approach to the state explosion problem. Kluwer Academic Publishers, Boston
29. Clarke E, Grumberg O, Peled D (2000) Model checking. The MIT press, Cambridge
30. Srivas M, Bickford M (1990) Formal verification of a pipelined microprocessor. IEEE Softw 7(5):52–64
31. Wilding M, Greve D, Hardin D (2001) Efficient simulation of formal processor models. Formal Methods Syst Des 18(3):233–248

32. Kuehlmann A, Eijk C (2001) Combinational and sequential equivalence checking. In: Logic synthesis and verification, Kluwer Academic Publishers, Norwell
33. Clarke E, Biere A, Ramimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Formal Methods Syst Des 19(1):7–34
34. Prasad M, Biere A, Gupta A (2005) A survey of recent advances in SAT-based formal verification. Int J Softw Tools Technol Transfer (STTT) 7(2):156–173
35. Ammann P, Black P, Majurski W (1998) Using model checking to generate tests from specifications. In: Proceedings of international conference on formal engineering methods (ICFEM), pp 46–54
36. Foster H, Krolnik A, Lacey D (2004) Assertion-based design, 2nd edn. Kluwer Academic Publishers, Boston
37. PSL working group (2005) Property Specification Language. http://www.eda.org/ieee-1850/
38. SVA committee (2004) SystemVerilog Assertion. http://www.eda.org/sv-ac/

Chapter 2
Modeling and Specification of SoC Designs

2.1 Introduction

As a system-level specification, SystemC transaction level modeling (TLM) [1] establishes a standard to enable fast simulation and easy model interoperability for hardware/software co-design. It mainly focuses on communication between different functional components of a system and data processing in each component. Although UML is being used as a de facto software modeling tool, UML Profile for SoC [2] has been proposed as an extension of UML 2.X to enable SoC modeling. It can be used to capture the system behavior for both SoC software and hardware components [3–5]. However, both SystemC TLM and UML diagrams are not formal enough for automatic analysis, especially for the validation using model checking techniques [6]. The inherent ambiguity, incompleteness, and contradiction in specifications can lead to different interpretations. Therefore, it is necessary to formalize the semantics of such SoC specifications.

This chapter focuses on the formal modeling of the two widely used SoC specifications. It describes how to automatically extract the formal models from specifications. Such formal models can be used for automated generation of directed tests (see the details in Chap. 3). This chapter is organized as follows. Section 2.2 presents both graph and finite state machine (FSM)-based modeling of systems. Section 2.3 describes the formal modeling of SystemC TLM designs. Section 2.4 describes formal modeling techniques of UML activity diagrams. Finally, Sect. 2.5 summarizes the chapter.

2.2 Modeling of Complex Systems

Modeling plays a central role in design automation of SoC architectures. The formal modeling can not only help designers accurately describe the syntax and semantics of a design, but can also enable the automatic analysis using corresponding tools.

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_2,

© Springer Science+Business Media New York 2013

This section presents two widely used formal models: the graph model for structure modeling and the FSM model for behavior modeling. The combination of both models can be used to capture the high-level abstraction of various complex SoC designs.

2.2.1 Graph-Based Modeling

In system level, there is no explicit boundary between software and hardware designs. A system can be considered as an interconnection between different functional components (e.g., tasks, modules, etc). Such structural information can be captured as a *graph model*.

Definition 2.1 For a system-level design, its graph model G is a directed graph denoted by the two-tuple *(V, E)*, where

- V is a set of nodes indicating different kinds of functional components.
- E is a set of edges indicating the communication channels between the components. ■

Consider $G = (V, E)$ as a graph model of a pipelined processor, which is derived from the architecture description language (ADL) specification [7]. In this graph model, the node notation V denotes two types of components in the processor: *units* (e.g., fetch, decode, etc) and *storages* (e.g. register file, memory, etc.). The edge notation E indicates two types of edges: *pipeline edges* and *data transfer edges*. A pipeline edge transfers an instruction (operation) from a parent unit to a child unit. A data-transfer edge transfers data between units and storages. This graph model is similar to the pipeline-level block diagram available in a typical architecture manual.

For illustration, consider a simplified version of a MIPS processor [8]. Fig. 2.1 shows the graph model of the processor. In this figure, rectangular boxes denote units, dashed rectangles are storages, bold edges are instruction-transfer (pipeline) edges, and dashed edges are data-transfer edges. A path from a root node (e.g., Fetch) to a leaf node (e.g., WriteBack) consisting of units and pipeline edges is called a *pipeline path*. For example, one of the pipeline paths is *Fetch, Decode, IALU, MEM, WriteBack*. A path from a unit to main memory or register file consisting of storages and data-transfer edges is called a *data-transfer path*. For example, *MEM, DataMemory, MainMemory* is a data-transfer path.

2.2.2 FSM-Based Behavior Modeling

Although the graph model can be used to describe the structural information, it is not suitable to capture behavioral details. FSM is widely used for describing the internal behavior of software/hardware components. For example, in hardware design, FSM

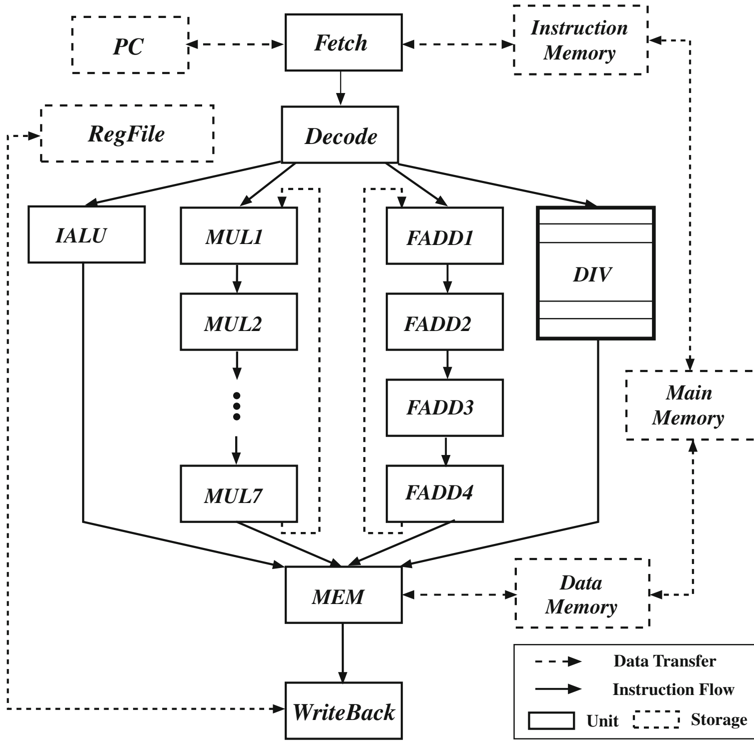


Fig. 2.1 Graph model of the MIPS processor

can be used to describe the state change of registers. In software design, FSM can be used to describe the execution of a piece of sequential code. FSM models can be derived from system-level specifications (e.g. ADL [7], SystemC TLM, UML).

Definition 2.2 A finite state machine M is a seven-tuple $(I, O, S, \delta, \lambda, s_i, s_F)$ where

- I is a finite set of inputs.
- O is a finite set of outputs.
- S is a finite set of states.
- δ is the state transition function $\delta : S \times I \rightarrow S$.
- λ is the output function $\lambda : S \times I \rightarrow O$.
- s_i is an initial state, an element of S.
- s_F is the set of final states, a subset of S. ■

When the model M is in the state s ($s \in S$) and receives an input a ($a \in I$), it moves to the next state specified by $\delta(s, a)$ and produces an output given by $\lambda(s, a)$. For an initial state s_1, an input sequence $x = a_1, \ldots, a_k$ takes the M successively to states $s_{i+1} = \delta(s_i, a_i)$, $i = 1, \ldots, k$ with the final state s_{k+1}.

It is important to note that multiple smaller FSMs can be composed to form a larger FSM. In other words, each functional unit can be modeled as an FSM, and the composition of all the functional units is also an FSM [9]. As an example in the composite FSM model of a pipelined processor, a state indicates the value stored in all the registers of the processor. Assuming that each input a_i corresponds to instruction(s) fetched from instruction cache (or memory), the input instruction sequence $x = a_1, \ldots, a_k$ can be used as a test program to exercise the path consisting of the states and the state transitions from s_1 to s_{k+1}.

2.3 Specification Using SystemC TLMs

As a framework built on C++, SystemC [10] mimics the hardware description languages such as VHDL and Verilog. With an event-driven simulation kernel, SystemC can be used to simulate the behavior of concurrent processes which can communicate with each other using procedure calls or other mechanisms offered by the SystemC library. Generally, SystemC is often associated with TLM [1, 11], because SystemC TLM provides a wrapper to facilitate the process of communication modeling. Since SystemC TLM provides a rapid prototyping platform for the architecture exploration and hardware/software integration [12], it is widely used to enable early exploration for both hardware and software designs. It can reduce the overall design and validation efforts of complex SoC architectures.

To enable automated analysis, various researchers have tried to extract formal representations from SystemC TLM specifications. Abdi et al. [13] introduced *Model Algebra*, a formalism for representing SoC designs at system level. The work by Kroening et al. [14] formalized the semantics of SystemC by means of labeled Kripke structures. Moy et al. [15] provided a compiler front-end that can extract architecture and synchronization information from SystemC TLM designs using HPIOM. Karlsson et al. [16] translated SystemC models into a Petri-Net based representation PRES+. This model can be used for model checking of properties expressed in a timed temporal logic. Habibi et al. [17] proposed a method that adopts the formal model AsmL. A state machine generated from AsmL can be verified, and then can be translated into both SystemC code and properties for low-level validation. All these modeling techniques focus on the formal modeling of SystemC specifications. This section discusses how to extract the formal models from SystemC TLM specifications to enable automated test generation.

2.3.1 Modeling of SystemC TLM Designs

As a system-level specification, SystemC TLM emphasizes the functionality of the data transfers instead of actual implementation. A SystemC TLM design interconnects a set of processes communicating with each other using transactions data tokens

(i.e., C++ objects). The initial process starts a communication, and the target process passively responds to the communication. Similar to the producer/consumer models, each process does the following tasks: consuming data, processing data, and producing data.

Since SystemC is based on C++, it supports various programming constructs (e.g., template, inheritance, etc.). Although the concept of some TLM components (signals, ports, etc.) is easy, their C++ implementation details are really complex. Therefore, it is difficult to directly translate their behaviors to enable automated validation. In this chapter, abstraction of certain SystemC components is used to hide the implementation details using the predefined SMV constructs. The underlying complex SystemC scheduler aggravates the modeling complexity. For SystemC TLM, to mimic the parallel execution of processes, the SystemC scheduler activates the *ready-to-run* processes in a "non-deterministic" way. Depending on the target model, translation of SystemC scheduler may or may not be required. For example, while translating SystemC TLM specification into SMV model, it is not necessary to model the SystemC scheduler explicitly since SMV inherently supports parallel execution.

For TLM, two most important factors are the *transaction data token* and the *transaction flow*. So the extracted formal model of TLM specifications should reflect both information. The extracted models should not only guide the generation of SMV specification, but should also enable automatic generation of properties and tests. Definition 2.3 provides the formal model of SystemC TLM designs.

Definition 2.3 The **formal model** of a SystemC TLM design is an eight-tuple $(\Sigma, P, T, A, E, M, I, F)$ where

- Σ is a set of transaction data tokens.
- $P = \{p_1, p_2, \ldots, p_m\}$ is a set of places.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a set of transitions.
- $A \subseteq \{P \times T\} \cup \{T \times P\}$ is a set of arcs between places and transitions.
- $E = \{e_1, e_2, \ldots, e_k\}$ is a set of arc expressions. The mapping $Expression(a_i) = e_i$ $(a_i \in A, 1 \le i \le k)$ provides the enable condition e_i for a_i. A token can pass arc a_i only when e_i is true.
- $M : 2^{P \times \Sigma} \times T \rightarrow 2^{P \times \Sigma}$ is a function that describes the internal operations on input transaction data and output transaction data of a transition.
- $I \in 2^{P \times \Sigma}$ specifies the initial state.
- $F \subseteq 2^{P \times \Sigma}$ specifies the final states. ■

Graph model can be used as an immediate form to capture the execution as well as interconnection of processes.

Figure 2.2a shows an interconnection of six modules. Each arrow indicates a port binding between two modules. Figure 2.2b shows the graph representation of its corresponding formal model. In the graph model, each circle (node) is called a *place* that is used to indicate the input or output buffer of a module. It can temporarily hold the transaction data for later processing. The edges (vertical bars with incoming and outgoing arrow lines) are *transitions*, which are used to indicate modules that contain processes to manipulate input and output transaction data tokens. The places

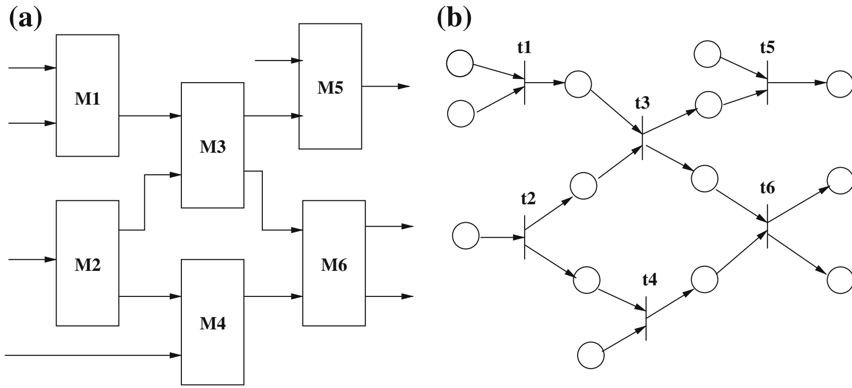


Fig. 2.2 Mapping from a SystemC structure to corresponding graph model. **a** Interconnection of modules. **b** Graph model of the module interconnections

without incoming arcs are *initial places* which start a transition. The places without outgoing arcs are *target places*. A transaction data token flows from the initial places to the target places and token values may change in transitions when necessary. The internal logic of a transition determines the flow of the transaction. It is a piece of code which can be modeled by an FSM model.

2.3.2 Transformation from SystemC TLM to SMV

As a popular model checking specification, SMV [18] is widely adopted to describe both the structure and behavior information of SystemC TLMs. This is because of the following reasons. First, the underlying semantics of SMV is similar to the semantics of SystemC scheduler. So we can mimic most TLM behaviors using SMV without modeling complex scheduler behavior. Second, SMV and TLM have the similar structure hierarchy. Each processing unit encapsulated by a TLM module corresponds to an SMV module. The interconnections (e.g. channels, ports and sockets) between TLM modules can be abstracted by using module parameters in SMV. Third, like SystemC, SMV provides a rich set of programming language constructs such as *if–then-else*, *case-switch* and *for loop* statements. Fourth, SMV main module connects, similar to SystemC, each component of the system. Finally, SMV supports various kinds of data types and data operations. Especially users can define their own data type. All of these SMV features facilitate the translation from TLMs into SMV specification.

As an intermediate form for TLM to SMV translation, the formal model defined in Definition 2.3 provides both structure and behavior information. Such information needs to be collected for translation into a SMV representation. The structure information includes the data-type definition and connectivity between modules. It

```
class packet{                    typedef packet struct{
public:
  sc_uint<2> to_chan;              to_chan : 0..3;
  sc_uint<6> payload_sz;           payload_sz : 0..63;
  sc_uint<8> payload[4];           payload : array 3..0 of 0..255;
  sc_uint<8> parity;               parity : 0..255;
};                               }
 a)packet in SystemC TLM           b) packet in SMV
```

Fig. 2.3 An example of data type transformation

corresponds to the description of transaction data token as well as interconnection of transitions and places in the graph model. The behavior information contains token processing and token routing. In the formal model, it represents the internal processing of a transition. This section discusses how to extract both structural and behavioral information (i.e., the graph model and FSM models) and transform it into an SMV specification. We use the example shown in Fig. 2.8 to illustrate how to extract the formal model from a router example.

2.3.2.1 Structure Extraction

In TLM, the content of a transaction data token indicates the transaction flow and the output of each component. Generally, a transaction token consists of several attributes of different types. Because data type determines the size of the specified variable which in turn affects the model checking performance, it is necessary to figure out the data type of a token. Besides all native C++ types, SystemC defines a set of data type classes within the namespace *sc_dt* to represent values with application-specific word lengths applicable to digital hardware. SMV also supports various data types such as array, Boolean, integer, struct, and so on. Such data type definitions facilitate the mapping of data types between SystemC TLM and SMV specification. During the transformation, the word lengths of user-defined type need to be considered. Figure 2.3 shows an example of the router *packet* in the form of SystemC TLM and SMV respectively. For example, $sc_uint < 2 >$ has 2 bits and will be transformed into a range $0-3$ in SMV.

Derived from the base class *sc_module*, TLM modules are the main processing units for the transaction data. Generally each *sc_module* contains the definitions of processes whose types are SC_METHOD or SC_THREAD. Modules communicate with each other by sending and receiving transaction data tokens via output and input ports. SystemC provides a communication wrapper for the system components (modules). Various binding mechanisms exist in SystemC (e.g., port to export binding, export to export binding, and port to channel binding) to establish interconnection between modules. Usually each binding corresponds to a channel (e.g., a FIFO channel) to temporarily hold transaction tokens.

```
class router : public sc_module{
 public:
     sc_export<tlm_put_if<packet> > packet_in;
     sc_export<tlm_fifo_get_if<packet> > packet_out0;
     sc_export<tlm_fifo_get_if<packet> > packet_out1;
     sc_export<tlm_fifo_get_if<packet> > packet_out2;

     router(sc_module_name module_name);
     void route();
 private:
      tlm_fifo<packet> chan0, chan1, chan2, input_;
      packet tmp_packet;
};
```

Fig. 2.4 An example of SystemC TLM module

Figure 2.4 shows the TLM module structure of a router. The class *sc_export* can be used as a port to communicate with other modules. Because the interface type of port *packet_in* is $tlm_put_if<packet>$, it is an input port. In contrast, $packet_outx$ ($x = 0, 1, 2$) have the interface $tlm_fifo_get_if<packet>$, so they are output ports. During the router communication, each connection between a port and an export uses a FIFO channel to temporarily hold a packet.

Structurally similar to SystemC TLMs, SMV specification is also modularized and hierarchically organized. So the extraction of structure information needs to map the TLM constructs in the right place of the SMV specification. Figure 2.5 shows the SMV module skeleton corresponding to example in Fig. 2.4 after the structure extraction. In SMV, a module uses the parameters as the input and output ports to both communicate with other modules and configure the system status defined in the *main* module. In the example of Fig. 2.5, the SMV module has one input port and three output ports. The type of the input and output ports is packet. All the declarations of member variables except for the FIFO channels are declared in the SMV specification. Because a FIFO channel together with its port pairs are abstracted as an SMV parameter, it is not necessary to create a variable in SMV explicitly. Based on context during the elaboration, some of the declared variables will be initialized. In SMV specification, each output ports and local variables need to be initialized. For example, $packet_out0$ is a parameter which refers to an output port, so it will be initialized with a value “0”. During the translation, it is required that all such module connections should be defined in the module *sc_top*.

2.3.2.2 Behavior Extraction

TLM behavior describes the run-time information of TLM including transaction creation, transaction manipulation, and module communication. Transaction creation initializes a transaction by creating a data token (i.e., a C++ object) with proper

```
module router(packet_in, packet_out0, packet_out1, packet_out2){
    input   packet_in : packet;
    output  packet_out0, packet_out1, packet_out2: packet;
    tmp_packet : packet;

    init(packet_out0):=0;
    init(packet_out1):=0;
    init(packet_out2):=0;
    init(tmp_packet):=0;
    ......
}
```

Fig. 2.5 An example of SMV module

```
router::router( sc_module_name mname ): sc_module(mname){
  packet_in(input_);   packet_out0(chan0);
  packet_out1(chan1);  packet_out2(chan2);
  SC_METHOD(route);
  sensitive << input_.ok_to_get();
  dont_initialize();
}
void router::route() {
  input_.nb_get(tmp_packet);
  if(tmp_packet.to_chan == (sc_uint<2>)0)
     chan0.nb_put(tmp_packet);
  else if(tmp_packet.to_chan == (sc_uint<2>)1)
     chan1.nb_put(tmp_packet);
  else chan2.nb_put(tmp_packet);
}
```

Fig. 2.6 An example of TLM process

values. Transaction execution describes the transaction flow among the modules. A module is a container which has a cluster of relevant processes. Such processes will handle the incoming transaction tokens and decide where to send them according to the specified conditions. Thus, a different value of a token will lead to different transaction flows. The following two types of process communication are widely supported in transaction flows: (1) direct procedure call from one process to another process, and (2) channel-based events triggered by the procedure call. For example, in the blocking mode, a process can fetch a transaction data token from the specified input port only when the corresponding channel is not empty. Otherwise, the operation "get" will be blocked until there is an event triggered by the "put" operation by other processes.

Figure 2.6 shows the module process $route$ of the router example. The process receives a packet from the driver via channel $input_$, and then it decides where to send data based on the packet header information to_chan.

TLM modeling provides some synchronization mechanism for the communications between modules. As shown in Fig. 2.6, the router can fetch the data from the

Fig. 2.7 An example of SMV process

```
module router(packet_in, packet_out0,
                packet_out1, packet_out2){
    ......
    next(tmp_packet) := packet_in;
    if(tmp_packet.to_chan = 0){
        next(packet_out0) := tmp_packet;
        next(packet_out1) := 0;
        next(packet_out2) := 0;
    }else if(tmp_packet.to_chan = 1){
        next(packet_out0) := 0;
        next(packet_out1) := tmp_packet;
        next(packet_out2) := 0;
    }else{
        next(packet_out0) := 0;
        next(packet_out1) := 0;
        next(packet_out2) := tmp_packet;
    }
}
```

FIFO queue $input_$ only when the driver puts a package and the FIFO channel event ok_to_get is triggered. Thus the synchronization between two modules is implicitly achieved.

SMV supports many constructs similar to the common programming language such as *if–then-else*, *switch-case* and *for loop*. So these constructs facilitate the behavior modeling of processes from TLMs to SMV specifications. Figure 2.7 shows the translated SMV specification of the TLM example presented in Fig. 2.6. During the translation from TLM into SMV, a channel is abstracted as an implicit buffer between two ports. So an SMV module will get the input data from its input ports. There is no mapping of the channel in the transformed SMV specification. For example, the tmp_packet is assigned the value of the $packet_in$ instead of the value of $input_$ shown in the TLM example in Fig. 2.6.

2.3.3 Case Study: A Router Example

A prototype tool called $TLM2SMV$ was developed to enable the transformation from SystemC TLM specifications to corresponding SMV models for automated directed test generation. The details of the implementation are described in Sect. 12.3.3.

Figure 2.8 shows the TLM structure of a router design. The router consists of five modules: one master, one router, and three slaves. The SystemC program consists of four classes (one class for packet definition, one class for the driver, one class for the router, and one class for the slave), eight functions, and 143 lines of code. The main function of the router is to analyze and distribute the packets received from the master to target slaves.

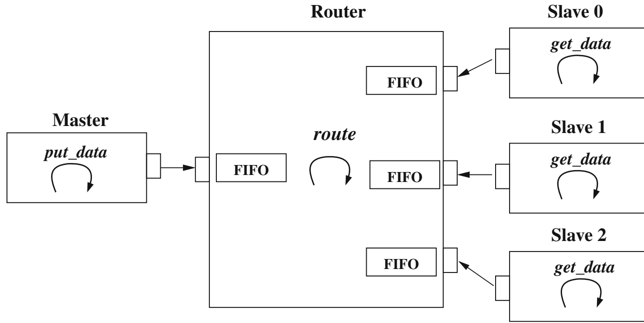


Fig. 2.8 The TLM structure of the router

At the beginning of a transaction, the master module creates a packet. Then, the driver sends the packet to the router for package distribution. The router has one input port and three output ports. Each port is connected to a FIFO buffer (channel) which temporarily stores packets. The router has one process $route$ which is implemented as an SC_METHOD. The $route$ first collects a packet from the channel connected to the driver, decodes the header of the packet to get the target address of a slave, and then sends the packet to the channel connected to the target slave. Finally, the slave modules read the packets when data are available in the respective FIFOs. The transaction data (i.e. packet) flows from the master to its target slave via the router. The flow is controlled by the address to_chan in the packet header. By using the proposed approach in Sect. 2.3.2, the automatically generated SMV model contains four modules and 145 lines of code.

2.4 Specification Using UML Activity Diagrams

Formal verification can be used to verify the correctness of specifications, so it can be used to guarantee the quality of UML models [19]. UML activity diagram adopts Petri-net semantics which is promising to describe the concurrent behavior [20, 21]. There are several approaches that use model checking techniques to verify UML activity diagrams. Eshuis [22] presented a translation procedure from UML activity diagrams to the input language of NuSMV [23]. However, the translation is used to verify the consistency between UML activity diagrams and class diagrams. It focuses on checking the consistency between two different models. Guelfi and Mammar [24] provided a formal definition for timed activity diagrams. They outlined the translation from the semantic specifications into PROMELA—an input language of the SPIN model checker. Das et al. [25] proposed a method to deal with timing verification of UML activity diagrams. All these verification work primarily focus on checking the consistency or correctness of the model itself instead of generating directed test cases.

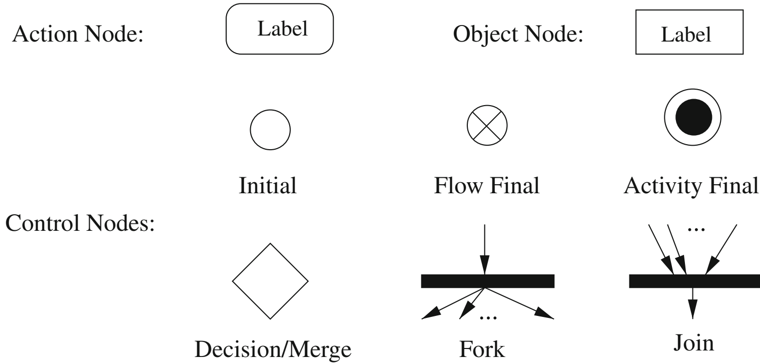


Fig. 2.9 UML activity nodes

In this section, UML 2.1.2 [26] is used as the SoC specification. To reduce the complexity of the testing work, we restrict the testing target and investigate a subset of activity diagrams. The subset mainly contains action nodes, control nodes, object nodes, and control and data flow. Especially for the object node, we assume that it can hold at most one object at a time and it does not support *competition* and *data store*.

2.4.1 Graphic Notations

UML activity diagram is used to coordinate the execution of actions. An action takes a set of inputs and converts them into corresponding outputs. An activity (behavior) consists of a set of actions and flow edges. The actions are connected by object flow edges to show how object tokens flow through and connected by control flow edges to indicate the execution order.

UML activity diagrams adopt the semantics like Petri-net [27]. It is a type of directed graphical representation. Tokens which indicate control or data values flow along the edges from the source node to the sink nodes driven by the actions and conditions. An activity diagram has two kinds of modeling elements: activity nodes and activity edges. More specially, there are three kinds of nodes in activity diagrams:

- *Action Node*. Action nodes consume all input data/control tokens when they are ready; generate new tokens; and send them to output activity edges.
- *Object Node*. Object nodes provide and accept data tokens, and may act as buffers, collecting data tokens as they wait to move downstream.
- *Control Node*. Control nodes route tokens through the graph. The control nodes include constructs to choose between alternative flows (decision/ merge), to split or merge the flow for concurrent processing (fork/ join).

Figure 2.9 shows the basic constructs of activity nodes. An action node is denoted by a round cornered box. It represents an execution of operations on input tokens, and the generated new tokens will be delivered to outgoing edges. An object node denoted by a rectangle box is used to temporarily hold the data tokens waiting to be processed or delivered. For simplicity, it is assumed that object nodes do not support *competition* and *data store* for test case generation. A flow in an activity starts from the initial node. When a token arrives at a flow final node, it will be destroyed. The flow final node has no outgoing edges, so there is no downstream effect. When no tokens exist in an activity diagram, the activity will be terminated. The activity final nodes are similar to flow final nodes, except that when a token reaches one activity final node, the entire flow will be terminated. Decision nodes and merge nodes use the same shape of diamond. Decision nodes choose one of the outgoing flows according to the value of Boolean expressions labeled on the outgoing edge. Merge nodes select only one of the incoming flows to deliver to the next activity node. Forks or joins are shown by multiple arrows leaving or entering the synchronization bar, respectively, to describe the concurrent behavior of a system. When a token arrives at a fork node, it will be duplicated across the outgoing edges. Join nodes synchronize multiple flows. The tokens must be available on every incoming edge in order to be passed to outgoing edges.

Activity nodes are connected by activity edges along which tokens may flow under some condition. Activity edges include control and data flow edges as follows:

- *Control Flow Edge*. Control flow edges indicate the execution sequence of actions.
- *Object Flow Edge*. Object flow edges indicate the relation of data token transmissions. It provides the inputs to actions.

To simplify the discussion, we combine the control and data token together as a new kind of token which contains both control and data information. Such token can flow through activity edges. In other words, we do not distinguish control flow edges and object flow edges.

Figure 2.10 shows an example which uses most of the elements shown in Fig. 2.9. It describes the functionality of withdrawing money from an automated teller machine (ATM) [28]. A user needs to enter the access code first. In case of failure, the user can input the access code again. The operation will abort if access code is wrong in both cases. If the input access code is right, the user can enter the amount of money he wants to withdraw. At the same time, the printer will be ready to print a receipt. Once the ATM decides whether there is enough money the user can withdraw, it provides the cash and generates the information for this transaction. Finally, the printer prints the receipt and the transaction is complete.

The token for this example contains the ATM transaction information such as the input access code and input cash amount, the context information such as the available cash amount and correct access code. In general, a token reflects all the data information required for this activity. Table 2.1 shows the composition of a token of the ATM activity diagram. It consists of five variables which will be used to make the decisions illustrated in Table 2.2.

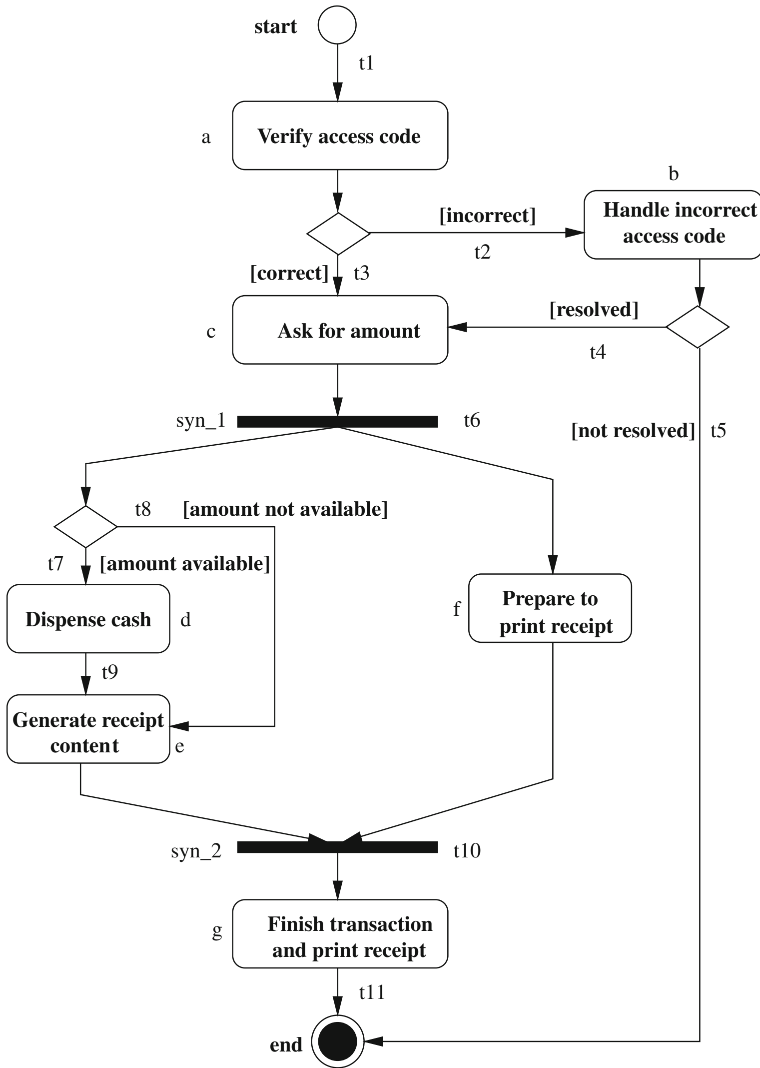


Fig. 2.10 The UML activity diagram of an ATM

2.4.2 Formal Modeling of UML Activity Diagrams

Without formalism, it is hard to describe and model the activity diagrams accurately. UML activity diagram itself is a semi-formal specification that cannot be directly mapped to a model checker input (e.g., SMV models). In practice, Petri-net is used as an intermediate formal model between activity diagrams and SMV model, because

Table 2.1 Breakdown of a token in Fig. 2.10

Variable	Type	Description
$access_code$	String	User's access code
$access_code_input$	String	User access code input
$access_code_resolve$	String	User access code input correction
$amount_input$	Integer	User cash amount input
$amount_available$	Integer	Cash amount available

Table 2.2 Condition on the flow edges in Fig. 2.10

Flow edge	Condition	Description
t2	Incorrect	$access_code! = access_code_input$
t3	Correct	$access_code = access_code_input$
t4	Resolved	$access_code = access_code_resolve$
t5	Not resolved	$access_code! = access_code_resolve$
t7	Amount available	$amount_input <= amount_available$
t8	Amount not available	$amount_input > amount_available$

the Petri-net formalism can capture the major functional scenarios as well as guide the translation.

Definition 2.4 describes the relation between the activity nodes and flow edges with a Petri-net semantics. Although it does not model the full features of activity diagrams, it formally depicts the static abstracted structure of activity diagrams which can be used to describe the scenarios that need to be tested.

Definition 2.4 An activity diagram is a directed graph described using an eight-tuple $(A, T, F, C, V, A, a_I, a_F)$ where

- $A = \{a_1, a_2, \ldots, a_m\}$ is a set of action nodes.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a set of completion transitions.
- $F \subseteq \{A \times T\} \cup \{T \times A\}$ is a set of flow edges between activity nodes and completion transitions.
- $C = \{c_1, c_2, \ldots, c_n\}$ is a finite set of guard conditions. Here, c_i $(1 \leq i \leq n)$ is a predicate (expression) based on the input variables. There is a mapping from $f_i \in F$ to c_i, referred as $Cond(f_i) = c_i$.
- Let V be the set of all possible assignments for input variables $V_1, V_2, \ldots, V_k$ where k is a positive integer.
- $M : A \times V \rightarrow V$ is a mapping that describes the value change of the input variables inside an activity node.
- $a_I \in A$ is the *initial node*, and $a_F \in A$ is the *final node*. There is only one completion transition $t \in T$ and $c \in C$ such that $(a_I, t) \in F$, and for any $t' \in T$, $(t', a_I) \notin F$ and $(a_F, t') \notin F$. ■

A node can be an action node, an initial node or a final node. The *completion transition* and *flow edge* are used to model the behavior of the control nodes. In the graph, the nodes are connected by flow edges associated with a completion transition. Because activity diagrams allow tokens to exist in the flows concurrently, the completion transition can be used to synchronize the token flows. If a completion transition has multiple incoming flow edges, it will do the join operation. If there are multiple outgoing flow edges, then it will do the fork operation. For each flow, e.g., there may be a condition which can guide the token traverse. The graph has one initial node that indicates the start of control and data flows. Activity diagrams have two kinds of final nodes: flow final nodes and activity final nodes. We can combine them together and use a join operation to get a new activity final node. So in the definition there is only one final node.

When analyzing dynamic behaviors of an activity diagram, we need to use the *states* (a set of actions executing concurrently) to model the status of a system. Current state (denoted by CS) of an activity diagram indicates the actions which are being activated.

Definition 2.5 Let D be an activity diagram. The *current state* CS of D is a subset of A. For any transition $t \in T$,

- ${}^\bullet t$ denotes the preset of t, then ${}^\bullet t = \{ a \mid (a, t) \in F\}$.
- $t^\bullet$ denotes the postset of t, then $t^\bullet = \{a \mid (t, a) \in F\}$.
- $enabled(CS)$ denotes the set of completion transitions that are associated with the outgoing flow edges of CS, then $enabled(CS) = \{ t \mid {}^\bullet t \subseteq CS \}$.
- *firable(CS)* denotes the set of transitions that can be fired from CS, then *firable(CS)*=$\{ t \mid t \in enabled(CS) \bigwedge {}^\bullet t$ *are all completed* $\bigwedge \exists n \in A.\ Cond((t, n))$ is satisfied $\bigwedge (CS - {}^\bullet t) \cap t^\bullet = \emptyset\}$. After some t is fired, the new current state $CS' = fire(CS, t) = (CS - {}^\bullet t) \cup t^\bullet$. ■

The current state of an activity diagram indicates which activity nodes are holding the tokens. For example, when $\{d, f\}$ is the current state of the activity diagram in Fig. 2.10, two tokens are in the activity nodes d and f individually. At this time, only the transition associated with t_9 is firable. If it is fired, then the next state is $\{e, f\}$.

Because of the inherent concurrency, several transitions can be fired at the same time. For an activity diagram, all the firable transitions in a state form a *concurrent transition*.

Definition 2.6 Let D be an activity diagram. For a state CS of D, a concurrent transition τ is a set of completion transitions $t_1, t_2, ..., t_n \in firable(CS)$ where

1. $\forall i, j (1 \leq i < j \leq n), {}^\bullet t_i \cap {}^\bullet t_j = \emptyset$;
2. $\forall t \in (enabled(CS) - \{t_1, t_2, \ldots, t_n\})$, there exists an i $(1 \leq i \leq n)$ such that ${}^\bullet t \cap {}^\bullet t_i \neq \emptyset$.

After firing τ from state CS, the current state $CS' = fire(CS, \tau) = \bigcup_{i=1}^{n} (fire(CS, t_i)) = \bigcup_{i=1}^{n}((CS - {}^\bullet t_i) \bigcup t_i^\bullet)$. ■

An instance of dynamic behavior of an activity diagram can be represented by a sequence of states and concurrent transitions. We call it a *path* of the activity diagram. Because a path may have cycles, during the model checking, it is hard to determine the cycle numbers, so we neglect the cycles on a path. We call such a path as *key path*.

Definition 2.7 A path ρ of the activity diagram D is a sequence of states and concurrent transitions, let

$$\rho = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{n-1}} s_n$$

where $s_0 = \{a_I\}$, $s_n = \{a_F\}$, and $s_{i+1} = fire(s_i, \tau_i)$ for any i $(0 \leq i < n)$. ρ is a *key path* if there is no state repetition in ρ, i.e. $\forall i, j$ $(0 < i < j \leq n)$, $s_i \bigcap s_j = \emptyset$. ■

There are five key paths in Fig. 2.10:

- $\rho_1 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t5\}} \{end\}$
- $\rho_2 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- $\rho_3 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t8\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- $\rho_4 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- $\rho_5 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t8\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$.

The *dummy* node is inserted here because it assumes that outgoing edges of the fork node must connect to an activity rather than a selection node. For a key path, when firing transitions, we need to consider guard conditions. For clarity, in Fig. 2.10, we did not label the condition guards for each transition.

Definition 2.8 Let D be an activity diagram. An interaction of the activity diagram is a set of activity nodes (actions) that can be activated simultaneously. A "k-interaction" is a set that contains k activity nodes. ■

In order to detect whether a concurrent state of an activity diagram is reachable or can be activated, we use the term *interaction*[1] to describe the scenario that a set of actions can be activated simultaneously. For example, in Fig. 2.10, $\{d, f\}$ is an example of " 2-interaction" in the ATM.

[1] Unlike the interaction in UML Interaction overview diagram, the interaction here means that several actions are activated at the same time.

2.4.3 Transformation from UML Activity Diagrams to SMV

By parsing a UML activity diagram, both the control and data flows can be extracted. The translation consists of two parts: static information extraction and dynamic information extraction. Static information extraction analyzes the structure of an activity diagram and then generates a skeleton of the SMV input. The dynamic information extraction analyzes the dynamic behavior of the system by focusing on control and data flow analysis (i.e., the state change of activities, data manipulation in activities and the condition of the transitions).

2.4.3.1 Static Information Extraction

This step collects both the input data manipulated by the activities and the predicates used as guard conditions of the transitions. For example in Fig. 2.10, there are five input data variables that determine the data and control flows: $access_code$, $access_code_input$, $access_code_resolve$, $amount_input$, and $amount_available$. Since there may be a large number of possible values for a variable, during model checking it will cause the state space explosion. SMV does not support complex data types (e.g., float, double, etc.). For each variable, it is required that the value range should be specified explicitly. To avoid state space explosion, the following methods can be used to reduce the complexity of data types:

- *Scaling*. Scaling is to proportionally reduce the value range of a variable.
- *Reduction*. Reduction is to reduce the cardinality of possible values for a variable.

Since it is hard to implement the above techniques automatically, before the SMV translation, the variable type information is tuned manually for activity diagrams.

During translation, each activity is assigned with a *state variable* which has three possible state values: *unvisited* (0), *unvisited* (1) and *visited* (2). *visited* indicates that no token has passed through this activity node. *Visiting* indicates currently the activity is holding one or more tokens. *visited* indicates that some token has passed through this activity node and currently there is no token in this activity node. The extraction procedure instantiates the activity state variables and assigns suitable values to them. During initialization, the initial activity node is assigned *unvisited* that means there is a token ready at the initial state. Other nodes are initialized to $unvisited$. Also, each flow edge is assigned with a state variable which has two possible values: $fired$ (1) and $unfired$ (0). $Fired$ means some tokens have flowed from the incoming activity nodes to its outgoing activity nodes. $Unfired$ means no token has passed through this activity edge. Initially, they are set with value 0.

Figure 2.11 shows the generated skeleton of Fig. 2.10 in SMV format [18, 23]. There are three modules in this skeleton. The module $state$ defines the token information (described in Table 2.1) as well as the state variable for activity nodes and flow edges. For example, $verify_access_code$ is a state variable for an action with three states. Initially it is assigned the state $unvisited$ (0). Module ATM shows a

Fig. 2.11 The generated skeleton after structure extraction

```
MODULE state
 VAR
   access_code: { A1, B1, C1 };
   access_code_input: { A1, B1, C1, D1 };
   start: 0..2;
   syn_1: 0..2;
   verify_access_code: 0..2;
   t2_cond: 0..1;
   t3_cond: 0..1;
   ......
 ASSIGN
   init(start):=1;
   init(syn_1):=0;
   init(verify_access_code):=0;
   ......
MODULE ATM(st)
 ASSIGN
   next(st.start):=
   next(st.t2_cond):=
   ......
   next(st.prepare_print_receipt):=
   ......
   next(st.dispense_cash):=
   next(st.t7_cond):=
   ......
MODULE main() {
   st: state;  atm: ATM(st);
   p_print:  prepare_print(st);
   check:  check_amount(st);
}
```

static skeleton without dynamic behavior information. In this phase, variables are collected without any processing. The missing state transition details will be described in Sect. 2.4.3.2. The module *main* creates the module instances and elaborates them together. For example, *st* is an instance of state module and *atm* is an instance of ATM module. Both the *st* and *atm* are bound together, because *atm* will handle the state changes of variables in *st*.

2.4.3.2 Dynamic Information Extraction

After static information extraction, it needs to extract both data manipulations and transitions of state variables, because they will determine the data and control flows. Figure 2.12 defines a set of rules that specify the state transition for each activity node and the value changes of each data. In these rules, the preset and postset notations are used. In these rules, the assignment and constraint to a set means the assignment and constraint to each element in the set. For example, if ${}^{\bullet}t = \{a_1, a_2, \ldots a_k\}$,

Rule 1: If n is an initial node
$init(n) := 1;$
$next(n) := 2;$

Rule 2: If n is a final node, and there are k incoming transitions $t_1, t_2 \ldots t_k$.
$init(n) := 0;$
$next(n) := case$
$((^\bullet t_1 = 1 \ \& \ cond(t_1)) \mid (^\bullet t_2 = 1 \ \& \ cond(t_2)) \mid$
$\ldots \mid (^\bullet t_k = 1 \ \& \ cond(t_k))) : 2;$
$1 : n;$
$esac;$

Rule 3: If n is an activity node (not join or fork), and there are k incoming transitions $t_1, t_2 \ldots t_k$.
$init(n) := 0;$
$next(n) := case$
$n = 1 : 2;$
$(^\bullet t_1 = 1 \ \& \ cond(t_1)) \mid (^\bullet t_2 = 1 \ \& \ cond(t_2)) \mid$
$\ldots \mid (^\bullet t_k = 1 \ \& \ cond(t_k))) : 1;$
$1 : n;$
$esac;$

Rule 4: If n is a fork node, and the corresponding transition is t.
$init(n) := 0;$
$next(n) := case$
$n = 1 \ \& \ t^\bullet > 0 : 2;$
$^\bullet t = 1 : 1;$
$1 : n;$
$esac;$

Rule 5: If n is a join node of transition t, and $a_1, a_2 \ldots a_k$ are k elements of $^\bullet t$.
$init(n) := 0;$
$next(n) := case$
$n = 1 : 2;$
$n = 0 \ \& \ (a_1 + a_2 + \ldots + a_k = 2 * k) : 1;$
$n = 2 \ \& \ (a_1 + a_2 + \ldots + a_k < 2 * k) : 0;$
$1 : n;$
$esac;$

Rule 6: If t is a transition which corresponds to the flow edges.
$init(t) := 0;$
$next(t) := case$
$!\ cond(t) \ \& \ ^\bullet t = 1 : 0;$
$cond(t) \ \& \ ^\bullet t = 1 : 1;$
$1 : t;$
$esac;$

Rule 7: If v is a variable whose new value is changed by exp_i in activity act_i $(1 \leq i \leq n)$.
$next(v) := case$
$act_1 = 1 : exp_1;$
$act_2 = 1 : exp_2;$
$\ldots\ldots$
$act_n = 1 : exp_n;$
$1 : v;$
$esac;$

Fig. 2.12 Translation rules for state and data transitions

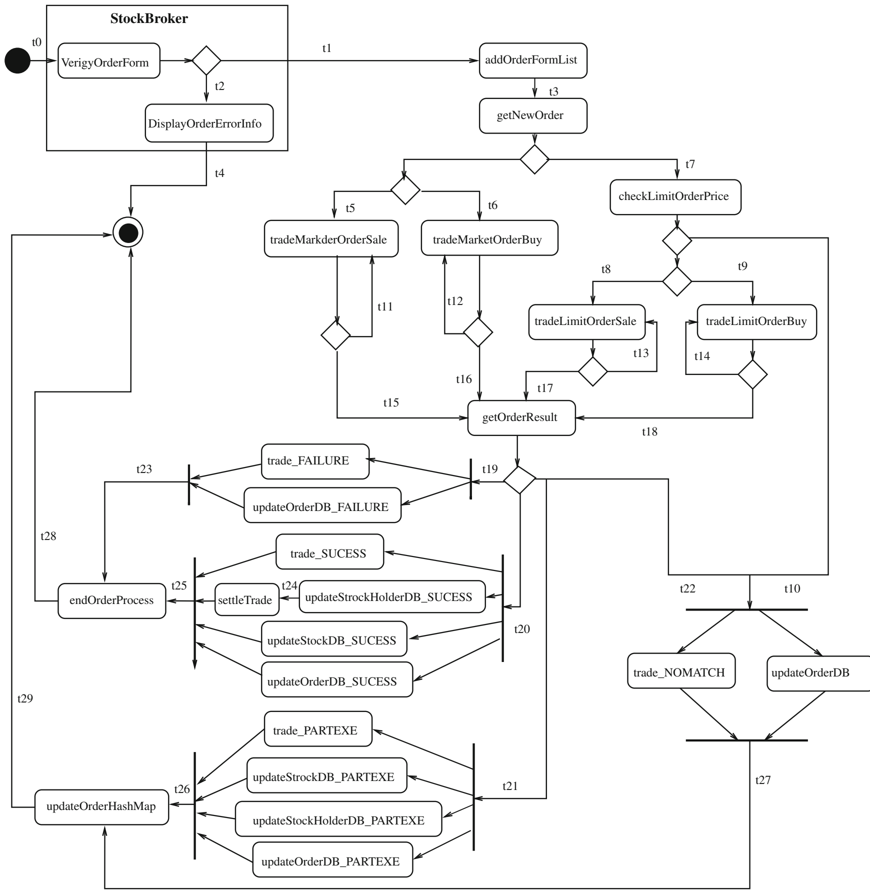


Fig. 2.13 The activity diagram for a stock exchange system

then ${}^{\bullet}t = 1$ means $a_1 = 1$ & $a_2 = 1$ &... & $a_k = 1$ and $cond(t)$ means $cond((a_1, t))$ &$cond((a_2, t))$ &... &$cond((a_k, t))$.

Rule 1 specifies the translation rule for the initial node. The token will be first put at the initial state and the node is marked as *visited* in the next step. Rule 2 specifies the translation rule for the final node. At first, the state is *unvisited*, when one of the incoming edges is activated, its state will become *visited*. Rule 3 defines the state changes of an activity. Initially, the state of an activity is *unvisited*. If the incoming edge is activated, the state will become *unvisited* in the next step. If the current state is *unvisited*, the state will change to be *visited* in the next step. Rule 4 presents the state transition of the fork nodes. When the incoming edge is activated, the fork node will maintain the *unvisited* status until all the outgoing edges are visiting or visited. Rule 5 provides the state transition of join nodes. The join node is used to synchronize the token flows. When all the incoming flows are ready, the transition corresponding

to the join node can be fired. In this rule, if we want to fire the transition, it needs to wait until all the activity nodes in the preset of the transition are visited. Rule 6 shows how to manipulate the state change of the transition when it is fired. Rule 7 presents the translation for value change of the variables. If an activity performs some operation on the variable, the value of the variable can be modified only when the activity state is *unvisited*.

2.4.4 Case Study: A Stock Exchange System

Based on the framework proposed in Sect. 2.4.3, a prototype tool is developed to automate the process of test generation. The tool takes three inputs: type definition of the data which is used in the activity diagram, the context information which sets the parameters for the execution of an activity diagram (e.g. when to trigger the initial node and so on), and UML activity diagrams. The UML activity diagrams are stored in the format of XML Metadata Interchange (XMI) files. The tool can parse the XMI files to get the static and dynamic information for formal model translation. Combined with the context information and data type information, a formal model can be generated using the proposed mapping rules.

The purpose of the Online Stock Exchange System (OSES) is to process three scenarios: accept, check, and execute the customers' orders (market orders and limit orders). The system uses the UML activity diagram as its behavior specification. Fig. 2.13 shows the specification of the stock system. It has 27 activities, 29 transitions, and 18 key paths. The generated SMV model has 756 lines of code.

2.5 Chapter Summary

This chapter presented the basic concepts of graph/FSM-based modeling of systems. It also introduced two system-level design specifications for SoC designs: SystemC TLM and UML activity diagrams. Based on the structural and behavior models extracted from these two specifications, this chapter presented mechanisms to convert them into formal SMV models to enable automatic analysis and directed test generation, which will be discussed in the following chapters.

References

1. Rose A, Swan S, Pierce J, Fernandez J (2005) Transaction level modeling in SystemC. Open SystemC Initiative. http://www.systemc.org
2. Object Management Group (2006) UML profile for system on a chip (SoC), v 1.0.1. http://www.omg.org/technology/documents/formal/profile_soc.htm

3. Riccobene E, Scandurra P, Rosti A, Bocchio S (2005) A UML 2.0 profile for systemc: toward high-level soc design. In: Proceedings of ACM international conference on embedded software, pp 38–141
4. Chureau A, Savaria Y, Aboulhamid EM (2005) The role of model-level transactors and UML in functional prototyping of systems-on-chip: a software-radio application. In: Proceedings of design automation and test in Europe (DATE), pp 698–703
5. Mueller W, Rosti A, Bocchio S, Riccobene E, Scandurra P, Dehaene W, Vanderperren Y (2006) UML for ESL design: basic principles, tools, and applications. In: Proceedings of international conference on computer-aided design (ICCAD), pp 73–80
6. Ammann P, Black P, Majurski W (1998) Using model checking to generate tests from specifications. In: Proceedings of international conference on formal engineering methods (ICFEM), pp 46–54
7. Mishra P, Dutt N (2008) Processor description languages. Morgan Kaufmann Publishers, San Francisco
8. Hennessy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann Publishers, San Francisco
9. Hopcroft JE, Motwani R, Ullman FD (2006) Introduction to automata theory, language, and computation 3rd edn. Addison-Wesley
10. Open SystemC Initiative (OSCI) (2006) Systemc. http://www.systemc.org
11. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of international conference on hardware/software codesign and system, synthesis (CODES+ISSS), pp 19–24
12. Ghenassia F (2005) Transaction level modeling with systemC. Springer, Dordrecht
13. Abdi S, Gajski D (2005) A formalism for functionality preserving system level transformations. In: Proceedings of Asia and South Pacific design automation conference (ASPDAC), pp 139–144
14. Kroening D, Sharygina N (2005) Formal verification of systemc by automatic hardware/software partitioning. In: Proceedings of international conference on formal methods and models for co-design (MEMOCODE), pp 101–110
15. Moy M, Maraninchi F, Maillet-Contoz L (2005) Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In: Proceedings of the international conference on application of concurrency to system design, pp 26–35
16. Karlsson D, Eles P, Peng Z (2006) Formal verification of systemc designs using a petri-net based representation. In: Proceedings of design, automation, and test in Europe (DATE), pp 1228–1233
17. Habibi A, Tahar S (2006) Design and verification of systemC transaction-level models. IEEE Trans Very Large Scale Integr Syst (TVLSI) 14(1):57–68
18. McMillan KL (OSCI) (2006) SMV model checker. http://www.kenmcmil.com/
19. Unhelkar B (2005) Verification and Validation for Quality of UML 2.0 Models. Wiley, New York
20. Chen M, Qiu X, Li X (2006) Automatic test case generation for uml activity diagrams. In: Proceedings of international workshop on automation on software test, pp 2–8
21. Chen M, Qiu X, Xu W, Wang L, Zhao J, Li X (2009) UML activity diagram based automatic test case generation for java programs. Comput J 52(5):545–556
22. Eshuis R (2006) Symbolic model checking of UML activity diagrams. ACM Trans on Softw Eng Methodol 15(1):1–38
23. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NUSMV: A new symbolic model verifier. In: Proceedings of international conference on computer aided verification (CAV), pp 495–499
24. Guelfi N, Mammar A (2005) NUSMV: A formal semantics of timed activity diagrams and its Promela translation. In: Proceedings of Asia-Pacific software engineering conference (APSEC), pp 283–290
25. Das D, Kumar R, Chakrabarti PP (2006) Timing verification of UML activity diagram based code block level models for real time multiprocessor system-on-chip applications. In: Proceedings of Asia-Pacific software engineering conference (APSEC), pp 199–208

26. Object Management Group (2007) UML superstructure V2.1.2. http://www.omg.org/docs/formal/07-11-02.pdf
27. Peterson J (1981) Petri nets theory and the modeling of systems. Prentice-Hall, Englewood Cliffs
28. Ericsson M (2004) Activity diagrams: what they are and how to use them. The Rational Edge. http://www.ibm.com/developerworks/rational/library/2802.html

Chapter 3
Automated Generation of Directed Tests

3.1 Introduction

Increasing complexity combined with decreasing time-to-market requirement make the functional validation a major bottleneck in the HW/SW design flow. As a most widely used validation method, simulation employs tests in the following three categories: random, constrained-random tests, and directed tests. Random and constrained-random testing [1] are easy to implement; nevertheless, it is hard to guarantee the convergence to the testing target (i.e., functional coverage). In contrast, directed testing [2] uses fewer tests to obtain the required functional coverage since it exploits the design structure information. By applying the directed tests, the validation effort can be drastically reduced. However, most directed test generation methods assume the expert knowledge of the design under validation (DUV). Due to the inevitable human intervention, current directed test generation methods are laborious and error-prone. Therefore, it is necessary to develop efficient techniques to automate the process of directed test generation.

Model checking [3] is a formal method which can verify whether a temporal property is satisfied for a finite state concurrent system. In model checking, a design is modeled as a state transition graph, called a Kripke structure [3], which is a four-tuple model $M = (S, S_0, R, L)$. S is a finite set of states. S_0 is a set of initial states, where $S_0 \subseteq S$. $R : S \rightarrow S$ is a transition relation between states, where for every state $s \in S$, there is a state $s' \in S$ such that the state transition $(s, s') \in R$. $L : S \rightarrow 2^{AP}$ is the labeling function to mark each state with a set of atomic propositions (AP) that hold in that state. Properties are expressed as linear temporal logic (LTL) and computational tree logic (CTL) formulas to describe expected design behaviors. For a formal model M of the design and a property p, the model checking is to find whether all states in S satisfy p or not. If there does not exist a reachable error state from initial states, the design satisfies the property, i.e., $M \models p$. Otherwise, the property does not hold for the design, and a variable assignment trace (counterexample) from an initial state to the error state will be reported. Such a trace can be used as a test to activate the scenario described by the negation of the checked property p.

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_3,
© Springer Science+Business Media New York 2013

Generally, date structures like binary decision diagrams (BDDs) [4] are efficient to represent and manipulate the transition relation of the finite state model. However, they are not scalable to handle complex systems. Due to the high complexity of realistic designs, the number of states of the design can be very large and the explicit traversal of the state space becomes infeasible, known as state space explosion. To alleviate this problem, model checking algorithm based on boolean satisfiability (SAT) procedures [5] have emerged as a promising approach, especially for the bounded model checking (BMC) [6, 7]. The basic idea behind SAT-based BMC is to encode the property checking to a SAT problem with a restricted search range. If the SAT problem is satisfiable, it means that the property is false and a counterexample of the property will be reported. SAT-based BMC can not only generate counterexamples much faster than traditional BDD-based model checking, but also the generated counterexample is of shorter length which can locate the design bug quickly.

The rest of this chapter is organized as follows. Section 3.2 introduces the related work on various SoC validation approaches including model checking based techniques. Section 3.3 presents the workflow of directed test generation approach using model checking. Section 3.4 describes the details of the coverage-driven property generation. Section 3.5 presents the test generation methods using both unbounded model checking and SAT-based BMC. Section 3.6 presents two case studies. Finally, Sect. 3.7 summarizes the chapter.

3.2 Related Work

Traditionally, validation of SoC designs has been performed by applying a combination of random and directed test programs using simulation techniques [8]. There are many successful test generation frameworks in industry today. Genesys-Pro [9], used for functional verification of IBM processors, combines architecture and testing knowledge for efficient test generation. In Piparazzi [10], a model of micro-architectural processor and the user's specification are converted into a constraint satisfaction problem (CSP) and the dedicated CSP solver is used to construct an actual test program. Many techniques have been proposed for directed test program generation based on an instruction tree traversal [11], micro-architectural coverage [12, 13], and functional coverage using Bayesian networks [2]. Recently, Gluska [14] described the need for coverage directed test generation in coverage-oriented verification of the Intel Merom microprocessor. However, none of these techniques can automatically generate directed tests based on comprehensive functional coverage metrics. In other words, these techniques either generate constrained-random tests automatically, or they generate directed tests for specific scenarios in semi-automated fashion.

The model checking based approaches presented in this chapter can automatically generate the required directed tests to achieve a given functional coverage goal [15]. However, traditional BDD-based unbounded model checking [3] cannot handle large designs because it suffers state space explosion problem. As a complementary

technique of model checking, Biere et al. [6] introduced BMC combined with satisfiability SAT solving [16]. The recent developments in SAT-based BMC techniques have been presented in [5]. BMC is an incomplete method that cannot guarantee a true or false determination when a counterexample does not exist within a given bound. However, once the exact bound of a counterexample is known, large designs can be falsified very fast since SAT solvers [17, 18] do not require exponential space, and searching counterexample in an arbitrary order consumes much less memory than breadth first search in model checking. Amla et al. [19] have analyzed the performance of bounded and unbounded algorithms using a set of industrial benchmarks. The capacity increase of the BMC technique has become attractive for industrial use. Intel study [20] showed that BMC has better capacity and productivity over unbounded model checking for real designs taken from the Pentium-4 processor. SAT-based BMC can be used as a test generation engine due to its capacity and performance if the bound is selected appropriately. A major challenge in these approaches is how to determine the exact bound. This chapter presents a method to determine the bound for each test generation scenario, thereby making SAT-based BMC more feasible for directed test generation in SoC designs.

3.3 The Workflow of Model Checking Based Test Generation

Figure 3.1 presents a methodology for specification-driven test generation using model checking techniques. First, a design is described using a specification language that can capture both structure and behavior of SoC systems. Next, the design specification is translated to a formal model (described in Chap. 2), and the properties in the form of LTL or CTL formulas are generated based on the functional fault models (see Sect. 3.4). Finally, the properties are applied on the formal model using a model checker to generate required tests (counterexamples). The model checker exhaustively searches all reachable states of the model to check if any state violates the property. If a violation is found, it will produce a counterexample. The counterexample contains a sequence of input assignments from an initial state to a state, where the specified property fails. If we assume that the design is correct and the property is a false safety property,[1] the model checker will always generate a valid counterexample. The generated counterexamples can be used for validating both specifications and implementations.

There are three major challenges in implementing this test generation flow in practice: (i) automatic extraction of formal models from SoC specifications, (ii) development of efficient functional fault models and associated coverage-driven property generation, and (iii) state space explosion problem. We have discussed the formal model generation in Chap. 2. Chapters 5–9 will present efficient approaches to minimize the effect of the state space explosion problem. The following sections focus on the automatic generation of properties and corresponding directed tests.

[1] A safety property asserts that a specified scenario can never happen. A false safety property is a safety property which can be proved to be false.

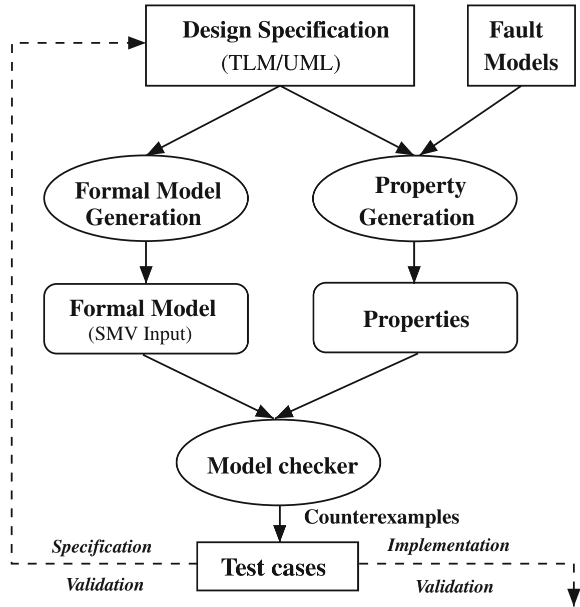


Fig. 3.1 Test generation using model checking

3.4 Coverage-Driven Property Generation

For model checking based test generation, a test is derived from the counterexample of a false safety property. A safety property in the temporal logic form $\neg F(p)$ asserts that a specified scenario cannot happen (i.e., property p cannot be true). If $\neg F(p)$ can be falsified, a counterexample which explains the reason of the error will be reported by a model checker. In other words, such a counterexample can be used as a test to activate the specified scenario. To achieve complete confidence of correctness of the design, the specification validation should include all the properties that the design should satisfy. Based on the structural information of the formal models, this section describes several fault models, which can be used to derive the properties automatically.

3.4.1 Safety Property and Its Negation

Focusing on automated directed test generation, this book only involves LTL safety properties using model checking based approaches. An LTL property is composed of three elements as follows:

- Atomic propositions: variables in the design.

```
Property 1: The activity dispense_cash is not reachable.
LTL formula: ~F (st.dispense_cash=2)

Property 2: The transition with condition [amount available] cannot
            be fired.
LTL formula: ~F(st.t7_cond = 1 )

Property 3: The key path 4 can not be covered.
LTL formula: ~F (st.start=2 & st.verify_access_code=2
    & st.handle_access_code=2 & st.ask_for_amount=2
    & st.prepare_print_receipt=2 & st.dispense_cash=2
    & st.generate_receipt_content=2
    & st.finish_transaction_print_receipt = 2
    & st.end = 2 & st.t2_cond=1 & st.t4_cond=1 & st.t7_cond=1 )

Property 4: The activities dispense_cash and prepare_to_print_receipt
            cannot be activated simultaneously.
LTL formula: ~F(st.dispense_cash=1 & st.prepare_to_print_receipt=1)
```

Fig. 3.2 Fault model examples

- Boolean connectives: $\wedge$, $\vee$, $\neg$,[2] and $\rightarrow$, etc.
- Temporal operators, assuming p is a state or path formula:

1. Fp (Eventually p): True if there exists a state on the path where p is true.
2. Gp (Always p): True if p is true at all states on the path.
3. Xp (Next p): True if p is true at the state immediately after the current state.
4. $p_1 U p_2$ (Until p): True if p_2 is true in a state and p_1 is true in all preceding states.

For example, the property $G(req \rightarrow F(ack))$ describes that if req is asserted then the design must eventually reach a state where ack is asserted.

Since a property describes a desired scenario, to achieve a test to activate this scenario using model checking methods, it is required to figure out the negated version of the property first. The De Morgan's laws as well as the following rules can be used for the property negation.

$$\begin{aligned} \neg X(p) &= X(\neg p) \\ \neg G(p) &= F(\neg p) \\ \neg F(p) &= G(\neg p) \\ \neg pUq &= pR\neg q \end{aligned} \tag{3.1}$$

As an example in pipeline interaction checking, pipeline interactions can be converted in the form of a property such as $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ that combines activities p_i over n modules using logical *"AND"* operator. The atomic proposition p_i is a

[2] In conventional LTL formulas, the sign "¬" denotes the negation. For the real property checking, both notations "∼" and "!" are used to indicate the negation of properties and expressions, respectively.

functional activity at a node i such as operation execution (i.e., active), stall, exception, or NOP. The property is true when all the p_i $(1 \leq i \leq n)$ holds at some time step. The negation of $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$, can be described as $\neg\mathrm{F}(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ (or $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$) whose counterexamples will satisfy the original property.

3.4.2 Testing Adequacy Using Model Checking

In model checking based test generation method, the quality of the generated tests is determined by the corresponding properties. During the property generation, it is required to guarantee that the generated properties can sufficiently validate the system.

The coverage metrics [21] are widely used in measuring testing adequacy. Test generation using model checking techniques requires that the automatically generated properties can cover as many desired scenarios in the design as possible. In practice, properties can be derived from a *fault model*, which represents a complete set of specific errors. Each *fault* in the fault model indicates a potential "design error" which can be described by a temporal logic property. The test generated from such a property can be applied on the design to check the specific scenario (negation of the fault). For example, when validating a desired scenario described by a LTL formula p, we use the negation $\neg p$ as a fault. By checking the property $\neg p$, we can derive a test to check the scenario where property p holds. Since this book focuses on safety property generation for above fault models, majority of the properties will be in the form of ¬F(p) or G(¬p). However, other forms of safety properties are also possible and allowed in test generation using model checking based methods.

3.4.3 Fault Models

Fault model [22] plays an important role in directed test generation. Each fault model represents a kind of "false functional scenarios". The efficiency of directed tests is directly related to the generated properties, which in turn are related to the associated fault model. The following three subsections present the fault models for the graph/FSM models as well as their two variants: fault models for SystemC TLM designs and fault models for UML activity diagrams. It is important to note that these fault models are by no means the "golden" model rather it is a representative model, which can be refined or modified for improved verification methodology.

Generally, for a complex design, a large number of properties needs to be generated. Depending on the design, the generated properties may lead to redundant tests. Therefore, proper property compaction, which will be described in Chap. 4, can be employed to reduce the number of properties without affecting the coverage goal.

3.4.3.1 Generic Fault Models for Graph/FSM Based Models

For a simple graph/FSM model, there is only node and edge information. By investigating the status of the nodes and edges, we can infer various system behaviors. There are four widely used fault models for graph models as follows.

- *Node/State fault*. Each node/state is faulty. For example, a node/state cannot be activated.
- *Edge/Transition fault*. Each edge/transition is faulty. For example, the respective nodes/states cannot be activated in that order.
- *Path fault*. Each execution path is faulty. For example, the associated nodes/states and edges/transitions are either faulty or their behavior cannot be composed correctly to activate the path.
- *Interaction fault*. Each interaction is faulty. For example, an interaction involving a set of nodes/states cannot be activated simultaneously.

The transformation from a fault model to corresponding properties in the form of temporal logic is a one-to-one mapping. This means exactly one property is generated for each fault in the fault model. Because a fault is already a negation of the system required behavior, it can be directly used to derive a property for test generation.

Let us consider Fig. 2.1 in Chap. 2 as an example of a graph model. The following example shows four properties (one for each fault type) for the graph model. Chapter 4 will discuss the FSM-based property generation in detail.

```
Prop. 1: The node FETCH cannot be activated.
LTL formula: ~ F (FETCH.active = 1)

Prop. 2: The edge between node MUL4 and MUL5
cannot be activated.
LTL formula: ~ F(MUL4.active = 1
   -> X(MUL5.active = 1))

Prop. 3: The path of FADD cannot be activated.
LTL formula: ~F (FETCH.active=1 & Decode.active=1
   & FADD1.active=1 & FADD2.active=1
   & FADD3.active=1 & FADD4.active=1
   & MEM.active=1 & WRITEBACK.active=1)

Prop. 4: DIV, FADD4 and MUL7 cannot be activated
simultaneously.
LTL formula: ~ F(DIV.active=1 & FADD4.active=1
   & MUL7.active=1)
```

3.4.3.2 Fault Models for SystemC TLM Specifications

In TLM, transaction data and transaction flow are two most important factors. They reflect both the structure and behavior information of system-level hardware designs. In addition to the fault models presented in Sect. 3.4.3.1, the following two fault models are identified to be useful in transaction validation.

- *Transaction data fault model* investigates the content of the variables relevant to the transaction. For each variable, it is assumed that a specific value can/cannot be assigned in some scenario.
- *Transaction flow fault model* investigates the controls along the path where the transaction flows. For each branch condition along the transaction path, it is assumed that it can/cannot be activated in some scenario.

Transaction data fault model deals with the possible value assignment for each part of the transaction data. However, during property generation, due to the large size of value space, trying all possible values of a data is time-consuming and impractical. By checking each bit of a variable (data bit fault) separately, the data content coverage can be partially guaranteed. Transaction flow fault model deals with the controls along with the transaction flow. To ensure transaction flow coverage, one can cover branch conditions which exist in *if–then-else* and *switch-case* statements. The goal is to check all possible transaction flows. Section 12.3.1.1 gives an example for each type of TLM transaction faults.

3.4.3.3 Fault Models for UML Activity Diagrams

In traditional software testing, testing adequacy [21] is defined as a measurement function. The case of UML activity diagrams is different because it is in the form of model instead of code. Especially, the coverage of activity diagram is more complex because of the concurrency. Similar to the generic fault models presented in Sect. 3.4.3.1, the following four fault models can be used for UML activity diagram validation.

- *Activity fault model*. For each activity node of AD, the model assumes that such activity node is not reachable.
- *Transition fault model*. For each transition of AD, the model assumes that such transition cannot be fired.
- *Key path fault model*. For each key path of AD, there is no corresponding executable $path$.
- *Interaction fault model*. For each interaction of AD, the activities associated with the interaction cannot be activated at the same time.

From these four different models, various properties can be generated to validate activity diagrams. The activity fault model can be used to check the reachability of each activity. So it can be used to check whether there exists infinite loops in the system. The transition fault model can be used to check the execution orderof the

activities. It can also be used to check whether the condition guard of the transition can be satisfied. To check all the dynamic behaviors of the system, the key path fault model is a preferable choice. The interaction fault model can be used to check whether several activities can be activated simultaneously. In general, if all the interactions involve only one activity, the interaction fault model is the same as the activity fault model.

The following example shows four properties (one for each fault type) for the UML activity diagram shown in Fig. 2.10.

3.4.4 Functional Coverage Based on Fault Models

The *functional coverage* of a system-level design is defined based on the overall faults of a fault model and the faults activated by the derived tests.

Definition 3.1 Let D be a design, F be a fault model, and T be a test suite. F indicates a complete set of faults, which are the negation of required functional behaviors of D. T is a set of directed tests which are derived from F. By applying T on D, the functional coverage D_F can be calculated as:

$$D_F = \frac{\#\ of\ exercised\ F - type\ functional\ scenarios}{|F|}$$ ■

3.5 Test Generation Using Model Checking Techniques

Model checking [3] is a formal method that can enumerate all the possible state to check whether a finite state system M satisfies a property p in the form of temporal logic (e.g. LTL or CTL [3]), i.e., $M \models p$. When the property fails at some state, it will report a counterexample to falsify the specified property p. Let us consider a test generation example for a pipelined processor. To activate a fault in the stall functionality of a decode unit (i.e., the decode unit can never be stalled), the system will generate the property "$\sim F(Decode.stall = 1)$". Taking the property and the processor model as inputs, the model checker will generate a counterexample to stall the decode unit, which can be used as a test to activate the stall functionality of the decode unit. The counterexample contains a sequence of instructions from an initial state to a state where the property fails. In this section, we briefly introduce two kinds of test generation methods based on different model checking techniques.

3.5.1 Test Generation Using Unbounded Model Checking

Symbolic model verifier (SMV [3, 23]) is a widely used model checker. By taking model of the design and temporal logic properties as inputs, SMV can determine

whether the design satisfies the property. SMV abstracts the given model into a formal Kripke structure, and does the state space search on this Kripke structure. The model checking algorithm stops when: (i) it encounters a false state for the property, then the counterexample which leads to this state will be generated, or (ii) all the states have been explored and no error is detected. Generally, the implementation of the state search adopts the data structure based on BDDs. However, they are not scalable to handle large systems in practice.

Algorithm 1 outlines the general test generation approach using Unbounded Model Checking (UBMC) [24–26]. The algorithm takes a SMV model M and a set of false safety properties P as inputs and generates a test suite extracted from counterexamples. For each property P_i, one test is generated. The algorithm iterates until all the properties are checked.

Algorithm 1: Test Generation using UBMC

Input: i) SMV Model, M
    ii) A set of false safety properties P (based on fault models)
Output: Testsuite
TestSuite = ϕ;
for *each property* P_i *in the set* P **do**
  $test_i$ = ModelChecking(P_i, M);
  TestSuite = TestSuite $\cup$ $test_i$;
end
return TestSuite

3.5.2 Test Generation Using Bounded Model Checking

For complex designs and properties, BDD-based methods usually encounter the state space explosion problem. As an alternative, SAT-based approaches have emerged, especially for the BMC. SAT-based BMC [6] is a promising method which can prove whether there is a counterexample for the property within a given bound. Given a model M, a safety property p, and a bound k, SAT-based BMC will unfold the model k times and encode it using the the Boolean formula Eq. (3.2).

$$BMC(M, p, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \tag{3.2}$$

Here, $I(s_0)$ means the initial state of the system, $T(s_i, s_{i+1})$ describes the state transition from state s_i to state s_{i+1}, and $p(s_i)$ tests whether property p holds on state s_i. This formula will be transformed to a conjunctive normal form (CNF) and checked by a SAT solver. If there is a satisfying assignment, the property is false and a satisfying variable assignment will be reported, i.e., $M \not\models_k p$. Otherwise, it implies that the property is true within the specified time steps. It means that there is

no counterexample with length k for this property, written $M \models_k p$. Test generation using BMC is similar to model checking based approach except that it needs to determine the bound for each property. SAT-based BMC takes model M, negated property p_i, and $bound_i$ as inputs and generates a counterexample (test).

3.5.2.1 Test Generation Algorithm

Algorithm 2 describes the widely used test generation procedure using BMC [27, 28]. This algorithm takes the model M generated from a design model and properties as inputs and generates test suite extracted from the counterexamples. For each property P_i, one test is generated. The algorithm iterates until all the properties are covered. In each iteration, the bound k_i of each property P_i is decided. SAT-based BMC takes model M, negated property P_i, and bound k_i as inputs and generates a counterexample (test).

Algorithm 2: Test Generation using BMC

Input: i) Design Model, M
ii) A set of false properties P (based on fault models)
Output: Testsuite
TestSuite = ϕ;
for *each property* P_i *in the set* P **do**
  $bound_i$ = DetermineBound(M, P_i);
  $test_i$ = BoundedModelChecking(P_i, M, $bound_i$);
  TestSuite = TestSuite $\cup$ $test_i$;
end
return TestSuite

During the test case generation, bound determination plays an important role. If it can be known a priori, SAT-based BMC can be more effective than BDD-based model checking techniques. However, any incorrect bound determination will increase test case generation time as well as memory requirement. Therefore, the techniques of deciding property bounds determine the efficiency of test case generation using SAT-based BMC.

3.5.2.2 Determination of Bound

Biere et al. [6] described several ways to determine the bound. If $M \not\models_k p$ for all k within the bound, then $M \not\models p$. However, there is no deterministic way to compute the bound of the property. In fact, determining the minimal bound for a property is as hard as the model checking itself [6].

According to the definition of the diameter in [6], the bound for each node error instance is decided by the temporal distance between the root node and the node under verification. For example, in UML activity diagrams, the bound for the key path error

is determined by the activities and transitions along the path. In Fig. 2.10, the length of the key path $\rho_4 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$ is 9. The property derived for this key path is shown in the Fig. 3.2. In the translation rules described in Sect. 2.4.3.2, an activity state transition needs one-step delay. Fork node needs one-step delay, and join node needs two steps delay. One-step delay at the *start* node is also required. The bound size will be $9 + 1 + 2 + 1 = 13$. The bound of the activity error or transition error is determined by the delay of activities and transitions on a valid shortest path from the $start$ node to the activity or transition which needs to be verified in the UML activity diagram. For example, when checking the activity error model instance "*prepare_to_print_receipt* can not be activated", the system will generate the property $\sim F(st.prepare_print_receipt = 2)$. The shortest path from start to such an activity is $\rho = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\}$. In a similar way, the bound for this property is $4 + 1 + 1 = 6$. Sometimes in the system, there is a counter that acts like a clock which counts the execution steps. Such variable in a property will affect the bound of the property. For example, because of the introduction of a counter, the property $\sim F(clk = 10 \,\&\, st.prepare_print_receipt = 2)$ has a bound of 10 instead of 6.

Different properties based on different fault models have different methods to compute the bounds. Assume that there is no counter variables, the determination of the bound of a graph-based model can use the following rules:

- *Node/state or edge/transition faults*. Extract all the paths without loops from the initial node/state to the target node/state or edge/transition. Calculate the bound for each extracted path and choose the shortest one as the property bound.
- *Path faults*. Calculate the bounds for the path based on the delay of nodes/states and edges/transitions on the path.
- *Interaction faults*. Calculate the bound for each element (node or edge) in the interaction. Choose the largest bound as the property bound.

If a property contains a counter variable. Then the bound of the property is the larger one of the counter value and the bound calculated using the above rules. Therefore, the complexity of bound determination is polynomial to the nodes in the graph-based models.

3.6 Case Studies

This section demonstrates two case studies for UML activity diagrams: a control system and an Online stock exchange system (OSES). Section 12.4 provides the details of automated directed test generation for TLM designs. This section compares model checking based approach with the random test-based method [29, 30], which is the well-known result in the category of test generation for UML activity diagrams.

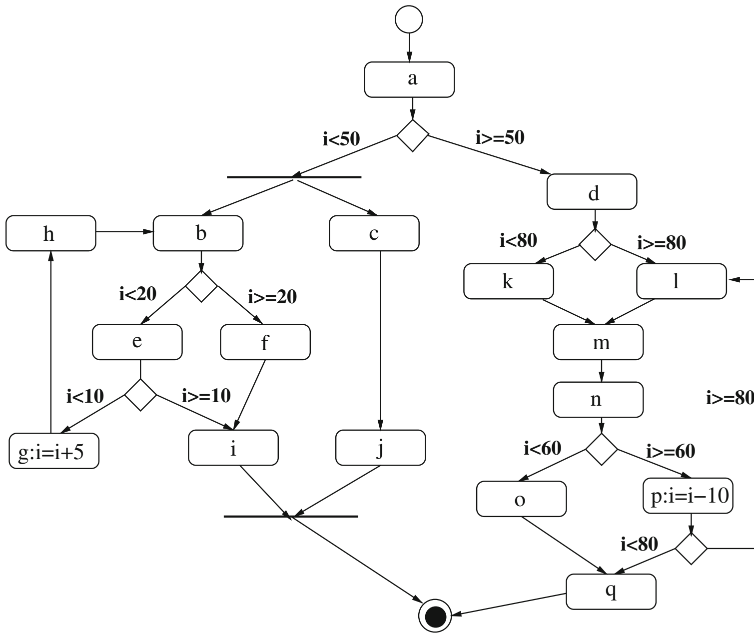


Fig. 3.3 The activity diagram for a control system

The experimental results indicate that the model checking based method can drastically reduce the overall validation effort by producing fewer tests. Furthermore, for UML activity diagrams, the generated high-level tests can be directly applied on the low-level implementations (e.g. Java code). Therefore, it can be used to check the consistency between UML activity diagrams and its low-level implementations. This case study adopted the Cadence SMV [23] as the model checker. All the experiments were conducted using 2.0 GHz Intel Core2 Duo CPU with 1 GB RAM.

3.6.1 A Control System

The first case study is a small control system using a UML activity diagram shown in Fig. 3.3. The UML activity diagram representation of the control system consists of 17 activities, 23 transitions, and 6 key paths. It has a global integer variable i which determines token flows. The generated SMV files have 365 lines of code.

Table 3.1 shows the comparison between UMC-based approach and the random test-based method [30]. For generating tests with highest coverage, the random method requires 8.83 s to run 150 random tests; however, UMC-based approach just needs 0.91 s. In this case study, UMC-based approach improves the test generation time by an order of magnitude.

Table 3.1 Comparison of two methods

Method	Coverage (%)			Time (s)
	Activity	Transition	Path	
Random 30	90	85	50	1.33
Random 50	95	93	67	2.35
Random 100	100	100	83	5.13
Random 150	100	100	100	8.83
UMC	100	100	100	0.91

Table 3.2 Implementation level coverage of the control system

Package (%)	Class (%)	Method (%)	Block (%)	Line (%)
100	100	90	88	93

The generated tests are also applied to the Java implementation of the control system. Table 3.2 shows the coverage of the Java code. The generated tests obtained 100 % *package* as well as *class* coverage. However, the *method*, *block*, and *line* coverage are around 90 %. The analysis showed that the Java implementation has many "try" and "catch" blocks to handle exceptions whereas the specification does not have any information on the exception scenarios. As a result, the generated tests did not activate any of the exception blocks which resulted in low coverage of methods, blocks as well as lines. Clearly, this is an issue of incomplete specification. Based on this observation, detailed exception information of the design is added into the specification. The derived corresponding tests indicate the required coverage in all the categories of the implementation.

3.6.2 A Stock Exchange System

The stock exchange system is based on the example presented in Sect. 2.4.4. It uses the UML activity diagram as its behavior specification. The system is implemented in JAVA and consists of 7 packages, 39 classes, 372 methods, and 2510 lines.

In Table 3.3, the first three rows depict the results by using 800, 1,000, 1,500 random tests, respectively. The results using model checking methods (i.e., UMC and BMC) are shown in the last two rows. In the case of *random* 800, two key paths are missing due to the randomness. So the coverage metrics are not 100 %. If the number of the random tests is increased to 1,000, one key path is still missing. Based on the observation, in the random method, it is hard to determine what is an appropriate upper bound for the number of required random tests. As a result, it is hard to obtain 100 % specification coverage using the random tests. The result of the UMC shows that we can get an order of magnitude improvement compared

Table 3.3 Comparison of three methods

Method	Coverage (%)			Time
	Activity	Transition	Path	(min)
Random 800	96	83	89	19.06
Random 1000	96	86	94	24.26
Random 1500	100	100	100	30.25
UMC	100	100	100	3.47
BMC	100	100	100	0.15

Table 3.4 Implementation level coverage of OSES

Package (%)	Class (%)	Method (%)	Block (%)	Line (%)
100	100	58	55	51

to the random method. Because the bounds of the properties of OSES system are shallow and can be predetermined, SAT-based BMC can be applied in this situation. The result shows that BMC method can be an order of magnitude faster than UMC method. Clearly, BMC approach reduces the validation effort by 200 times compared to the best known result [30] in this category.

Table 3.4 presents the coverage of the implementation by applying the generated tests. The coverage of method, block, and line are not sufficient because the activity diagram does not consider all the scenarios of the system, such as the registration of the customers and so on. In this case, it needs to add the missing details in the specification to obtain the required coverage.

3.7 Chapter Summary

It is widely acknowledged that automatic test case generation from system-level specifications can have double impact: (i) the generated test cases can be used to verify both the specification and the implementation, and (ii) it can drastically reduce the overall validation effort. However, due to lack of comprehensive fault models and associated test case generation techniques, it is not possible to automatically generate directed test cases to activate all the interesting scenarios and corner cases in SoC specifications. This chapter presented a framework to automatically generate directed tests from SoC specifications. The experimental results demonstrated that the generated tests can produce the required functional coverage and also can make a significant reduction in validation effort for specifications as well as implementations. Model checking based test generation is promising for automated test generation but it can lead to state space explosion in the presence of complex designs and properties. The following chapters will present various optimization techniques to reduce the overall test generation complexity.

References

1. Duran JW, Ntafos SC (1999) An evaluation of random testing. IEEE Trans Softw Eng 10(4):438–444
2. Fine S, Ziv A (2003) Coverage directed test generation for functional verification using Bayesian networks. In: Proceedings of design automation conference (DAC), pp 286–291
3. Clarke E, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge
4. Bryant R (1986) Graph-based algorithms for boolean function manipulation. IEEE Comput Soc 35(8):677–691
5. Prasad M, Biere A, Gupta A (2005) A survey of recent advances in SAT-based formal verification. Int J Softw Tools Technol Transf 7(2):156–173
6. Biere A, Cimatti A, Clarke E. M., Zhu Y (1999) Symbolic model checking without BDDs. In: Proceedings of tools and algorithms for the construction and analysis of systems (TACAS) pp 193–207
7. Biere A, Cimatti A, Clarke EM (2003) Bounded model checking. Adv Comput 58(3):117–148
8. Wagner I, Bertacco V, Austin T (2005) StressTest: An automatic approach to test generation via activity monitors. In: Proceedings of design automation conference (DAC), pp 783–788
9. Adir A, Almog E, Fournier L, Marcus E, Rimon M, Vinov M, Ziv A (2004) Genesys-pro: Innovations in test program generation for functional processor verification. IEEE Des Test 21(2):84–93
10. Adir A, Bin E, Peled O, Ziv A (2003) A test program generator for micro-architecture flow verification. In: Proceedings of high-level design validation and test workshop (HLDVT), pp 23–28
11. Aharon A, Goodman D, Levinger M, Lichtenstein Y, Malka Y, Metzger C, Molcho M, Shurek G (1995) Test program generation for functional verification of PowerPC processors in IBM. In: Proceedings of design automation conference (DAC), pp 279–285
12. Koo H, Mishra P, Bhadra J, Abadir M (2006) Directed micro-architectural test generation for an industrial processor: A case study. In: Proceedings of microprocessor test and verification (MTV), pp 33–36
13. Ur S, Yadin Y (1999) Micro architecture coverage directed generation of test programs. In: Proceedings of design automation conference (DAC), pp 175–180
14. Gluska A (2006) Practical methods in coverage-oriented verification of the Merom microprocessor. In: Proceedings of design automation conference (DAC), pp 332–337
15. Ammann PE, Black PE, Majurski W (1998) Using model checking to generate tests from specifications. In: Proceedings of international conference on formal engineering methods (ICFEM), pp 46–55
16. Marques-Silva J, Sakallah K (1999) Grasp: a search algorithm for propositional satisfiability. IEEE Trans Comput 48(5):506–521
17. Goldberg E, Novikov Y (2002) BerkMin: a fast and robust SAT-solver. In: Proceedings of design automation and test in Europe (DATE), pp 142–149
18. Moskewicz M, Madigan C, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: Proceedings of Design Automation Conference (DAC), pp 530–535
19. Amla N, Du X, Kuehlmann A, Kurshan R, McMillan K (2005) An analysis of SAT-based model checking techniques in an industrial environment. In: Proceedings of correct hardware design and verification methods (CHARME), pp 254–268
20. Copty F, Fix L, Fraer R, Giunchiglia E, Kamhi G, Tacchella A, Vardi M (2001) Benefits of bounded model checking at an industrial setting. In: Proceedings of computer aided verification (CAV), pp 436–453
21. Zhu H, Hall P, May J (1997) Software Unit Test Coverage and Adequacy. ACM Comput Surv 29(4):366–427
22. Ferrandi F, Fummi F, Gerli L, Sciuto D (1999) Symbolic functional vector generation for VHDL specifications. In: Proceedings of design, automation and test in Europe (DATE), pp 442–446

23. Cadence SMV. http://www-cad.eecs.berkeley.edu/kenmcmil/smv
24. Koo HM, Mishra P (2009) Functional test generation using design and property decomposition techniques. ACM Trans Embed Comput Syst (TECS) 8(4):32:1–32:33
25. Mishra P, Dutt N (2004) Graph-based functional test program generation for pipelined processors. In: Proceedings of design automation and test in Europe (DATE), pp 182–187
26. Mishra P, Dutt N (2005) Functional coverage driven test generation for validation of pipelined processors. In: Proceedings of design automation and test in Europe (DATE), pp 678–683
27. Koo HM, Mishra P (2006) Test generation using (SAT)-based bounded model checking for validation of pipelined processors. In: Proceedings of ACM great lakes symposium on VLSI (GSLVLSI), pp 362–365
28. Mishra P, Koo HM, Huang Z (2005) Language-driven validation of pipelined processors using satisfiability solvers. In: Proceedings of international workshop on microprocessor test and verification (MTV)), pp 119–126
29. Chen M, Qiu X, Li X (2006) Automatic test case generation for UML activity diagrams. In: Proceedings of international workshop on automation on software test, pp 2–8
30. Chen M, Qiu X, Xu W, Wang L, Zhao J, Li X (2009) UML activity diagram based automatic test case generation for java programs. Comput J 52(5):545–556

Chapter 4
Functional Test Compaction

4.1 Introduction

The cost of manufacturing test is greatly affected by the size of a test set, since it is generated once but applied to every single chip. Similarly, System-on-Chip (SoC) functional validation cost is proportional to the size of a test set; therefore, it is desirable to apply shorter test set that provides the same functional coverage. In manufacturing test domain, considerable research has been done to reduce the cost of manufacturing test by reducing the volume of test sets. However, it is interesting to note that in functional validation domain there has been little progress in test compaction because functional tests are considered as one-time effort in design methodology. As described in Chap. 1, functional validation is a primary bottleneck in SoC design cycle due to the combined effects of increasing design complexity and decreasing time-to-market.

To minimize the design validation cost, the effectiveness and the volume of a test set are key factors that need to be considered. Compared to random or constrained-random tests, coverage-directed test generation can typically reduce the volume of test suites with higher functional coverage. Comprehensive functional coverage metrics can be used for development of effective test suites. For further reduction of the volume of test sets, functional compaction techniques can be applied to remove redundancy in directed tests.

The rest of the chapter is organized as follows. Section 4.2 provides an overview of the existing test reduction techniques in manufacturing test domain. Section 4.3 presents functional property compaction techniques that can be applied before test generation. Finally, Sect. 4.4 concludes the chapter.

4.2 Manufacturing Test Reduction Techniques

During manufacturing test process, one or more sets of tests are applied to integrated circuits (ICs) in order to distinguish between the correct chips and the faulty chips caused by defects. Various test reduction techniques have been developed in

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_4,
© Springer Science+Business Media New York 2013

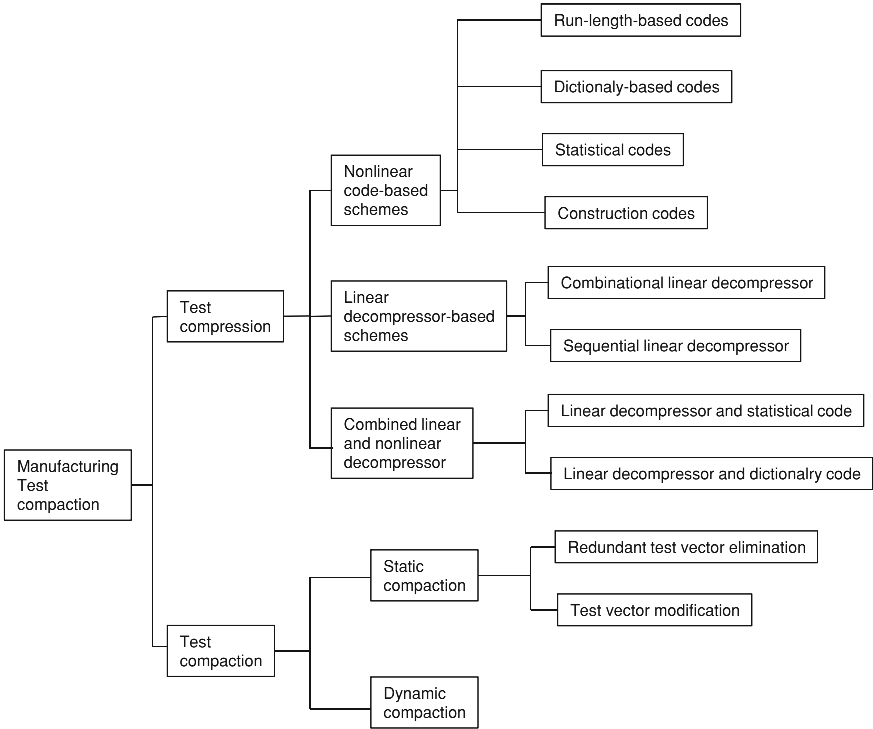


Fig. 4.1 Manufacturing test reduction techniques

manufacturing test domain because the same set of tests are applied on millions of chip copies and the volume of the test set makes a significant impact on overall testing cost and time. The tests are usually created by Automatic Test Pattern Generator (ATPG) and they are executed in Automatic Test Equipment (ATE). As a complementary technique, built-in self-test (BIST) is used by incorporating additional hardware logic into integrated circuits to allow them to perform self-testing, thereby reducing dependence on ATE.

Manufacturing test spends a significant portion of the overall SoC product cost due to the huge amount of test volume and test application time. Therefore, effective test reduction techniques are required to reduce the amount of test data. Over the past decades, researchers have proposed a variety of techniques and vendors have successfully developed commercial tools for test reduction. They can be divided into two categories: test compression and compaction. Figure 4.1 illustrates general categories of test reduction techniques.

4.2.1 Test Compression

Each test vector is designed for one or more modeled faults. Test vectors typically contain high percentage of "*don't cares*" in their bits which can take either 0 or 1 with no impact on the fault coverage. In addition, test vectors are closely correlated because faults are structurally related in the circuit. Due to this fact, high compression of test vectors can be achieved by removing redundancy among test vectors. The compression ratio can be improved by appropriately assigning the "*don't care*" bits to either 0 or 1.

In test compression, lossless compression techniques are used to preserve fault coverage. The compressed form of test data is stored in the tester and transferred to the Chip Under Test (CUT). The compressed test vector is decompressed on CUT. Although test vector compression requires additional on-chip hardware logic for test decompression, it can drastically reduce the test volume and the test time. As shown in Fig. 4.1, test data compression techniques [29] can be broadly categorized into nonlinear code-based schemes and linear decompressor-based schemes.

Code-based schemes encode a test vector into a code word to form the compressed data and an on-chip decoder converts the code word back into the corresponding test vector. Many coding forms have been proposed such as run-length-based codes [3, 4, 11, 15], dictionary codes [19, 26, 36], statistical codes [16], and constructive codes [25, 34]. Linear decompressor-based schemes decompress the input variables using a linear decompressor that consists of only wires, XOR gates, and flip-flops [1, 18, 24, 33, 35]. Combinational linear decompressors use simple hardware and control logic where an XOR of the tester channels drives each scan chain. Sequential linear decompressors use linear finite state machine such as linear feedback shift registers or ring generator.

According to comparison analysis of two schemes, linear decompression-based schemes provide greater compression for test sets with very large percentage of "*don't care*" bits than code-based schemes. Since industrial test vectors typically have more than 95 % "*don't care*" bits, linear decompression-based schemes are adopted in almost all commercial tools for test data compression. In order to achieve further reduction in the volume of test sets, it is necessary to look beyond the on-chip compression techniques. Some techniques for reducing the number of test vectors, called test compaction, will be described in the following section.

4.2.2 Test Compaction

A primary goal of test compaction is to minimize the size of a test set while maintaining the same fault coverage, which is the same as the goal of test compression. However, test compaction is more focused on reduction of overall volume of test sets by removing and modifying redundant tests while test compression is more focused on reduction of each test by encoding each test into a smaller size code. Test

compaction techniques are categorized broadly into static compaction and dynamic compaction as shown in Fig. 4.1. If compaction algorithms are applied after test generation, those schemes belong to static compaction. If compaction algorithms are integrated into the test generation process and they are applied during test generation, those schemes fall into the category of dynamic compaction that performs test compaction concurrently with the test generation process.

The classic procedure in dynamic compaction is to generate a test for one fault, and then modify the test in order to increase the number of faults detected by the test. Fault simulation is used to check what other faults are detected through the modified test, and those detected faults are dropped in the fault list. In [27], genetic algorithms are used to fill the "*don't care*" bits in a partially specified test and better tests are evolved to detect more faults during test generation. Pomeranz and Reddy [23] have proposed a test vector improvement technique for both random and deterministic test generation. It is based on the observation that even if a test is generated using dynamic test compaction heuristics, it is possible to improve the test further so as to increase the number of yet-undetected faults. To address the recent need for multiple fault models to better defect coverage, Venkatesh et al. [31] have developed dynamic test compaction across multiple fault models instead of using separate ATPG and test compaction for each fault model.

Dynamic compaction can typically achieve greater reduction of test set size than static compaction. But it requires high processing complexity for iterative modification of partially specified test vectors to detect more yet-detected faults during test generation. In contrast, static compaction is independent of any test generator. Static compaction schemes can be categorized into redundant test vector elimination and test vector modification.

A redundant test vector can be removed from a test set, because its corresponding faults are all detectable by other test vectors. Redundant test elimination can be modeled as a set covering problem to cover all target faults using the minimum number of test vectors. Dimopoulos and Linardis [7] have modeled static compaction for sequential circuits as a set-covering problem. Set covering has also been applied to combinational circuits using the fault detection matrix [10, 14]. In addition to the compacted size, early termination of test compaction is also important due to the high computational complexity of test compaction algorithms. Based on the fact that identifying redundant tests is dependent on the test ordering during fault simulation, the test order for fault simulation can be different from the test order of generation. For example, in reverse order fault simulation [22], a test vector that was generated later is a fault simulated earlier because the later test typically targets to the harder-to-detect fault and it also detects many other faults.

Test vectors can be modified to detect more faults. For example, two test vectors can be merged if they do not have any conflict values in any bit position. Bit conflict occurs when two tests have different values, i.e., 0 and 1 or 1 and 0 at the same bit position. Since each test vector typically has many "*don't care*" bits in it, there is a high chance to avoid the bit conflict. By applying this merging operation repeatedly, two or more test vectors can be combined into one test vector [9, 21]. As a complementary approach, El-Maleh and Osais [8] have presented decomposition-based

Fig. 4.2 Test matrix for FSM coverage

$$
\begin{array}{c c}
 & \begin{array}{ccccc} s_1 & s_2 & s_3 & \dots & s_n \end{array} \\
\mathbf{TM} = & \left(\begin{array}{ccccc} 1 & 0 & 0 & \dots & 1 \\ 0 & 1 & 1 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 1 & \dots & \dots & 1 \end{array}\right) \begin{array}{c} t_1 \\ t_2 \\ t_3 \\ \dots \\ t_n \end{array}
\end{array}
$$

static compaction algorithms where a test vector is decomposed into atomic components and the test vector is eliminated if its components can be all moved to other test vectors.

4.2.3 Applicability and Limitations

Due to the difference in test data structure and test methodology, most of manufacturing test reduction techniques cannot be directly applied to functional test compaction. This section investigates applicability of manufacturing test compaction techniques to functional test compaction. In manufacturing test compaction, test minimization using set covering can be applied after test generation. In general, due to the computational complexity, set covering-based approaches are suitable only for small size of test sets.

For illustration, we use the functional FSM model of pipelined processors and FSM state coverage described in Sect. 2.2.2. Let us consider a set of tests $T = \{t_1, t_2, \dots, t_n\}$ detecting the set of functional states $S = \{s_1, s_2, \dots, s_n\}$, where n is the number of states and directed tests are obtained through test generation process. The test minimization problem is a problem of selecting the minimal number of tests, i.e., a minimum subset of T such that all FSM states in S are covered. To represent a given test set, an $n \times n$ matrix can be used. Each row of the matrix corresponds to a test and each column corresponds to an FSM state. The element of the matrix with coordinates i, j holds the value 1, if the test t_i can activate the state s_j, otherwise it holds the value 0. This matrix is denoted as *Test Matrix*. Figure 4.2 shows an example of the *Test Matrix*. Diagonal elements in the matrix are all set to 1 in the case of directed test generation.

The test minimization problem can be formulated as a set covering problem [6]. Since many studies have been proposed for solving set covering problems, this formulation can take advantage of existing algorithms. However, finding the minimum test set suffers from exponential blow-up because the set covering problems are NP-complete. Therefore, there is a need to reduce the size of matrix before applying any algorithm to solve set covering problems.

The test matrix reduction techniques [32] can be used to reduce the complexity of set covering by eliminating redundant faults (rows) and test vectors (columns) in the fault detection matrix. The *Test Matrix* shrinks after iteratively applying the following rules: *test essentiality*, *test dominance* for row elimination, and *FSM state*

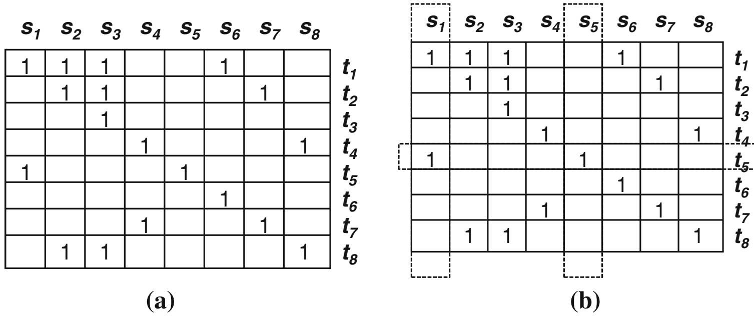


Fig. 4.3 Test matrix (**a**) and essential test t_5 (**b**)

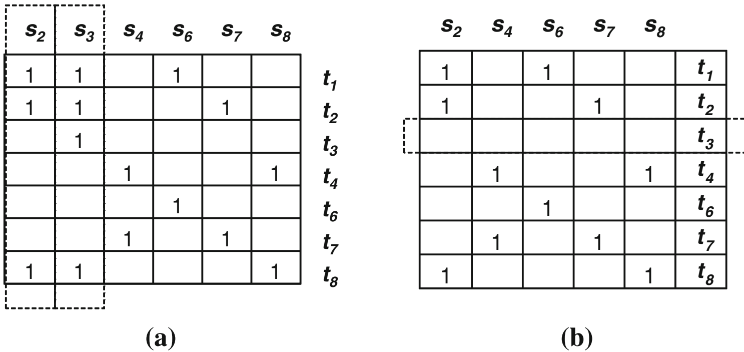


Fig. 4.4 State dominance. **a** s_2 is dominant to s_3, therefore s_3 is removed; **b** t_3 covers no fault, so is removed

dominance for column elimination. If ith column is covered by only one test, the test is an essential test that cannot be removed from the test set, that is, test essentiality. The columns that are covered by the essential tests can be removed from the matrix. If all states of t_i are covered by t_j, t_j dominates t_i and t_i (ith row) can be eliminated, that is, test dominance. If all tests of s_i detect s_j, s_j dominates s_i, and s_j (jth column) is removed, that is, FSM state dominance. After matrix reduction, the set covering can be used to obtain the minimum test set.

The matrix reduction techniques are shown in Figs. 4.3, 4.4, and 4.5. Figure 4.3a shows a test matrix of FSM states and tests. Figure 4.3b shows that t_5 is an essential test. Therefore, states s_1 and s_5 are removed and t_5 should be put in the test list. Figure 4.4 describes an example of FSM state dominance. The state s_3 can be removed because s_2 is dominant to s_3. The test t_3 can be removed because it does not cover any state any more. Figure 4.5 shows test dominance. The test t_6 can be removed because t_1 is dominant to t_6. After removing t_6, t_1 is identified as an essential test. As a result, states s_2 and s_6 are removed and t_1 is put in the test list.

By applying the redundancy reduction rules, a test matrix can be considerably reduced into smaller one. After matrix reduction, a set covering optimization

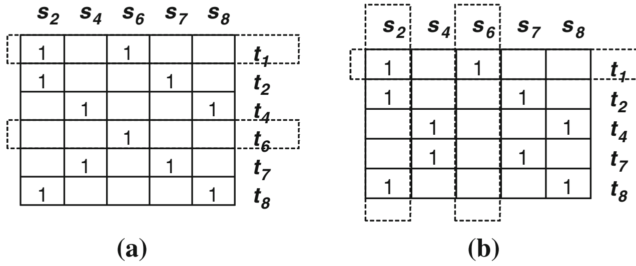


Fig. 4.5 Test dominance. **a** t_1 is dominant to t_6, therefore, t_6 can be removed; **b** t_1 is essential

technique can be applied to minimize the test set. For a manageable volume of test sets after test generation, the combined scheme of set covering and matrix reduction can be applied to functional test compaction. Investigation of relationship between functional tests and manufacturing tests is expected to enable the common aspects of the two activities to be reused for improved functional test compaction.

4.3 Functional Test Compaction

A lot of structural test compaction techniques have been proposed in manufacturing test domain because they have a significant impact on overall testing cost and time. In contrast, there has been little progress in functional test compaction in the validation domain. Considering that millions of regression tests are conducted almost everyday during design cycle in the current industrial practice, functional test compaction will have significant impact on overall design effort.

Compared to random or pseudo-random tests, directed test generation can significantly reduce overall validation effort since the shorter tests can achieve the same coverage goal. However, the number of directed tests can still be extremely large. Although each test is generated for activating a particular functional fault, it may go through several pipeline stages and paths of hardware designs over multiple clock cycles to reach the target fault. Therefore, there is a high probability that the test can accompany multiple pipeline interactions before and after it reaches the target functionality. In other words, the directed functional tests have correlation with each other due to the hardware pipeline characteristics. Based on this fact, we can achieve functional test compaction by removing redundant tests. Both in functional validation and manufacturing test domains, pruning of redundant tests from a test set is performed during or after test generation. In contrast to these test compaction approaches, a property compaction technique needs to be applied before test generation. Therefore, a test generator only needs to generate a reduced set of tests without sacrificing the coverage requirement [17]. This approach can reduce the generation cost as well as the overall validation effort.

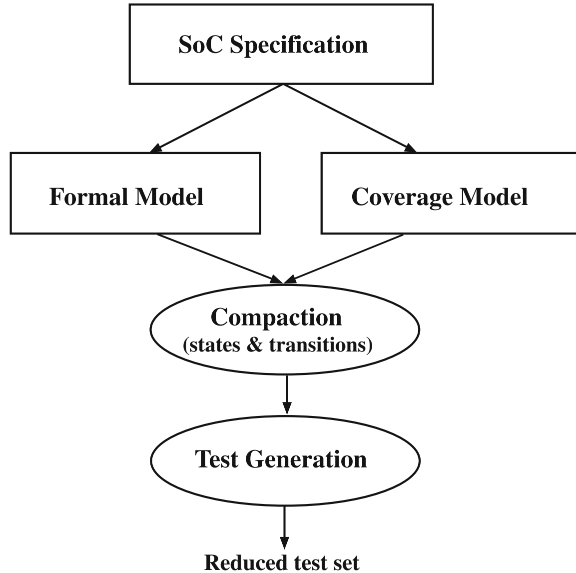


Fig. 4.6 Functional test compaction methodology

Figure 4.6 shows the overall flow of functional compaction and associated test generation methodology. To illustrate the test compaction technique, this chapter uses the FSM model of pipelined processors described in Sect. 2.2.2. From specification of a processor, we create an FSM model and define FSM state and transition coverage metrics based on pipeline interactions. The FSM states and transitions are represented in the binary format as described in Sect. 4.3.1. Each binary code corresponds to a pipeline interaction property for test generation. FSM property compaction is performed before test generation by eliminating the states and the transitions that are illegal, redundant, or unreachable based on design constraints as described in Sect. 4.3.3. One of the uncovered states or transitions is chosen as a target for directed test generation. A path in FSM is traversed from a given point by tracing backward to an initial state and tracing forward to a final state from that point. During the traversal, an uncovered transition is selected to minimize test redundancy as well as test volume as described in Sect. 4.3.4. The number of uncovered states and transitions are reduced by eliminating the states and transitions on the path. For illustration, model checking is used to generate a test to exercise the path. Path selection and test generation continues until all states and transitions are covered.

4.3.1 Binary Format of FSM Models

Functional states *S* of a pipelined processor FSM model described in Sect. 2.2.2 can be expressed in the form of binary data that contains both the pipelined structure and the behavior of the processor. The FSM model is based on interactions among functional units of the pipelined processor. A group of bits is assigned to describe the

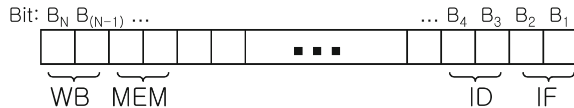


Fig. 4.7 Binary format of the states in FSM model of a pipelined processor

functional status of each functional unit. A functional state of the entire processor consists of bit concatenation of local states of all functional units. We denote the number of activities in the functional unit fu_j by r_j and the number of bits to be assigned to the unit by b_j where $j = 1, \ldots, U$ and U is the number of functional units in the processor. Therefore, the total number of bits to describe the processor FSM states is $N = \sum_{j=1}^{U} b_j$. We denote the number of states in the machine M by $NS = |S| = 2^N$. The state of functional unit fu_j is denoted as ss_j using b_j bits and the state s_k of the processor FSM can be defined by concatenating $ss_1, ss_2, \ldots, ss_U$.

For example, we assign two bits to represent four functional states of Fetch (IF) unit: '00' for idle, '01' for normal operation (instruction fetch), '10' for stall, and '11' for exception. Figure 4.7 shows an example of the FSM states of the pipelined processor. Assume that all the functional units have only four possible states, each unit requires 2 bits for its four functionalities. This binary format of functional FSM model provides an efficient indexing mechanism to access and analyze each functional state. In addition, next states can be described as Boolean functions. For example, assuming the state transitions (s_i, s_j) and (s_i, s_k) with s_j = '0011' and s_k = '0010', the next states of s_i are expressed as $\bar{B}_4\bar{B}_3B_2B_1 + \bar{B}_4\bar{B}_3B_2\bar{B}_1 = \bar{B}_4\bar{B}_3B_2$. For each state, a corresponding index number has a list of the next and previous states that are produced using transition functions. The list of neighboring states are used for selecting a uncovered path during test generation.

The state transition functions are based on pipeline behavior of processors. The pipeline behavior are the rules in each pipeline stage that determine when instructions can move to the next stage and when they cannot. For pipeline behavior modeling, we decompose the entire processor FSM into smaller FSMs at functional unit level. Since not all the functional units affect the next states of other functional units, the transition functions of the FSM can be decomposed into subfunctions each of which is dedicated to a specific functional unit.

Figure 4.8 shows the general behavior of pipelined processors. Each instruction goes through the current pipeline stage to the next stage as shown in Fig. 4.8a, where *fu* is a functional unit, $1 \leq i, k, l \leq U$, $1 \leq j \leq D$, and D is the pipeline depth. Each functional unit $fu_{i,j}$ can interact with different number of functional units at stage $j-1$ and $j+1$. For example, a decode unit may have multiple execution units at its following stage while a fetch unit typically has only one unit (decode unit) at the following stage.

Figure 4.8b shows the pipeline interaction of the functional unit $fu_{i,j}$. The state of $fu_{i,j}$ at time step t is decided by the previous and current states of units $fu_{k,j-1}$ and $fu_{l,j+1}$ as well as itself. For example, if $fu_{l,j+1}$ and $fu_{i,j}$ are on the same pipeline and $fu_{l,j+1}$ is in the stall state at time step t, then $fu_{i,j}$ should be in stall state because the instruction in $fu_{i,j}$ cannot go to the next stage $fu_{l,j+1}$. Considering

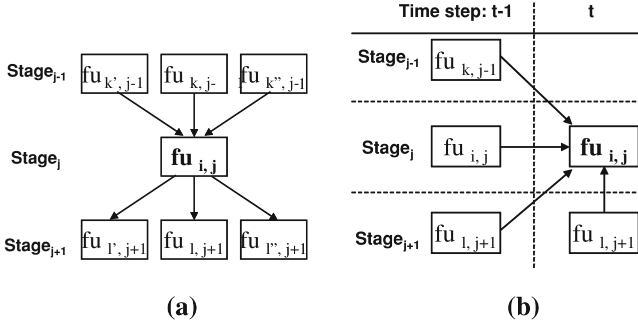


Fig. 4.8 Pipeline behavior: **a** instruction flow, **b** pipeline interaction

feedback loop such as data forwarding in the pipelined processor, $fu_{i,j}$ at time t will be affected by the state of $fu_{i,j+\alpha}$ at $(t-1)$ where $0 \leq \alpha \leq D$.

Based on the pipeline behavior, the state transition to the functional unit $fu_{i,j}$ at time step t is defined as $ss_{i,j}(t) = f(ss_{k,j-1}(t-1), ss_{i,j}(t-1), ss_{l,j+1}(t-1), ss_{l,j+1}(t))$. Here, $ss_{i,j}(t)$ represents a set of bits to describe the functional state of $fu_{i,j}$ at time t, and f represents a transition function decided by unit interactions. Therefore, the state s of the processor FSM can be expressed by concatenating $ss_{i,j}$ where $i = 1, \ldots, U$ and $1 \leq j \leq D$.

4.3.2 Number of FSM States and Transitions

State coverage ensures that every state of an FSM has been visited. Transition coverage ensures that every transition between FSM states has been traversed. When state coverage and transition coverage are used as coverage metrics to generate a test set, it will be interesting to estimate how many tests are required in naive directed test generation methods.

The state coverage of the FSM model is identical to the pipeline interaction coverage that tries to detect whether a set of pipeline interactions (between functional units) have been activated at a given clock cycle. Therefore, a test that covers the FSM state will activate the corresponding pipeline interaction. Each FSM state consists of multiple substates of each functional unit in the processor. We can compute the number of theoretically possible FSM states based on the number of functional units and the number of activities at each unit. In general, the number of activities varies for different units depending on what activities need to be tested, thereby each unit may require different number of bits for its functional states. Considering an FSM model with m units where each unit has on average r activities, the FSM will have r^m states which can be extremely large even for small number of activities. For example, the MIPS processor with 17 functional units and 4 activities described in Sect. 2.2.1 has approximately seventeen billion states.

From the point of functionality, the state transition coverage represents temporal pipeline interactions. Based on the state transition functions, each state has a list of their next states. When a test visits the state and goes to one of its next state, we put the next state off the list since the transition between the two states is covered. State transition coverage of the FSM is achieved when the next state lists are empty for every state. The number of state transitions is determined by the processor's functional behavior. Theoretically, the maximum number of state transitions is N^2, where N is the number of states, assuming any state can be reached from another state in one step. This theoretically large number of functional states and transitions can be reduced by eliminating unreachable states using functional constraints described in the processor specification.

4.3.3 Property Compaction of FSM States and Transitions

The state and transition compaction of an FSM plays a major role in efficient test generation since reduction of one state or transition implies one less test vector to generate and apply on RTL implementation. The basic idea is to identify and eliminate all the unreachable and redundant states as well as transitions with respect to coverage-driven test generation.

4.3.3.1 Identifying Unreachable States

Constraints described in the processor specification can be used to distinguish unreachable states from reachable ones. According to the specification, for example, the functional constraints may include the number of issuable instructions at issue stage, stall conditions of each functional units, exception cases, etc. These constraints are represented as binary patterns of FSM states and we can remove the states with these patterns, because they are unreachable. In other words, illegal behaviors are expressed as unreachable binary patterns and those patterns are not considered during test generation and coverage analysis. For example, assume that decode (*ID*) unit has single instruction issue constraint and there are two parallel execution units *EX1* and *EX2* in the next pipeline stage. Since only one instruction can be passed to either *EX1* or *EX2*, both execution units cannot be in normal operation (executes a valid instruction) at the same time. Assuming that *EX1* and *EX2* correspond to the state variables B_6B_5 and B_4B_3, respectively, in 10-bit FSM processor state model, the binary pattern 'xxxx0101xx' represents all of unreachable states for the single issue constraint, where '01' represents the unit state of normal operation and 'x' represents '0' or '1', that is, "*don't care*" bit.

For n-bit FSM model, the number of total states is $N_{\mathrm{T}} = 2^n$. The number of reachable states N_{rs} is computed by removing the unreachable states: $N_{\mathrm{rs}} = N_{\mathrm{T}} - N_{\mathrm{us}}$, where N_{us} represents the number of unreachable states. If an unreachable binary pattern has m bits of "*don't care*" bits, then $N_{\mathrm{us}} = 2^m$ and $N_{\mathrm{rs}} = 2^n - 2^m$. For the

Table 4.1 Transition rules between $ss_{k,j-1}$ and $ss_{i,j}$

$ss_{k,j-1}(t-1)$	$ss_{i,j}(t)$
Idle	Idle, stall
Normal operation	Normal operation, stall, exception
Stall	Idle, stall
Exception	Idle, stall

Table 4.2 Transition rules between $ss_{i,j}$ and $ss_{i,j}$

$ss_{i,j}(t-1)$	$ss_{i,j}(t)$
Idle	Idle, normal operation, stall, exception
Normal operation	Idle, normal operation, exception
Stall	Idle, normal operation, stall, exception
Exception	Idle

Table 4.3 Transition rules between $ss_{l,j+1}$ and $ss_{i,j}$

$ss_{l,j+1}(t-1)$	$ss_{i,j}(t)$
Idle	Idle, normal operation, stall, exception
Normal operation	Idle, normal operation, stall, exception
Stall	Idle, normal operation, stall, exception
Exception	Idle

above example, $N_{\text{rs}} = 2^{10} - 2^{6} = 960$. By applying all the functional constraints described in the processor specification, we can identify and remove the unreachable states in the FSM and compute the number of reachable states.

4.3.3.2 Identifying Illegal State Transitions

Extracting FSM state transitions at the processor level is very difficult since specification documents do not include relation of processor-level states in general. However, the processor specification provides the rules in each pipeline unit about when instructions can move to the next pipeline stage and when they cannot. These pipeline behaviors are used to identify illegal state transitions. We can also leverage the decomposition of a processor state transition into functional unit level transitions. For example, if the state of a functional unit $ss_{i,j}$ is in normal operation at time t, then the state of the unit in the previous stage ($ss_{k,j-1}$) cannot be in idle state at time $t-1$ since the instruction in $fu_{i,j}$ must be ready at the previous pipeline stage at time $t-1$.

Substate transition rules between units are shown in Tables 4.1, 4.2, and 4.3 assuming four functional activities at each unit and one register between consecutive pipeline stages. For example, in Table 4.1, if $ss_{k,j-1}(t-1) = \text{stall}$, then $ss_{i,j}(t)$ can

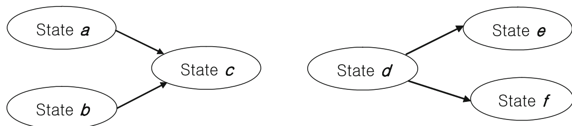


Fig. 4.9 Single transitions between states

be either in idle or stall state because no instruction moves from the previous stage. In Tables 4.2 and 4.3, if $ss_{k,j-1}(t-1)$ or $ss_{l,j+1}(t-1) = \text{exception}$, then $ss_{i,j}(t)$ should be in idle state to flush the following instructions in the pipeline.

4.3.3.3 Identifying Inevitable States and Transitions

In terms of coverage-driven test generation, if multiple tests go through the same FSM state, then those tests are redundant to each other. Especially, if a test generated for activating any other states or transitions *must* go through the state (transition) under consideration, the state (transition) is called an *inevitable state* (*transition*).

Figure 4.9 shows inevitable states and transitions that have single outgoing transition (from states a and b) and single incoming transition (to states e and f). The state c is an inevitable state because all the paths from a and b should include the state c. Similarly, the state d is an inevitable state because all the paths to e and f should include the state d. The transitions $(a \rightarrow c)$, $(b \rightarrow c)$, $(d \rightarrow e)$, and $(d \rightarrow f)$ are inevitable transitions to their neighbors. We can eliminate the test cases that activate these inevitable states and transitions since any test program that exercises their neighboring states will also activate the inevitable states. The next state lists of each state are used to identify the inevitable states of the single outgoing transitions. If a state has only one state in its next state list, the next state is an inevitable state. In the same way, the previous state lists are used to identify the single incoming transitions. Before test generation, those inevitable states (transitions) can be identified and removed from a test list. Other redundant states (transitions) are reduced during test selection as described in the following section.

4.3.4 FSM Coverage-Driven Test Selection and Generation

In FSM coverage-driven test generation, tests are created to activate a target coverage point (state or transition). For validation of pipelined processors, many FSM model-based test generation techniques have been developed where an FSM model is used to generate a test suite for state, transition, or path coverage [2, 37]. A significant bottleneck in these methods is the high complexity of FSM models, resulting in state explosion problem. To alleviate FSM complexity, abstraction techniques have been proposed [5, 13, 20, 28, 30]. The abstract FSM models provide feasible ways of concrete test generation. Abstract tests are generated from the abstract FSM and then they are converted into tests. Compared to the existing approaches, the property

compaction described in the previous section is applied before test generation and the binary format of FSM model makes it easy to create a fault list and analyze FSM coverage.

Once all the unreachable and inevitable tests are removed, one of the uncovered states or transitions is chosen as a target for directed test generation. Each state (binary index number) has a flag to indicate whether the state is covered or not. The flag is called *StateCovered* flag and is initialized to 0. Each state also has a list of its neighboring states, i.e., a list of its transitions. In the list, each neighboring state has two flags to indicate whether the state is a next state or a previous state and whether the transition to/from the neighboring state is covered or not. The second flag is called *TransitionCovered* flag and is initialized to 0.

Beginning from a target state, we search for an FSM path that can cover maximum number of states and transitions. For backward path, one of the previous neighboring states is selected that has *TransitionCovered* $= 0$. A pair of state and transition is covered by setting value 1 for *StateCovered* of the current state and *Transition Covered* of the previous state. This process continues until the path traversal reaches an initial state. If all of the previous states in the neighbor list are covered (*Transition-Covered* $= 1$) at the current state, the neighbor list of the previous states needs to be checked to determine whether the path includes any uncovered state and transition. In this way, the number of redundant tests can be reduced by avoiding redundant states and transitions. The search space is extended until an uncovered state/transition is encountered, or until the number of backward transitions reaches its upper bound (maximum number of clock cycles in which an instruction can stay in the processor pipeline). Similarly, the path is completed by tracing forward to a final state from the target point. The path generation continues until all states and transitions in the FSM are covered, i.e., all of the *StateCovered* and *TransitionCovered* flags are set to 1.

Each selected FSM path represents a desired functional behavior and is expressed in the form of temporal logic property for directed test generation using model checking. Negated version of a desired property is applied to model checking to produce a counterexample and corresponding input requirements. The input requirement of the processor model contains a sequence of instructions that can be used for validation of the selected FSM path.

4.3.5 A Case Study

The functional test compaction methodology is applied on a single-issue MIPS architecture [12] as described in Sect. 2.2.1. There are 17 functional units. We consider four functional states (activities) of each unit: ‘00’ for idle, ‘01’ for normal operation, ‘10’ for stall, and ‘11’ for exception. Figure 4.10 shows the binary format of a functional FSM model of the processor in the form of 33-bit binary. We assume that WB has only two states (idle and normal operation), and IALU and DIV have the exception state for overflow and divide-by-zero, respectively. All other functional units have three states (idle, normal operation, and stall). In the figure, the term

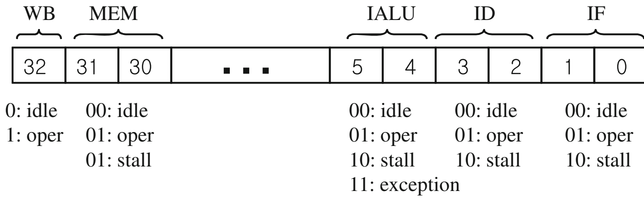


Fig. 4.10 33-bit FSM state model

Table 4.4 Test compaction results for MIPS processor

	Tests for state coverage	Tests for transition coverage
Reachable (states/transitions)	87.2×10^6	693.1×10^6
Legal and required (states/transitions)	83.5×10^6	376.3×10^6
Selected tests (states/transitions)	16.9×10^6	16.9×10^6
Overall reduction	97.8 %	

"oper" represents "normal operation". Therefore, theoretically possible number of states is $2 \times 4 \times 4 \times 3^{14} \simeq 153 \times 10^6$.

Unreachable states are removed by using the constraints of processor behavior such as single issue requirement. For example, the unreachable binary pattern 'xxxx...0101xxxx' (where x is a don't-care bit) represents the single issue constraint that two execution units IALU and MUL1 cannot be in normal operation at the same clock cycle. The corresponding number of states is 12.7×10^6. Those states can be eliminated in the FSM model since the states with this pattern are not allowed due to single-issue constraint. After removing all unreachable states, the number of states is reduced to 87.2×10^6 (43 % reduction).

Table 4.4 presents the results of property compaction for the MIPS processor. The first column indicates various compaction steps. The second and third column present compaction results for states and transitions, respectively. For example, the value 87.2×10^6 (in second row, second column) indicates the total number of reachable states after performing reachability analysis (on original 153×10^6 states). For FSM state compaction, the incoming and outgoing single transitions that have inevitable neighboring states are searched. We do not need to generate a test for those states since the test to exercise their neighbors will cover them.

After state compaction, the number of tests can be reduced by 3.7×10^6 (4 % reduction). As a result, the number of directed tests before test selection is 83.5×10^6. The FSM transition coverage needs to activate 693.1×10^6 transitions. The framework of selecting tests (minimum number of states/transitions required to achieve 100 % state and transition coverage) produced 16.9×10^6 tests. Therefore, the property compaction approach generates an overall 97.8 %[1] reduction of directed tests without sacrificing the functional coverage goal.

[1] $(87.2 + 693.1 - 16.9)/(87.2 + 693.1) = 97.8\,\%$.

4.4 Chapter Summary

This chapter presented a functional test compaction technique that can significantly reduce the number of directed tests without sacrificing the functional coverage goal. FSM model of pipelined processors and FSM coverage are used to illustrate the test compaction technique. It is important to note that compaction of required properties is performed before test generation and therefore the framework only needs to generate a reduced set of tests. This is in contrast with the existing test compaction approaches, especially in the manufacturing test domain where compaction is performed during or after test generation. Applying test compaction before test generation can reduce the test generation cost, and the compacted test set reduces the overall validation effort.

References

1. Barnhart C, Brunkhorst V, Distler F, Farnsworth O, Ferko A, Keller B, Scott D, Koenemann B, Onodera T (2002) Expending OP-MISR beyond 10x scan test efficiency. IEEE Des Test Comput 19(5):65–73
2. Campenhout D, Mudge T, Hayes J (1999) High-level test generation for design verification of pipelined microprocessors. In: Proceedings of design automation conference (DAC), pp 185–188
3. Chandra A, Chakrabarty K (2001) System-on-a-Chip test data compression and decompression architectures based on golomb codes. IEEE Trans Comput Aided Des Integr Circuits Syst 20(3):355–368
4. Chandra A, Chakrabarty K (2003) Test data compression and dtest resource partitioning for system-on-a-chip using frequency-directed run-length (FDR) codes. IEEE Trans Comput 52(8):1076–1088
5. Cheng K, Krishnakumar S (1996) Automatic generation of functional vectors using the extended finite state machine model. ACM Trans Des Autom Electron Syst 1(1):57–79
6. Corno F, Prinetto P, Rebaudengo M, Reorda M (1997) New static compaction techniques of test sequences for sequential circuits. In: Proceedings of European conference on design and test (ED&TC), pp 37–43
7. Dimopoulos M, Linardis P (2004) Efficient static compaction of test sequence sets through the application of set covering techniques. In: Proceedings of design automation and test in Europe (DATE), pp 194–199
8. El-Maleh A, Osais Y (2003) Test vector decomposition-based static compaction algorithms for combinational circuits. ACM Trans Des Autom Electron Syst 8(4):430–459
9. El-Maleh A, Al-Suwaiyan A (2001) An efficient test relaxation technique for combinational and full-scan sequential circuits. In: Proceedings of VLSI test symposium (VTS), pp 53–59
10. Flores P, Neto H, Marques-Silva J (1999) On applying set covering models to test set compaction. In: Proceedings of Great Lakes symposium on VLSI (GLSVLSI), pp 8–11
11. Gonciari P, Nicolici N (2003) Variable-length input Huffman coding for system-on-a-chip test. IEEE Trans Comput Aided Des Integr Circuits Syst 22(6):783–796
12. Hennessy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann, San Fransisco
13. Ho R, Yang C, Horowitz M, Dill D (1995) Architecture validation for processors. In: Proceedings of international symposium on computer architecture (ISCA), pp 404–413

14. Hochbaum D (1996) An optimal test compression procedure for combinational circuits. IEEE Trans Comput Aided Des Integr Circuits Syst 15(10):1294–1299
15. Jas A, Touba N (1998) Test vector compression via cyclical scan chains and its application to testing core-based designs. In: Proceedings of international test conference (ITC), pp 458–464
16. Jas A, Ghosh-Dastidar J, Mom-Eng N, Touba N (2003) An efficient test vector compression scheme using selective Huffman coding. IEEE Trans Comput Aided Des Integr Circuits Syst 22(6):797–806
17. Koo H, Mishra P (2008) Specification-based compaction of directed tests for functional validation of pipelined processors. In: Proceedings of international symposium on hardware/software codesign and system synthesis (CODES + ISSS), pp 137–142
18. Krishna C, Jas A, Touba N (2001) Test vector encoding using partial LFSR reseeding. In: Proceedings of international test conference (ITC), pp 885–893
19. Li L, Chakrabarty K, Touba N (2003) Test data compression using dictionaries with selective entries and fixed-length indices. ACM Trans Des Autom Electron Syst 8(4):470–490
20. Moundanos D, Abraham J, Hoskote Y (1998) Abstraction techniques for validation coverage analysis and test generation. IEEE Trans Comput 47(1):2–13
21. Miyase K, Kajihara S, Reddy S (2002) A method of static test compaction based on don't care identification. In: Proceedings of international workshop on electronic design, test, and application (ITC), pp 392–395
22. Pomeranz I, Reddy S (2001) Forward-looking fault simulation for improved static compaction. IEEE Trans Comput Aided Des Integr Circuits Syst 20(10):189–194
23. Pomeranz I, Reddy S (2010) On test generation with test vector improvement. IEEE Trans Comput Aided Des Integr Circuits Syst 29(3):502–506
24. Rajski J, Tyszer J, Kassab M, Mukherjee N (2004) Embedded deterministic test. IEEE Trans Comput Aided Des Integr Circuits Syst 23(5):776–792
25. Reda S, Orailoglu A (2002) Reducing test application time through test data mutation encoding. In: Proceedings of design automation and test in Europe (DATE), pp 387–393
26. Reddy S, Miyase K, Kajihara S, Pomeranz I (2002) On test data volume reduction for multiple scan chain designs. In: Proceedings of VLSI test symposium (VTS), pp 103–108
27. Rudnick E, Patel J (1999) Efficient techniques for dynamic test sequence compaction. IEEE Comput Soc 48(3):323–330
28. Shen J, Abraham J (2000) An RTL abstraction technique for processor microarchitecture validation and test generation. J Electron Test Theory Appl 16(1):67–81
29. Touba N (2006) Survey of test vector compression techniques. IEEE Des Test Comput 23(4):294–303
30. Utamaphethai N, Blanton R, Shen J (2000) Effectiveness of microarchitecture test program generation. IEEE Des Test Comput 17(4):38–49
31. Venkatesh R, Shanmugasundaram P, Parekhji R (2011) An efficient test data reduction technique through dynamic pattern mixing across multiple fault models. In: Proceedings of VLSI test symposium (VTS), pp 285–290
32. Villa T, Kam T, Brayton R, Sangiovanni-Vincenteili A (1997) Explicit and implicit algorithms for binate covering problems. IEEE Trans Comput Aided Des Integr Circuits Syst 16(7): 677–691
33. Wang L, Wen X, Wu S, Wang Z, Jiang Z, Sheu B, Gu X (2008) VirtualScan: test compression technology using combinational logic and one-pass ATPG. IEEE Des Test Comput 25(2): 122–130
34. Wang Z, Chakrabarty K (2005) Test data compression for IP embeded cores using selective encoding of scan slices. In: Proceedings of international test conference (ITC), pp 581–590

35. Wohl P, Waicukauski A, Patel S, DaSilva F, Williams T, Kapur R (2005) Efficient compression of deterministic patterns into multiple PRPG seeds. In: Proceedings of international test conference (ITC), paper 36.1
36. Wurtenberger A, Tautermann C, Hellebrand S (2004) Data compression for multiple scan chains using dictionaries with corrections. In: Proceedings of international test conference (ITC), pp 926–935
37. Zhang Y, Wang D, Wang J, Zheng W (2005) Using model-based test program generator for simulation validation. In: Proceedings of international conference on embedded software and systems, pp 549–556

Chapter 5
Property Clustering and Learning Techniques

5.1 Introduction

Although model checking techniques are promising for automated directed test generation, it is costly for complicated designs due to the state space explosion problem. For complex designs, there will be a large number of properties to be validated. When validating a specific system component, it is common that several properties have a large overlap on sub-functionalities. Validating the properties individually will be a waste of time due to the repeated validation efforts on the same functional scenarios. Potentially these redundancy can be avoided and consequently the overall test generation time can be significantly reduced.

The aim of property clustering and learning is to reduce the overall test generation time by exploiting the similarities among properties. Figure 5.1 shows the test generation framework using the property clustering and learning approaches [1, 2]. The presented methodology has three important steps: coverage-driven property generation, clustering of similar properties, and test generation using learning techniques. It is important to note that each of these three steps is independent. For example, this method uses the coverage of fault models to derive properties. The other two steps will produce beneficial results even if other fault models are used to generate properties. Designers can add various properties manually to the set of generated properties without affecting the usefulness of this approach.

The rest of this chapter is organized as follows. Section 5.2 presents the related approaches of SAT-based test generation. Section 5.3 introduces the implementation details of DPLL-based SAT solvers. Section 5.4 proposes property clustering approaches. Section 5.5 presents how to efficiently generate tests using the conflict clause based learning. Section 5.6 presents case studies on both hardware and software designs. Finally, Sect. 5.7 summarizes the chapter.

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_5,

© Springer Science+Business Media New York 2013

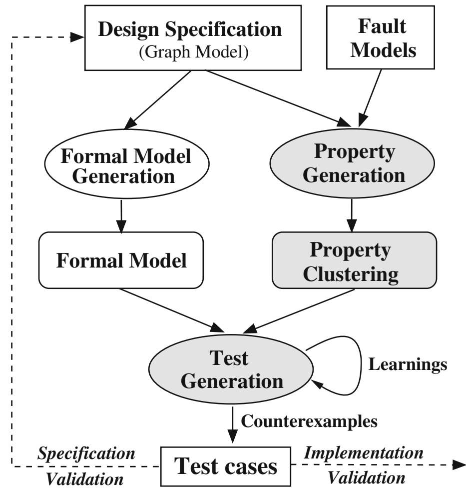


Fig. 5.1 Test generation methodology using property clustering and learning

5.2 Related Work

Due to the scalability issues of conventional binary decision diagram (BDD) based methods [3], SAT-based BMC is proposed as a complementary solution for large designs. Many studies in both software and hardware domains [4] show that BMC has better capacity and productivity over unbounded model checking for real designs.

Currently, various techniques based on conflict clause forwarding [5] are proposed to further improve the efficiency of BMC-based test generation. As a promising learning based approach, incremental SAT [6–9] tries to leverage the similarity between the elements of a sequence of SAT instances—most do so by re-utilizing learned knowledge based on conflict clauses. When many closely related instances need to be solved, caching solutions [10] and incremental translation [11] can also be effective. If a SAT instance is obtained from another by augmenting some clauses as described in [12], all conflict clauses of the first can be forwarded to the second. Therefore, when clauses are only added through a sequence of instances, there is no need to screen conflict clauses to determine which ones can be forwarded. This, on the other hand, is necessary when arbitrary clauses are both added or deleted to create a new instance. A common approach for such a general case is to have incremental SAT solvers keep track of whether a conflict clause depends on some removed clauses. Majority of the existing approaches exploit incremental satisfiability to improve the test generation time involving only one property with different bounds. There are very few approaches such as [13] where both static and dynamic learning are used across test generation instances for path-delay fault model by dynamically excluding the untestable path during test generation. Since the learning is employed across all test scenarios without efficient clustering methods, the improvement in test generation time is small (6 % on average) and has a wide variation (−7 to 27 %) on different ISCAS circuits.

5.3 Background: SAT Solver Implementation

This section introduces the preliminary knowledge of SAT solver implementation.

5.3.1 DPLL Algorithm

Most modern SAT solvers such as GRASP [14] and zChaff [5, 18] employ the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm [15, 16] for the searching of satisfying solutions.

Algorithm 1: DPLL search procedure of zChaff

```
while TRUE do
    run_periodic_functions();
    if decide_next_branch() then
        while deduce() == CONFLICT do
            blevel = analyze_conflicts();
            if blevel < 0 then
                return UNSAT;
            end
        end
    else
        return SAT
    end
end
```

Algorithm 1 shows the DPLL implementation in zChaff. It contains three major parts:

- **Periodic function** updates the SAT configuration triggered by some specified events, such as updating the scores of literals after a certain number of backtracks.
- **Boolean Constraint Propagation** (BCP) is implemented in *deduce*. It figures out all possible implications by previous decision assignment.
- **Conflict analysis** does a proper backtrack when encountering a conflict. It analyzes the reason for the conflict and makes it as a conflict clause to avoid the same conflict in future processing.

Studies in [5] show that modern SAT solvers spend approximately 80 % of time to carry out BCP. In addition, during the conflict analysis, long distance backtracks will increase the burden of SAT solvers.

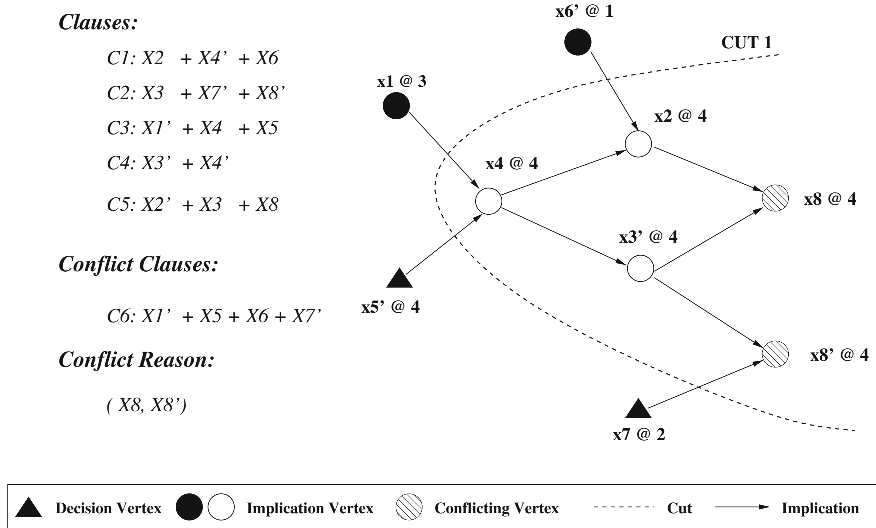


Fig. 5.2 Conflict analysis using an implication graph

5.3.2 Conflict Clause

As shown in Algorithm 1, SAT solvers use the conflict analysis technique to trace the reason for a conflict. The conflict analysis contains two part: conflict-driven backtracking and conflict-driven learning. Conflict-driven backtracking enables the non-chronological backtracking up to the closest decision which caused the conflict. Conflict-driven learning produces conflict clauses and adds them to the original clauses, in order to avoid the same conflict in the future. Both techniques can drastically boost the performance of the SAT solvers.

The kernel of the conflict analysis technique is the implication graph [14, 17]. The graph keeps the current state and the implication history of the search during the SAT solving by recording the dependence of variable assignments. The implication graph is a directed acyclic graph where each vertex represents an assignment to a Boolean variable and each edge implies that all the in-edges implicate the assignment of the vertex.

Figure 5.2 shows a small example of conflict analysis using an implication graph. As shown at the left of the figure, there are five original clauses *C1-C5*. The right part is a scenario of implication graph for *C1-C5*. In this example, $x4@4$ means variable $x4$ is assigned with value 1 at decision level 4. The node has a corresponding clause $(x1' + x4 + x5)$, we call it the antecedent clause of $x4$, i.e., the assignments $x1 = 1$ and $x5 = 0$ imply $x4 = 1$. Only the implication vertex (non-decision vertex) has an antecedent clause. A conflict happens when two nodes in the implication graph have different value assignments for the same variable. For example, the implications in the graph lead to the ambiguous assignment to variable $X8$ ($X8 = 0$ and $X8 = 1$).

When encountering a conflict, conflict analysis will trace back along the implication relations to find the reason for the conflict and encode the reason using a conflict clause. A conflict clause can be found by a bipartition of the implication graph. The side containing the conflicting vertex is called conflict side, and the other side is called reason side which can be used to form the conflict clause. In Fig. 5.2, $CUT1$ is a cut that divides the implication graph into two parts. The conflict analysis stops at $CUT1$. The left part of $CUT1$ in the implication graph is the reason side, and the right part is the conflict side. From the reason side, we can get the conflict clause C6=(X1′ + X5 + X6 + X7′). That means, the assignment of variables $X1 = 1$, $X5 = 0$, $X6 = 0$, and $X7 = 1$ will always lead to a conflict because of the clauses $C1 - C5$. Lemma 5.1 indicates that the generated conflict clauses during the SAT search can be added to original clause set as an assignment constraint. Therefore, we can add the clause $C6$ to the original clause set to avoid the same conflict in the future.

Lemma 5.1 *Given a set of Conjunctive Normal Form (CNF) clauses $S1$, ψ is a conflict clause derived during the conflict analysis, then $S1$ is satisfiable iff $S1 \bigwedge \psi$ is satisfiable.*

Proof Because $S1 \bigwedge \psi$ is a super set of $S1$, so if $S1 \bigwedge \psi$ is satisfiable then $S1$ is satisfiable. According to the definition of the conflict clause, the assignments that make the clause ψ false will make the clause set $S1$ false. If $S1$ is satisfiable, then there exists a variable assignment that makes $S1$ true. This assignment should make ψ true. So the assignment will make $S1 \bigwedge \psi$ true. ■

For two SAT instances, if one SAT instance is a subset of the other SAT instance, according to Theorem 5.1, the conflict clauses generated from the smaller-size SAT instance can be forwarded to the larger-size SAT instance. In other words, the local learning can be forwarded as a knowledge for global searching. Usually the average cost of locally learned conflict clauses is much cheaper than the globally learned conflict clauses.

Theorem 5.1 *Given two CNF clause sets $S1$ and $S2$, where $S1 \subseteq S2$, ψ is a conflict clause derived from the clauses in $S1$, written $S1 \vdash \psi$, then $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable.*

Proof Since $S2 \bigwedge \psi$ is a super set of $S2$, if $S2 \bigwedge \psi$ is satisfiable then $S2$ is satisfiable. Because $S1 \vdash \psi$ and $S1 \subseteq S2$, then ψ is also a conflict clause of $S2$. According to Lemma 5.1, $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable. ■

According to the Eq. 3.2 in Chap. 3, similar properties share a large part of the CNF clauses. Regardless of the cone of influence, the equation shares the system part (i.e., transition relation $T(s_i, s_{i+1})$) and the part of property testing (i.e., $p(s_i)$). For two similar properties in this case, sharing a large part of CNF clauses indicates that when checking the first property, the learned knowledge (conflict clauses) derived only from their CNF intersection can be forwarded to the second property without affecting the satisfiability of the CNF clauses of the second property.

Theorem 5.2 *Assume that we have two sets of CNF clauses* $S1$ *and* $S2$*, and let* $\omega = S1 \bigcap S2$ *be the common clause set shared by both* $S1$ *and* $S2$*.* ψ *is a conflict clause derived only by the clauses in* ω*, written* $\omega \vdash \psi$*. Then* $S2$ *is satisfiable iff* $S2 \bigwedge \psi$ *is satisfiable.*

Proof Because $S2 \bigwedge \psi$ is a super set of $S2$, so $S2 \bigwedge \psi$ is satisfiable then $S2$ is satisfiable. Because $\omega \vdash \psi$ and $\omega \subseteq S2$, then $S2 \vdash \psi$. According to Lemma 5.1, $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable. ∎

5.4 Property Clustering

Given a set of properties, a property clustering method determines how to divide the properties into several groups such that each group contains similar properties that can benefit from each other during test generation. The similarity can be structural or behavioral, but the assumption is that there is a significant overlap among the counterexample assignments derived from a set of similar properties.

Algorithm 2: Property Clustering

Input: i) A set of properties, P
  ii) Similarity strategy CS, and threshold W_{th}
Output: Clusters consisting of similar properties
PropertyClusters = ϕ;
1. Construct a graph, G where each node is a property;
for *each pair of nodes* (n_i, n_j) *in* G **do**
  Weight w_i^j = ComputeSimilarity $CS(n_i, n_j)$;
  if($w_i^j \geq w_{th}$) Create an edge between n_i and n_j with weight w_i^j;
end
2. k = 1; /* first cluster */
while G *is not empty* **do**
  $Base_k$ = Node with highest overall edge weight;
  $Cluster_k$ = all the nodes connected to $Base_k$;
  $G = G - Cluster_k$;
  PropertyClusters = *PropertyClusters* $\cup$ $Cluster_k$;
  $k = k + 1$;
end
return *PropertyClusters*

Algorithm 2 outlines the major steps in property clustering. The first step constructs a property graph[1] where the properties are nodes and edges represent similarity. An edge is added between two properties (nodes) when they are similar.

[1] This chapter uses three different types of graphs for three different purposes. The graph model of the design (or *design graph* in short) is used to model the design. The *implication graph* is used to store the dependence of variable assignments that is used for conflict analysis. The *property graph* models the similarity between properties and used for clustering.

Each edge e_j includes weight information to quantify the similarity. An edge with weight 0 or 1 is not possible since an weight of 0 means no similarity, and an weight of 1 implies same (identical) property. To compute the weight information for each e.g., four methods can be used—structural, textual, influence, and CNF intersection based similarity. Each method will use a similarity threshold for clustering. In other words, there will be no edge between two properties when the weight value is below certain threshold. The second step determines the clusters based on the base property. The base property is the property (node) with highest weight (summation of weights of all edges connected to that node). The cluster is formed by adding all the adjacent nodes with the base property. All the nodes selected for a cluster are deleted from the property graph for the next iteration. The remainder of this section describes four different ways of computing similarity between properties.

5.4.1 Similarity Based on Structural Overlap

A simple and natural way to cluster properties is to exploit the structural information of the design model and its properties. The intuition is that two similar properties will share similar variable assignments (global and local variables[2]) in the counterexamples. In fact, a conflict clause is a constraint on the assignment of the variables. Therefore, properties with similar structural information will share a lot of conflict clauses.

As mentioned earlier, in the context of directed test generation, properties are generated based on functional coverage of the design. These properties try to cover different parts of the design (e.g., all computation nodes, various interactions, etc.). Therefore, we can cluster the properties that try to cover a specific functionality or interactions. For example, in an SoC environment, the properties can be clustered based on whether they are related to verifying the processor, co-processor, FPGA, memory, bus synchronization, or controllers. Each cluster can be further refined based on structural details of each component. For example, the processor related properties can be further divided based on which execution path they activate, such as ALU pipeline, load-store pipeline, etc.

In the pipelined processor example in Fig. 2.1, there are four execution pipelines: $IALU$, MUL, $FADD$, and DIV. The corresponding paths are as follows.

- $\rho_1 = FET \rightarrow DEC \rightarrow IALU \rightarrow MEM \rightarrow WB$
- $\rho_2 = FET \rightarrow DEC \rightarrow MUL1 \cdots \rightarrow MUL7 \rightarrow MEM \rightarrow WB$
- $\rho_3 = FET \rightarrow DEC \rightarrow FADD1 \cdots \rightarrow FADD4 \rightarrow MEM \rightarrow WB$
- $\rho_4 = FET \rightarrow DEC \rightarrow DIV \rightarrow MEM \rightarrow WB$

Consider two properties $p1 =\sim F(FADD3.active = 1)$ and property $p2 =\sim F(FADD4.active = 1)$. They share the same path ρ_3, and the bound of $p1$ is

[2] In a graph model, a local variable is defined locally inside a node whereas the scope of a global variable is valid across nodes.

just one smaller than $p2$. So we can cluster them together. Also for the interaction property $p3 =\sim F(FADD4.active = 1 \,\&\, MUL3.active = 1)$ and $p4 =\sim F(FADD3.active = 1 \,\&\, MUL4.active = 1)$, the two interactions are related to the same set of paths ρ_2 and ρ_3 and have similar bounds. Therefore, clustering them together is a good choice.

5.4.2 Similarity Based on Textual Overlap

Another simple way to quantify similarity is to measure the textual differences between two properties. For example, the similarity between $\neg F(a \,\&\, b \,\&\, c)$ and $\neg F(b \,\&\, c \,\&\, d)$ is 67 % since they share a common part consisting of two subexpressions b and c.

This section focuses on bounded model checking of invariants (safety properties) such as the property in the form ¬ F(p). Informally, $BMC(M, p, k)$ is true means from cycle 0 to cycle k, the property will be false in some cycle. So the invariant cannot always be true and one counterexample will be reported. Because the part $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ comes from the design, for different properties this part is same. The part $\bigvee_{i=0}^{k} \neg p(s_i)$ usually determines the difference among the properties.

The negative format of each literal in the conflict clause is a false assignment for the logic formula $BMC(M, p, k)$. In fact, the conflict clause can be regarded as a constraint for the variable assignment. Let P and Q be two properties of the model, the properties P, $P \wedge Q$, and $P \vee Q$ can be expanded as follows:

- $BMC_1(M, P, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i)$
- $BMC_2(M, P \wedge Q, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg(P \wedge Q)(s_i)$
  $= I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k}(\neg P(s_i) \vee \neg Q(s_i))$
- $BMC_3(M, P \vee Q, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k}(\neg P(s_i) \wedge \neg Q(s_i))$

In the expanded Boolean formulas above, we assume that the same variable in respective expansion has the same meaning. Let A be a partial assignment of the CNF variables. Therefore, $BMC_1 \not\models A$ implies $BMC_3 \not\models A$, $BMC_2 \not\models A$ implies $BMC_1 \not\models A$, and $BMC_2 \not\models A$ implies $BMC_3 \not\models A$. In other words, the conflict clauses of BMC_1 can be forwarded to BMC_3, and conflict clause of BMC_2 can be forwarded to both BMC_1 and BMC_3.

In most existing BMC tools, the variables in the generated CNF file do not have specific meaning. The conflict clauses of the stronger property cannot be directly forwarded to some weaker properties. For example, some conflict clauses of property $P \wedge Q$ cannot be forwarded to check property $P \vee Q$. However, when properties have the relation of implication, and their textual similarity is high, clustering them together will have a positive effect. If two properties are in the same format and have a significant (more than 50 %) textual overlap, the two properties generally can benefit from each other.

Textual clustering is very fast, but it may not be very accurate in clustering similar properties in some cases. For example, the properties $\neg F(a)$ and $\neg F(c)$ have no textual overlap. However, it is possible that both properties are very closely related in the design model (e.g., they can activate the same path), and therefore they can actually be a good candidate for clustering. In practice, textual clustering is beneficial when information regarding the designs or original fault models are not available and/or when there are many textual overlaps.

5.4.3 Similarity Based on Influence

An assignment to a global variable determines the state transition of various components in the design (graph) model. For example, in the MIPS model, when the instruction buffer contains only division instruction, only the components in DIV path will be activated. However, it is time-consuming to analyze all the global and local variables of the model since it requires to consider the state transition of each component.

Based on the graph model structure, we can determine various cause-effect relations. For example, the state change of $MUL6$ will be one clock cycle later than $MUL5$. That means the execution of $MUL5$ has an influence on the execution of $MUL6$. The influence nodes indirectly reflect the assignment of the global variables, since the assignment of global variables is relevant to the variable assignment in the counterexample.

Prior to clustering, it is important to figure out the influence node set for each node in the graph model. We can compute the influence node set for each node using depth first search (DFS) algorithm. If there is a path starting from the start node to the current node, then all the nodes on this path are influence nodes for the current node. DFS can explore all such paths (except the paths with loops) from the start node to the current node. For example the influence node sets for $MUL2$, $FADD3$, and WB are as follows:

- $Influence(MUL2) = \{FET,\ DEC,\ MUL1,\ MUL2\}$
- $Influence(FADD3) = \{FET,\ DEC,\ FADD1,\ FADD2,\ FADD3\}$
- $Influence(WB) = \{n|$ n is a node in the MIPS graph model.$\}$

A property may correspond to several nodes (modules) in the graph model. So the influence node set of a property is the union of the influence of all relevant nodes. When comparing the similarity of two properties, we need to compute the intersection of influence sets. For example, the influence set of property $\sim F(MUL2.active = 1\ \&\ FADD3.active = 1)$ is $S_1 = \{FET, DEC, MUL1, MUL2, FADD1, FADD2, FADD3\}$ and the influence set for $\sim F(MUL3.active = 1\ \&\ FADD3.active = 1)$ is $S_2 = \{FET, DEC, MUL1, MUL2, MUL3, FADD1, FADD2, FADD3\}$. The two sets share a large intersection. For set S_1, the similarity with S_2 is $7/7 = 100\,\%$. For set S_2 the similarity with S_1 is $7/8 = 87.5\,\%$. Based on previous experience, when the overlap of influence sets are larger than 70 %, forwarding conflict clauses is beneficial. In this example, S_1 and S_2 can be clustered together.

5.4.4 Similarity Based on CNF Intersection

One obvious, but costly, way to determine property similarity for clustering is to compute intersections of CNF clauses between properties. We can cluster properties that have a relatively large number of clauses in the intersection. Based on the experience, a threshold of 0.9 is beneficial. In other words, when two properties share at least 90 % common clauses, it is beneficial to forward conflict clauses between two instances.

CNF intersection based method is very time-consuming because it requires $O(n^2)$ intersections for n properties. When n is large, this method is not feasible, because the calculation of intersection of irrelevant properties may waste more time than actual SAT solution time. Moreover, in certain scenarios, forwarding conflict clauses may not improve the overall test generation time for a cluster, since it may change variable ordering and searching heuristics. CNF intersection based clustering is a good choice only when the number of properties is small or when other methods fail to find beneficial clusters.

5.4.5 Determination of Base Property

Determination of base property in a cluster is crucial for test generation using learning techniques. The base property is solved first and its conflict clauses are shared by the remaining properties in the cluster. Although, any property in the cluster can be used as the base property for that cluster, previous studies have shown that certain properties serve better as base property and thereby generate better overall savings for the cluster. We need to consider two important factors while choosing a base property for a cluster. First, the base property should be able to generate a large number of conflict clauses. In other words, a weak base property may find the satisfying assignment quickly without making mistakes (generating conflict clauses). In this scenario, the remaining properties have nothing to learn from the base property. Moreover, the SAT checking time for the base property should be relatively small. This will ensure that the overall gain is maximized by reducing the solution time of the properties which takes longer time to solve. It is important to note that none of these requirements can be determined without actually solving them. The experience shows that the following heuristics work well most of the time.

- Choose a property that has significant variable and/or sub-expression overlap with other properties in the cluster.
- If bound for each property is known, choose the property whose bound is closest to the remaining properties.
- Compute intersection of every pair of properties in the cluster, and choose the one that shares the most with the remaining properties.

5.5 Conflict Clause Based Test Generation

Since a conflict clause can be used to avoid the repetitive occurrences of the same conflict, it can be used as a learning that can be forwarded from the checking SAT instances to the unchecked SAT instances. Based on this observation, this section shows how to utilize the derived conflict clauses to reduce the overall test generation time of a cluster of similar properties.

5.5.1 Conflict Clause Forwarding Techniques

The basic idea of conflict clause forwarding is to reuse the learning (i.e., conflict clauses) from the solved properties to improve the test generation time of the unchecked properties in the same cluster. While solving the first property (base property), the SAT solver may have taken many wrong decisions (lead to conflicts) and therefore needs long time to find a counterexample. Forwarding conflict clauses ensures that these wrong decisions are avoided while solving the similar properties. An important question is whether all the wrong decisions of the first property are relevant to all the other properties in the clusters? Since the properties are similar but not the same, some of the decisions are not relevant. To identify the conflict clauses that can be forwarded to help the solving of unchecked properties, one feasible way is to compute the intersection of CNF clauses of such properties.

Algorithm 3 describes the test generation methodology by reusing the learned conflict clauses. It accepts a list of clusters where each cluster consists of a set of similar properties. Since one property is used to generate a test, the number of input properties is exactly the same as the number of output tests. The first step generates the CNF clauses for all the properties in each cluster using the design and respective bounds. The second step performs name substitution to maximize knowledge sharing. The third step computes the intersection of CNF clauses between the base property and all the remaining properties in the cluster. The first three steps can be omitted, if CNF intersection based clustering is employed. The fourth step marks the clauses in the base property to indicate whether a particular clause is also in the clause set of another property in the cluster. The next step uses a SAT solver to generate the conflict clauses and the counterexample for the base property. Based on the intersection information with the base property, the set of conflict clauses is filtered to identify the relevant ones for solving the remaining properties in step 6. Step 7 uses the forwarded conflict clauses to solve the remaining properties. Finally, the algorithm reports all the generated directed tests.

Algorithm 3: Test Generation using Conflict Clauses based Learning

Input: i) Design model D
ii) Clusters of similar properties
Output: Tests
for *each cluster, i, of properties* **do**
  Generate CNF for the base property P_1^i, CNF_1^i;
  for j *is from 2 to the* $size_i$ *of cluster* i **do**
    /* P_j^i is the jth property in the ith cluster */;
    1. Generate CNF, $CNF_j^i = BMC(D, P_j^i, bound_j^i)$;
    2. Perform name substitution on CNF_j^i;
    3. INT_j^i = ComputeIntersection(CNF_1^i, CNF_j^i);
    4. Mark the clauses of CNF_1^i using INT_j^i;
  end
  /* Generate a counterexample and record conflict clauses */;
  5. ($ConflictClauses_i$, $test_1^i$) = SAT(CNF_1^i);
  Tests = $\{test_1^i\}$;
  for j *is from 2 to the* $size_i$ *of cluster* i **do**
    /* Find relevant ones for P_j^i from conflict clauses */;
    6. CC_j^i = Filter ($ConflictClauses_i$, j);
  end
  for j *is from 2 to the* $size_i$ *of cluster* i **do**
    7. $test_j^i$ = SAT($CNF_j^i \bigcup CC_j^i$);
    Tests = *Tests* $\cup$ $test_j^i$;
  end
end
return *Tests*

A simple example is used to illustrate how Algorithm 3 works. Let us assume that we are generating tests using n properties for a design. The input is a list of m ($m \leq n$) clusters based on property similarities. Each cluster can have different number of properties. In the worst case, each cluster can have only one property which will be verified normally. However, this scenario is rare in practice since a typical design uses thousands of properties for directed test generation and majority of them share significant parts of the design functionality. For ease of illustration, let us assume that there is a cluster with three similar properties, $\{P_1, P_2, P_3\}$. Let us further assume that the second step selects P_1 as the base property. The fourth step computes intersection of CNF clauses of P_1 with P_2, and P_1 with P_3. This information is used to filter conflict clauses (generated while solving P_1) which are beneficial to P_2 and P_3 in step 6. The last step adds the relevant conflict clauses while solving the respective properties to reduce the test generation time.

The following subsections describe two important techniques: name substitution for computation of intersections, and identification of relevant conflict clauses.

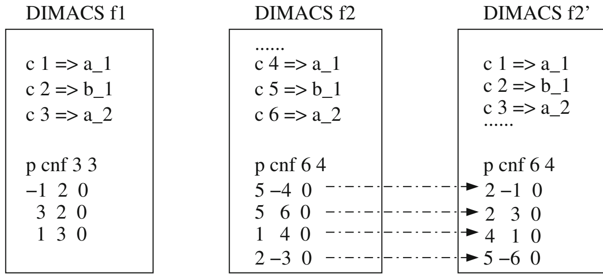


Fig. 5.3 An example of name substitution

5.5.2 Name Substitution for Computation of Intersections

Name substitution is an important preprocessing step. Currently, few BMC tools support the name mapping from the variables of the CNF clauses to the names in the model of the unrolled design. As a result, the variables of the CNF clauses of two different properties may not have any name correspondence. In other words, the same variable in two properties may have different names in their respective CNF clauses. Therefore, without name substitution (mapping), it will miss the overlap information. As a result, the computed intersection will be small and will adversely affect the sharing of learned conflict clauses. It is observed that the improvement in test generation time without using name substitution is negligibly small due to very small number of clauses being forwarded as a result of small number of clauses in the intersection. Since the properties are similar and the design is exactly the same, the size of the intersection is very large when the name substitution method is employed.

The proposed framework uses the SAT solver zChaff [18] which accepts the input in the DIMACS format. The generated DIMACS file for each property provides the name mapping from the CNF variable to the unrolled design. For example, "c 8 = V1_var[6]" shows that the variable 8 is used in the CNF file to refer to the 7th bit of variable *var* in the design specification at time step 1.

Given two DIMACS files $f1$ and $f2$ for two properties P_1 and P_2 respectively, the name substitution is a procedure that changes the names of clause variables of $f2$ using the name mapping defined in $f1$. Figure 5.3 shows an example for name substitution. Before the name substitution, the intersection ($f1 \cap f2$) is empty. However, after name substitution, there are two common clauses in the intersection ($f1 \cap f2'$). The complexity of both name substitution and computation of intersection is linear (using hash table) to the size of the DIMACS file of the properties. Therefore, the time required by name substitution and intersection computation is negligible compared to the SAT solving time for complex properties.

It is important to note that the same variable at different time steps can be assigned a different number. Therefore, the name mapping (substitution) method needs to consider the same variable at different time steps in the CNF clauses of the same property as well as in the CNF clauses for the different properties in the same cluster.

Moreover, the name mapping routine needs to remap some of the variables in the CNF clauses. For example in Fig. 5.3, when the variable 4 in file $f2$ is replaced with the variable 1 (in $f2'$), the name mapping routine needs to remap the original variable 1 in file $f2'$ to a different variable.

5.5.3 Identification and Reuse of Common Conflict Clauses

The implementation of relevant conflict clause determination is motivated by the work of Strichman [9] which proved that for two sets of CNF clauses C_1 and C_2, and their intersection φ, use of conflict clauses generated from φ when checking C_1 will not affect the satisfiability of the CNF clauses $C_2 \bigcup \varphi$ (proved in Theorem 5.2). Therefore, the conflict clauses generated from the intersection when checking the base property can be shared by other properties in the cluster.

Strichman [9] suggested an isolation procedure that can isolate the conflict clauses which are deduced solely from the intersection of two CNF clause sets. The isolation procedure is modified to improve the efficiency of test generation for a cluster of properties. In addition, the SAT solver zChaff [18] has also been modified to be incorporated in the test generation framework.The zChaff provides utilities for implementing incremental satisfiability. For each clause, it uses 32 bits to store a group id to identify the group where this clause belongs. Use of group id allows us to generate the conflict clauses for different properties when checking the base property. If the *i*th bit of the clause's group id is 1, it implies that the clause is shared by the CNF clauses of property P_i. If the clause of the base property is not shared by any property, the field will be 0.

Assume that there are $k + 1$ properties in a cluster with C_i as the set of CNF clauses for the property P_i. Moreover, assume that P_0 is the base property. In other words, there are $k + 1$ sets of clauses with C_0 as the base set, and $C_1, C_2, ..., C_k$ are k similar sets with C_0. The following steps can be used to calculate the conflict clauses for $C_1, C_2, ..., C_k$ when solving C_0.

- During preprocessing, for each clause cl in C_0, if this clause also exists in C_i $(2 \leq i \leq k)$, then mark the *i*th bit of cl's group id as 1.
- When one conflict clause is encountered during the checking of the base property, collect all the group ids of the clauses in the conflict side. The group id of the conflict clause is logical "AND" of all these group ids.
- For each conflict clause, if the *i*th bit of the group id is 1, then this conflict clause can be shared by C_i.

As described above, each conflict side clause has a group id which is marked during the preprocessing step or marked during the conflict analysis if it is a conflict clause. The procedure of group id determination of a conflict clause is described in Algorithm 4.

Algorithm 4: Determination of conflict clause and its group ID

Input: i) Conflicting node N
Output: Conflict clause with its group id
Visited = { N};
ConflictAssign = {};
groupId $=$ group id of $N's$ antecedent clause;
while *the set Visited is not empty* **do**
  1. v = RemoveOneElement(*Visited*);
  2. *clause* = AntecedentOf(v);
  groupID = *groupID "AND"* group id of *clause*;
  if v *is on the conflict side* **then**
    3. Put all the nodes of *clause* in implication graph except v to *Visited*;
  else
    4. *ConflictAssign* = *ConflictAssign* $\cup$ {v};
  end
end
5. *ConflictClause* = Logical disjunction of negated assignments of all elements in *ConflictAssign*;
return *ConflictClause* and *groupId*

This algorithm traces back from the conflicting assignment to a cut such as *first Unique Implication Point* (UIP) [17] in zChaff. The conflict side will contain all the implications of the variable assignments of the reason side. For UIP, they are implication variable assignments in the same decision level as the conflicting variable assignment which led to the conflict. The group id of the conflict clause is the logical "AND" value of all the group ids of the conflict side clauses. This algorithm can guarantee that if the ith bit of the group id of the conflict clause is 1, then this conflict clause can be forwarded to the ith CNF clause set.

Figure 5.4 illustrates how this computation is done. The implication graph belongs to a base property of a cluster. Each clause in this graph is marked with the group id information. Here four bits are used to express the group id. For example, the group id "1010" of clause $(x3' + x4')$ means that this clause exists both in CNF clause set C_2 and CNF clause set C_4. The group id of the conflict clause is the logical "AND" of all conflict side clauses, and the result is 0010. That means, this conflict clause can be forwarded to clause set C_2. Therefore, the use of this conflict clause in solving P_2 will reduce the SAT solving (test generation) time.

5.6 Case Studies

To demonstrate the effectiveness of the presented test generation methodology, various software and hardware designs have been checked. This section presents two case studies: an implementation of the MIPS architecture, and a stock exchange

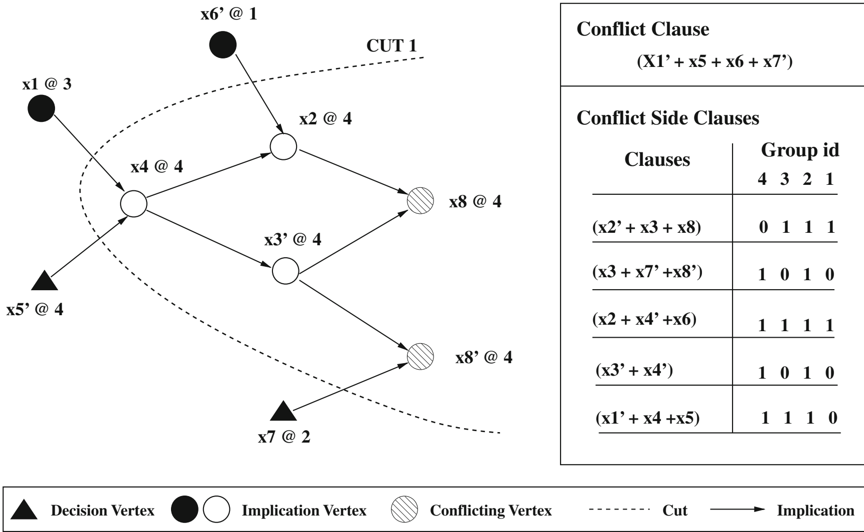


Fig. 5.4 An example of conflict clause reuse

system. Both experiments were performed on a Linux PC using 2.0GHz Core 2 Duo CPU with 1 GB RAM. In these experiments, NuSMV [19] is used as the BMC tool to generate the CNF clauses (in the DIMACS format) for the design and properties. A tool called *PropertyCluster* is developed, which accepts formal models (i.e., graph models, FSM models), the fault models, and the clustering strategies as inputs. This tool generates the required properties and clusters them using the clustering strategies proposed in Sect. 5.4. The zChaff [18] is modified to incorporate the presented techniques including name substitution, clause intersection, and constraint sharing. The modified zChaff can accept a cluster of properties and check them together.

5.6.1 A MIPS Processor

The details of the MIPS processor have been presented in Fig. 2.1. To validate the design, the tool *PropertyCluster* generated 171 properties using the node coverage, 2-interaction coverage, and the path coverage criteria. In this section, we first present results for each clustering technique, and then present a summary to compare the clustering techniques.

5.6.1.1 Structure-Based Clustering

The graph model of MIPS processor has four parallel pipeline paths. Each of them shares four units (fetch, decode, memory, and writeback), and differs only in the execution units. The structural similarity is established based on the path that a set

Table 5.1 Verification results for a structure-based cluster

Property	Type	Bound	Size	Forward	Original (s)	New(s)	Speedup
$p13$	Interaction	8	461122	–	15.61	21.99	0.71
$p28$	Edge	7	395566	32576	8.31	0.16	51.94
$p133$	Edge	7	395564	32576	11.99	0.18	66.60
$p134$	Inter.	7	395564	32576	9.07	0.19	47.74
$p150$	Node	6	330002	21748	4.70	0.16	29.38
$p165$	Path	8	461132	35121	22.87	0.27	84.70
$p170$	Path	8	461142	35121	24.45	0.26	94.04
Average	–	7.29	414299	–	13.86	3.32	4.18

of properties activates. For example, the following seven properties is grouped in a cluster because all of them refer to the division path.

- $p13 \; =\sim F(FET.active = 1\&DIV.active = 1)$
- $p28 \; =\sim F(DEC.active = 1\&DIV.active = 1)$
- $p133 =\sim F(DIV.active = 1\&MEM.active = 1)$
- $p134 =\sim F(DIV.active = 1\&WB.active = 1)$
- $p150 =\sim F(DIV.active = 1)$
- $p165 =\sim F(FET.active = 1\&DEC.active = 1\&DIV.active = 1)$
- $p170 =\sim F(FET.active = 1\&DEC.active = 1\&DIV.active = 1$
  $\&MEM.active = 1\&WB.active = 1)$

Table 5.1 presents the verification details for the above cluster. This cluster has seven properties where $p13$ is the base property. The second column shows the property type (node coverage, edge coverage, interaction coverage, etc.). The third column indicates the bound for that property. The fourth column shows the number of CNF clauses (size) for that property. The fifth column presents the number of conflict clauses forwarded from the base property. The next column presents the test generation time (original, in seconds) using unmodified zChaff. The seventh column presents the test generation time using the forwarded learning. For the base property, the new time is larger than the original time, since it includes the intersection calculation time with other properties in the cluster. The speedup is computed using the formula (Original Time/New Time). The overall speedup for this cluster is 4.18x.

Table 5.2 provides the test generation details of the clusters generated using the structural similarity. The total 171 properties are grouped into 16 clusters shown in the first column. The example presented in Table 5.1 is the expansion of the fifth cluster in Table 5.2 (row 5). The second column presents the size of that cluster in terms of number of properties. The base time is the execution time of the base property. The original time is the running time of the remaining properties (except the base property) without using any knowledge sharing techniques. Since intersection calculation is necessary before executing the base property, we show the

Table 5.2 Structure-based clustering results for MIPS processor

Cluster index	Size (# Prop)	Base time (s)	Original time (s)	Improved time		Speedup
				Verify(s)	Overhead(s)	
1	10	1.21	68.26	32.01	5.91	1.78
2	8	1.84	83.26	37.43	6.02	1.88
3	17	15.90	193.21	2.49	15.44	6.18
4	17	18.31	173.20	3.81	14.47	5.23
5	7	15.61	81.40	1.22	6.38	4.18
6	7	2.03	120.38	40.05	5.71	2.56
7	4	2.15	15.94	5.79	2.62	1.71
8	1	8.56	8.56	8.56	0.00	1.00
9	17	30.92	582.80	59.44	17.57	5.69
10	17	2.30	149.75	50.74	12.83	2.31
11	7	10.54	140.31	30.77	6.78	3.14
12	17	9.40	669.83	164.34	17.39	3.55
13	11	21.21	365.79	44.1	12.26	4.99
14	4	10.62	46.58	3.84	3.54	3.18
15	14	15.84	142.78	4.00	11.47	5.07
16	13	2.65	263.93	149.19	11.92	1.63
Average	10.69	10.57	194.12	39.86	9.39	3.42

improved time in two parts: new verification time, and overhead (intersection calculation time). The last column shows the speedup using the formula (Base time + Original time)/(Base time + Improved time). In this table, we can find that the overhead has a linear relation with the number of properties in the cluster. Using structure-based clustering, a speedup of 3.42x [3] can be achieved.

5.6.1.2 Clustering Based on Textual Similarity

Since the properties are generated based on fault models, they use similar format and therefore are helpful for clustering based on textual similarity. In this case, it is assumed that 50 % is a reasonable threshold for textual similarity. For example, the following properties are textually similar. In this case, $p49$ is the base property, and other 6 properties has 50 % similarity with it. So they can be clustered together.

- $p49 =\sim F(MUL1.active = 1 \& MUL6.active = 1)$
- $p50 =\sim F(MUL1.active = 1 \& MUL7.active = 1)$
- $p61 =\sim F(MUL2.active = 1 \& MUL6.active = 1)$

[3] Clustering time using structural similarity is negligible and not shown in the table.

Table 5.3 Verification results for a textual cluster

Property	Type	Bound	Size	Forward	Original (s)	New(s)	Speedup
$p49$	Interaction	10	592239	–	59.54	68.81	0.87
$p50$	Interaction	11	657806	78826	81.09	5.88	51.94
$p61$	Interaction	10	592239	78826	60.72	0.31	195.87
$p72$	Interaction	10	592239	78826	62.37	0.31	201.19
$p82$	Interaction	10	592239	78826	61.91	0.31	199.71
$p91$	Edge	10	592239	78826	67.96	0.31	219.23
$p100$	Edge	11	657806	78826	84.17	6.08	13.84
Average	–	10.29	610972	–	68.25	11.72	5.82

- $p72 =\sim F(MUL3.active = 1\&MUL6.active = 1)$
- $p82 =\sim F(MUL4.active = 1\&MUL6.active = 1)$
- $p91 =\sim F(MUL5.active = 1\&MUL6.active = 1)$
- $p100 =\sim F(MUL6.active = 1\&MUL7.active = 1)$

Table 5.3 shows the test generation details for a cluster consisting of above seven properties. The numbers in the table are in the same format as Table 5.1. Due to knowledge sharing, the speedup for this cluster is 5.82x.

Table 5.4 shows the test generation details for all 32 clusters using textual similarity. Table 5.3 is the expansion of the 22nd cluster of Table 5.4 (row 22). In this case, it is able to obtain a $3.72X$ overall speedup.

5.6.1.3 Influence-Based Clustering

The following 7 properties are grouped using influence-based clustering with $p111$ as the base property. The threshold of the similarity is set to be 70 %. For instance, the influence nodes of $p111$ are $\{FET$, DEC, $MUL1$, $MUL2$, $MUL3$, $MUL4$, $MUL5$, $MUL6$, $MUL7$, $FADD1$, $FADD2$, $FADD3$, $FADD4\}$, and the influence of $p108$ is $\{FET$, DEC, $MUL1$, $MUL2$, $MUL3$, $MUL4$, $MUL5$, $MUL6$, $MUL7$, $FADD1\}$. The similarity between $p108$ and $p111$ is $10/13 = 77\,\%$.

- $p111 =\sim F(MUL7.active = 1\&FADD4.active = 1)$
- $p104 =\sim F(MUL6.active = 1\&FADD4.active = 1)$
- $p110 =\sim F(MUL7.active = 1\&FADD3.active = 1)$
- $p103 =\sim F(MUL6.active = 1\&FADD3.active = 1)$
- $p109 =\sim F(MUL7.active = 1\&FADD2.active = 1)$
- $p102 =\sim F(MUL6.active = 1\&FADD2.active = 1)$
- $p108 =\sim F(MUL7.active = 1\&FADD1.active = 1)$

Table 5.4 Textual clustering results for MIPS processor

Cluster index	Size (# Prop)	Base time (s)	Original time (s)	Improved time Verify(s)	Improved time Overhead(s)	Speedup
1	1	0.11	0.11	0.11	0	1.00
2	1	0.12	0.12	0.12	0	1.00
3	1	0.35	0.35	0.35	0	1.00
4	1	0.35	0.35	0.35	0	1.00
5	3	1.28	4.62	2.57	1.53	1.10
6	5	2.75	15.63	6.02	3.34	1.52
7	8	5.56	72.61	15.23	6.55	2.86
8	11	11.30	183.44	26.31	10.57	4.04
9	11	17.72	249.19	40.57	12.03	3.80
10	10	30.58	456.97	48.44	12.38	5.33
11	1	0.30	0.30	0.30	0.00	1.00
12	3	1.28	4.65	2.00	1.57	1.22
13	5	2.69	17.78	7.82	3.40	1.47
14	8	5.00	77.04	21.91	6.62	2.45
15	11	4.7	100.19	34.17	9.16	2.18
16	3	1.55	4.77	1.22	1.62	1.44
17	5	2.73	18.17	4.28	3.42	2.00
18	2	1.21	1.84	1.42	0.97	0.85
19	17	15.67	269.53	6.18	16.45	7.39
20	13	7.74	127.90	4.49	11.24	5.78
21	4	2.04	7.78	1.13	2.38	1.77
22	7	59.54	418.22	13.22	9.27	5.82
23	7	10.34	69.91	9.16	5.82	3.17
24	3	29.07	61.34	0.32	3.39	2.76
25	4	95.77	288.45	0.61	5.66	3.77
26	6	21.63	104.19	0.85	5.98	4.42
27	4	4.02	29.97	4.24	3.05	3.00
28	2	10.46	10.50	0.15	1.72	1.70
29	5	18.64	81.71	0.83	5.08	4.09
30	5	21.07	78.80	6.61	5.22	3.04
31	3	22.25	44.91	0.46	3.05	2.61
32	1	28.78	28.78	28.78	0	1.00
Average	5.34	13.64	88.44	9.07	4.74	3.72

Table 5.5 shows the verification results for an influence-based cluster consisting of the above 7 properties. In this case, the overall speedup using the clustering and learning techniques is 4.52x.

Table 5.5 Verification results for an influence-based cluster

Property	Type	Bound	Size	Forward	Original (s)	New(s)	Speedup
$p111$	Interaction	10	592239	–	54.80	63.40	0.87
$p104$	Interaction	9	526687	66773	25.98	0.22	118.09
$p110$	Interaction	10	592239	70975	54.26	0.25	217.04
$p103$	Interaction	9	526687	66773	25.83	0.22	117.41
$p109$	Interaction	10	592239	70975	49.16	0.25	196.64
$p102$	Interaction	9	526687	66773	33.27	0.22	151.23
$p108$	Interaction	10	592239	70975	49.74	0.26	191.31
Average	–	9.57	564145	–	41.86	9.26	4.52

Table 5.6 shows the verification results using influence-based clustering for all 27 clusters. The details of the first cluster (row 1) is shown in Table 5.5. The overall speedup is 4.30x.

5.6.1.4 Intersection-Based Clustering

Intersection-based clustering is intuitive and easy to be implemented since it does not require any prior knowledge about the structure of the graph model or the format of the properties. It only uses the mapping of the variables for name substitution and the intersection between the CNFs. Due to use of data structure *hashmap*, the intersection time is linear to the size of the CNF file. The following properties are grouped as a cluster using a threshold for the intersection as 90 %.

- $p50 \ =\sim F(MUL1.active = 1\&MUL7.active = 1)$
- $p62 \ =\sim F(MUL2.active = 1\&MUL7.active = 1)$
- $p73 \ =\sim F(MUL3.active = 1\&MUL7.active = 1)$
- $p83 \ =\sim F(MUL4.active = 1\&MUL7.active = 1)$
- $p92 \ =\sim F(MUL5.active = 1\&MUL7.active = 1)$
- $p100 =\sim F(MUL6.active = 1\&MUL7.active = 1)$

Table 5.7 presents the verification details for the above cluster using $p50$ as the base property. The speedup for this cluster is 5.96x.

Table 5.8 presents the intersection clustering verification for all the 171 properties. The details of the 9th cluster are shown in Table 5.7. The overall speedup using the clustering and learning techniques is 5.90x.

Table 5.6 Influence-based clustering results for MIPS processor

Cluster index	Size (# Prop)	Base time (s)	Original time (s)	Improved time		Speedup
				Verify(s)	Overhead(s)	
1	7	54.80	238.24	1.42	8.60	4.52
2	15	55.31	874.07	38.38	19.18	8.23
3	6	0.07	72.30	83.01	5.18	0.82
4	11	21.22	173.93	4.81	10.44	5.35
5	17	25.94	570.77	48.36	19.22	6.38
6	7	10.49	62.39	4.89	5.92	3.42
7	14	8.98	188.18	22.39	12.64	4.48
8	6	9.41	19.76	0.86	4.45	1.98
9	17	11.76	192.75	20.44	14.62	4.37
10	7	4.06	44.33	10.76	5.29	2.41
11	8	4.39	49.22	7.26	5.91	3.05
12	4	24.29	49.00	0.90	3.92	2.52
13	6	15.54	73.46	0.72	5.74	4.05
14	5	2.19	8.99	2.25	2.86	1.53
15	6	2.18	12.60	1.42	3.44	2.10
16	7	12.98	84.54	8.65	6.45	3.47
17	6	19.49	63.14	1.01	5.59	3.17
18	2	4.58	1.83	0.11	1.27	1.08
19	1	2.31	2.31	2.31	0.00	1.00
20	9	10.57	107.50	16.85	8.14	3.32
21	2	1.54	0.35	0.08	0.74	0.80
22	3	18.24	26.83	0.43	2.90	2.09
23	1	0.35	0.35	0.35	0.00	1.00
24	1	0.30	0.30	0.30	0.00	1.00
25	1	1.21	1.21	1.21	0.00	1.00
26	1	0.12	0.12	0.12	0.00	1.00
27	1	0.12	0.12	0.12	0.00	1.00
Average	6.33	11.94	108.1	10.35	5.65	4.30

5.6.1.5 Comparison of Clustering Techniques

Table 5.9 compares the four clustering techniques. The first row shows the proposed clustering methods. The second row indicates the number of clusters using the respective clustering methods, and the third row shows the corresponding clustering time (in seconds). The fourth row presents the test generation time for the base property. Similar to the previous tables, the original time refers to traditional (no clustering)

Table 5.7 Verification results for an intersection-based cluster

Property	Type	Bound	Size	Forward	Original (s)	New(s)	Speedup
$p50$	Interaction	11	657806	–	80.91	89.41	0.90
$p62$	Interaction	11	657806	91548	95.87	0.58	165.29
$p73$	Interaction	11	657806	91548	95.75	0.46	208.15
$p83$	Interaction	11	657806	91548	96.29	0.59	163.20
$p92$	Interaction	11	657806	91548	96.83	0.59	164.12
$p100$	Interaction	11	657806	91548	83.99	0.59	142.36
Average	–	11	657806	–	91.61	15.37	5.96

Table 5.8 Intersection-based clustering results for MIPS processor

Cluster index	Size (# Prop)	Base time (s)	Original time (s)	Improved time		Speedup
				Verify(s)	Overhead(s)	
1	4	1.22	4.08	0.27	1.75	1.64
2	13	1.82	28.44	1.31	7.48	2.85
3	17	15.68	266.61	2.76	16.99	7.97
4	17	7.72	147.75	1.80	14.51	6.47
5	17	3.65	66.50	2.00	11.96	3.98
6	14	26.19	383.10	2.28	15.91	9.22
7	13	60.61	691.41	2.68	16.58	9.42
8	17	8.51	172.23	3.10	14.20	7.00
9	6	80.91	468.73	2.81	8.50	5.96
10	17	20.57	323.98	2.73	16.71	8.61
11	12	13.01	120.28	2.17	10.26	5.25
12	4	4.74	15.29	0.41	2.88	2.49
13	2	0.11	0.11	0.04	0.30	0.49
14	3	0.35	0.65	0.16	0.89	0.71
15	13	18.91	249.84	2.40	13.29	7.77
16	1	30.63	30.63	30.63	0	1
17	1	29.54	29.54	29.54	0	1
Average	10	19.07	176.42	5.12	8.95	5.90

verification time for all the properties excluding the base property. The sixth row presents the verification time for all the properties except the base property using the respective clustering method. The speedup is computed using the formula (Base time + Original time)/(Clustering time + Base time + Improved time). For the first three clustering methods, the clustering is very fast and the associated cost (time) is negligible. However, for the intersection-based clustering, the intersection

Table 5.9 Property clustering and verification for MIPS processor

Methods	Structure	Textual	Influence	Intersection
Cluster number	16	32	27	17
Clustering Time	0.24	0.06	0.22	187.90
Base Time	169.09	436.60	322.44	324.18
Original time	3105.98	2830.13	2918.56	2999.16
Improve time	788.09	442.53	431.92	239.28
Speedup	3.42	3.72	4.33	5.90 (4.42)

time is longer compared to other three methods and is not negligible. Therefore, for intersection-based clustering, the speedup values are provided for both scenarios–without considering clustering time (the first number) as well as with the clustering time (the number in parenthesis).

It is important to note that, when the conflict clause based learning is used, intersection-based clustering is most beneficial for reducing overall test generation time. However, the clustering overhead is much more than other strategies. When a large number of complex properties are involved, the intersection overhead may become prohibitively large. In such cases, influence-based clustering is most beneficial. Interestingly, textual clustering consumes least amount of clustering time but generates better results than structure-based clustering. When detailed information about the design is not available, textual clustering is most beneficial.

5.6.2 A Stock Exchange System

This section presents the test generation results of the online stock exchange system (OSES) described in Sect. 2.4.4. The specification is used to generate 51 properties based on the fault model. The clustering methods discussed in Sect. 5.4 are applied on all the properties to generate the tests.

Table 5.10 presents the test generation results using structure-based clustering for all the 51 properties, with a 2.26x overall speedup. Table 5.11 presents the test generation results using textual clustering for all the 51 properties, with a 2.33x overall speedup. Table 5.12 presents the test generation results using influence-based clustering for all the 51 properties, with a 2.44x overall speedup. Table 5.13 presents the test generation results using intersection-based clustering for all the 51 properties. The overall speedup without considering clustering overhead is 2.84x. If the clustering overhead is considered, the overall speedup is 2.69x.

Table 5.14 summarizes the results using both the conflict clause based learning and four clustering methods, where 2–3 times improvement is achieved. It is important to note that the results for OSES are consistent with the results for MIPS in Table 5.9.

Table 5.10 Structure-based clustering results for OSES

Cluster index	Size (# Prop)	Base time (s)	Original time (s)	Improved time		Speedup
				Verify(s)	Overhead(s)	
1	2	4.48	3.72	0.63	0.97	1.35
2	4	6.14	45.5	13.13	1.92	2.44
3	2	1.76	2.03	0.60	0.97	1.14
4	4	59.56	160.99	15.16	1.90	2.88
5	2	9.34	11.09	19.58	0.98	0.68
6	4	10.74	123.79	5.97	1.95	7.21
7	2	0.40	0.32	0.25	0.97	0.44
8	4	96.44	150.45	31.11	1.91	1.91
9	2	6.62	7.40	0.71	1.13	1.66
10	4	10.08	82.61	48.02	2.26	1.54
11	2	3.36	4.69	1.22	1.13	1.41
12	4	101.16	154.62	38.48	2.22	1.80
13	2	29.55	36.5	2.90	1.14	1.97
14	4	106.51	168.30	2.24	2.24	1.95
15	2	0.21	0.20	19.34	1.14	0.02
16	4	95.91	588.49	120.00	2.26	3.14
17	2	18.91	15.53	1.16	0.82	1.65
18	1	0.88	0.88	0.88	0.00	1.00
Average	2.83	31.23	86.51	19.51	1.44	2.26

Table 5.11 Textual clustering results for OSES

Cluster index	Size (# Prop)	Base time (s)	Original time (s)	Improved time		Speedup
				Verify(s)	Overhead(s)	
1	1	0.68	0.68	0.68	0.00	1.00
2	2	15.55	18.86	7.73	0.81	1.43
3	9	4.33	196.59	60.88	4.26	2.89
4	8	60.25	135.37	36.83	3.80	1.94
5	1	33.57	33.57	33.57	0.00	1.00
6	6	11.62	246.23	2.05	2.86	15.60
7	9	6.44	469.61	130.68	5.01	3.35
8	8	10.61	155.82	95.90	4.50	1.50
9	7	0.21	760.38	390.69	3.91	1.93
Average	5.67	15.87	224.12	84.33	2.79	2.33

Table 5.12 Influence-based clustering results for OSES

Cluster index	Size (# Prop)	Base time (s)	Original time (s)	Improved time		Speedup
				Verify(s)	Overhead(s)	
1	5	22.97	147.84	50.48	2.75	2.24
2	8	10.10	369.97	120.27	4.40	2.82
3	3	36.62	59.65	38.78	1.69	1.26
4	5	10.66	135.98	11.37	2.37	6.01
5	4	0.32	4.00	3.28	1.90	0.78
6	1	93.48	93.48	93.48	0	1.00
7	7	28.89	629.39	132.41	3.89	3.98
8	2	12.87	9.85	0.37	0.98	1.58
9	6	14.23	302.63	115.31	2.83	2.40
10	7	34.66	261.80	69.81	3.34	2.75
11	2	15.87	18.98	7.63	0.81	1.43
12	1	0.75	0.75	0.75	0	1.00
Average	4.25	23.12	169.50	53.65	2.08	2.44

As Table 5.14 shows, when conflict clause based learning is used, intersection-based clustering is most beneficial for reducing overall test generation time. However, when clustering overhead is prohibitively large, influence-based clustering is beneficial. Similarly, when detailed information about the design is not available, textual clustering is the best choice.

On two case studies (MIPS and OSES), the approach using efficient integration of property clustering and conflict clause forwarding based learning techniques demonstrated a 3–5 times improvement in overall test generation time.

5.7 Chapter Summary

Directed test vectors can reduce overall validation effort since fewer tests can obtain the same coverage goal compared to the random tests. The applicability of the existing approaches for directed test generation is limited due to capacity restrictions of the automated tools. This chapter addresses the test generation complexity by clustering similar properties and exploiting the commonalities between them. To enable knowledge sharing across multiple properties, a number of conceptually simple but extremely effective techniques have been developed, including name substitution and selective forwarding of learned conflict clauses. The experimental results using both hardware and software designs demonstrated the effectiveness of the combination of the clustering and learning techniques.

Table 5.13 Intersection-based clustering results for OSES

Cluster index	Size (# Prop)	Base time (s)	Original time (s)	Improved time Verify(s)	Improved time Overhead(s)	Speedup
1	7	4.84	53.91	16.64	3.31	2.37
2	3	10.93	94.79	6.2	1.46	5.69
3	2	7.13	56.72	5.81	0.98	4.59
4	2	35.32	68.96	24.97	0.98	1.70
5	3	5.06	20.60	22.56	1.45	0.88
6	7	84.18	243.60	22.78	3.30	2.97
7	8	6.54	393.75	147.45	4.53	2.53
8	6	3.37	98.46	42.39	3.32	2.07
9	3	29.45	68.71	19.07	1.74	1.95
10	3	107.27	457.52	39.59	1.69	3.80
11	4	0.20	247.46	62.83	2.24	3.79
12	2	18.74	15.35	1.17	0.82	1.64
13	1	0.7	0.7	0.7	0	1.00
Average	3.92	24.13	140.04	31.70	1.99	2.84

Table 5.14 Property clustering and verification for OSES

Methods	Structure	Textual	Influence	Intersection
Cluster number	18	9	12	13
Clustering time	0.05	0.01	0.05	42.77
Base Time	562.05	142.81	277.42	313.73
Original time	1557.11	2017.11	2034.05	1820.53
Improve time	377.15	784.16	668.72	437.98
Speedup	2.26	2.33	2.44	2.84 (2.69)

References

1. Mishra P, Chen M (2009) Efficient techniques for directed test generation using incremental satisfiability. In: Proceedings of international conference of VLSI design, pp 65–70
2. Chen M, Mishra P (2010) Functional test generation using efficient property clustering and learning techniques. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 29(3):396–404
3. Bryant R (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput 35(8):677–691
4. Amla N, Du X, Kuehlmann A, Kurshan R, McMillan K (2005) SATIRE: an analysis of SAT-based model checking techniques in an industrial environment. In: Proceedings of conference on correct hardware design and verification methods (CHARME), pp 254–268

5. Moskewicz MW, Madigan CF, Zhao Y, Zhang L (2001) Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th design automation conference (DAC), pp 530–535
6. Jin H, Somenzi F (2004) An incremental algorithm to check satisfiability for bounded model checking. In: Proceedings of BMC, pp 51–65
7. Whittemore J, Kim J, Sakallah K (2001) SATIRE: a new incremental satisfiability engine. In: Proceedings of design automation conference (DAC), pp 542–545
8. Zhang L, Prasad M, Hsiao M (2004) SATIRE: incremental deductive and inductive reasoning for SAT-based bounded model checking. In: Proceedings of international conference on computer-aided design (ICCAD), pp 502–509
9. Strichman O (2001) Pruning techniques for the SAT-based bounded model checking problem. In: Proceedings of correct hardware design and verification methods (CHARME), pp 58–70
10. Kim J, Whittemore J, Marques-Silva J, Sakallah K (2000) On solving stack-based incremental satisfiability problems. In: Proceedings of international conference on computer design (ICCD), pp 379–382
11. Benedetti M, Bernardini S (2004) Incremental compilation-to-SAT procedures. In: Proceedings of international conference on theory and applications of satisfiability testing (SAT), pp 46–58
12. Hooker J (1993) Solving the incremental satisfiability problem. J Logic Program 15(12):177–186
13. Chandrasekar K, Hsiao MS (2005) Integration of learning techniques into incremental satisfiability for efficient path-delay fault test generation. In: Proceedings of design automation and test in Europe (DATE), pp 1002–1007
14. Marques-Silva J, Sakallah K (1999) Grasp: a search algorithm for propositional satisfiability. IEEE Trans Comput (TC) 48(5):506–521
15. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5(7):394–397
16. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7(3):201–215
17. Zhang L, Madigan CF, Moskewicz MH, Malik S (2001) Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of international conference on computer-aided design (ICCAD), pp 279–285
18. Princeton University (2007) zChaff. http://www.princeton.edu/~chaff/zchaff.html
19. FBK-irst and CMU (2006) NuSMV. http://nusmv.irst.itc.it/

Chapter 6
Decision Ordering Based Learning Techniques

6.1 Introduction

For SAT-based BMC, the performance of efficient test generation is determined by how to quickly get satisfying assignments for SAT instances. Since SAT problem is NP-complete, various heuristic methods and tools [9, 6] have been proposed to improve the SAT searching time. Decision ordering [7] plays an important role during the search because different decision ordering implies different decision trees as well as different search paths which strongly affect the search time. Existing decision ordering methods focus on exploiting the useful information of general SAT problem with a single SAT instance. Most of them are based on the statistics of SAT instances without considering any other learning information. For test generation, a design may have various properties and generally model checking techniques will check each of them individually. For a given design, similar properties describe correlated functional scenarios. Therefore the respective counterexamples are expected to have a significant overlap which can be used for sharing learning. Furthermore, even for a single SAT instance, the result of the local search can also benefit the global search. The method proposed in this chapter exploits the learning from decision ordering in the context of test generation involving one or more properties of a design. This chapter mainly focuses on the following three techniques for efficient SAT-based BMC test generation: i) investigates the decision ordering based learning for a single SAT instance (i.e., *intra-property learning*); ii) applies the decision ordering based learning between similar SAT instances (i.e., *inter-property learning*); and iii) exploits the relation between the decision ordering and conflict clause forwarding based methods.

The rest of the chapter is organized as follows. Section 6.2 presents the related work on decision ordering based heuristics. Section 6.3 describes the property learning techniques based on decision ordering. Section 6.4 proposes the test generation methodology using efficient decision ordering heuristics. Section 6.5 presents the experimental results. Finally, Sect. 6.6 summarizes the chapter.

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_6,

© Springer Science+Business Media New York 2013

6.2 Related Work

Different variable ordering will lead to different search trees, therefore branching heuristics can improve the SAT searching performance significantly [7]. As a popular SAT solver, zChaff uses the variable state independent decaying sum (VSIDS) heuristic [9]. This heuristic contains two parts: i) the static part collects the statistics of the conjunctive normal form (CNF) literals prior to SAT solving and sets the initial decision ordering, and ii) during the SAT solving, the dynamic part periodically updates the priority based on conflict clauses. Although the above general-purpose heuristics are promising for propositional formulas, they neglect some unique information about BMC. Strichman [10] exploited the characteristics of the BMC formulas for a variety of optimizations including decision ordering. When the bound is unknown, SAT-based BMC needs to increase the unrolling depth one-by-one until finding a counterexample. Wang et al. [13] analyzed the correlation among different SAT instances of a property. They used the *unsatisfiable core* of previously checked SAT instances to guide the variable ordering for the current SAT instance.

Most of the existing approaches exploit variable ordering to improve the SAT solving time involving only one property (one SAT instance or several correlated SAT instances with different bounds). The technique developed by Chen et al. [4, 3] was the first attempt to use both decision and conflict clauses to reduce the BMC-based test generation time for a cluster of similar SAT instances. The following sections will describe this approach in detail.

6.3 Decision Ordering Based Learnings

Decision ordering plays an important role during the SAT search. It indicates which variable will be selected first and which value (true or false) will be first assigned to this variable. Similar to BDD-based methods [1], variable ordering determines the performance of the SAT solving time. In the VSDIS heuristics implementation of zChaff, each literal l is associated with a $zchaff_score(l)$ which is used for decision ordering at *decide_next_branch()* (see Algorithm 3 in Sect. 5.3.1). Initially the score is equal to the literal count in corresponding CNF file. During the SAT solving, the literal score will be updated in periodic function after a certain number of backtracks. The calculation of the new literal score is as follows:

$$chaff_score(l) = chaff_score(l)/2 + lits_in_new_confs(l) \tag{6.1}$$

where $lits_in_new_confs(l)$ is the number of newly added conflict clauses which contain literal l since last update.

Similar properties usually have similar counterexamples, which indicates that they may have similar Boolean constraints during the test generation. Consequently,

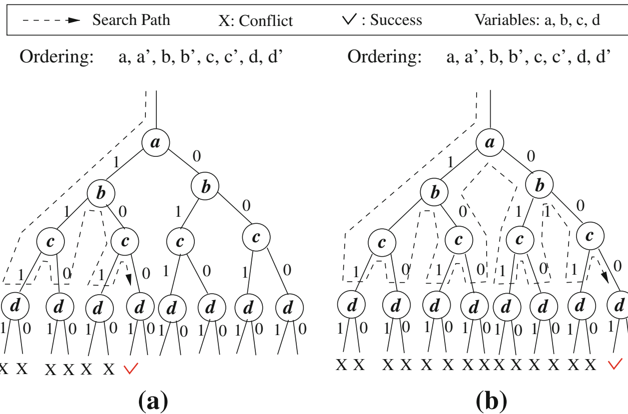


Fig. 6.1 Two examples of SAT search. **a** Partial view of the first example. **b** Partial view of the second example

the generated SAT instances should have a large overlap in CNF clauses and can be clustered to share the learning. This section presents the decision ordering heuristics which will be incorporated in the test generation approaches presented in Sect. 6.4.

6.3.1 Motivation

As discussed in Sect. 5.3.1, in SAT searching, the most time-consuming parts are Boolean constraint propagation (BCP) and long distance backtracking. They are indicated by the implication number and conflict clause number which represent the successful decision ratio and backtrack number respectively. To reduce the test generation of a cluster of similar properties, one feasible way is to reduce the number of implications and conflict clauses of unchecked properties by incorporating the learned decision ordering knowledge from previously checked properties.

Assuming that we have two similar properties, both properties will have a large overlap on CNF clauses and counterexample assignments. Figure 6.1 shows the partial views of search trees and search paths of the two properties. The search paths are formed according to the decision ordering (shown on top of the search trees). For each variable *v* in the ordering, there are two literals (*v* means *v=1* and *v'* means *v=0*). As shown in Fig. 6.1a, there are six conflicts encountered. The search stops after finding a satisfying assignment $a = 1, b = 0, c = 0, d = 1$ in this scenario. In Fig. 6.1b, the search will be successful only when $a = 0, b = 0, c = 0, d = 1$ after encountering 14 conflicts. Therefore the search for the second example will be more time-consuming because of more backtracks.

Due to the large overlap in the assignment of counterexamples, the result of previously checked properties can be used as a learning for unchecked properties. For example, in Fig. 6.1, the result of the first example strongly indicates the assignment

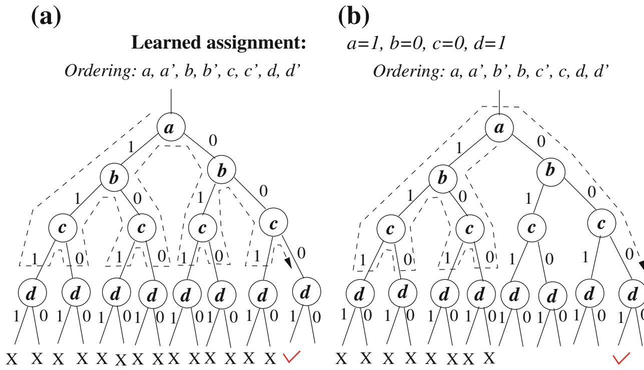


Fig. 6.2 A scenario where bit-value ordering works. **a** Without bit-value ordering. **b** With bit-value ordering

of the second example because of the satisfying assignment intersection $b = 0$, $c = 0, d = 1$. If the second example uses the decision ordering based on the variable assignments in the first example, the searching time of the second example can be drastically reduced. The examples will be described in the following sections.

6.3.2 Bit Value Ordering

Similar properties generally have a large intersection on both corresponding CNF clauses and counterexample assignments. This indicates that the satisfying assignment of checked SAT instances contain rich decision ordering knowledge for unchecked satisfiable SAT instance. In SAT search, incorrect value selection for Boolean variables will cause conflicts which will result in backtracks to remove the reason for the conflicts. A good decision ordering can mostly avoid such faulty assignments. Unlike pruning the search tree using conflict clause forwarding [8], bit value ordering changes the *search path*. By setting the *bit priority* (choose 0 or 1 first) for each variable using the knowledge of previous property checking, the length of the search path can be reduced.

Figure 6.2 shows an example where bit-value ordering works. As shown in Fig. 6.1a, we can get a satisfying assignment $a = 1$, $b = 0$, $c = 0$ and $d = 1$. This assignment can be used to change the bit-value ordering of the second example. That means, when node b is encountered, the search chooses $b = 0$ first in its search path. The same rule also applies on other nodes. Applying such heuristic in Fig. 6.2b, there are only 8 conflicts encountered compared to 14 conflicts in Fig. 6.2a. In addition, the search path is also shortened. Therefore, the searching time is reduced.

It is important to note that the bit-value ordering itself is not always helpful for the SAT searching. For example in Fig. 6.3, $a = 1$, $b = 1$, $c = 0$, $d = 1$ is the only

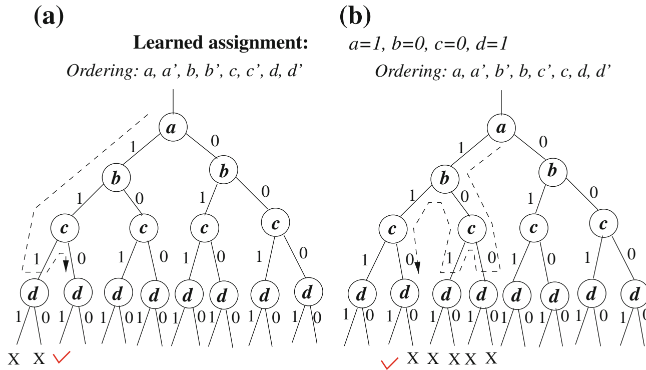


Fig. 6.3 A scenario where bit-value ordering fails. **a** Without bit-value ordering. **b** With bit-value ordering

satisfying assignment in the given scenario. The searching in Fig. 6.3a without bit value ordering is faster than the searching in Fig. 6.3b because of less conflicts. If the learning assignment in Fig. 6.3 was $a = 0$, $b = 1$, $c = 0$ and $d = 1$, the searching performance will be much worse than the search in Fig. 6.3b. Clearly, in the search tree, the high-level variables (e.g., node a) strongly affect the performance of the searching if they are not consistent with learned bit-value ordering.

6.3.3 Variable Ordering

Although bit-value ordering is promising in general, there are still a lot of conflicts encountered during the search. According to the example shown in Fig. 6.3, if high-level nodes (e.g., node a) make the wrong decision, the search path will be lengthened due to the long distance backtracking. To reduce the searching time, it is necessary to restrict the conflict detection and reasoning in a small area.

Efficient combination of variable ordering and bit-value ordering is very promising. As shown in Fig. 6.4b, the search time is better than that in Fig. 6.4a due to a shorter search path and less conflicts. This improvement is due to the enhancement of the priority of variables b and c. Since a is the variable with different values between the two satisfying assignments shown in Fig. 6.1, lowering down the priority of such variables (those with different values between two CNFs) can efficiently avoid long distance backtracks. Generally, before SAT solving, it is hard to figure out the difference between two satisfying CNF variable assignments. However, based on the value assignment statistics of the checked properties, the variable ordering can be constructed. For a variable with the lower assignment value variation, which

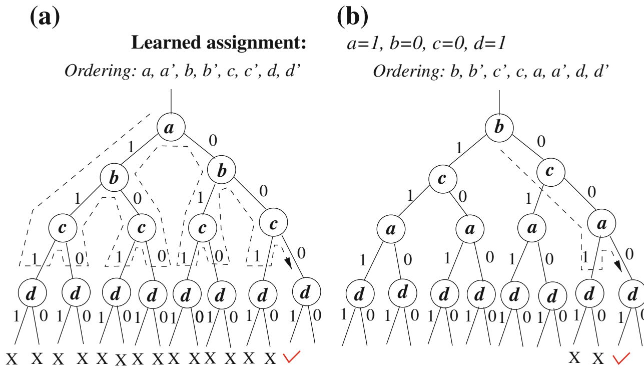


Fig. 6.4 An example of bit-value and variable ordering. **a** Without any learning. **b** With bit-value and variable ordering

indicates high chance of the same value, its priority is enhanced by increasing the score of its two literals.

6.3.4 Hybrid Learning from Conflict Clauses and Decision Ordering

Conflict clause is promising to avoid repeated conflicts during the SAT searching. Therefore it can be used as a learning during the test generation. In essence, conflict clause forwarding can be used to prune the decision tree and can be utilized as a complementary approach for the decision ordering techniques proposed in Sect. 6.3.2 and Sect. 6.3.3. For two similar SAT instances, if the conflict clauses of the checked SAT instance can be forwarded to the unchecked one, it will reduce the conflicts, thus further shortening the search path.

Figure 6.5a shows application of bit-value ordering on the example shown in Fig. 6.1b. There are eight conflicts during the SAT search in this case. Let us assume the conflict clauses generated from Fig. 6.1a can be forwarded to the CNF clauses of Fig. 6.1b. The generated six conflict clauses are as follows:

$$\left.\begin{array}{l}(a' \vee b' \vee c' \vee d')\\(a' \vee b' \vee c' \vee d)\\(a' \vee b' \vee c \vee d')\\(a' \vee b' \vee c \vee d)\end{array}\right\} \Rightarrow (a' \vee b')$$
$$\left.\begin{array}{l}(a' \vee b \vee c' \vee d')\\(a' \vee b \vee c' \vee d)\end{array}\right\} \Rightarrow (a' \vee b \vee c') \qquad (6.2)$$

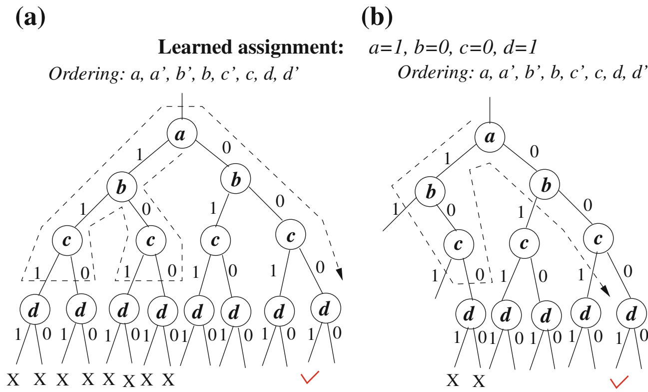


Fig. 6.5 An example of conflict clause based decision ordering. **a** With bit-value ordering. **b** Bit-value ordering + conflict clauses

Equation 6.2 shows the resolution of the forwarded conflict clauses. Based on the result, we can prune the search tree as shown in Fig. 6.5b. It indicates that there are only two conflicts by applying the bit value ordering on the pruned search tree. Therefore the test generation time can be significantly reduced. For the example shown in Fig. 6.4b, the conflict clause forwarding is not beneficial since the search does not traverse the pruned part of the decision tree. However, in general cases, the conflict clause forwarding can further improve the performance of the decision ordering based methods.

6.4 Test Generation Using Decision Ordering Techniques

For test generation using model checking, each property is described as a negation of a desired system behavior so that a model checker can produce a counterexample. Since SAT-based BMC is adopted for test generation in this chapter, we assume that the bounds can be predetermined and the generated SAT instances are satisfiable. The goal of the test generation for the property with a known bound is to figure out a satisfying assignment for this SAT instance.

To reduce the overall test generation effort, Sect. 6.4.1 applies the learning based on the decision ordering for test generation of a single property, and Sect. 6.4.2 presents an algorithm which shares learning from the decision ordering among a cluster of similar properties.

6.4.1 Test Generation for a Single Property

When checking a base property using property clustering techniques, or when checking only a single property, current methods solve the SAT instance alone since there is no source of learning. Therefore it is time-consuming and it can be a major bottleneck of the clustering-based test generation.

During test generation, if the bound of a property is increased by one, the test generation time will be drastically increased. According to Strichman's observation [10], the reason for time-consuming search is due to the long distance backtracks. Since a large set of clauses that belongs to different distant cycles is being satisfied independently (locally), Strichman found that there are three typical scenarios which can cause the conflicts:

- Distant cycles are being satisfied independently until they collide each other with assignment conflict.
- Some cycle assignment collides with the constraints imposed by the initial state.
- Some cycle assignment collides with the constraints imposed by the negation of the specified property.

The resolution of such conflicts needs to cancel a large number of variable assignments between the conflicting cycles. Especially for the SAT instance with large bound, the cost of non-chronological backtracking is still huge since large bound indicates huge number of clauses and variables.

To alleviate long distance backtracks during test generation, learning is required to guide the SAT search. Conflict clause is a promising learning that can prune the decision tree. However, in a SAT instance with large bound, deriving a conflict clause is costly due to large interleaving of irrelevant variables during the SAT search. Furthermore, a large set of CNF clauses is likely to generate a large number of conflict clauses which can affect the search performance. Therefore, if we can get conflict clauses from a smaller SAT instance, then the average cost of conflict clause generation will be reduced. As an alternative, decision ordering can be used as learning. Since the SAT instance is assumed to be satisfiable, each segment[1] of the CNF clauses should be satisfiable. The searching time for a segment is much shorter than the original SAT instance. Although a segment cannot reflect the global view of the system, if the satisfying assignment of the segment is consistent with the partial variable assignment of the original SAT instance, it will be helpful to reduce the overall test generation time of the original SAT instance.

6.4.1.1 Heuristic Implementation

The basic idea of the heuristic for test generation involving a single property is to use the learnings from a small part of the SAT instance to guide the search of the whole

[1] A CNF SAT instance can be viewed as a union of a set of segments where each segment consists of a set of CNF clauses.

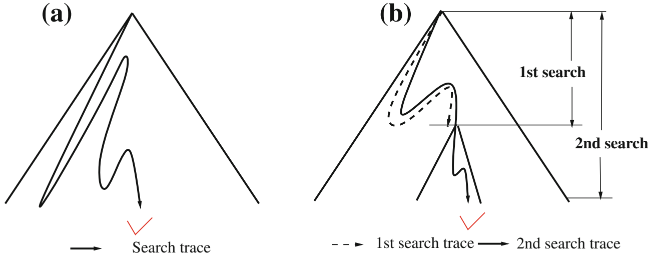


Fig. 6.6 Learning techniques for a single property. **a** A search without any learnings. **b** A search with two kinds of learnings

SAT instance. By dividing the SAT instance into two segments, we can get the first segment which contains the initial state constraints and the second segment which contains property constraints. After checking any one of them, we can get the partial variable assignments which can be used as decision ordering learning, and we can get the conflict clauses which can be forwarded to the original property according to Theorem 5.2

Figure 6.6 demonstrates an example of using such learnings. In Fig. 6.6b, we first check one part of the SAT instance and get the corresponding learnings. Then during the checking of the whole SAT instance, under the guidance of the learned knowledge, the overall search path is shortened compared to Fig. 6.6a.

The decision ordering heuristics implementation uses an array var[$sz + 1$] (sz is the largest variable number for CNFs) to indicate the satisfying assignment result of the first search. Each element of the array var[i] ($0 < i \leq sz$) has three values: 1 means that the ith variable is assigned with 1; 0 means that the ith variable is assigned with 0; and -1 implies that the variable is not assigned during the first search. So during the second search, the literal score is calculated using the following formula where $\max(v_i) = \text{MAX}(\text{chaff_score}(v_i), \text{chaff_score}(v_i'))+1$

$$\text{score}(l_i) = \begin{cases} \max(v_i) & (\text{var}[i] == 1 \;\&\; l_i = v_i) \\ & \text{or} \quad (\text{var}[i] == 0 \;\&\; l_i = v_i') \\ \text{chaff_score}(v_i) & \text{otherwise} \end{cases} \tag{6.3}$$

6.4.1.2 Test Generation Using Intra-Property Learning

Algorithm 1 describes the test generation procedure for a single property using learnings from some part of the SAT instance corresponding to the original property. Step 1 initializes all the elements of *var* with -1. Step 2 generates the CNF clauses for the property p. After dividing the CNF into two parts in step 3, step 4 solves the clauses in any one part and derives the learning in the form of decision ordering and

conflict clauses. Step 5 updates the *var*. Finally, step 6 uses the learning to guide the test generation of the original property. For intra-property learning, a SAT instance is equally divided into two segments.

Algorithm 1: Test Generation for a Single Property

Input: i) Formal model of the design, D
ii) Property p with bound b
Output: A test t for p with generated conflict clauses
1. Initialize *var*;
2. $CNF = BMC(D,\ p,\ b)$;
3. Divide CNF into CNF_1 and CNF_2;
4. $(assign, conf_clauses1)$ = SAT(CNF_1 or CNF_2, *var*, NULL);
5. Update *var* using $assign$;
6. $(t, conf_clauses2)$ = SAT(CNF, *var*, $conf_clauses1$);
return t

It is important to note that the heuristic for a single property is based on the assumption that the decision ordering knowledge learned from the first search has a large overlap with a satisfying assignment of the second search. Although the forwarded conflict clauses can prune the decision space, it is still possible that the first search may mislead the second search which will aggravate the overall searching time. Since we halve the SAT instance and each part can be checked individually, for test generation, we use the following three strategies in parallel:

- Directly solve the original SAT instance.
- Solve the first part and use the learnings to solve the original instance.
- Solve the second part and use the learnings to solve the original instance.

Once one of the above methods finds a satisfying assignment, the remaining two processes will be terminated. Therefore, it can be guaranteed that the worst case of the test generation time is the same as directly solving the original SAT instance.

6.4.2 Test Generation for Similar Properties

For similar properties, there exists a large overlap between corresponding counterexamples. Therefore the satisfying assignments of checked properties can be used as a learning for other properties in the cluster. Some of the derived conflict clauses can also be forwarded as learning. This section will discuss how to extract the bit-value ordering and variable ordering based learnings from the checked properties in detail. Also, we will describe an algorithm to utilize the learning based on decision ordering for test generation of a cluster of similar properties.

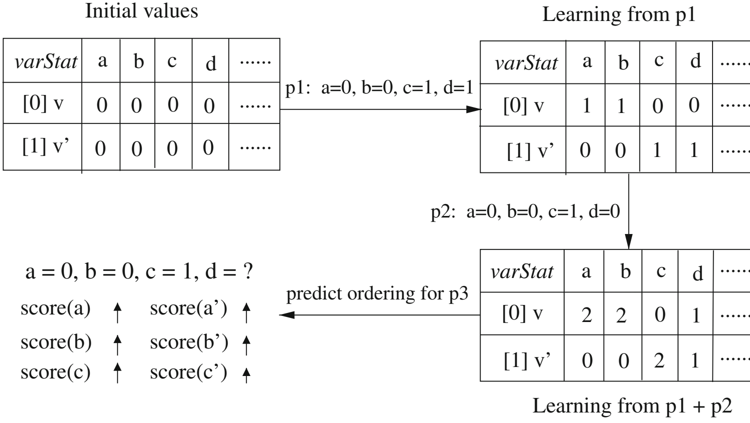


Fig. 6.7 Statistics for two properties

6.4.2.1 Heuristic Implementation

In the heuristic implementation developed by Chen et al. [4, 3], the decision ordering is predicted based on the statistics collected from the checked properties. Let varStat[$sz + 1$][2] (sz is the largest variable number for CNFs) be a 2-D array to keep the count of variable assignments. Initially, varStat[i][0] $=$ varStat[i][1] $= 0$ ($0 < i \leq sz$). varStat will be updated after checking each property. Assuming we are now checking property p_j, if the value of variable v_i in the assignment of the p_j is 0, then varStat[i][0] will be increased by one; otherwise, varStat[i][1] will be increased by one. This updated information of varStat will be utilized when checking property p_{j+1}.

For example, if we have three properties p_1, p_2, and p_3, the statistics after checking p_1 and p_2 are shown in Fig. 6.7. When checking p_3, we can predict its decision ordering based on the collected information saved in varStat. The content of varStat indicates that variables a and b are more likely to be 0, c is more likely to be 1 and d can be assigned any value. Furthermore, varStat implies that the assignments for variable a, b, and c are more consistent than the assignment for variable d. Thus the score of variable a, b, and c will be increased. In other words, they will be searched first as described in Sect. 6.3.3.

Assuming l_i is a literal of v_i, the following equation is used to predict the bit value assignment of v_i when checking p_{j+1}.

$$\text{potential}(l_i) = \begin{cases} 1 & (\text{varStat}[i][1] > \text{varStat}[i][0] \& l_i = v_i) \\ & \text{or}(\text{varStat}[i][1] < \text{varStat}[i][0] \& l_i = v_i') \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

Here, potential$(l_i) = 0$ means that value of l_i is more likely to be 0 in the satisfying assignment of p_{j+1}. For example, in Fig. 6.7, potential$(a) = 0$ which means that a is more likely to be assigned with 0. Let

$$\text{ratio}(i) = \frac{\max(\text{varStat}[i][0], \text{varStat}[i][1]) + 1}{min(\text{varStat}[i][0], \text{varStat}[i][1]) + 1} \tag{6.5}$$

indicates the assignment variance of variable v_i. The larger ratio$_i$ means that the value assignments for variable v_i are more consistent. So it can be used for variable ordering.

The proposed decision ordering heuristic is based on VSIDS [9]. The only difference is that the proposed method incorporates the statistics of previously checked properties. For each literal l_i, score(l_i) is used to describe its priority. Initially, score(l_i) is equal to the literal count of l_i. At the beginning of search as well as periodically decaying time, the literal score will be recalculated using the following equation where $\max(v_i) = \text{MAX}(\text{score}(v_i), \text{score}(v_i'))+1$.

$$\text{score}(l_i) = \begin{cases} \max(v_i) * \text{ratio}(i) & \text{potential}(l_i) = 1 \\ \text{score}(l_i) * \text{ratio}(i) & \text{otherwise} \end{cases} \tag{6.6}$$

6.4.2.2 Test Generation Using Inter-Property Learning

Algorithm 2 describes the test generation methodology for a property cluster. The inputs of the algorithm are a formal model of the design and a cluster of similar properties. The first step initializes varStat which is used to keep statistics of the variable assignments. Step 2 generates the CNF clauses for the base property p_1. Step 3 generates the CNF clauses for other properties. Step 4 derives the test for a base property using the method proposed in Sect. 6.4.1.2. Steps 5–6 generate tests for the remaining properties in the cluster. When solving each property, it needs to update the varStat accordingly in step 5. Step 6 solves the current property using the learnings based on decision ordering. Finally, the algorithm reports all the generated counterexamples (tests). Note that sharing conflict clauses among properties needs to calculate the intersection between the base property and other properties in a cluster. The overhead of intersection calculation is not negligible and can be larger than the test generation time for the non-base properties. Therefore, the hybrid learning is used for the base property only. For the remaining properties, the decision ordering based learning is used. This approach is referred as Hybrid $\rightarrow$ DOL.

Algorithm 2: Test Generation for a Property Cluster

Input: i) Formal model of the design, D
  ii) Property cluster, P, with satisfiable bounds
Output: *Test-suite*
1. Initialize *varStat*;
2. Select the base property p_1 and generate CNF, CNF_1;
for *i is from 2 to the size of cluster P* **do**
  3. Generate CNF, $CNF_i = BMC(D,\ p_i,\ bound_i)$;
end
4. $test_1$ = Algorithm 1(D, p_1, $bound_1$);
Test-suite = $test_1$;
for *i is from 2 to the size of cluster P* **do**
  5. Update *varStat* using $test_{i-1}$;
  6. $test_i$ = SAT(CNF_i, *vatStat*, NULL);
  Test-suite = *Test-suite* $\cup$ $test_i$;
end
return *Test-suite*

6.5 Case Studies

This section presents case studies for efficient test generation using decision ordering as well as conflict clause based heuristics. Section 6.5.1 presents the case studies using intra-property learning for checking individual SAT instances. The collected benchmarks are all pre-generated satisfiable SAT instances. Section 6.5.2 presents two case studies: an implementation of the MIPS architecture and a stock exchange system. Each case study generates a number of properties and groups them into several clusters according to their similarity. By using intra-property learning for the base property of the cluster and inter-property learning for the other properties in the same cluster, the overall test generation time can be reduced. NuSMV [5] was used to generate the CNF clauses (in DIMACS format). A SAT solver based on the modified zChaff [14] is used to incorporate the presented decision ordering heuristic on top of VSDIS. The experimental results are obtained on a Linux PC using 2.0 GHz Intel Core i7 CPU with 3 GB RAM.

6.5.1 Intra-Property Learning

The benchmarks are collected from [11] and [12]. In [11], there are 13 SAT instances given in the benchmark set which are all taken from real industrial hardware designs (contribution of IBM research and Galileo). Four complex instances were chosen from them, because most SAT instances provided in [11] take a short time during

Table 6.1 Test generation results using intra-property learning

SAT	CNF size		zChaff [14]	Intra-property learning			Max
instance	#Variable	#Clause	time (s)	CCF (s)	DOL (s)	Hybrid (s)	speedup
BMC-galileo-8	58074	294821	1.60	0.67	1.18	**0.63**	2.54
BMC-galileo-9	63624	326999	2.84	1.59	1.47	**0.86**	3.30
BMC-ibm-10	59056	323700	13.24	**6.65**	13.60	12.94	1.99
BMC-ibm-11	32109	150027	12.29	8.00	**3.17**	12.44	3.88
VLIW-1	521188	13378461	1332.46	1047.5	2002.25	**474.95**	2.81
VLIW-2	521158	13378532	196.58	**65.84**	215.38	290.93	2.99
VLIW-3	521046	13376161	145.35	150.81	54.70	**51.73**	2.81
VLIW-4	520721	13348117	1104.79	288.39	605.09	**93.50**	11.82
VLIW-5	520770	13380350	858.61	742.23	686.27	**165.59**	5.19
VLIW-6	521192	13378781	209.85	**52.66**	526.58	308.22	3.98
VLIW-7	521147	13378010	**87.42**	189.79	345.48	391.73	1.00
VLIW-8	521179	13378617	1200.68	931.99	441.59	**369.33**	3.25
VLIW-9	521187	13378624	941.25	**180.99**	1476.84	1539.59	5.20
VLIW-10	521182	13378625	1725.82	**896.52**	902.19	1540.70	1.93
PIPE-1	138917	4678756	1327.92	777.63	284.67	**284.60**	4.84
PIPE-2	138918	4678718	1710.66	1767.09	412.45	**412.32**	4.29
PIPE-3	138917	4678757	825.78	374.07	**376.47**	995.45	2.28
PIPE-4	138563	4675040	1080.10	33.00	418.74	**14.49**	77.14
PIPE-5	138918	4678760	626.9	583.25	614.63	**117.92**	5.48
PIPE-6	138795	4671352	**0.43**	0.61	118.72	118.72	1.00
PIPE-7	138918	4678760	1734.26	1013.67	1391.29	**549.90**	3.26
PIPE-8	138711	4688614	113.07	2.50	0.62	**0.62**	190.9
PIPE-9	138916	4676007	6062.27	2664.06	364.36	**359.18**	17.48
PIPE-10	138918	4678760	1430.29	1086.73	**285.67**	986.53	5.09
Statistics	–	–	23238.77	12866.24	11534.41	**9092.87**	2.56

falsification. Apart from these four benchmarks, the benchmarks of two complex designs from [12] are selected as described below. Since this chapter focuses on test generation, the collected SAT instances are all satisfiable.

- **VLIW-SAT-4.0**, buggy VLIW processors with instruction queues and 9-stage pipelines. The processors support advanced loads, predicated execution, branch prediction, and exceptions.
- **PIPE-SAT-1.1**, buggy variants of the pipe benchmarks as presented in [12].

Table 6.1 shows the test generation details using various intra-property learning techniques. The first column shows the names of the SAT instances. The second and third columns indicate the CNF size information including the variable number and clause number. The fourth column indicates the checking time by directly using zChaff [14] without any other learning information. The fifth column shows the checking time using intra-property learning based on conflict clause forwarding, and the sixth column shows the test generation time using our decision ordering based

technique. The seventh column presents the result of our hybrid learning which incorporates both conflict clause forwarding and decision ordering techniques as described in Sect. 6.3.4 and implemented in Algorithm 1. The execution time in columns 5–7 includes the learning time from divided/segmented CNFs. It is important to note that all the learning methods are not always helpful for the test generation. This is because the learning methods may lead the search in a wrong way with more conflicts. However, since different methods are executed on different computers with the same settings, when one machine gets the satisfiable assignment, all the remaining SAT searches on the other machines will be terminated. Therefore the SAT searching time is the minimum searching time among these techniques. Based on such minimum time, the last column indicates the maximum speedup using the following equation:

$$\text{speedup} = \frac{\text{zChaff}}{\text{MIN(zChaff, CCF, DOL, Hybrid)}} \tag{6.7}$$

where zChaff, CCF, DOL, and Hybrid indicate the results of columns 4–7 in Table 6.1, respectively. The last row provides the maximum speedup for the 24 SAT instances using different methods. To explicitly compare the performance of the four methods, in this table, the best results are highlighted using bold fonts.

Table 6.1 shows that the proposed methods can drastically reduce the test generation time (up to 191 times) in most cases (22 out of 24 SAT instances). It can be observed that the conflict clause forwarding based method achieves better performance than zChaff for 20 out of 24 examples. For decision ordering based method, it has a better overall test generation time (11534.41 s) than conflict clause forwarding based method (12866.24 s). However, this is largely due to the test generation improvement in the SAT instance "PIPE-9". Therefore, compared to the conflict clause forwarding based method, decision ordering is not quite a promising intra-property learning for the test generation of a single SAT instance. As shown in the last row of the table, the hybrid method achieves the best overall performance (with a speedup of 2.56 times in total test generation time). The hybrid method outperforms conflict clause forwarding based method in 15 SAT instances and outperforms decision ordering based method in 14 out of 24 SAT instances. Moreover, the hybrid method can achieve the best performance in 14 out of 24 SAT instances. Therefore the hybrid method is the first choice of intra-property learning when there is only one computer available.

Figures 6.8 and 6.9 show the statistics of conflicts and implications for the collected benchmarks using various intra-property learning methods. The figures employ the normalized total conflict clauses and implications generated by the four different methods shown in Table 6.1. The vertical axes of the stacked graphs show the normalized percentage of conflict clauses and implications respectively. It can be found that the results of the percentages of conflict clauses and implications are consistent. In other words, less conflicts will result in less implications. Furthermore, these figures also are consistent with the test generation performance results shown in Table 6.1. For example, in the case of "PIPE-1" in Table 6.1, the order of the test

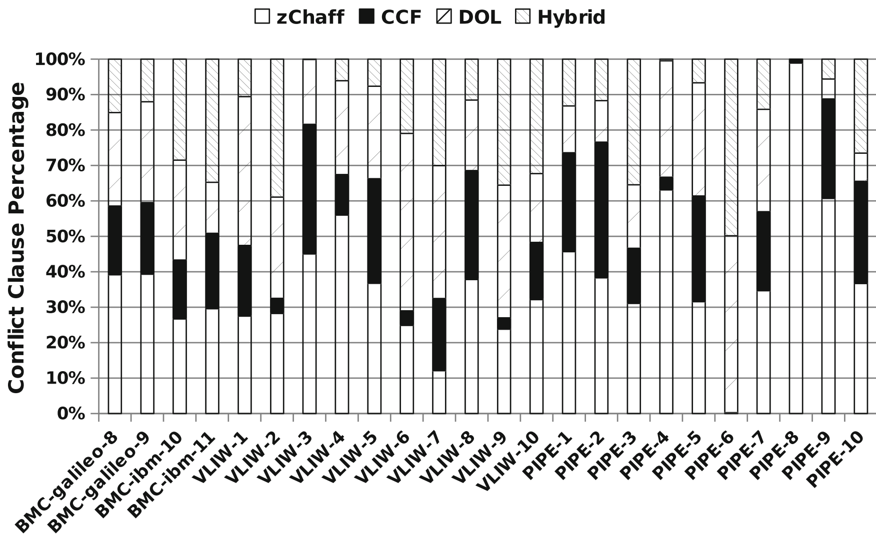


Fig. 6.8 Conflict statistics using various intra-property learnings

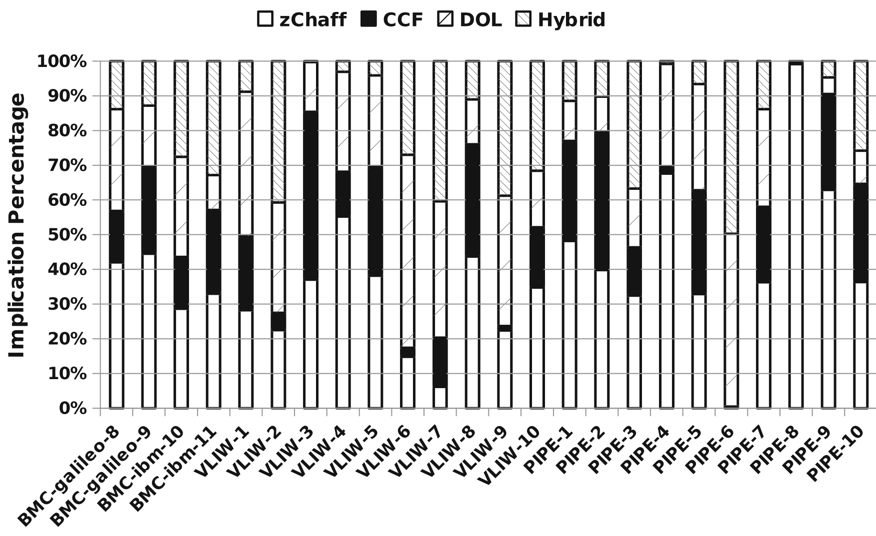


Fig. 6.9 Implication statistics using various intra-property learning

generation time is zChaff $>$ CCF $>$ DOL $>$ Hybrid. In Figs. 6.8 and 6.9, we can find that the percentage of the evaluated metrics is also in the same order. Therefore, if the parallel invocation of methods is not applicable, hybrid learning should be employed.

6.5.2 Inter-Property Learning

6.5.2.1 A MIPS Processor

This case study investigates the MIPS processor presented in Sect. 2.2.1. The MIPS processor consists of five pipeline stages: fetch, decode, execute, memory and write-back. This case study focuses on the validation of pipeline paths (ALU, DIV, FADD and MUL) in the execute stage. Targeting to check whether each pipeline path can give the correct outputs, a set of 16 properties was derived to generate the required directed tests by applying the inter-property learning.

According to the structure similarity proposed in [2] (see Chap. 5), the properties of each pipeline path are grouped together to share the learning. There are four clusters and each cluster has four properties. Table 6.2 shows the test generation results for each cluster. The first column indicates the component under test. The second column shows the properties used for test generation. The third and fourth columns show the CNF size information including the variable number and clause number. The fifth column gives the test generation time using zChaff [14]. The sixth, seventh, and eighth columns present the results using the method proposed in [2]. Since the conflict clause forwarding based method needs to explore the common clauses, it needs to figure out the intersection between SAT instances. Therefore the sixth column gives the intersection time. The seventh column gives the checking time under the learning of conflict clauses. The eighth column gives the speedup over zChaff (speedup = $\frac{\text{zChaff Time}}{\text{Intersection Time}+\text{Checking Time}}$). The ninth and tenth columns give the test generation result only using our decision ordering based learnings. They indicate both the result of test generation time and speedup over zChaff. It is important to note that DOL does not consider how to reduce the test generation time for the base property. To further reduce the overall test generation time, we adopt the Hybrid $\rightarrow$ DOL method implemented in Algorithm 2 which is a combination of intra- and inter- property learnings. The last two columns show the result using this method.

Table 6.2 shows that the decision ordering is a better inter-property learning than conflict clauses. The decision ordering learning-based method can achieve 3.5–4.5 times improvement compared with the method using zChaff. Furthermore, Hybrid $\rightarrow$ DOL method outperforms three other methods. Since the base property is a major bottleneck of the clustering methods [2], the test generation time reduction of the base property using hybrid learning can drastically improve the overall performance. Therefore, Hybrid $\rightarrow$ DOL method can achieve the best performance with 6–9 times improvement compared with the method using zChaff.

Table 6.2 Test generation results for MIPS processor

MIPS unit	Prop. (Tests)	CNF size #Variable	CNF size #Clause	zChaff [14] Time (s)	CCF [2] Inter. (s)	CCF [2] Time (s)	CCF [2] Speedup	DOL Time (s)	DOL Speedup	Hybrid → DOL Time (s)	Hybrid → DOL Speedup
	p_1 [a]	40881	467869	19.55	0	19.61	1	19.75	0.99	10.60	1.84
ALU	p_2	41327	467861	19.24	1.90	1.44	5.76	0.10	192.40	0.23	83.65
Unit	p_3	41774	467869	16.10	1.21	0.36	10.25	0.11	146.36	0.10	161.00
	p_4	42228	467861	17.38	1.47	0.21	10.35	0.26	66.8	0.33	52.64
Summary	all	-	-	72.27	26.20		2.76	20.22	3.57	11.26	**6.42**
	p_5 [a]	40903	467880	13.34	0	13.28	1	13.49	0.99	8.13	1.64
DIV	p_6	41341	467799	17.84	1.42	1.26	6.66	0.18	99.11	0.10	178.40
Unit	p_7	41808	467880	15.10	1.13	1.08	6.83	0.16	94.38	0.35	43.14
	p_8	42274	467880	14.84	1.79	0.31	7.07	0.17	87.29	0.46	32.26
Summary	all	-	-	61.12	20.27		3.02	14.00	4.37	9.04	**6.76**
	p_9 [a]	40879	467892	16.50	0	16.37	1	16.55	1	9.61	1.72
FADD	p_{10}	41368	467892	19.53	1.48	0.26	11.22	0.25	78.12	0.17	114.88
Unit	p_{11}	41838	467892	21.29	1.25	0.91	9.86	0.15	141.93	0.10	212.90
	p_{12}	42309	467892	19.44	1.78	0.52	8.45	0.15	176.73	0.10	176.73
Summary	all	-	-	76.76	22.57		3.40	17.06	4.50	9.99	**7.68**
	p_{13} [a]	52099	600928	49.94	0	49.94	1	49.90	1	22.53	2.22
MUL	p_{14}	52824	600928	49.28	2.21	5.20	9.48	0.21	234.67	0.18	273.78
Unit	p_{15}	53551	600928	45.58	1.80	2.35	19.40	0.26	175.31	0.18	253.22
	p_{16}	54276	600928	54.55	1.52	3.83	14.24	0.14	389.64	0.17	320.88
Summary	all	-	-	199.35	61.32		3.25	50.51	3.94	23.06	**8.64**

[a] Base property

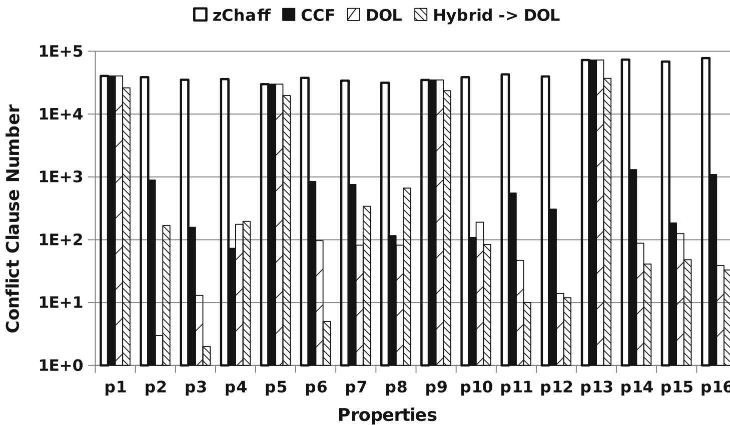


Fig. 6.10 Conflict statistics for MIPS processor

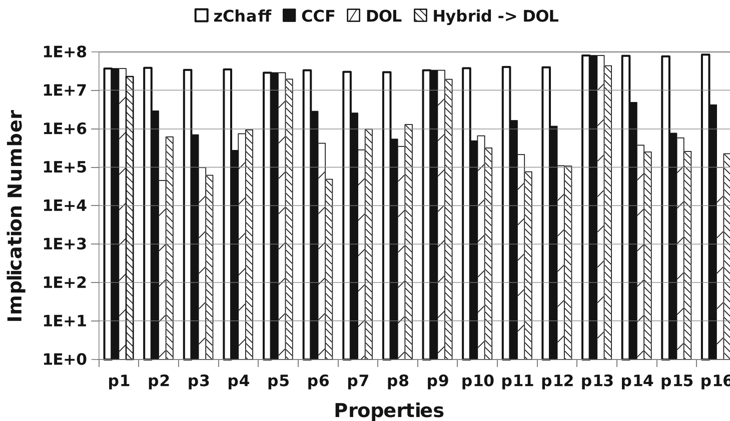


Fig. 6.11 Implication statistics for MIPS processor

During the SAT searching, the number of conflict clauses and the number of implications strongly indicate the searching time. Figure 6.10 illustrates the conflict clause generation for each property during the search using different methods. Figure 6.11 shows the corresponding implication numbers. It can be seen that, by using the proposed methods, the number of conflict clauses and implications can be reduced drastically by several orders of magnitude, which results in significant improvement in test generation time. It can be found that the decision ordering based method outperforms the conflict clause forwarding based method because of less conflicts and implications encountered. Furthermore, the decision ordering method does not need to calculate the CNF intersections which is time-consuming. Among the four methods, Hybrid $\rightarrow$ DOL method can achieve least number of

Table 6.3 Test generation results for stock exchange system

Cluster	Size	zChaff [14] (s)	Conflict clause (s)	Decision ordering (s)	Hybrid → DOL (s)
C_1	3	26.95	25.32	14.86	9.04
C_2	4	74.05	38.91	3.71	4.02
C_3	8	350.99	313.77	17.99	28.47
C_4	4	4.12	5.75	2.67	1.27
C_5	4	62.33	71.55	8.97	5.44
C_6	8	535.06	269.22	23.06	40.07
C_7	2	10.13	6.94	4.66	4.59
C_8	8	768.32	332.68	73.15	51.26
C_9	8	241.99	145.86	40.95	11.59
Total	**49**	**2073.94**	**1210.00**	**190.02**	**155.75**
Speedup	–	**1**	**1.71**	**10.91**	**13.32**

conflicts and implications for base properties (i.e., p_1, p_5, p_9, and p_{13}), which justifies the discussion in Sect. 6.3.4. It can achieve the best performance in 8 out of 12 non-base properties (i.e., p_3, p_6, p_{10}, p_{11}, p_{12}, p_{14}, p_{15} and p_{16}). Therefore, hybrid→ DOL method gives the best performance in the overall test generation time.

6.5.2.2 A Stock Exchange System

The formal NuSMV description of the online stock exchange system (OSES) is derived in the same way as that presented in Sect. 2.4.4. A path in the UML activity diagram indicates a stock transaction flow. There are a total of 49 properties generated based on path coverage criteria. According to the structural similarity, they are grouped into nine clusters.

Table 6.3 shows the test generation results involving all the nine clusters. The first column indicates the clusters. The second column indicates the size of each cluster (i.e., number of properties). The third column presents the test generation time (including base property) using zChaff. The fourth column gives the result using conflict clause based property learnings [2]. The fifth column presents the result using the decision ordering based property learnings. In this method, we do not consider the intra-property learning for the base property. The last column indicates the test generation time using the method proposed in Algorithm 2. In this case study, it can be found that the hybrid → DOL method can produce an average of 13.32 times overall improvement in test generation time compared with zChaff. It is important to note that the hybrid → DOL method can achieve the best performance, which is consistent with the results obtained in Sect. 6.5.2.1.

6.6 Chapter Summary

To address the complexity of test generation using SAT-based BMC, this chapter presented a novel methodology which explores the intra-property learnings within a SAT instance and inter-property learnings between similar SAT instances. All these learnings are based on decision ordering heuristics as well as conflict clause forwarding techniques. By exploiting the commonalities during the search of satisfiable assignments, the test generation time of a single property as well as a set of similar properties can be reduced. The experimental results using both hardware and software designs demonstrated the effectiveness of the intra- and inter- property learning approaches. According to the experimental results, hybrid learning is more profitable for solving one SAT instance, whereas Hybrid $\rightarrow$ DOL approach is more beneficial for solving a set of similar SAT instances.

References

1. Bryant R (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput 35(8):677–691
2. Chen M, Mishra P (2010) Functional test generation using efficient property clustering and learning techniques. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 29(3):396–404
3. Chen M, Mishra P (2011) Property learning techniques for efficient generation of directed tests. IEEE Trans Comput 60(6):852–864
4. Chen M, Qin X, Mishra P (2010) Efficient decision ordering techniques for SAT-based test generation. In: Proceedings of design, automation, and test in europe (DATE), pp 490–495
5. FBK-irst and CMU (2006) NuSMV. http://nusmv.irst.itc.it/
6. Marques-Silva J, Sakallah K (1999) Grasp: a search algorithm for propositional satisfiability. IEEE Trans Comput (TC) 48(5):506–521
7. Marques-Silva JP, Sakallah KA (1999) The impact of branching heuristics in propositional satisfiability. In: Proceedings of the 9th portuguese conference on artificial intelligence, pp 62–74.
8. Mishra P, Chen M (2009) Efficient techniques for directed test generation using incremental satisfiability. In: Proceedings of international conference of VLSI design, pp 65–70
9. Moskewicz MW, Madigan CF, Zhao Y, Zhang L (2001) Chaff: engineering an efficient SAT solver. In Proceedings of design automation conference (DAC), pp 530–535
10. Shtrichman O (2000) Tuning SAT checkers for bounded model checking. In: Proceedings of the international conference on computer aided verification (CAV), pp 480–494
11. The satisfiability library (2003) SAT benchmark problems. http://www.satlib.org/Benchmarks/SAT/BMC/description.html
12. Velev M (2006) Boolean satisfiability (SAT) benchmarks. http://www.miroslav-velev.com/sat_benchmarks.html
13. Wang C, Jin H, Hachtel GD, Somenzi F (2004) Refining the SAT decision ordering for bounded model checking. In: Proceedings of design automation conference (DAC), pp 535–538
14. zChaff (2007) zChaff. http://www.princeton.edu/~chaff/zchaff.html

Chapter 7
Synchronized Generation of Directed Tests

7.1 Introduction

Existing test generation techniques using SAT-based bounded model checking (BMC) [1] can be divided into two categories based on whether it addresses one property or multiple properties. The first category is applicable for test generation for one design and one property with varying bounds [2, 3]. However, the knowledge obtained is not shared when solving for other properties on the same design. In contrast, the methods in the second category try to accelerate the test generation for multiple properties with known bounds [4]. They first group similar properties into clusters. Next, the knowledge is shared by all properties in the same cluster. This approach exploits the fact that although each test generation instance is created for a different property, these instances still have a large overlap, because the design remains unchanged. The major drawback of this solution is that it assumes that the bound is known prior to SAT solving. In general, it is difficult to determine the bound upfront without actually solving the SAT instance, which limits the applicability of this solution.

This chapter presents an approach [5] that combines the advantages of both approaches by developing a novel BMC-based test generation technique for multiple properties of the same design, which enables the reuse of learned knowledge across different bounds as well as across properties in the same cluster. The basic idea of this approach is to synchronize the solving process of multiple properties for different bounds, so that the utilization of learned knowledge can be maximized. One may think that solving many SAT instances together can be dramatically complex than solving one instance, and therefore may be impractical. On the contrary, since all these instances are generated by unrolling the same design several times, this approach significantly reduces the overall SAT solving time by forwarding knowledge among different solving processes. The experimental results demonstrate an order-of-magnitude reduction in overall test generation time.

The rest of the chapter is organized as follows. Section 7.2 describes related work on BMC and directed test generation. Section 7.3 describes the synchronized test

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_7,
© Springer Science+Business Media New York 2013

generation (STG) methodology for multiple properties and bounds. Section 7.4 presents the experimental results. Finally, Sect. 7.5 concludes this chapter.

7.2 Related Work

A great deal of work has been done to accelerate the SAT solving process during BMC [2–4, 6]. Hooker [7] was the first to introduce the notion of incremental satisfiability based on DPLL algorithm. After that, many incremental SAT solvers have been designed to reduce the solving time by recording and utilizing the previously learned conflict clauses. A common approach is to keep generated conflict clauses as long as the clauses which led to the conflicts are not removed from the database. In [8], the clauses that are responsible for the deduction of a new conflict clause in the implication graph is recorded, so that such conflict clauses can be used in the future when applicable. The only problem is that keeping track of such dependencies might be expensive. Interestingly, such dependency in BMC are much easier to track than general cases. For single property checking, Strichman [2, 3] observed that if a conflict clause is deduced only from the transition part of a SAT instance, it can be safely forwarded to all instances with larger bounds, because the transition part of the design will still be in the SAT instance when we unroll the design for more times. Nevertheless, this approach was designed to check one property on one design at a time and cannot be directly applied to accelerate the SAT solving of multiple properties.

In directed test generation, tests can be created based on the graph model of the design specification and automatically generated properties [9] using BMC. To accelerate the SAT solving process, Mishra et al. [4] forwarded clauses between properties to speedup the SAT solving process. They found that different properties can be divided into several clusters based on their textual or structural similarity. Within the same cluster, although properties are still different, the SAT instances usually have a large overlap, because these similar properties are checked on the same design. By solving a "base" property, many common conflict clauses can be generated and shared to accelerate the solving processes of other properties in the same cluster. However, this method requires the bound as an input. Since the bound calculation itself is usually time-consuming, and may be impossible in many scenarios without solving the SAT instances, the applicability of their approach is restricted.

7.3 Synchronized Test Generation

Figure 7.1 shows the framework of the STG approach. In order to create directed tests, the formal model of the design, a set of properties for the desired behaviors (faults) that should be activated, and the corresponding cluster information is accepted as input. Next, the SAT instances for each property are grouped into different clusters

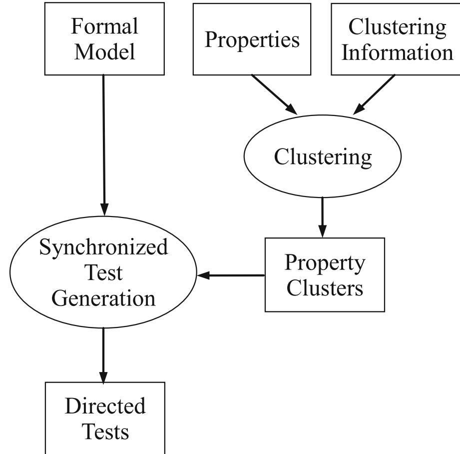


Fig. 7.1 Synchronized test generation

based on their similarity and then solved simultaneously to create the test suite, which can be used to trigger the desired behaviors during simulation-based validation. As discussed in Sect. 5.4, the clustering is performed based on the similarity on structural or textual overlap among different properties. The properties in the same cluster are describing behaviors of the same functional unit or component. Algorithm 1 outlines the key steps in the directed test generation framework.

To highlight the contribution of this technique, Fig. 7.2 compares it with two closely related techniques: (i) incremental SAT for single property with unknown bound [3] and (ii) test generation for multiple properties with known bounds [4]. In this example, there are three properties p_1, p_2, and p_3 with bounds 3, 2, and 1 respectively. We use solid dots to represent different SAT instances and lines to indicate the conflict clause forwarding paths. Strichman et al. [3] solved each property separately, and passed the knowledge (deduced conflict clauses) "horizontally" within instances for the same property (Fig. 7.2a). In contrast, Chen et al. [4] solved one "base" property first, (e.g., p_2 in this case), then forward the learned clause "vertically" between other SAT instances for different properties, as shown in Fig. 7.2b.

Algorithm 1: Test generation framework

Input: i) Design D ;
    ii) Properties P for fault activation ;
Output: Tests for corresponding faults
Cluster similar properties into groups.;
$TestSuite = \emptyset$;
for *each property cluster* PC **do**
    Perform Synchronized Test Generation on PC;
    Add generated tests into TestSuite.
end
return *TestSuite*

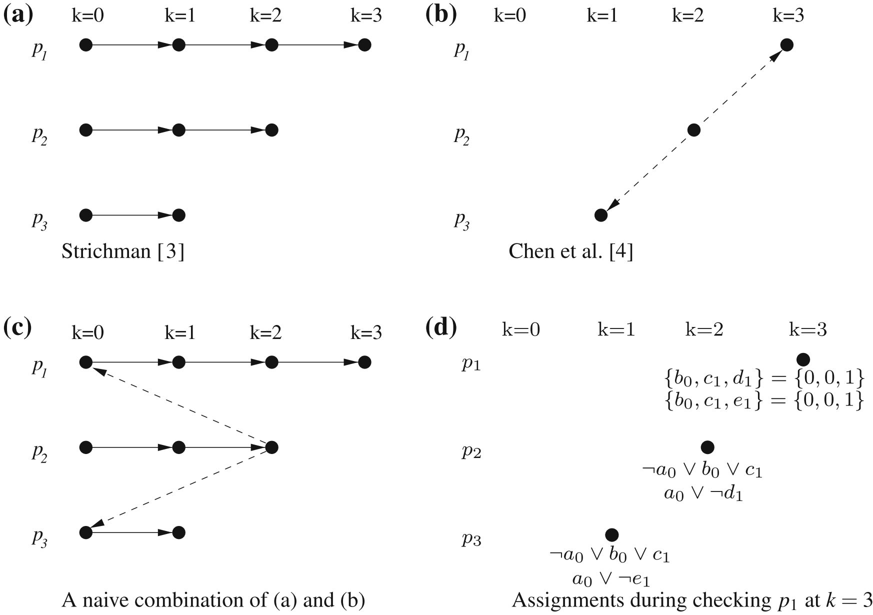


Fig. 7.2 Different incremental SAT solving techniques. **a** Strichman [3]. **b** Chen et al. [4]. **c** A naive combination of **a** and **b**. **d** Variable assignments during checking p_1 at $k = 3$

Clearly, it should be profitable if we can appropriately forward conflict clauses "vertically" between properties while solving for each property "horizontally". In this way, the knowledge learned during checking a property for a specific bound can benefit itself with larger bounds as well as across other properties. One intuitive way to combine the two approaches, as shown in Fig. 7.2c, is to choose some property as based property (p_2 in Fig. 7.2c), check this property for different bounds, and then forward the learned conflict clauses to other SAT instances for other properties. Unfortunately, this naive combination has three problems. First, it is very hard to choose the base property, that should yield a large number of conflict clauses which can be shared by other properties. Unlike [4], where each property has only one SAT instance, we do not know how many SAT instances we have to solve. As a result, it is impossible to apply the clustering technique proposed in [4], to determine the base property. Second, even if we correctly find the optimal base property, it is still difficult to choose the suitable bound of the receiving property to forward clauses, because SAT instances with inappropriate bounds may be solved trivially. Moreover, the learning during checking non-base properties is wasted. For example, in Fig. 7.2d, suppose $(\neg a_i \vee b_i \vee c_{i+1})$, $(a_i \vee \neg d_{i+1})$ and $(a_i \vee \neg e_{i+1})$ are clauses within the transition constraint of the system at time step $i + 1$.

In the SAT solving process of p_2 with bound $k = 2$, a conflict clause $(b_0 \vee c_1 \vee \neg d_1)$ is deduced based on $(\neg a_0 \vee b_0 \vee c_1)$ and $(a_0 \vee \neg d_1)$ to prevent the assignment $\{b_0, c_1, d_1\} = \{0, 0, 1\}$, which will result in a conflict on a_0. During the solving process of p_1 with bound $k = 2$, the SAT solver may explore the assignment

$\{b_0, c_1, d_1\} = \{0, 0, 1\}$ if Strichman's approach [3] is employed. Such assignment can be avoided by using [4] (as shown in Fig. 7.2b and c), because the learned conflict clause $(b_0 \vee c_1 \vee \neg d_1)$ is forwarded to p_1.

However, learned clauses are only allowed to be forwarded from the base property (p_2 in this case). The knowledge learned during solving non-base properties will not be reused. As indicated in Fig. 7.2d, conflict clause $(b_0 \vee c_1 \vee \neg e_1)$ is deduced based on $(\neg a_0 \vee b_0 \vee c_1)$ and $(a_0 \vee \neg e_1)$ during the solving process of p_3 with bound $k = 1$. Since p_3 is not a base property, this information will not be reused by p_1. Therefore, during the solving process of p_1 with bound $k = 2$, the SAT solver will still try to make the assignment $\{b_0, c_1, e_1\} = \{0, 0, 1\}$. When the number of properties is large, this may cause a great waste of computational power, because we have to explore the same search space many times, if the space is not visited during the solving process of the base property.

A promising approach to solve this problem is based on the effective identification of conflict clauses that can be shared by other SAT instances across properties and bounds. In fact, for any bound $k_0 \geq 0$, all SAT instances generated during BMC (Eq. 3.2) with $k \geq k_0$ clearly share the transition clauses $I(s_0) \wedge \bigwedge_{i=0}^{k_0-1} R(s_i, s_{i+1})$, although their property terms $\bigvee_{i=0}^{k} \neg p(s_i)$ are different. This observation implies that all conflict causes deduced based on these common clauses during solving process of *any* SAT instance can be forwarded to *any* other SAT instances with $k \geq k_0$, because all of them have the same set of clauses that led to the conflict clause. Therefore, if we check all properties together for $k = 0, 1, 2, \ldots$, i.e., "synchronously", all conflict clauses can be safely shared by all subsequent SAT instances.

Algorithm 2: Synchronized Test Generation for Properties in a Cluster

---

Input: i) Design D ;
ii) Properties P ;
iii) Maximum bound $K_{\max}$
Output: Test Set TS
Bound $k = 0$;
Common Conflict Clause Set $CCS = \emptyset$;
$TS = \emptyset$;
while $P \neq \emptyset$ *and* $k \leq K_{\max}$ **do**
  Clause Set $CS_T^k = BMC(D, true, k)$;
  for $p \in P$ **do**
    Clause Set $CS_p^k = BMC(D, p, k)$;
    Step1: In CS_p^k, mark all clauses that also exist in CS_T^k ;
    Step2: $(ConflictC, test_p) =$ SAT$(CCS \bigcup CS_p^k)$;
    Step3: $CCS = CCS \bigcup$ CheckMark $(ConflictC)$;
    if $test_p \neq null$ **then**
      remove p from P;
      $TS = TS \bigcup test_p$;
    end
  end
  $k = k + 1$;
end
return *TS*

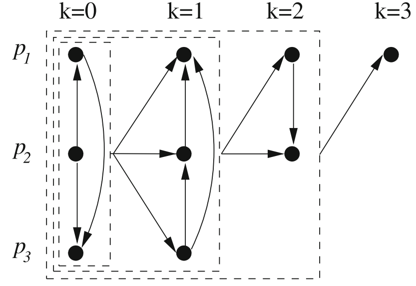


Fig. 7.3 Synchronized test generation for multiple properties

Algorithm 2 outlines the STG method for clustered properties. It accepts each property cluster and the design of the system as input and produces corresponding tests. As indicated before, this algorithm will check all properties synchronously for each bound. In each iteration, the transition clause set CS_T^k (corresponding to $I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$) is generated first using BMC(D,true,k). Next, a property p from the property set P is randomly chosen to create its own clause set CS_p^k (corresponding to $I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$). Finally, the following three steps are performed.

1. Mark all clauses in CS_p^k which are also in CS_T^k. Since CS_T^k remains same for all properties at k, this step can be implemented efficiently by table lookup, as described in Sect. 7.3.2.
2. Use a SAT solver to solve the CNF formula $CCS \bigcup CS_p^k$, which contains not only CS_p^k, but also all previously learned conflict clauses in CCS.
3. For new conflict clauses $ConflictC$ learned by SAT solver, merge the clauses deduced purely by marked clauses into CCS. This step is similar to the isolation technique proposed in [2, 4].

If the satisfied assignment or a counterexample $test_p$ is found in step 2, we record it in test set TS and remove p from P. This process repeats until tests for all properties are found or the maximum bound $K_{\max}$ is reached. Finally, the algorithm returns all generated tests.

The same example in Fig. 7.2 is used to illustrate the flow of Algorithm 2. The clause forwarding path is shown in Fig. 7.3. In the first iteration for $k = 0$, suppose we randomly pick p_2 from the property set. At the beginning, the common conflict clause set CCS is empty. Thus, p_2 is solved directly. Since the bound of p_2 is 2, the SAT instance is not satisfiable and no test is generated. However, all conflict clauses deduced based on clauses in CS_T^0 are now recorded in CCS, and will be used to accelerate the solving process of both p_1 and p_3 at bound 0. Similarly, the conflict clauses generated during solving p_1 at $k = 0$ will be used to speedup p_3 at $k = 0$ (assumes p_3 is solved last). In the next iteration, all instances will be solved with the help of conflict clauses learned by all three SAT instances at $k = 0$, because all conflict clauses are recorded in CCS. Eventually, three tests will be generated at bound 3, 2, and 1 for p_1, p_2 and p_3 respectively. In the case of Fig. 7.2d, since

both $(\neg a_0 \vee b_0 \vee c_1)$, $(a_0 \vee \neg d_1)$ and $(a_0 \vee \neg e_1)$ are clauses from the transition constraint of the system, both $(b_0 \vee c_1 \vee \neg d_1)$ and $(b_0 \vee c_1 \vee \neg e_1)$ will be recorded in CCS based on Algorithm 2. Therefore, during the solving process of p_1 with bound $k = 2$, the SAT solver will skip the assignment $\{b_0, c_1, d_1\} = \{0, 0, 1\}$ and $\{b_0, c_1, d_1\} = \{0, 0, 1\}$. In this way, the unnecessary waste of time is avoided.

The remainder of this section proves the correctness of STG approach and discusses the implementation details of the STG algorithm.

7.3.1 Correctness of STG

To show the correctness of STG, we need to show that in Algorithm 2, solving $CCS \bigcup CS_p^k$ is equivalent to solving CS_p^k. Formally, let φ_p^k and ψ be the CNF formulae formed by clause set CS_p^k and CCS respectively, we need to prove that φ_p^k is satisfiable iff $\varphi_p^k \wedge \psi$ is satisfiable using the following lemma.

Lemma 7.1 $\varphi_p^k \vdash \psi$ *for all* $p \in P$ *and* $k \geq 0$.

Proof Let φ_T^k be the CNF formula formed by CS_T^k. We first show that

$$\varphi_T^k \vdash \psi \tag{7.1}$$

for $k \geq 0$ by induction on the size of ψ. In the basis step, formula 7.1 obviously holds because ψ is empty.

Considering the moment before a new conflict clause π is added to ψ in some iteration when the bound $k' \leq k$, π must be deduced from $\varphi_T^{k'} \wedge \psi$, i.e., $\varphi_T^{k'} \wedge \psi \vdash \pi$. By induction hypothesis, $\varphi_T^k \vdash \psi$ before π is added into ψ. We also know that $\varphi_T^k \vdash \varphi_T^{k'}$, because their original forms satisfy

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \vdash I(s_0) \wedge \bigwedge_{i=0}^{k'-1} R(s_i, s_{i+1})$$

Hence, $\varphi_T^k \vdash \varphi_T^{k'} \wedge \psi$. As a result, we have $\varphi_T^k \vdash \pi$ and $\varphi_T^k \vdash \psi \wedge \pi$, which means formula 7.1 still holds, after any new clause is added to ψ, as long as $k' \leq k$.

On the other hand, notice that

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \vdash I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$$

or

$$\varphi_p^k \vdash \varphi_T^k$$

Therefore, it can be concluded that

$$\varphi_p^k \vdash \psi$$

for all $p \in P$ and $k \geq 0$.

Since $\varphi_p^k \vdash \psi$, we have $\varphi_p^k \leftrightarrow \varphi_p^k \wedge \psi$.This leads to the following theorem. ∎

Theorem 7.1 *$\forall p \in P$ φ_p^k is satisfiable iff $\varphi_p^k \wedge \psi$ is satisfiable. The correctness of STG is therefore justified.*

7.3.2 Implementation Details

The STG algorithm is built around zChaff SAT solver [11], which provides clause management scheme to support incremental SAT solving. zChaff maintains all input clauses and generates conflict clauses within an internal clause database DB. When invoked, it will solve the CNF formed by all clauses currently in DB. The management of clauses within database DB is based on "group". For each clause, zChaff assigns a 32-bit group ID. Each bit identifies whether that clause belongs to a certain group or not. When a conflict clause is deduced by clauses from multiple groups, its group ID is a "OR" product of the group ID of all its parent clauses, i.e., this clause belongs to multiple groups. zChaff also allows user to add or remove clauses by group ID between successive solving processes. If one clause belongs to multiple groups, it is removed when any of these groups are removed.

With these utilities, steps 1 and 3 in Algorithm 2, can be implemented efficiently as follows:

1. In the clause marking step, add all clauses in $CS_T^k \bigcap CS_p^k$ into DB with group ID 1.
2. Add other clauses in CS_p^k into DB with group ID 2.
3. After solving all clauses in DB with zChaff, remove clauses with group ID 2.

In this way, CCS is implicitly maintained within DB, because only conflict clauses generated purely based on clauses in $CCS \bigcup CS_T^k$ are kept after each iteration.

There is another potential overhead in step 1. Before it is marked in CS_p^k, we have to identify whether it is in CS_T^k. Since CS_T^k remains the same for all properties at k, a hash table is built to record all clauses in CS_T^k. It takes $O(1)$ time to determine whether a clause from CS_p^k is in CS_T^k. Therefore, the overall time consumption of steps 1 and 3 in Algorithm 2 is negligible compared to the SAT solving time.

7.4 Case Studies

The STG approach has been evaluated using different software and hardware designs. In this section, STG is compared with existing methods [3, 4], which are described in Chap. 5, in two scenarios: a stock exchange system and a VLIW implementation of the MIPS architecture. The systems and properties are described in SMV language and converted into CNF clauses (DIMACS files) using NuSMV [12]. We used zChaff [11] as the SAT solver to implement STG algorithm. The experiments were performed on a PC with 3.0 GHz AMD64 CPU and 4 GB RAM.

7.4.1 A Stock Exchange System

The design in the first case study simulates the behavior of a common online stock exchange system (OSES), which is described in Chap. 2. It can accept, check, and execute the customer's orders (market order and limit order). The system is specified using UML activity diagram and implemented in JAVA. Its UML behavior specification has 27 activities, 29 transitions, and 18 key paths. The specification is translated into NuSMV input to generate corresponding SAT instances. STG is applied to find the satisfiable assignments, which can be used as tests. We compared STG with Strichman's approach [3] and a naive combination of [3, 4] on different properties with unknown bounds. For Strichman's approach [3], it is used to solve a sequence of SAT instances for the same property with varying bounds until a satisfiable instance is found. The naive combination of [3, 4] is developed as described in Sect. 7.3. After SAT instance generation, the cone of influence (COI) is applied to speedup Strichman's approach. When STG was applied, COI is not used.

Table 7.1 shows the results of 20 most time-consuming properties using Strichman's approach [3]. The first column shows the properties used for test generation. The second column indicates the corresponding bounds of each property. The third column shows the test generation time (in seconds) for each property using STG. The time consumed by steps 1 and 3 in Algorithm 2 is also counted in this column. The fourth column indicates the time required by Strichman's approach [3] to generate the test for the same property. The time is calculated as the summation of the time to solve all the SAT instances from $k = 0$ to the bound of the property. The fifth column shows the speedup[1] of STG over [3]. The last two columns present the test generation time using the naive combination of [3, 4] and the speedup of STG. It can be seen that STG can produce more than 10 times improvement compared to [3], because many more conflict clauses are reused by subsequent iterations. This is especially important for "hard" SAT instances, which have to explore a potentially large assignment space. For example, the "hardest" property p_1 for [3] actually consumes less than 3 s in STG. Clearly, the time consumption for solving multiple SAT instances using STG is significantly smaller than the summation of time to solve each instances

[1] It is calculated as (previous column/third column).

Table 7.1 Test generation time comparison for OSES

Property	Bound	STG Time (s)	[3] versus STG		[3] + [4][a] versus STG	
			Time (s)	Speedup	Time (s)	Speedup
1	15	2.94	180.31	61.24	67.58	22.96
2	14	2.55	150.49	59.06	57.70	22.64
3	14	3.12	149.89	48.04	61.11	19.59
4	15	10.54	139.56	13.25	42.53	4.04
5	14	19.38	130.58	6.74	55.74	2.88
6	14	2.97	107.13	36.09	61.66	20.77
7	16	6.61	101.67	15.39	35.86	5.43
8	16	3.54	89.31	25.20	3.76	1.06
9	15	1.73	84.19	48.72	38.97	22.55
10	12	1.96	84.07	42.80	5.51	2.80
11	13	1.21	83.94	69.48	22.54	18.66
12	15	2.83	83.80	29.59	39.77	14.04
13	15	5.60	83.01	14.81	23.49	4.19
14	14	1.34	80.25	59.88	22.60	16.86
15	14	11.16	79.79	7.15	22.53	2.02
16	15	0.85	78.72	92.39	10.94	12.85
17	15	0.88	78.28	88.95	14.51	16.49
18	15	0.86	78.19	90.49	12.60	14.58
19	12	79.40	74.96	0.94	75.10	0.95
20	12	1.38	73.46	53.23	5.43	3.93
Total	–	160.87	2011.62	**12.50**	679.93	**4.23**

[a]This is an intuitive combination of [3, 4] (Fig. 7.2c). We have shown these results to demonstrate how STG is superior to any naive combination of existing methods [3, 4]

independently. The overall time consumption is reduced by knowledge sharing during solving all properties synchronously.

One interesting observation is that the most time-consuming property p_{19} in STG has a bound of only 12. The reason for this is that the clauses learned during the solving process of easier properties like p_{19} eliminated some useless searching attempts for the solution of harder properties like p_1. More importantly, these clauses are more effective than the conflict clauses learned during solving SAT instances of the same property with smaller bounds. Although p_{19} itself, which was solved first, did not benefit from other properties, the overall time consumption was dramatically reduced. As a result, STG outperforms [3], which only forwards clauses within SAT instances of the same property.

For the naive combination of [3, 4], p_{19} is chosen as the base property and forwarded the clauses learned during solving it to other properties at bound 11. These parameters are selected to illustrate the best possible performance of the combination. It is remarkably faster compared to Strichman's approach [3], although it is still four times slower than STG. It should be noted that in reality, it is impossible to choose the optimal parameter for this combination because the bounds are unknown

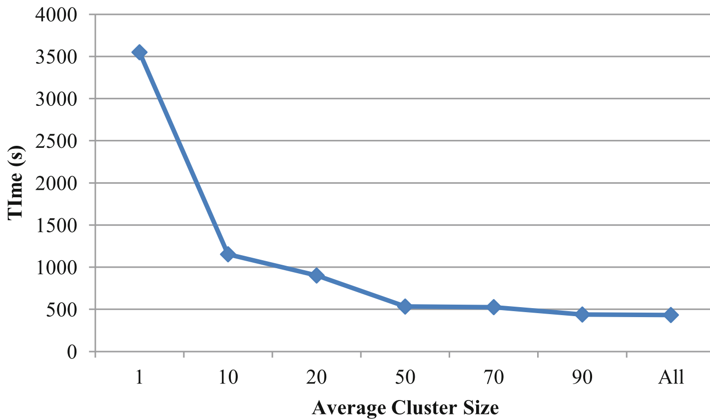


Fig. 7.4 Test generation time for OSES

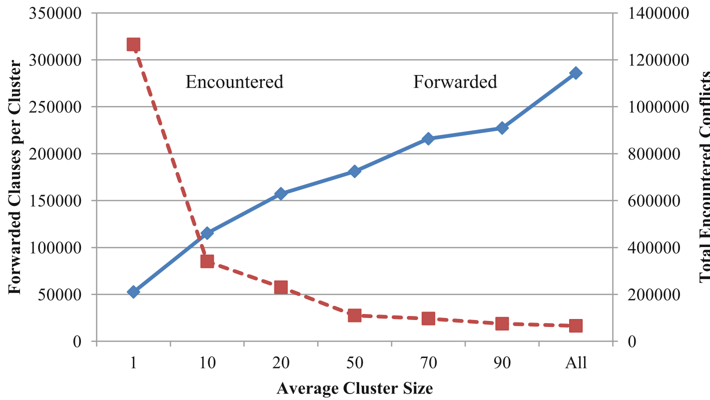


Fig. 7.5 Forwarded clauses and encountered conflicts during test generation for OSES

for all properties. In other words, the performance of the naive combination of [3, 4] will be much worse than the results illustrated here. Thus, STG will outperform it more significantly in practical scenarios.

We also investigated the impact of cluster size on the overall solving time. All the 135 properties are clustered into groups of different sizes. Figure 7.4 presents the overall solving time with respect to different average cluster size. Figure 7.5 shows the corresponding average number of forwarded clauses per cluster (solid curve) and the total number of conflicts encountered for different cluster sizes (dotted curve). Their

Table 7.2 Test generation time comparison for MIPS

Property	Bound	STG Time (s)	[3] versus STG		[3] + [4]* versus STG	
			Time (s)	Speedup	Time (s)	Speedup
1	8	0.78	139.29	179.48	18.66	24.04
2	8	0.74	132.07	178.46	19.45	26.29
3	8	0.76	125.18	164.70	18.18	23.93
4	8	0.76	120.02	158.74	18.45	24.40
5	8	0.76	115.84	151.61	27.14	35.53
6	9	0.86	111.13	129.81	58.26	68.06
7	8	0.81	108.09	133.76	26.63	32.95
8	9	0.95	104.56	110.29	53.59	56.52
9	8	0.75	96.25	128.67	16.77	22.41
10	8	0.77	87.24	113.00	16.47	21.33
11	8	0.76	87.23	114.77	17.37	22.85
12	8	0.77	84.98	110.64	16.45	21.42
13	7	0.65	81.08	125.11	13.35	20.60
14	9	32.31	80.25	2.48	31.61	0.98
15	8	0.76	75.47	99.30	7.25	9.54
16	8	0.76	72.05	94.30	20.63	26.99
17	7	76.54	71.72	0.94	72.30	0.94
18	8	1.00	70.05	70.33	19.46	19.53
19	8	0.76	69.85	91.90	6.98	9.19
20	8	0.76	65.80	87.03	11.08	14.65
Total	–	122.99	1898.13	**15.43**	490.06	**3.98**

values can be found on the left and right y-axes respectively. The result suggests that larger clustering is generally helpful to reduce the overall solving time. The reason is that the number of forwarded clauses usually increases with the average cluster size, which can effectively reduce the total number of conflicts encountered during the solving process.

7.4.2 A MIPS Processor

The STG approach has also been applied to a single-issue MIPS processor [13, 14]. There are five pipeline stages: fetch, decode, execute, memory access, and writeback. The execute stage has four parallel execution paths: integer ALU, 7 stage multiplier (MUL1 -MUL7), four stage floating-point adder (FADD1 - FADD4), and multi-cycle divider (DIV).

The design is translated into the NuSMV format. The three approaches are employed to solve the generated SAT instances for different properties and bounds. For the combination of [3, 4], p_{17} is chosen as the base property and forwarded

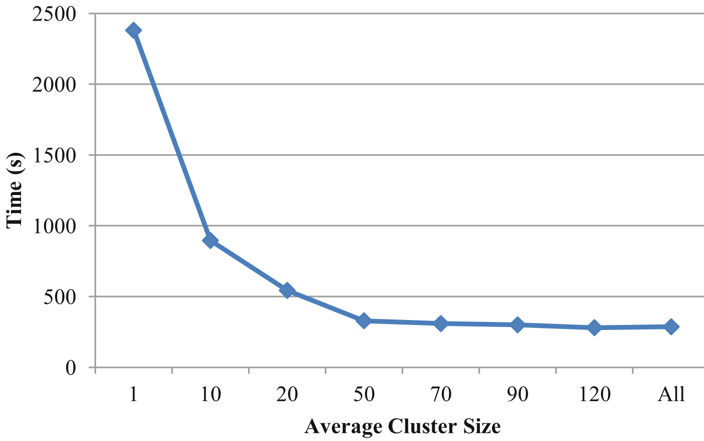


Fig. 7.6 Test generation time for MIPS processor

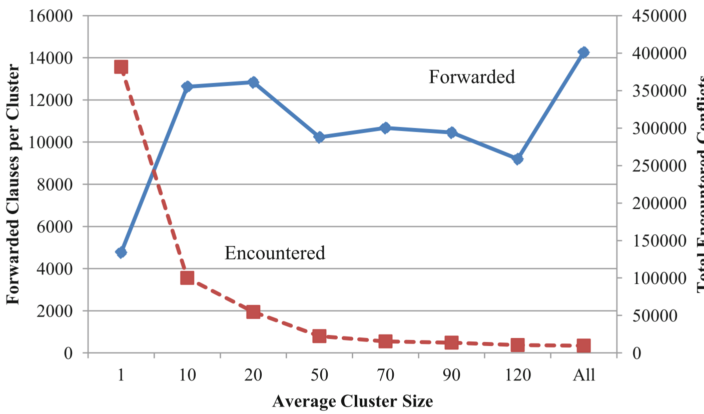


Fig. 7.7 Forwarded clauses and encountered conflicts during test generation for MIPS

learned clauses to bound 7. Table 7.2 shows the results on 20 most time-consuming properties using Strichman's approach. It can be seen that STG outperforms both Strichman's approach [3] and the naive combination of [3, 4] by 15 and 3 times respectively.

The impact of cluster size on overall solving time is shown in Fig. 7.6. There are 170 properties in total. It can be observed that the overall solving time becomes constant after the average cluster size is more than 50. At the same time, the number of forwarded clauses per cluster is not increasing, as indicated by the dotted curve in Fig. 7.7. This phenomenon can be explained by the fact that once the clusters are

large enough to include all the similar properties, the overall solving time will not be further improved and the number of forwarded clauses becomes stable.

7.5 Chapter Summary

Automatic generation of directed tests is promising for simulation-based functional validation because it requires less number of test vectors to achieve the same coverage requirement. However, its applicability is limited due to the capacity restriction of current model checking tools. Existing incremental SAT approaches are suitable only for a single property with unknown bound or for multiple properties with known bounds. To enable knowledge sharing among properties as well as bounds, we presented an STG technique for multiple properties with different bounds. SAT instances for different properties are solved together, so that the discovery and utilization of the common conflict clauses can be maximized. The overall time consumption of checking multiple properties using STG is remarkably smaller than the summation of time to check each property independently. The experimental results on both hardware and software designs demonstrated an order-of-magnitude reduction in overall test generation time.

References

1. Prasad M, Biere A, Gupta A (2005) A survey of recent advances in SAT-based formal verification. Int J Softw Tools Technol Transf (STTT) 7(2):156–173
2. Strichman O. (2001) Pruning techniques for the SAT-based bounded model checking problem. In: Proceedings of IFIP WG 10.5 advanced research working conference on correct hardware design and verification, methods, pp 58–70
3. Strichman O (2004) Accelerating bounded model checking of safety properties. Formal Methods Syst Des 24(1):5–24
4. Mishra P, Chen M (2009) Efficient techniques for directed test generation using incremental satisfiability. In: Proceedings of international conference on VLSI design, pp 65–70
5. Qin X, Chen M, Mishra P (2010) Synchronized generation of directed tests using satisfiability solving. In: Proceedings of international conference on VLSI design, pp 351–356
6. Khasidashvili Z, Nadel A, Palti A, Hanna Z (2005) Simultaneous SAT-based model checking of safety properties. In: Proceedings of haifa verification conference, pp 56–75
7. Hooker JN (1993) Solving the incremental satisfiability problem. J Log Progr 15(1–2):177–186
8. Whittemore J, Kim J, Sakallah K (2001) SATIRE: a new incremental satisfiability engine. In: Proceedings of design automation conference, pp 542–545
9. Koo HM, Mishra P (2006) Functional test generation using property decompositions for validation of pipelined processors. In: Proceedings of the conference on design, automation and test in Europe, pp 1240–1245
10. Chen M, Mishra P (2010) Functional test generation using efficient property clustering and learning techniques. IEEE Trans Comput-Aided Des Integr Circuits Syst 29(3):396–404
11. Fu Z, Mahajan Y, Malik S (2001) zChaff, Princeton University. http://www.princeton.edu/chaff/zchaff.html

12. Cavada R, Cimatti A, Jochim CA, Keighren G, Olivetti E, Pistore M, Roveri M, Tchaltse A (2010) NuSMV. ITC-Irst. http://nusmv.irst.itc.it/
13. Hennessy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann Publishers, San Francisco
14. Mishra P, Dutt N (2004) Graph-based functional test program generation for pipelined processors. In: Proceedings of the conference on design, automation and test in Europe, pp 182–187

Chapter 8
Test Generation Using Design and Property Decompositions

8.1 Introduction

Over the last couple of decades, model checking has become an important part of automatic test generation and considerable progress has been made on its applications. Despite these successes, the state space explosion problem remains a major hurdle in applying model checking to test generation for large design validation of industrial complexity. This section presents a promising approach to overcome the state explosion problem by decomposing design model and property into smaller ones.

Figure 8.1 shows a functional test generation methodology based on design and property decompositions. The design model can be generated from the architecture specification or can be developed by the designers. Similarly, the properties can be generated from the specification based on specific fault models. Additional properties can be added based on interesting scenarios and corner cases. To address the state explosion problem, the properties and the design model are decomposed. In this framework, either unbounded model checking (UMC) or bounded model checking (BMC) can be used to generate automatically partial counterexamples for the partitioned designs and decomposed properties. These partial counterexamples are integrated to construct the final test program.

This methodology has seven important steps: (i) design model generation, (ii) generation and negation of properties, (iii) design decomposition, (iv) property decomposition, (v) determination of bound for each property, (vi) test generation using model checking as well as SAT-based BMC, and (vii) merging partial (local) counterexamples to generate the global counterexample (test). The last two steps work in an integrated fashion to ensure that the generated test cases can activate the intended test scenarios.

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_8,
© Springer Science+Business Media New York 2013

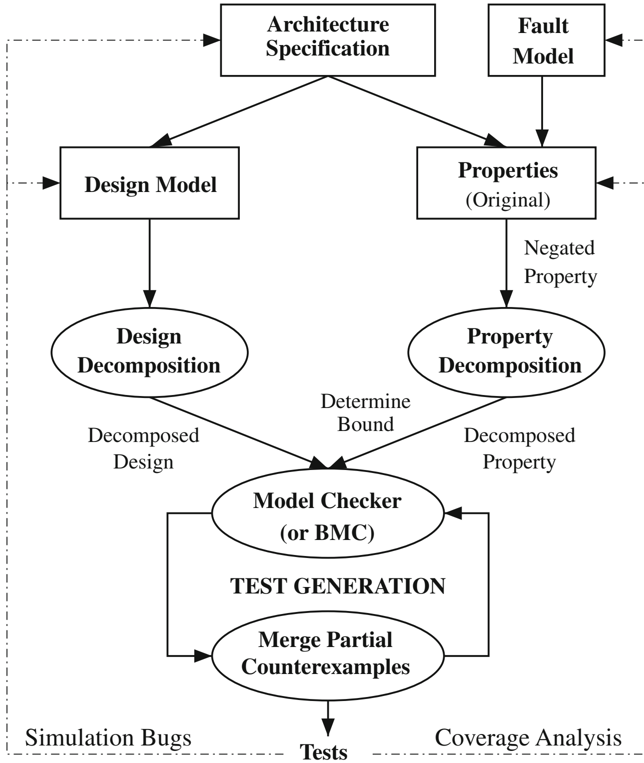


Fig. 8.1 Test generation methodology using design and property decompositions

---

Algorithm 1: Test Generation Using Design and Property Decompositions

---

Input: (i) Design specification S
(ii) Set of faults F derived from specific fault models and corner cases
Output: Tests
Test = ϕ;
M = CreateDesignModel (S);
for *each fault* F_i *in the set* F **do**
  P_i = CreateProperty(F_i);
  $bound_i$ = DecideBound(P_i);
  $\overline{P_i}$ = Negate(P_i);
  $test_i$ = DecompositionalModelChecking($\overline{P_i}$, M, $bound_i$);
  Tests = Tests $\cup$ $test_i$;
end
return Tests;

Algorithm 1 outlines the major steps in the test generation methodology shown in Fig. 8.1. This algorithm takes the model M and a set of desired faults F as inputs and generates a set of tests. For example, a set of faults F can include pipeline

interaction faults in the graph model of pipelined processors, and each functional interaction among pipeline modules is converted and negated into a temporal logic property. The exact bound for BMC is determined for each property. The design model, the negated version of the property, and the required bound are applied to the decompositional model checking framework to generate the test for the property. The algorithm iterates over all the faults based on the functional coverage and corner cases.

The remainder of this chapter presents *DecompositionalModelChecking*($\overline{P_i}$, M, $bound_i$) in Algorithm 1. Section 8.3 describes design and property decomposition techniques. Section 8.4 presents test generation techniques based on decompositional model checking and time step-based integration of partial counterexamples to construct a final test. Integration of partial counterexamples is a complicated task due to the fact that the relationships among decomposed modules and subproperties may not be preserved at the top level. Section 8.5 presents a conflict resolution technique during merging of partial counterexamples.

8.2 Related Work

Several formal model based test generation techniques have been developed for validation of processor designs. Finite state machines (FSM) have been widely used for representing the behavior of sequential systems and FSM coverage metrics can be used for effective verification. In FSM-based test generation, FSM coverage is used to generate test programs based on reachable states and state transitions [1–4]. Since complicated micro-architectural mechanisms in modern processor designs include interactions among many pipeline stages and buffers, the FSM-based approaches suffer from the state space explosion problem. To alleviate the state explosion, Utamaphethai et al. [5] have presented an FSM model partitioning technique based on micro-architectural pipeline storage buffers. Shen and Abraham [6] have proposed an RTL abstraction technique that creates an abstract FSM model while preserving clock accurate behaviors. Wagner et al. [7] have presented a Markov model driven random test generator with activity monitors that provides assistance in locating hard-to-find corner case design bugs and performance problems. In spite of much progress in FSM-based test generation, the state explosion problem remains a major challenge in applying FSM to test generation for large industrial designs.

Model checking [8, 9] has been widely used in the context of falsification by generating counterexamples. Clarke et al. [10] have presented an efficient algorithm for generation of counterexamples and witnesses in symbolic model checking. Bjesse et al. [11] have used counterexample guided abstraction refinement to find complex bugs. Automatic test generation techniques using model checking have been proposed in software [12] as well as in hardware validation [13]. However, traditional model checking based techniques do not scale well due to the state space explosion problem. To alleviate this problem, a design decomposition technique at the module level was introduced when the original property contains variables for only a single

Table 8.1 Design and property decomposition scenarios

Design	Property	Comments
0	0	Traditional model checking
0	1	Merging of counterexamples is not always possible
1	0	Similar to traditional model checking
1	1	Both property and design decompositions

0: Original; 1: Decomposed/partitioned

module [14, 15]. In order to handle common properties that have variables from multiple modules, a new framework has been developed by decomposing both the properties and the processor model [16].

As a complementary technique of model checking, Biere et al. [17, 18] introduced bounded model checking (BMC) combined with satisfiability (SAT) solving. The recent developments in SAT-based BMC techniques have been presented in [19]. BMC is an incomplete method that cannot guarantee a true or false determination when a counterexample does not exist within a given bound. However, once the exact bound of a counterexample is known, large designs can be falsified very fast since SAT solvers [20–22] do not require exponential space, and searching counterexample in an arbitrary order consumes much less memory than breadth first search in model checking. Amla et al. [23] have analyzed the performance of bounded and unbounded algorithms using a set of industrial benchmarks. The capacity increase of the BMC technique has become attractive for industrial use. Intel study [24] showed that BMC has better capacity and productivity over unbounded model checking for real designs taken from the Pentium-4 processor. SAT-based BMC can be used as a test generation engine due to its capacity and performance if the bound is selected appropriately. A major challenge in these approaches is how to determine the exact bound. Incremental SAT solvers [25–27] tried to mitigate the impact of choosing an incorrect initial bound by exploiting similarity and forwarding conflict clauses, but it is disadvantageous for deep counterexamples due to accumulation of iterative running time. A method to determine the bound for each test generation scenario has been developed in [28, 29], thereby making SAT-based BMC useful for directed test generation in pipelined processors.

8.3 Decomposition of Design and Property

It is important to note that the property and design decompositions are not independent. Table 8.1 shows four possible scenarios of design and property decompositions. The first scenario indicates the traditional model checking where original property is applied to the whole design. The second scenario implies that the decomposed properties are applied to the whole design. In certain applications this may improve overall model checking efficiency. However, in general this procedure is not

applicable since merging counterexamples may not generate the expected result. Consider an example property that can be used for generating a test to activate two simultaneous unit stalls. The property can be decomposed to generate two subproperties. These subproperties may generate counterexamples to stall the respective units in a pipelined processor but the combined test may not simultaneously stall both the units. Chapter 9 will present an efficient approach based on decision ordering to generate efficient tests. The third scenario is meaningless since design decomposition is not useful if the original property is not applicable to the partitioned design components. The last scenario depicts the approach where both design and properties are partitioned.

8.3.1 Design Decomposition

Decomposition of a design plays a central role in efficient test generation for large design validation. Ideally, the design should be decomposed into components such that there is very little interaction among the partitioned components. For a pipelined processor, the natural partition is along the pipeline boundaries. In other words, the partitioned pipelined processor can be viewed as a graph where nodes consist of units (e.g., fetch, decode etc.) or storages (e.g., memory or register file), and edges consist of connectivity among them. Typically, instruction is transferred between units, and data is transferred between units and storages.

It is important to note that the design decomposition is dependent on the property decomposition. The pipelined processor can be partitioned into modules. However, we need to change the partitioning policy based on the properties. It is hard to decompose the properties when they involve multiple modules or in the complicated forms such as pUq, $F(p \rightarrow G(q))$, $G(p \rightarrow F(q))$, and so on. For example, a property related to checking data-forwarding path is not decomposable on a basis of a module level partitioning, but it may be decomposable on a basis of a pipeline path level partitioning.

Various forms of design partitioning are possible, for example, module-level, path-level, or pipe stage-level partitioning. Figure 8.2 shows module (or node) level partitioning that gives the lowest level of granularity in the graph model. There are 17 modules in the module-level partitioned processor and each module is corresponding to each pipeline functional unit. Figure 8.3 shows path-level partitioning as gray highlighted modules. For example, the integer-ALU pipeline path {*Fetch, Decode, IALU, Mem, WriteBack*} is treated as one of the path-level partitions. Similarly, the multiplier path, the floating-point adder path, and the divider path are other examples of path level partitioning for the pipelined processor. Figure 8.4 shows stage-level decomposition based on pipeline stages that are determined by the distance from the root node (i.e., *Fetch*).

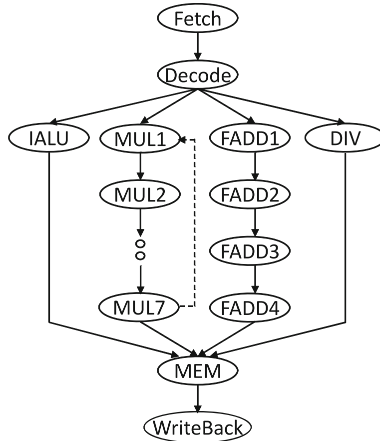


Fig. 8.2 Module-level design decomposition

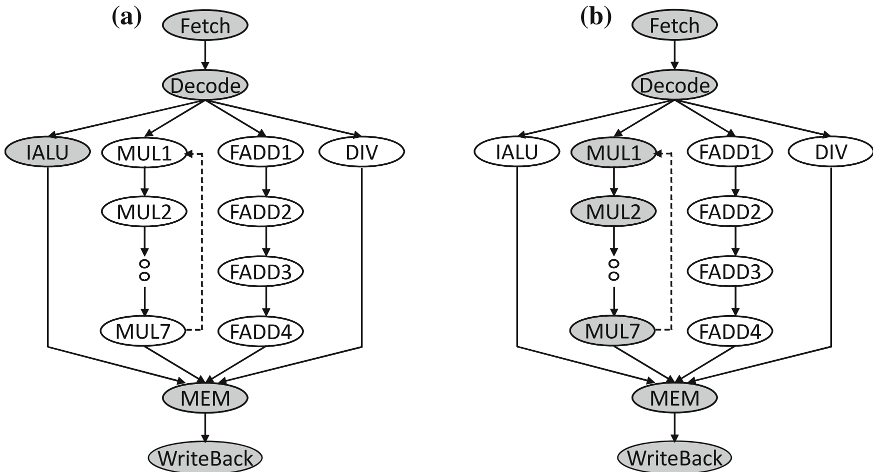


Fig. 8.3 Path-level decomposition **a** IALU Path **b** MUL Path

8.3.2 Property Decomposition

Various combinations of temporal operators and Boolean connectives are possible to express desired properties in temporal logic. If a property is decomposable, the decomposed properties can be used to generate partial counterexamples. However, not all properties are decomposable and in certain situations decompositions are not beneficial compared to traditional model checking based test generation. This section describes how to decompose pipeline interaction properties with respect to generation of a counterexample. We assume that a set of counterexamples always

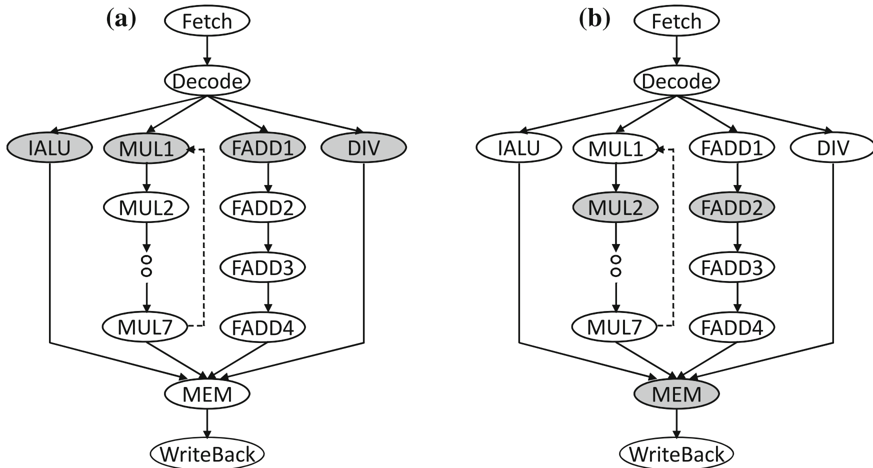


Fig. 8.4 Stage-level decomposition **a** Pipeline stage 3 **b** Pipeline stage 4

exist for the property since it is the negated version of the original property and the design model is assumed to be correct.

8.3.2.1 Pipeline Interaction Property

Today's test generation techniques and formal methods are very efficient to find logical bugs in a small module design. Hard-to-find bugs arise often from the intermodule interactions, for example, functional interactions among many pipeline stages and buffers of modern processor designs. This section primarily focuses on such hard-to-verify interactions among modules in a pipelined processor. If we consider the graph model of the pipelined processor shown in Sect. 2.2.1, the pipeline interactions imply the interactions between the nodes in the graph model. All the properties can be specified using temporal logic, since the functional coverage model of pipeline interactions does not require complex property specifications supported in Property Specification Language [30]. When a different or complex coverage model is employed, PSL-based specification may be required. However, the overall flow presented in this section will remain the same in the presence of properties specified using PSL, since the existing model checkers support PSL property specifications [31].

We first define the possible pipeline interactions based on the number of nodes in the graph model and the average number of activities in each node. For example, an IALU node can have four activities: operation execution, stall, exception, and no operation (NOP). In general, the number of activities for a node will be different based on what activity we would like to test. For example, execution of ADD and SUB operations can be treated as the same activity because they go through the same pipeline path. Separation of them into different activities will refine the functional

tests but increase the test generation complexity. Furthermore, the number of activities may vary for different nodes. In a graph model with n nodes where each node can have on average r activities, the total number of possible interactions can be expressed using the following expression:

$$\sum_{i=1}^{n} {}_nC_i \times r^i \tag{8.1}$$

Although the total number of interactions can be extremely large, in reality the number of simultaneous interactions can be small, and many other realistic assumptions and test compaction techniques can reduce the number of properties to a manageable one. The generated properties are expressed in linear temporal logic (LTL) [9]. A property is generated for each pipeline interaction from the specification. Pipeline interactions can be converted in the form of a property such as $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ that combines activities p_i over n modules using logical 'AND' operator. Since we are interested in counterexample generation, we need to generate the negation of the property. For example, the negation of $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$, *interaction fault*, can be described as $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$ whose counterexamples will satisfy the original property. In the following section, we describe how to decompose these properties (already negated) for efficient test generation using model checking.

8.3.2.2 Decomposable Properties

The following types of properties allow simple decompositions. Lemmas 8.1–8.4 prove that the decomposed properties can be used for test generation.

$$\begin{aligned} G(p \wedge q) &= G(p) \wedge G(q) \\ F(p \vee q) &= F(p) \vee F(q) \\ X(p \vee q) &= X(p) \vee X(q) \\ X(p \wedge q) &= X(p) \wedge X(q) \end{aligned} \tag{8.2}$$

Lemma 8.1 *Counterexamples of the decomposed properties $G(p)$ and $G(q)$ can be used to generate a counterexample of $G(p \wedge q)$.*

Proof Let $C_{G(p)}$ denote the set of counterexamples for $G(p)$ that satisfies $F(\neg p)$, $C_{G(q)}$ denote the set of counterexamples for $G(q)$ that satisfies $F(\neg q)$, and $C_{G(p \wedge q)}$ denote the set of counterexamples for $G(p \wedge q)$ that satisfies $F(\neg p \vee \neg q)$. Since $F(\neg p \vee \neg q) = F(\neg p) \vee F(\neg q)$, so the sets $C_{G(p)}$ and $C_{G(q)}$ are subsets of $C_{G(p \wedge q)}$, that is, $C_{G(p)} \cup C_{G(q)} \equiv C_{G(p \wedge q)}$. Therefore, any counterexample of the decomposed properties $G(p)$ or $G(q)$ can be used as a counterexample of $G(p \wedge q)$. □

Lemma 8.2 *Counterexamples of the decomposed properties $F(p)$ and $F(q)$ can be used to generate a counterexample of $F(p \vee q)$.*

Proof Since $G(\neg p \wedge \neg q) = G(\neg p) \wedge G(\neg q)$, so the set $C_{F(p \vee q)}$ is equal to the intersection set between $C_{F(p)}$ and $C_{F(q)}$, that is, $C_{F(p)} \cap C_{F(q)} \equiv C_{F(p \vee q)}$. Therefore, a common counterexample between $F(p)$ and $F(q)$ can be used as a counterexample of $F(p \vee q)$. □

Lemma 8.3 *Counterexamples of the decomposed properties $X(p)$ and $X(q)$ can be used to generate a counterexample of $X(p \wedge q)$.*

Proof Since $X(\neg p \vee \neg q) = X(\neg p) \vee X(\neg q)$, so the sets $C_{X(p)}$ and $C_{X(q)}$ are subsets of $C_{X(p \wedge q)}$, that is, $C_{X(p)} \cup C_{X(q)} \equiv C_{X(p \wedge q)}$. Therefore, any counterexample of the decomposed properties $X(p)$ or $X(q)$ can be used as a counterexample of $X(p \wedge q)$. □

Lemma 8.4 *Counterexamples of the decomposed properties $X(p)$ and $X(q)$ can be used to generate a counterexample of $X(p \vee q)$.*

Proof Since $X(\neg p \wedge \neg q) = X(\neg p) \wedge X(\neg q)$, so the set $C_{X(p \vee q)}$ is equal to the intersection set between $C_{X(p)}$ and $C_{X(q)}$, $C_{X(p)} \cap C_{X(q)} \equiv C_{X(p \vee q)}$. Therefore, a common counterexample between $X(p)$ and $X(q)$ can be used as a counterexample of $X(p \vee q)$. □

8.3.2.3 Non-decomposable Properties

It is important to note that the property decomposition is impossible in various scenarios when the combination of decomposed properties is not logically equivalent to the original property. For example, $F(p \wedge q) \neq F(p) \wedge F(q)$, and $G(p \vee q) \neq G(p) \vee G(q)$. However, with respect to test generation, the counterexamples of the decomposed properties can be used to generate a counterexample of the original property as described below.

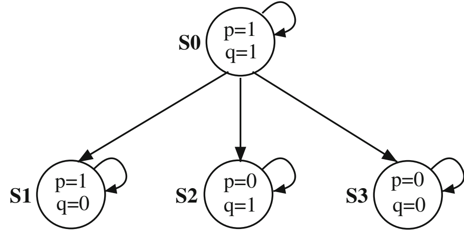
The property $F(p \wedge q)$ is true when both p and q hold at the same time step. But $F(p) \wedge F(q)$ is true even when p and q hold at different time steps. Therefore, $F(p \wedge q) \neq F(p) \wedge F(q)$. However, $F(p)$ and $F(q)$ can be used for test generation to activate the property $F(p \wedge q)$ based on the following Lemma 8.5.

Lemma 8.5 *Counterexamples of the decomposed properties $F(p)$ and $F(q)$ can be used to generate the counterexample of $F(p \wedge q)$.*

Proof Since the relation between $F(p \wedge q)$ and $F(p) \wedge F(q)$ is $F(p \wedge q) \rightarrow F(p) \wedge F(q)$, so $C_{F(p \wedge q)} \supset (C_{F(p)} \cup C_{F(q)})$. Therefore, any counterexample of the decomposed properties $F(p)$ or $F(q)$ is a counterexample of $F(p \wedge q)$. □

The property $G(p \vee q)$ is true when either p or q holds at every time step. But $G(p) \vee G(q)$ is true either when p holds at every time step or when q holds at

Fig. 8.5 An example of Kripke structure model



every time step. Therefore, $G(p \vee q) \neq G(p) \vee G(q)$. In this case, the counterexamples of the decomposed properties $G(p)$ and $G(q)$ cannot directly be used to generate a counterexample of $G(p) \vee G(q)$ since $G(p) \vee G(q) \rightarrow G(p \vee q)$, that is, $(C_{G(p)} \cap C_{G(q)}) \supset C_{G(p \vee q)}$. In other words, not all common counterexamples of $G(p)$ and $G(q)$ can be used as a counterexample of $G(p \vee q)$. Furthermore, it is hard to know whether the common counterexamples of $G(p)$ and $G(q)$ belong to $C_{G(p \vee q)}$. However, introducing the notion of clock allows the decomposed properties to produce a counterexample of $G(p \vee q)$ as described in Lemma 8.6.

Lemma 8.6 *Counterexamples of $G(p)$ and $G(q)$ can be used to generate a counterexample of $G(p \vee q)$ by introducing a specific time step.*

Proof The relation between $G(p \vee q)$ and $G(p) \vee G(q)$ with time step is $G((clk \neq t_s) \vee (p \vee q)) = G((clk \neq t_s) \vee p) \vee G((clk \neq t_s) \vee q)$ because both sides are evaluated to be true when $(clk \neq t_s)$, or when $(clk = t_s)$ and $p = true$ or $q = true$. Therefore, $C_{G((clk \neq t_s) \vee (p \vee q))} \equiv (C_{G((clk \neq t_s) \vee p)} \cap C_{G((clk \neq t_s) \vee q)})$. □

For example, Fig. 8.5 describes a Kripke structure [9] with four states $s0$, $s1$, $s2$, and $s3$, where $s0$ is the only initial state. The structure has three transitions: $(s0, s1)$, $(s0, s2)$, $(s0, s3)$, and self-loop in each state. There are two local variables p for $module1$ and q for $module2$: p holds on states $\{s0, s1\}$ and q holds on states $\{s0, s2\}$. Assuming the original property $F(p = 0 \wedge q = 0)$, we add a specific time step as $F(clk = t_s \wedge p = 0 \wedge q = 0)$[1] and its negation will be $G(clk \neq t_s \vee p = 1 \vee q = 1)$. Let us assume that $t_s = 2$. A set of counterexamples of $G(clk \neq 2 \vee p = 1 \vee q = 1)$ for the entire model is shown below:

$C_M = \{(s0, s0, s3), (s0, s3, s3)\}$

A set of counterexamples of $G(clk \neq 2 \vee p = 1)$ for $module1$ is shown below:

$C_{m1} = \{(s0, s0, s2), (s0, s0, s3), (s0, s2, s2), (s0, s3, s3)\}$

A set of counterexamples of $G(clk \neq 2 \vee q = 1)$ for $module2$ is shown below:

$C_{m2} = \{(s0, s0, s1), (s0, s0, s3), (s0, s1, s1), (s0, s3, s3)\}$

We can see that $C_{m1} \cap C_{m2} = \{(s0, s0, s3), (s0, s3, s3)\}$ is the same as C_M. Therefore, the decomposed properties can be used for test generation of the original property by introducing the specific time step.

Based on Lemma 8.6, the interaction fault $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$ is converted into $G((clk \neq t_s) \vee \neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n))$. The decomposed properties

[1] The clk variable is used to count time steps, and t_s is a specific time step during model checking.

$G((clk \neq t_s) \vee \neg p_1)$, $G((clk \neq t_s) \vee \neg p_2)$, ..., $G((clk \neq t_s) \vee \neg p_n)$ are repeatedly applied to model checker until a common counterexample is found as described in Sect. 8.4. The counterexample is one of the interactions that satisfies the property $F((clk = t_s) \wedge p_1 \wedge p_2 \wedge \ldots \wedge p_n))$. In this decomposition scenario, the time step (t_s) should be decided to guarantee that a counterexample exist within the given search bound (t_s). As described in the analysis of bounded model checking techniques [32], deciding the bound is a challenging problem since the depth of counterexamples is unknown in most cases. For deciding the bound, several techniques are described in Sect. 3.5.2.2 to enables test generation using SAT-based bounded model checking.

For certain properties such as pUq, $F(p \rightarrow F(q))$, $F(p \rightarrow G(q))$, $G(p \rightarrow G(q))$, or $G(p \rightarrow F(q))$, decompositions are not beneficial compared to traditional model checking because it is very difficult to decide a specific time step between their decomposed properties. Although many property decompositions are not possible, it is important to note that the scenarios described in this section are sufficient to generate the test programs in the context of pipeline interactions.

An important consideration during property decomposition is how to specify and handle the different types of variables in the property. In general, the properties are described as pairs of module name and variable name. An interaction fault property p_i can be either a local variable in a single module or a global variable over multiple modules. If p_i is a local variable, it is converted into $(m_i.p_i)$ where m_i is the corresponding module. If p_i is a global variable, p_i is decomposed into subproperties of corresponding modules. For example, for the property $G(\neg p_1 \vee \neg p_2)$, if p_1 is an interface variable between m_1 and m_2, and p_2 is a local variable of m_2, then the property is converted as $G(\neg m_1.p_1 \vee (\neg m_2.p_1 \vee \neg m_2.p_2))$. Decomposition of global variables is based on the decomposed modules of a processor model and their interfaces.

8.4 Decompositional Test Generation

Algorithm 2 presents the decompositional model checking procedure (invoked from Algorithm 1, DecompositionalModelChecking($\overline{P_i}, M, bound_i$)) for design and property decompositions. The basic idea is to apply the decomposed properties (subproperties) to the corresponding design partition using model checking, and compose the generated partial counterexamples to construct the final test. This algorithm accepts a property P_i (already negated in Algorithm 1), a design D, and search bound $bound_i$ as inputs and produces the required test program. The design and properties are decomposed as described in Sects. 8.3.1 and 8.3.2. The algorithm uses three lists to maintain the decomposed properties: $TaskList$ for the present clock cycle clk, $NextList$ for the next cycle i.e., $clk - 1$, and $AllList$ for all properties. Each entry in the $TaskList$ and the $NextList$ contains a collection of subproperties that are applicable to corresponding design partitions. Therefore, each list canhave up to

Algorithm 2: Decompositional Model Checking

```
Input: i) Property P_i, ii) Design D, and iii) bound_i
Output: Test
TaskList = φ; NextList = φ; AllList = φ;
PrimaryInputs = φ; clk = bound_i;
{P_i^1, P_i^2, ..., P_i^m} = DecomposeProperty(P_i);
{M_1, M_2, ..., M_n} = DecomposeDesign(D);
for each design partition M_j do
    /* P_i^j is applicable to M_j */
    TaskList[j] = AllList[clk][j] = P_i^j;
end
while TaskList is not empty and clk > 0 do
    outR_k = RemoveEntry(TaskList[k]);
    P_i^k = MergeRequirements(outR_k, AllList, clk);
    ¬P_i^k = Negate(P_i^k);
    Counterexample = ModelChecking(¬P_i^k, M_k, clk);
    inpR_k = input requirements for M_k from Counterexample;
    if inpR_k are not primary_inputs then
        for each applicable parent node M_r of M_k do
            outR_r = Extract output requirements for M_r from inpR_k;
            NextList[r] = NextList[r] ∪ outR_r;
            AllList[clk][r] = AllList[clk][r] ∪ outR_r;
        end
    else
        PrimaryInputs = PrimaryInputs ∪ inpR_k;
    end
    if TaskList is empty then
        clk = clk − 1;
        TaskList = NextList;
        NextList = φ;
    end
end
test_i = ExtractInstructions(PrimaryInputs);
return test_i
```

n entries where n is the number of design partitions in the processor model. The tasks in the TaskList need to be performed in the current time step (clk). The tasks in the NextList will be performed in the next time step ($clk - 1$). AllList contains all the entries of TaskList for each time step. This information is used to resolve the conflict among subproperties as described in Sect. 8.5. Initially these lists are empty.

The algorithm generates one test program for each property set DP_i that consists of one or more subproperties based on their applicability to different modules or partitions in the design. The algorithm adds the subproperties to the $TaskList$ and $AllList$ based on the partitions to which these properties are applicable. The

algorithm iterates over all the subproperties in the $TaskList$. It removes an entry (say kth location) from the $TaskList$ which is the output requirement $outR_k$ of kth partition. In general, this entry can be a list of subproperties (due to simultaneous output requirements from multiple children nodes) that need to be applied to partition M_k. These subproperties are composed to create the intermediate property P_i^k using $MergeRequirements$ described in Sect. 8.5. After negation of P_i^k, the property $\overline{P_i^k}$ is applied to the corresponding partition M_k using the model checker to generate a counterexample.

The generated counterexample is analyzed to find the input requirements inp_k for the partition M_k. If these are primary inputs (inputs of the root node in the graph model), then they are stored in $PrimaryInputs$ list. Otherwise, for each parent node M_r to which inp_k is applicable, we extract the output requirements for M_r. This output requirement is added to the rth entry of the $NextList$ as well as the $AllList$. Finally, if the tasks for the current time step is completed ($TaskList$ empty), $NextList$ is copied to the $TaskList$ and the time step clk is reduced by one. This process continues until both the lists are empty. Using a precise upper bound for the original property $\overline{P_i}$ enables the clk to be zero and two lists empty at the same time. However, if one chooses the diameter as the upper bound, two lists will be empty before the clk becomes zero. In both of these cases, it will ensure that we have obtained the primary input assignments for all the subproperties. These assignments are converted into a test program consisting of an instruction sequence.

For illustration, consider a simple property P_1 to verify a multiple stall scenario consisting of IALU and DIV nodes in Fig. 2.1 at clock cycle 5. We assume the module level partitioning of the design for this example. The property can be decomposed into two subproperties P_1^3 (IALU not stalled in cycle 5) and P_1^{15} (DIV not stalled in cycle 5). This implies that $TaskList$ will have two entries before entering the while loop: $TaskList[3] = P_1^3$ and $TaskList[15] = P_1^{15}$. At the first iteration of the while loop P_1^3 will be applied to M_3 (IALU) using model checker; the generated counterexample will be analyzed to find the output requirement for the Decode unit (in Fig.2.1) in clock cycle 4; and the requirement will be added to $NextList[2]$. During second iteration of the while loop P_1^{15} ($TaskList[15]$) will be applied to M_{15} (DIV); the generated counter example will be analyzed to find the output requirement for the Decode unit in clock cycle 4; and the requirement will be added to $NextList[2]$. At this point, the $TaskList$ is empty and the $NextList$ has only one entry with two requirements which is copied to the $TaskList$. At the third iteration of the while loop, these two requirements are composed into an intermediate property and applied to M_2 (Decode) that generates requirements for Fetch node. Finally, the fourth iteration applies the corresponding property to the Fetch unit that generates the primary input assignments. These assignments are converted to a test program. The following two examples show test generation using module level as well as pipeline path level partitioning of the processor model.

8.4.1 Test Generation Using Module-Level Partitioning

For illustration of test generation using module-level partitioning, this section describes an example of a multiple exception scenario at clock cycle 7 in which there are an overflow exception in IALU, divide by zero exception in DIV unit, and a memory exception in the MEM unit. The desired property P is shown as below:

```
P: F((clk=7) & (MEM.exception = 1)
             & (IALU.exception = 1)
             & (DIV.exception = 1))
```

The negated property, P', is shown below:

```
P': G((clk != 7) | (MEM.exception != 1)
                 | (IALU.exception != 1)
                 | (DIV.exception != 1))
```

P' is decomposed into three subproperties:

```
P1: G((clk != 7) | (MEM.exception != 1))
P2: G((clk != 7) | (IALU.exception != 1))
P3: G((clk != 7) | (DIV.exception != 1))
```

The subproperties $P1$, $P2$, and $P3$ will be applied to MEM, IALU, and DIV modules using SMV model checker as shown in Fig. 8.6a. The model checker will come up with a counterexample in each case as input requirements for the respective modules. For example, the counterexamples for $P1$, $P2$, and $P3$, respectively are: (C_{P1}) *load* operation with memory address zero, (C_{P2}) *add* operation with the maximum value for both source operands, and (C_{P3}) *divide* operation with second source operand value zero. These requirements are converted into properties and applied to the respective parent modules. In this case, $P1'$ (from C_{P1}) is applied to IALU, and $P23'$ (combine C_{P2} and C_{P3})[2] is applied to the Decode unit in the next step as shown in Fig. 8.6b. In each case, clock cycle value is reduced by one as shown below:

```
P1':  G((clk!=6) | (aluOp.opcode != LD) | (aluOp.src1Val != 0))
P23': G((clk!=6) | (decOp[0].opcode != ADD) | (decOp[0].src1Val != 2)
                 | (decOp[0].src2Val != 2) | (decOp[3].opcode != DIV)
                 | (decOp[3].src2Val != 0))
```

[2] Note that when multiple children create requirements for the parent (e.g, $P23'$), conflicts can occur. In such cases, alternative assignments need to be evaluated for the conflicting variable as described in Sect. 8.5.

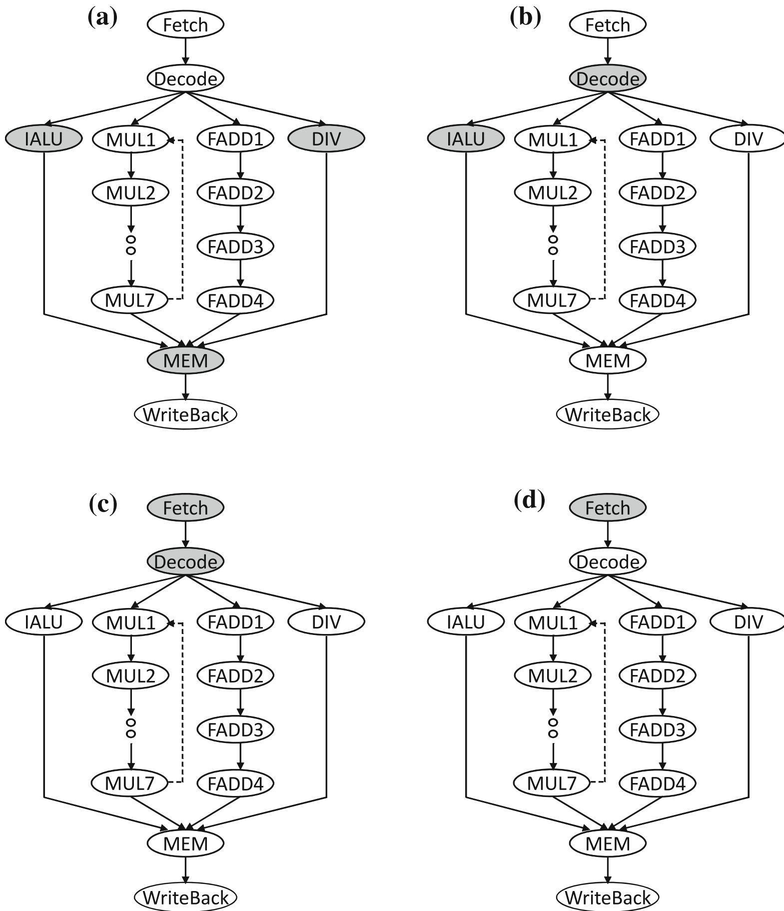


Fig. 8.6 Modules used for partial test generation at each time step **a** Modules at clk = 7 **b** Modules at clk = 6 **c** Modules at clk = 5 **d** Modules at clk = 4

Model checking using the module IALU and property $P1'$ generates the property $P1''$. Similarly, model checking using the module Decode and property $P23'$ will produce the property $P23''$. In time step 5, the property $P1''$ is applied to Decode unit, and $P23''$ are applied to Fetch unit and it generates primary inputs PI_i as shown in Fig. 8.6c.

```
P1'':  G((clk!=5) | (decOp[0].opcode != LD) | (decOp[0].src1Val != 0))
P23'': G((clk!=6) | (fetOp[0].opcode != ADD) | (fetOp[0].src1Val != 2)
                 | (fetOp[0].src2Val != 2) | (fetOp[3].opcode != DIV)
                 | (fetOp[3].src2Val != 0))
```

Model checking using the module Decode and property $P1''$ generates the property $P1'''$.

```
P1''': G((clk!=4) | (fetOp[0].opcode != LD) | (fetOp[0].src1Val != 0))
```

In time step 4, as shown in Fig. 8.6d, the property $P1'''$ will be applied to Fetch unit that generates the primary inputs PI_j. The primary inputs PI_i and PI_j are combined based on their time step (clock cycle) to generate the final test program.

8.4.2 Test Generation Using Path-Level Partitioning

The example shown above assumes a module-level partitioning of the processor model. However, it is not always possible to decompose a property based on module level partitioning. For example, if we are trying to determine whether two feedback (data-forwarding) paths are activated at the same time, it is not possible to decompose this property at module level because the "implication" relation between $feedOut$ and $feedIn$ (in the following property) will be lost.

```
/* Original Property */
 P: F((clk=9) & (FADD4.feedOut -> X(FADD1.feedIn))
              & (MUL7.feedOut -> X(MUL1.feedIn)))

/* Property after Negation*/
 P': G(((clk!=9 | !FADD4.feedOut) | (clk!=10
                | !FADD1.feedIn)) |
       ((clk!=9| !MUL7.feedOut) | (clk!=10
                | !MUL1.feedIn)))

/* Properties after Decomposition*/
 P1: G((clk!=9 | !FADD4.feedOut) | (clk!=10
               | !FADD1.feedIn))
 P2: G((clk!=9 | !MUL7.feedOut) | (clk!=10
               | !MUL1.feedIn))
```

To enable property decomposition in this example, we need to partition the design differently, that is, path-level partitioning as shown in Fig. 8.3. The floating-point adder path (FADD1 to FADD4) should be treated as a design partition F_{path}. Similarly, the multiplier path (MUL1 to MUL7) should be treated as another partition M_{path}. This new partitioning is applied for test generation. First, $P1$ and $P2$ can be applied on F_{path} and M_{path}, respectively that generates counterexamples $C1$ and $C2$. Next, $C1$ and $C2$ are combined and the corresponding property is applied to the Decode unit to generate the counterexample $C3$. Next, the property corresponding to $C3$ is

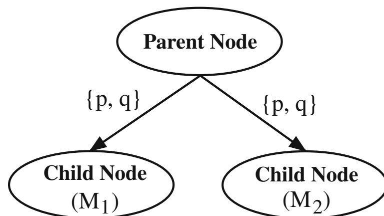


Fig. 8.7 An example structure with two children nodes

applied to the Fetch unit that generates the primary input requirements. Finally, these primary input requirements are converted into the required test program.

8.5 Merging Partial Counterexamples

If there is only a single child node, the output requirement ($outR_r$ in Algorithm 2) generated from the single child node can be directly used for the corresponding module (M_r) simply by negating the output requirement. In case of multiple children, the input requirements generated from children nodes need to be merged appropriately into the required property for the parent node. However, this is non-trivial since the input requirements can be conflicting due to the fact that the model checker assigns arbitrary values to the variables that do not have influence on falsification of the children nodes. For example, considering multiple instruction issue in Decode module, four children modules (IALU, MUL1, FADD1, DIV) share the parent node Decode. Counterexample generated from four children modules at the time step $clk = t_s + 1$ should be combined for creating the required property of Decode module at $clk = t_s$. However, they may require different output values for the same variable of the module Issue.

In case of output requirement conflict, we can adjust input requirements of the children nodes by excluding the current input requirement, called *false requirement*. For example, consider the scenario shown in Fig. 8.7. Assume that the output variables of the parent node are p and q, the input requirement of one child is $(p = 1 \wedge q = 0)$ that is generated by $G((clk \neq (t_s + 1)) \vee \neg(M_1.p = 1))$ at child node M_1, and the input requirement of the other child is $(p = 0 \wedge q = 1)$ that is generated by $G((clk \neq (t_s + 1)) \vee \neg(M_2.q = 1))$ at child node M_2. Obviously, there is no way to assign output p and q to satisfy these two conflicting inputs. We refine the subproperties of children nodes to resolve the conflict requirements by excluding the false requirement. The desired subproperties are stored in $AllList[t_s + 1]$ for children nodes and they are modified by adding the negated version of the conflict requirement as shown below:

$$F((clk = (t_s + 1)) \wedge (M_1.p = 1) \wedge \neg(M_1.p = 1 \wedge M_1.q = 0))$$
$$F((clk = (t_s + 1)) \wedge (M_2.q = 1) \wedge \neg(M_2.p = 0 \wedge M_2.q = 1))$$

To generate the input requirements of the $module1$, the above properties are negated as shown below:

$$G((clk \neq (t_s + 1)) \vee \neg(M_1.p = 1) \vee (M_1.p = 1 \wedge M_1.q = 0))$$
$$G((clk \neq (t_s + 1)) \vee \neg(M_2.q = 1) \vee (M_2.p = 0 \wedge M_2.q = 1))$$

These subproperties does not allow the counterexample $(p = 1 \wedge q = 0)$ any more. The generated counterexample will be $(p = 1 \wedge q = 1)$ as the input requirements of $module1$ and $module2$. As a result, we can merge them into the output requirement of the parent node as $(p = 1 \wedge q = 1)$ at $clk = t_s$. If there is an interface variable r between the parent and its child $module2$, it does not cause the output requirement conflict of the parent node since the input requirement of $module1$ does not influence the variable r. If there is another child node $module3$ that has the interface variables p and r, we need to adjust three input requirements of $module1$, $module2$, and $module3$ to resolve any conflict among them. It is possible that there is no common variable assignments for shared input variables among children nodes since their output requirements may be generated from false input requirements from the subsequent stages (child nodes of M_1 and M_2). In this case, we need to refine the subproperties of grandchildren nodes stored in $AllList[t_s + 2]$. The procedure of subproperty refinement continues until the conflict is resolved or clk is equal to $bound_i$ that is upper bound to search for a test program.

Although, this procedure of iterative conflict resolution may continue forever in the worst case, the experimental studies show that number of such iterations is small, 10–15 on average. In majority of these cases, the conflict happens in the fork node (e.g., Decode unit in MIPS) and it could be avoided if we maintain “don't care” value for each variable. In other words, when we compose the partial counterexamples in each stage, we can carry forward “don't care” values to the parent stages. To enable this, we need to enable/modify the existing model checking techniques to produce “don't cares” values for the variables where exact assignment is not required to produce the counterexample. To safeguard against the scenarios where the iteration may continue indefinitely (the worst case scenario), an iteration threshold can be applied to terminate the repeated conflict generation for the same scenario.

8.6 A Case Study

This section presents case studies of test generation using a MIPS processor [33]. Various test generation experiments are shown in this section for validating the pipeline interactions by varying different design partitions and property decompositions. This section also presents experimental results in terms of time and memory requirement in test generation.

Table 8.2 Comparison of test generation techniques for MIPS processor

Module interactions	Cadence SMV (UMC)		Cadence SMV (Cex-based abstraction)			NuSMV (Incr. BMC)		Decomposition (UMC+Dec.)	
	BDD (M)	Time	BDD (K)	Clauses (K)	Time	Clauses (K)	Time	BDD (K)	Time
None	6	165	23	33	0.37	135	8.47	3	0.06
2 modules	11	215	54	138	5.89	201	9.99	6	0.12
3 modules	21	240	76	139	7.67	268	11.57	9	0.19
4 modules		>1 h			>1 h	334	14.08	11	0.28
5 modules		>1 h			>1 h	401	16.65	15	0.35
6 modules		>1 h			>1 h	467	18.76	21	0.51

8.6.1 Module-Level Decomposition

The decomposed test generation technique using UMC and module level decomposition is applied to a multi-issue MIPS design model as shown in Fig. 8.2. NuSMV [34] is used to run incremental BMC experiments, and Cadence SMV [35] model checker is used to perform all other experiments. A few simplifications were applied to the MIPS processor model to enable comparison with existing other approaches. For example, if 32-bit registers are used in the register file, the UMC approach cannot produce any counterexample even for a simple property with no pipeline interaction due to the memory depletion during model checking. For comparison purpose, we assume eight 2-bit registers for the following experiments to ensure that the existing approaches can generate counterexamples.

Table 8.2 shows the results of the comparison of test generation techniques for MIPS processor. The first column defines the type of properties used for test generation based on number of module interactions. For example, “None” implies properties applicable to only one module; “Two Modules” implies properties that include two module interactions and so on. Each row presents the average memory requirement (M: Mbytes, K: Kbytes) for the BDD nodes (or CNF clauses) used as well as test generation time (in seconds). For example, the first row presents the average time and memory requirement for 68 ($n = 17$, $r = 4$, and $i = 1$ in Eq. 8.1) single module properties. Similarly, the second row shows the average values for test generation of 2,186 two module interactions and so on. The second and third columns present the average memory (number of BDD nodes) and time requirement for doing UMC using Cadence SMV. The next three columns present the average memory (number of BDD nodes as well as number of CNF clauses) and time requirement for doing counterexample-based abstraction [36] using Cadence SMV. The seventh and eighth columns present the average memory and time requirement for doing incremental BMC using NuSMV. The next two columns presents the results of the decompositional model checking approach—UMC with module-level design and property decompositions.

Clearly, the decompositional model checking approach requires much less memory and test generation time compared to other UMC based approaches. The decompositional model checking approach is also beneficial compared to existing BMC based approaches by providing at least an order-of-magnitude reduction in both test generation time and memory requirement.

8.6.2 Group-Level Decomposition Based on Time Step

Different decomposition techniques on a MIPS processor model can also be applied to test generation using SAT-based BMC. Input property is decomposed into module-level properties and those properties may be activated at different clock cycles, i.e., different time steps during model checking. In group-level decomposition based on time steps, module-level properties belonging to the same time step are grouped into a larger property and their corresponding modules are also grouped into a group modules. SAT-based BMC will take the group-level property, a group of modules, and counterexample bound as inputs.

```
P: F((clk=7) & (MEM.exception = 1)
             & (DIV.exception = 1)
       & ((clk=6) & (FE.opcode = LD)))
```

For example, the above property P can be negated and decomposed into two subproperties for clk $= 7$ and clk $= 6$:

```
P1: G((clk!=7) | (MEM.exception != 1)
               | (DIV.exception != 1))
P2: G((clk!=6) | (FE.opcode != LD))
```

The subproperty $P1$ and a group of two modules (MEM and DIV) will be taken as input of SAT-based BMC. As a result, a partial counterexample $P1'$ for clk=6 will be generated.

```
P1': G((clk!=6) | (decOp[3].opcode != DIV) | (decOp[3].src2Val != 0)
                | (aluOp.opcode != LD) | (aluOp.src1Val != 0))
P2: G((clk!=6)  | (FE.opcode != LD))
```

Because both $P1'$ and $P2$ are applied in the same time step (clk=6), two properties are merged into $P12'$. The property $P12'$ and a group of three modules (Decode, IALU, and Fetch) will be used for input of SAT-based BMC to generate a counterexample for clk=5.

```
P12': G((clk!=6)  | (decOp[3].opcode != DIV) | (decOp[3].src2Val != 0)
                  | (aluOp.opcode != LD) | (aluOp.src1Val != 0)
                  | (FE.opcode != LD))
```

This grouping and merging process will continue until $textclk = 0$ for construction of a final test.

Table 8.3 Comparison of test generation techniques for MIPS processor

Module interactions	SMV (UMC)	SMV (BMC)	Decompositions					
			Module-level decomposition			Group-level decomposition		
			UMC	BMC		UMC	BMC	
				Max. k	Each k		Max. k	Each k
None	165	5.63	0.06	2.24	0.42	0.21	3.87	0.22
2 modules	215	7.42	0.12	6.41	1.38	1.81	4.31	0.43
3 modules	240	7.74	0.19	6.75	1.45	8.06	5.72	0.52
4 modules	>1 h	8.79	0.28	7.63	1.97	37.13	6.98	0.64
5 modules	>1 h	9.29	0.35	9.03	2.18	83.25	8.31	0.62
6 modules	>1 h	9.58	0.51	10.70	2.50	126.01	9.04	0.88

Table 8.3 compares various approaches with the existing UMC and BMC based approaches combined with decomposition techniques. zChaff [22] is used as SAT solver and SMV is used as a model checker. The first and second columns in Table 8.3 have the same meaning as the first and third columns in Table 8.2. The third column presents the test generation time using BMC of Cadence SMV. This case adopted the maximum bound of 45 to ensure to get all the counterexamples. Comparison of the second and third columns shows BMC based test generation is faster than UMC based one.

The next three columns present the test generation times using module-level decomposition in three ways: UMC, BMC with maximum bound, and BMC using bound for each property. An interesting observation is that UMC with module level decomposition provides better performance than SAT-based BMC. This is because unfolding the processor model and converting it to the SAT problem takes some time. In other words, UMC might be better when properties are simple and can be decomposed at the module level.

The last three columns present test generation times using group-level decompositions. As the number of module interaction increases, test generation time using UMC grows exponentially because of exponential increase of UMC state search space. Therefore, BMC based test generation is more beneficial than UMC when many modules should be considered. For example, in six module interaction, UMC requires 126.01 s but BMC spends only 9.04 s, that is more than 90 % time saving. Compared to module-level decomposition, due to grouping modules, the number of calling SAT-based BMC is reduced, and the time per partial counterexample generation is increased. However, in terms of a final test generation time, group-level decomposition shows better performance because it does not need to spend as much as time to resolve partial counterexample conflicts in module-level decomposition.

8.6.3 Discussion: Applicability and Limitations

This chapter has presented various domain-specific (directed test generation) and application-specific (SoCs) heuristics. Clearly, the decompositional model checking approach is applicable only for test (counterexample) generation purposes. In other words, its domain is limited to automated generation of directed tests. However, the application is not limited to only pipelined processors. The approaches presented in this chapter can be used for directed test generation for any software/hardware designs where the interaction between the components are limited. In other words, the modeling, decomposition, and test generation techniques are applicable to any design that is decomposition friendly in the context of counterexample (test) generation. For example, in a SoC design if each component interacts heavily with all the other components in the design, the design and property decomposition may not be feasible. Similarly, in software-based systems if too many global variables are used for interaction between modules and/or there is strong coupling between components, decomposition will not be feasible. In practice, the interaction between components (modules) are well defined and limited in nature due to design-for-verification and other requirements. As a result, a vast majority of designs (hardware, software, or hardware+software) are decomposition friendly, and the design and property decompositions are applicable for directed test generation for those designs.

A directed test generation technique based on the design and property decompositions can alleviate the state space explosion problem in many scenarios. However, this decomposition based method may require manual intervention during composition of partial counterexamples where input requirements generated from children nodes are different or mutually exclusive as described in Sect. 8.5. In Chap. 9, we present a learning-oriented decomposition method that can remove the need for manual intervention.

8.7 Chapter Summary

This chapter presented a test generation methodology that is based on design and property decompositions. An efficient algorithm was developed to merge the partial counterexamples generated by the decomposed properties to create the final test program corresponding to the original property. Experimental results and comparison using MIPS architecture demonstrate the efficiency of the decompositional model checking by generating efficient directed tests.

References

1. Campenhout D, Mudge T, Hayes J (1999) High-level test generation for design verification of pipelined microprocessors. In: Proceedings of design automation conference (DAC), pp 185–188

2. Ho R, Yang C, Horowitz M, Dill D (1995) Architecture validation for processors. In: Proceedings of international symposium on computer architecture (ISCA), pp 404–413
3. Iwashita H, Kowatari S, Nakata T, Hirose F (1994) Automatic test program generation for pipelined processors. In: Proceedings of international conference on computer-aided design (ICCAD), pp 580–583
4. Kohno K, Matsumoto N (2001) A new verification methodology for complex pipeline behavior. In: Proceedings of design automation conference (DAC), pp 816–821
5. Utamaphethai N, Blanton R, Shen J (2000) Effectiveness of microarchitecture test program generation. IEEE Design Test 17(4):38–49
6. Shen J, Abraham J (2000) An RTL abstraction technique for processor microarchitecture validation and test generation. J Elect Test Theory Appl (JETTA) 16(1):67–81
7. Wagner I, Bertacco V, Austin T (2005) StressTest: an automatic approach to test generation via activity monitors. In: Proceedings of design automation conference (DAC), pp 783–788
8. Bryant R (1986) Graph-based algorithms for boolean function manipulation. IEEE Comput Soc 35(8):677–691
9. Clarke E, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge
10. Clarke E, Grumberg O, McMillan K, Zhao X (1995) Efficient generation of counterexamples and witnesses in symbolic model checking. In: Proceedings of design automation conference (DAC), pp 427–432
11. Bjesse P, Kukula J (2004) Using counter example guided abstraction refinement to find complex bugs. In: Proceedings of design automation and test in Europe (DATE), pp 156–161
12. Gargantini A, Heitmeyer C (1999) Using model checking to generate tests from requirements specifications. ACM SIGSOFT Softw Eng Notes 24:146–162
13. Mishra P, Dutt N (2002) Automatic functional test program generation for pipelined processors using model checking. In: Proceedings of high level design validation and test (HLDVT), pp 99–103
14. Mishra P, Dutt N (2004) Graph-based functional test program generation for pipelined processors. In: Proceedings of design automation and test in Europe (DATE), pp 182–187
15. Mishra P, Dutt N (2005) Functional coverage driven test generation for validation of pipelined processors. In: Proceedings of design automation and test in Europe (DATE), pp 678–683
16. Koo H, Mishra P (2006) Functional test generation using property decompositions for validation of pipelined processors. In: Proceedings of design automation and test in Europe (DATE), pp 1240–1245
17. Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y (2003) Bounded model checking. Adv Comput 58:117–148
18. Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Proceedings of tools and algorithms for the construction and analysis of systems (TACAS), pp 193–207
19. Prasad M, Biere A, Gupta A (2005) A survey of recent advances in SAT-based formal verification. Int J Softw Tools Technol Transf (STTT) 7(2):156–173
20. Goldberg E, Novikov Y (2002) BerkMin: a fast and robust SAT-solver. In: Proceedings of design automation and test in Europe (DATE), pp 142–149
21. Marques-Silva J, Sakallah K (1999) Grasp: a search algorithm for propositional satisfiability. IEEE Trans Comput 48(5):506–521
22. Moskewicz M, Madigan C, Zhao Y, Zhang L, Malik S (2001) Chaff: engineering an efficient SAT solver. In: Proceedings of design automation conference (DAC), pp 530–535
23. Amla N, Du X, Kuehlmann A, Kurshan R, McMillan K (2005) An analysis of SAT-based model checking techniques in an industrial environment. In: Proceedings of correct hardware design and verification methods (CHARME), pp 254–268
24. Copty F, Fix L, Fraer R, Giunchiglia E, Kamhi G, Tacchella A, Vardi M (2001) Benefits of bounded model checking at an industrial setting. In: Proceedings of computer aided verification (CAV), pp 436–453
25. Jin H, Somenzi F (2005) An incremental algorithm to check satisfiability for bounded model checking. In: Proceedings of international workshop on bounded model checking (BMC), pp 51–65

26. Strichman O (2001) Pruning techniques for the sat-based bounded model checking problem. In: Proceedings of correct hardware design and verification methods (CHARME), pp 58–70
27. Whittemore J, Kim J, Sakallah K (2001) SATIRE: a new incremental satisfiability engine. In: Proceedings of design automation conference (DAC), pp 542–545
28. Koo H, Mishra P (2006) Test generation using SAT-based bounded model checking for validation of pipelined processors. In: Proceedings of ACM great lakes symposium on VLSI (GLSVLSI), pp 362–365
29. Koo H, Mishra P (2009) Functional test generation using design and property decomposition techniques. ACM Trans Embed Comput Syst (TECS) 8(4):32:1–32:33
30. IEEE Standard for Property Specification Language. http://ieeexplore.ieee.org/servlet/opac?punumber=10222
31. Tuerk T, Schneider K, Gordon M (2007) Model checking PSL using HOL and SMV. In: Proceedings of international Haifa verification conference (HVC 2006), pp 1–15
32. Amla N, Kurshan R, McMillan K, Medel R (2003) Experimental analysis of different techniques for bounded model checking. In: Proceedings of tools and algorithms for the construction and analysis of systems (TACAS), pp 34–48
33. Hennessy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann, San Francisco
34. NuSMV. http://nusmv.irst.itc.it
35. Cadence SMV. http://www-cad.eecs.berkeley.edu/kenmcmil/smv
36. Amla N, McMillan K (2004) A hybrid of counterexample-based and proof-based abstraction. In: Proceedings of formal methods in computer-aided design (FMCAD), pp 260–274

Chapter 9
Learning-Oriented Property Decomposition Approaches

9.1 Introduction

To improve the test generation performance using SAT-based BMC [1], Chap. 5 described several efficient test generation techniques using clustering and learning techniques. In this approach, checking the first (base) property can be a major bottleneck during the test generation, because the base property cannot actively obtain learnings from others to improve its test generation time. Especially when checking a large design with complex properties [i.e., properties with large cone of influence (COI) or deep bounds], BMC-based methods are very costly since large SAT instances indicate long SAT search time.

To further improve the test generation performance, in Chap. 8, we have presented design and property decomposition techniques. By decomposing a complex property into several simple subproperties, and then composing the partial counterexamples derived from subproperties, a directed test for the original property can be generated. Since the test generation time of subproperties is typically several orders of magnitude smaller than the original property, the state space explosion problem can be avoided in many scenarios. However, the composition of tests of subproperties can be a major bottleneck in this method because human intervention and expert knowledge are required to resolve conflicts of partial counterexamples. As an alternative, this chapter presents a learning-oriented decomposition technique shown in Fig. 9.1b, which can be fully automated. Unlike the integration-based test generation, this approach is based on the decision ordering learned during the test generation of decomposed subproperties. As described in Chap. 6, such learnings can be used to drastically accelerate the original property falsification. Therefore the overall test generation effort can be significantly reduced.

The learning-based directed test generation is also based on the decomposition framework presented in Fig. 8.1. The inputs to this framework are the design specification and required properties. To reduce the test generation complexity, there are three important steps. First, the property decomposition techniques are used to

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_9
© Springer Science+Business Media New York 2013

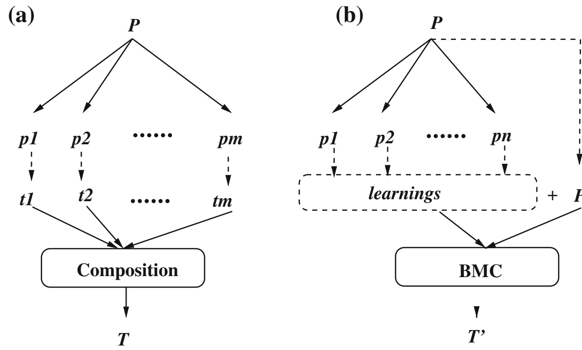


Fig. 9.1 Two property decomposition techniques. **a** Integration-based decomposition, **b** Learning-oriented decomposition

reduce the complexity during property falsification. Next, by checking the selected profitable subproperties, we can collect useful learnings for the original property checking. Finally, the learned knowledge (i.e., decision ordering) can be utilized to avoid the unnecessary conflicts during the original test generation. Therefore, the test generation time can be drastically reduced.

The rest of the chapter is organized as follows. Section 9.2 presents the related work on decomposition as well as learning techniques. Section 9.3 proposes two novel property decomposition methods based on learning techniques. Section 9.4 presents the decision ordering based learning techniques for original property checking. Section 9.5 describes how to utilize the learned knowledge from the decomposed properties for test generation. Section 9.7 presents case studies using both hardware and software designs. Finally, Sect. 9.8 summarizes the chapter.

9.2 Related Work

To overcome the complexity issues during design and verification, various decomposition techniques have been proposed. Case et al observed that some signals get fixed to constant values after some time frames [2]. They presented automated techniques to detect and eliminate redundancy related to transient signals and initialization inputs, which enable verification efficiencies in terms of logic reduction. Lin et al. [3] proposed a new formulation of Ashenhurst decomposition based on SAT solving. It can efficiently decompose a Boolean function into a network of smaller sub-functions. However, this method is mainly used in logic synthesis. As described in Chap. 8, although Koo et al. [4] presented a promising design and property decomposition method for test generation, it is hard to automate the composition of generated partial tests corresponding to the decomposed properties.

Sharing learnings across properties can improve overall performance since the repeated validation efforts can be avoided. Due to the commonality between different SAT instances of a property, Strichman [5] found that the conflict clauses can be replicated and forwarded as a constraint. Based on this observation, incremental SAT solvers [5, 6] are developed to reuse the learned conflict clauses from lower bound SAT instances to prune the SAT search for larger bound SAT instances. In [7], Chen and Mishra noticed that when checking a large set of relevant properties, SAT instances of similar properties have a large overlap of CNF clauses and can be clustered (described in Chap. 5). A large number of conflict clauses generated by a base property can be forwarded to other properties in the cluster. As an alternative of conflict clause forwarding, decision ordering heuristics [8] can be used as another learning to improve the SAT searching. In [9], Strichman presented an BMC optimization based on decision ordering by exploiting the characteristics of BMC formulas. Wang et al. [10] analyzed the correlation among different SAT instances of a property. They used the *unsatisfiable core* of previously checked SAT instances to guide the variable ordering for the current SAT instance. When checking a set of similar properties, Chen et al. [11] tuned the decision ordering of the current property based on the decision ordering results of the previously checked properties (described in Chap. 6). By sharing learnings (i.e., conflict clauses or decision ordering) among properties, the overall test generation time can be reduced. However, these learning techniques do not consider how to actively learn from other simpler properties. For example, when sharing learnings among a cluster of similar properties, checking the first property will be a major bottleneck because there is no knowledge that can be learned.

In contrast to conventional methods using learnings, the approach [12] presented in this chapter is the first attempt to use decision ordering based learning in property decomposition to enable automated test generation.

9.3 Learning-Oriented Property Decomposition

This chapter focuses on efficient falsification of safety linear temporal logic (LTL [13]) properties which consist of temporal operators (F, G, X, U) and Boolean connectives ($\wedge$, $\vee$, $\neg$ and $\rightarrow$). The basic idea is to utilize the learnings from simple properties for complex property checking.

9.3.1 Spatial Property Decomposition

For a complex property which involves multiple components of the design, it can be partitioned into multiple component-level subformulas. For example, a complex system-level property P can be broken into 2 subproperties P_1 and P_2 with different COI. Assuming that P_1 has a smaller COI than P, it usually needs less time and space

than that of checking the complex property P. If the partial counterexample generated by P_1 can be refined to guide the complex property falsification, the original property is *spatially decomposable*.

Definition 9.1 A false property P in conjunctive form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ or in disjunctive form $p_1 \vee p_2 \vee \ldots \vee p_n$ is spatially decomposable if all of the following conditions are satisfied.

- If the decomposed properties are in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$, then at least one property p_i $(1 \leq i \leq n)$ has a counterexample. In this case, the bound of P is the minimum bound of p_i which has a counterexample.
- If the decomposed properties are in the form $p_1 \vee p_2 \vee \ldots \vee p_n$, then each property p_i $(1 \leq i \leq n)$ has a counterexample. In this case, the bound of P is the maximum bound of all decomposed properties.
- The counterexamples generated from properties p_i $(1 \leq i \leq n)$ can guide the test generation for property P. ■

According to Definition 9.1, the following rules can be used for complex property decomposition.

$$\begin{aligned} \neg X(p \vee q) &\equiv \neg X(p) \wedge \neg X(q) \\ \neg X(p \wedge q) &\equiv \neg X(p) \vee \neg X(q) \\ \neg F(p \vee q) &\equiv \neg F(p) \wedge \neg F(q) \end{aligned} \tag{9.1}$$

The property in the form of $\neg F(p \wedge q)$ and $\neg F(p \rightarrow q)$ cannot be directly decomposed into conjunctive or disjunctive form. However, by introducing a clock clk for synchronization, they can be spatially decomposed (see the proof in Lemma 8.6). It is important to note that the value of the clk indicates the bound of the false property. Equation (9.2) shows that the counterexample of $\neg F(p \wedge q \wedge clk = k)$ can be refined by the counterexamples of $\neg F(p \wedge clk = k)$ and $\neg F(q \wedge clk = k)$.

$$\begin{aligned} &\neg F(p \wedge q \wedge clk = k) \equiv \neg F(p \wedge clk = k) \vee \neg F(q \wedge clk = k) \\ &\text{where} \neg F(p \wedge q \wedge clk = k) \text{is false.} \end{aligned} \tag{9.2}$$

For a property in the form $F(p \rightarrow q)$, p describes the precondition and q indicates the postcondition. When $G(\neg p)$ holds, $F(p \rightarrow q)$ will be vacuously true, and the checking of $\neg F(p \rightarrow q)$ will report a counterexample without satisfying the precondition p. This counterexample may not match the original intention. Equation 9.3 shows that the properties in the form of $\neg F(p \rightarrow q \wedge clk = k)$ can be transformed into $\neg F(p \wedge q \wedge clk = k)$ for test generation. The spatial decomposition in Eqs. (9.2) and (9.3) are similar.

$$\begin{aligned} \neg F(p \rightarrow q \wedge clk = k) &\equiv \neg F(p \wedge q \wedge clk = k) \\ &\equiv \neg F(p \wedge clk = k) \vee \neg F(q \wedge clk = k) \end{aligned} \tag{9.3}$$

where $\neg F(p \rightarrow q \wedge clk = k)$ and $\neg F(p \wedge clk = k)$ are false.

Equations (9.1)–(9.3) present several kinds of widely used properties which can be spatially decomposed. In fact, if a complex property can be decomposed in conjunctive form, it needs to sort the subproperties according to their bounds, and check them from the smallest to the largest bounds. The counterexample of the first falsified property can be used as a counterexample for the complex property.

When checking a complex property which can be decomposed in disjunctive form, it is not necessary to check all its subproperties. This is because, if the COI of a subproperty is similar to the original property, the complexity of such subproperty will be similar to the complex property. In this case, it is not economical to use learning. Therefore, it needs to figure out subproperties with smaller COI than the complex property.

According to the commutative law and associative law, for a complex property, we can classify its atomic subproperties into several clusters. For example, in Eq. 9.4, p_i and p_k are clustered together, and p_j belongs to another cluster.

$$p_i \vee p_j \vee p_k = (p_i \vee p_k) \vee p_j \tag{9.4}$$

For each cluster, we generate a *refined property* which represents all the atomic subproperties in the cluster to derive the learning. The following clustering rules based on experience work well for most of the time: (1) in each cluster, all the variables in the subformulas should come from the same component (e.g., fetch module in a processor); (2) in each cluster, all the subformulas should describe the related functional scenarios (e.g., fetching instructions and/or data in a processor).

Algorithm 1 outlines the spatial decomposition method which can derive a set of refined subproperties with small COI for learning. The inputs of the algorithm are a design model D and a complex property P in disjunctive form. Step 1 initializes the SD_props with an empty set. Step 2 tunes subproperties' order according to the commutative law and clusters subproperties using the similarity rules. Step 3 selects the ith cluster. If the COI of such cluster is smaller than $\frac{k}{n}$ of $P's$ COI, step 4 will generate a new refined property $newP$ for the ith cluster. Step 5 adds $newP$ to SD_props. The refined property $newP$ for learning represents a cluster of subproperties as shown in step 3. Finally, this algorithm will return a set of refined subproperties for deriving learnings (described in Sect. 9.4). Since the COI of a refined property in SD_props is small, its test generation time will be much smaller than that of the original complex property. It is important to note that this algorithm may return an empty set which means that it is not beneficial or impossible to spatially decompose the complex property.

Algorithm 1: Spatial Decomposition

Input: i) The design model, D
ii) A property P in the form $p_1 \vee p_2 \vee \ldots \vee p_n$
Output: A set of refined subproperties for learning, SD_props
1. $SD_props = \{\}$;
2. $(cluster_1, \ldots, cluster_m) = clustering(P, modular/functional)$;
for i *is from 1 to* m **do**
  3. $cluster_i = \{prop_1, \ldots, prop_k\}$;
  if $COI(cluster_i) \leq \frac{k}{n} COI(P)$ **then**
    4. generate a refined property $newP$ for the $cluster_i$;
    5. $SD_props = SD_props \bigcup newP$;
  end
end
return SD_props

9.3.2 Temporal Property Decomposition

To eclipse the bound effect, one intuitive way is to deduce a long bound property from a sequence of short bound properties. For example, $P1$, $P2$, and $P3$ ($P3 = P$) are properties indicating three different stages of property P. Their bounds are $K1$, $K2$, and $K3$, respectively, and $K1 < K2 < K3$. Because $P1$'s counterexample is similar to the prefix of the $P2$'s counterexample, $P1$'s counterexample contains rich knowledge that can be used when checking $P2$. Similarly, during the property checking, $P3$ can benefit from $P2$. Therefore we can quickly obtain the counterexample (test) for property P. If the counterexamples of lower bound properties can be used to reason about P, the property P is temporally decomposable.

Definition 9.2 A false safety property P is temporally decomposable if all the following conditions are satisfied.

- P can be divided into false properties $p_1, p_2, \ldots$ and p_n ($P = p_n$) with increasing bounds.
- $\neg p_i \rightarrow \neg p_{i+1}$ ($1 \leq i \leq k_n - 1$), which indicates that the counterexample generated from properties p_i can guide the test generation for property p_{i+1}. ■

In temporal decomposition, finding the implication relation ("$\rightarrow$") between properties is a key process. In the framework developed by [12], such implication relations are constructed by exploring the order between events, which are described by properties indicating different stages of the execution. Generally, a system behavior consists of a sequence of strongly relevant *events*. For example, in Fig. 9.2, there are 9 events. We classify the relation between these events in two categories. The cause–effect relation (marked by $\Rightarrow$) defines the relation of consequent events. For example, if $e1$ happens, then $e2$ should happen in the next stage. The happen before

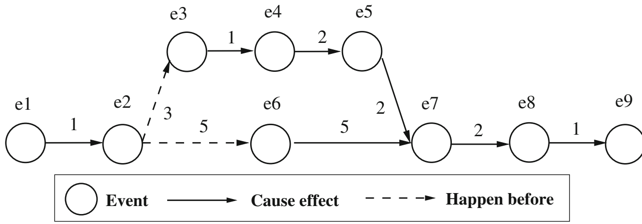


Fig. 9.2 A DAG of event relation

relation (marked by $\prec$) specifies the relation of conditional events. It indicates which events may happen before other events under some condition. For example, $e2 \prec e3$ means $e2$ may happen before $e3$.

During test generation, properties in the form of $\neg F(e)$ are used to indicate that the event e cannot be activated. According to definition 2, the "$\Rightarrow$" relation can be used to derive helpful learnings. For example, in Fig. 9.2, let property $P_1 = \neg F(e_1)$ and property $P_2 = \neg F(e_2)$. Since $e_1 \Rightarrow e_2$ implies $F(e_1) \rightarrow F(e_2)$, i.e., $\neg P_1 \rightarrow \neg P_2$, it shows that $P1$'s counterexample will be helpful for deriving $P2$'s counterexample. Such information can be used as a learning. The "$\prec$" relation can also be used to indicate the learning information. Assuming $e_2 \prec e_3$, the counterexample of $\neg F(e_2)$ is shorter than the counterexample of $\neg F(e_3)$. However, the counterexample of $\neg F(e_2)$ may have a large overlap of variable assignments with the counterexample of $\neg F(e_3)$. Therefore, the learning from $\neg F(e_2)$ can benefit the test generation of $\neg F(e_3)$.

When checking a large bound property for a transaction, there may be many events along the path to target events. Checking all these events to obtain learnings is time-consuming. For example, assuming that we want to check the property $\neg F(e_9)$, the relation between events is described using a directed acyclic graph (DAG) shown in Fig. 9.2. Each node indicates an event, and each directed edge indicates the relation of "$\Rightarrow$" or "$\prec$", and each edge is associated with the delay between events. In this DAG, there are eight events that happen before $e9$. However, it is not necessary to check all of them.

Since the branch nodes of a DAG indicate critical variable assignment information, it only needs to consider the events which determine the branches along the path from initial state $e1$ to the target state $e9$.

Algorithm 2: Temporal Decomposition

Input: i) An event DAG, D
ii) Initial event src, target event $dest$
Output: A property sequence TD_props
1. $path$ = Dijkstra(D, src, $dest$) to find the shortest delay path;
2. TD_props = (property for src);
for *i is from 2 to len (number of events in path)* **do**
    3. $(e_{i-1}, e_i) = (i-1)^{th}$ edge of $path$;
    if $out_degree(e_{i-1}) + in_degree(e_i) > 2$ **then**
        4. Append the property for e_i to TD_events;
    end
end
return TD_props

Algorithm 2 describes how to obtain a sequence of properties based on temporal decomposition. It accepts an event DAG with the initial and target events as inputs. Step 1 uses Dijkstra's algorithm [14] to find a shortest $path$. Step 2 initializes the sequence *TD_props* with a property for the initial event. Steps 3 and 4 select the branch events and append their corresponding properties to the *TD_props*. Finally the algorithm reports the property sequence for deriving learnings. By using this algorithm, $(\neg F(e1), \neg F(e3), \neg F(e7))$ is a property sequence from the temporal decomposition in Fig. 9.2.

9.4 Decision Ordering Based Learning Techniques

SAT-based BMC encodes a property checking problem into a SAT instance (a Boolean formula). A counterexample of the property is a satisfying variable assignment for this formula. Although the variable assignment of counterexamples derived from the decomposed subproperties may not satisfy the SAT instance of the complex property, it has a large overlap with the complex property on the variable assignment. Such information can be used as a learning to bias the decision ordering when checking the complex property.

During the SAT search, decision ordering plays an important role to quickly find a satisfying assignment. The learning approach presented in this chapter is based on variable state-independent decaying sum (VSIDS) method [15]. A major difference is that this learning method incorporates the statistics of decomposed properties. Since different subproperties have different bounds, such information needs to be considered in the learning heuristics.

Let bounds be an array which stores the bound of k subproperties. Because in spatial method the decomposed subproperties can be independent, the learning between subproperties is not significant. So we set bounds$[i] = 1 (1 \leq i \leq k)$. However, for temporal decomposition, the vstat information (introduced later) of lower bound

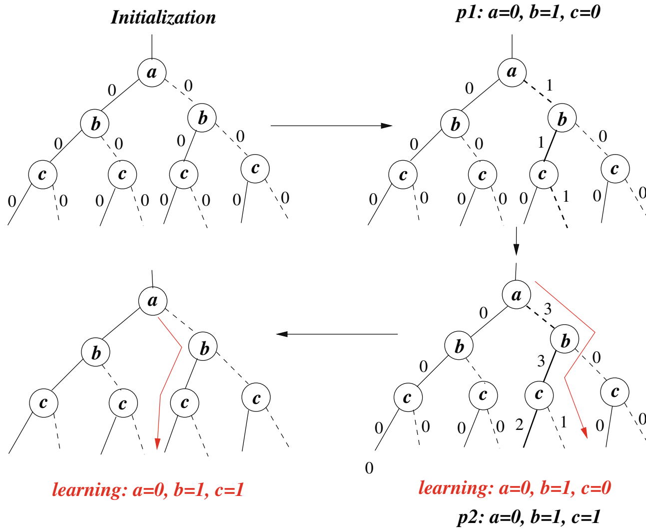


Fig. 9.3 Learning statistics applied on decision trees

properties can further benefit the larger bound property checking. Moreover, the larger bound subproperty is closer to the final properties than smaller bound subproperties. Therefore, for temporal decomposition-based method, the subproperties are sorted according to the increasing bounds, and bounds[i] indicates the bound of ith property. Let vstat[$sz+1$][2] (sz is the maximum Boolean variable index during the complex property checking) be a 2-D array to record the statistics of variable assignments. Initially, vstat[i][0] $=$ vstat[i][1] $= 0$ ($0 < i \leq sz$). vstat will be updated after checking each subproperty. When checking the subproperty p_j, if a variable v_i is evaluated and its value in the counterexample is 0 (false), vstat[i][0] will be increased by bounds[j]; otherwise if $v_i = 1$ (true), vstat[i][1] will be increased by bounds[j].

Assuming l_i is a literal of the variable v_i (v_i has two literals, v_i and v_i'), we use score(l_i) to indicate its decision ordering. Initially, score(l_i) is equal to the literal count of l_i. However, at the beginning of SAT searching and periodic score decaying, the literal score will be recalculated. Let

$$\text{bias} = \frac{\text{MAX}(\text{vstat}(v_i), \text{vstat}(v_i')) + 1}{\text{MIN}(\text{vstat}(v_i), \text{vstat}(v_i')) + 1}$$

indicate the variable assignment variance. And let

$$\text{score}(l_i) = \begin{cases} \max(v_i) * \text{bias} & (\text{vstat}[i][1] > \text{vstat}[i][0] \& l_i = v_i) \\ & \text{or } (\text{vstat}[i][1] < \text{vstat}[i][0] \& l_i = v_i') \\ \text{score}(l_i) & \text{otherwise} \end{cases}$$

The new literal score will be updated using the above formula where $\max(v_i) = \text{MAX}(\text{score}(v_i), \text{score}(v_i')) + 1$.

Figure 9.3 shows an example of temporal decomposition using the proposed heuristic. The complex property P is decomposed into three properties p_1, p_2, and $p_3(=P)$ with bounds 1, 2, and 3 respectively, and we assume that we always check the variables in the order of a, b, c. Initially, when checking p_1, there is no learning information. However, after checking p_1, we can predict the decision ordering for p_2 based on the collected vstat information from p_1. Also, we can predict the decision ordering of $p_3(=P)$ from the vstat of p_1 and p_2. When checking P, the content of vstat indicates that variables a is more likely to be 0, b and c are more likely to be 1.

9.5 Test Generation Using Decomposition and Learning Techniques

Algorithm 3 describes the test generation methodology using both property decomposition and learning techniques. The inputs of the algorithm are a formal model of the design, a set of decomposed properties props and their satisfiable bounds bounds, and the complex property P with its satisfiable bound bound_p. Step 1 generates CNF files in the DIMACS format [16] for each decomposed property in props. Step 2 sorts the CNFs by their DIMACS file size. Step 3 initializes vstat which is used to keep statistics of the variable assignments for decomposed subproperties. Then for each decomposed subproperty, its counterexample assignments are collected from steps 4 to 5. During each iteration, the vstat statistics is updated. In steps 6 and 7, the complex property P is checked using the decision ordering derived from the decomposed subproperties. Finally, the algorithm reports a test for the complex property P.

Algorithm 3: Test Generation Using Decomposition and Learning Approaches

Input: i) Formal model of the design, D
    ii) Decomposed properties $props$ and satisfiable $bounds$
    iii) The property P and its satisfiable bound $bound_p$
Output: A test $test_P$ for P
1. $CNFs = BMC(D, props, bounds)$;
2. $(CNF_1, \ldots, CNF_n)$ = sort $CNFs$ using increasing file size;
3. Initialize $vstat$;
for *i is from 1 to the n* **do**
    4. $test_i$ = SAT($CNF_i, vstat$);
    5. $Update(vstat, test_i, bounds[i])$;
end
6. Generate $CNF = BMC(D, P, bound_P)$;
7. $test_P$ = SAT($CNF, vstat$);
return $test_P$

9.6 An Illustrative Example

This section presents an example of how to apply decomposition methods on the graph model of an MIPS processor [7] (described in Sect. 2.2.1). It consists of five pipeline stages: fetch, decode, execute, memory, and write back. It has four pipeline paths in the execute stage: ALU for integer ALU operation, FADD for floating-point addition operation, MUL for multiply operation, and DIV for divide operation. Assume that we want to check a complex scenario that the units $MUL5$ and $FADD3$ will be active at the same time. We generate the property $P = \sim F(MUL5.active = 1 \;\&\; FADD3.active = 1)$ which is a negation of the desired behavior. The remainder of this section describes how to generate a test using spatial and temporal decomposition methods respectively.

9.6.1 Spatial Decomposition

In the MIPS design, each functional unit has a delay of one clock cycle. To trigger the functional unit $MUL5$, we need at least seven clock cycles (there are seven units along the path $Fetch \rightarrow Decode \rightarrow \cdots \rightarrow$ MUL5). Similarly, to trigger the functional unit $FADD3$, we need at least five clock cycles, plus one clock cycle for initialization, we need a total of eight clock cycles for triggering this interaction. Thus the bound of this property is eight. According to Eq. (9.2) and Algorithm 1, property P can be spatially decomposed into two subproperties as follows, assuming the COI of $P1$ and $P2$ are both smaller than half of COI of P.

```
/* Modified original complex property P' */
P': ~F(MUL5.active=1 & FADD3.active=1 & clk=8)
/* Spatially decomposed properties */
P1: ~F(MUL5.active=1 & clk=8)
P2: ~F(FADD3.active=1 & clk=8)
```

When checking $P1$ and $P2$ individually, we can get the following two counterexamples.

```
   Counterexamples for P1 and P2
Cycles  P1's Inst.        P2's Inst.
  1     NOP               NOP
  2     MUL R2, R2, R0    NOP
  3     NOP               NOP
  4     NOP               FADD R1, R1, R0
```

However, according to Algorithm 3, the test generation for $P2$ is under the guidance of $P1$'s result. Thus, the counterexample of $P2$ guided by $P1$ contains $P1$'s

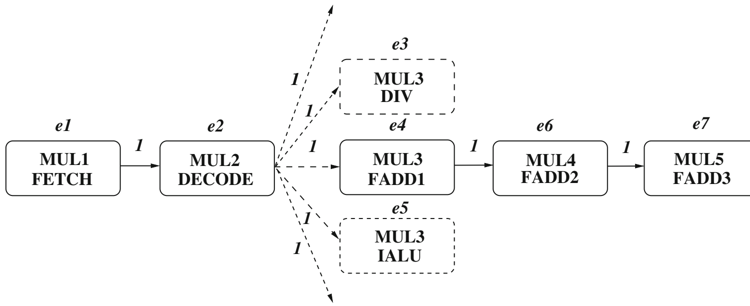


Fig. 9.4 Event implication graph for property P

partial behavior (see clock cycle 2 below). So the score of literals which have repetitive occurrences is enhanced.

```
Counterexample for P2 guided by P1
Cycles    P2's Inst. after learning
  1          NOP
  2          MUL R2, R2, R0
  3          NOP
  4          FADD R1, R1, R0
```

The statistics saved in vstat indicates an assignment which has a large overlap of the assignments with the counterexample that can activate property P. Thus it can be used as the decision ordering learning to guide the property checking of P.

9.6.2 *Temporal Decomposition*

Temporal decomposition requires figuring out event relation first. Since we want to check property $\sim F(MUL5.active = 1 \ \& \ FADD3.active = 1)$, the target event is $MUL5.active = 1 \ \& \ FADD3.active = 1$. Figure 9.4 shows the event implications. There are seven events in this graph, and e_7 is the target event.

Assuming $e1$ is the initial event, from $e1$ to $e7$, there is only one path $e_1 \rightarrow e_2 \rightarrow e_4 \rightarrow e_6 \rightarrow e_7$. Along this path there is a branch node e_2. According to Algorithm 2, we need to check two events e_1 and e_4 using the following properties. By using our learning technique, during the test generation, P_e4 can benefit from P_e1, and P can benefit from P_e4.

```
/* Temporally decomposed properties*/
P_e1: ~F(FETCH.active=1 & MUL1.active=1)
P_e4: ~F(MUL3.active=1 & FADD1.active=1)
```

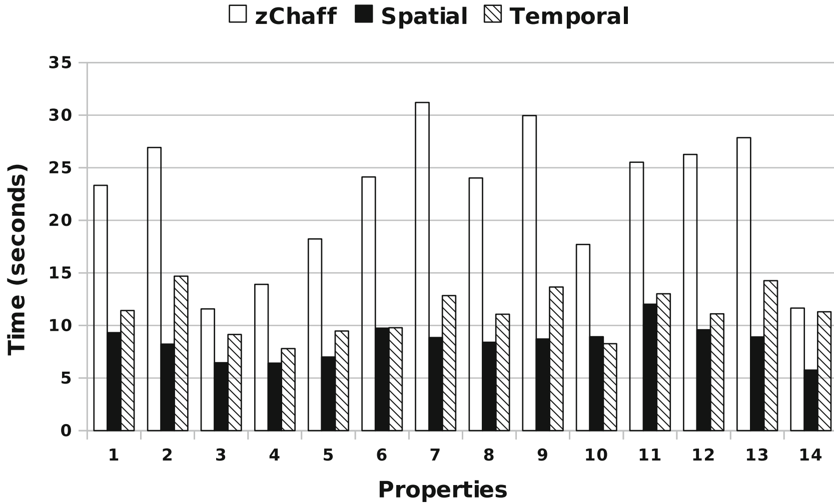


Fig. 9.5 Test generation results for MIPS processor

9.7 Case Studies

This section presents two case studies: a MIPS architecture [17] (described in Sect. 2.2.1) and a stock exchange system (described in Sect. 2.4.4). A tool [12] is developed, which takes a graph model of the design as an input for property decomposition. Based on the analysis of graph model, the process of bound determination and property decomposition can be automated in this tool. In the experiments, the cost of property decomposition was not considered since it is small (less than 0.01s). For test generation, NuSMV [18] was used to derive the CNF clauses (in DIMACS format) and integrated the proposed methods in the SAT solver zChaff [16]. The experimental results were obtained on a Linux PC using 2.0GHz Core 2 Duo CPU with 1 GB RAM.

9.7.1 A MIPS Processor

This section presents the experimental results using the same design illustrated in Sect. 9.6. For the MIPS design, we are focusing on the test generation of interaction faults. The properties were generated in the form of $\neg F(p_1 \& p_2 \& \ldots \& p_n)$ which indicate whether n pipelined components p_i $(1 \leq i \leq n)$ can be activated at the same time. For example, the property $\sim F(MUL6.active = 1 \& FADD3.active = 1 \& DIV.active = 1)$ asserts that there is no instruction sequence which can activate the components $MUL6$, $FADD3$, and DIV at the same time.

Table 9.1 Decomposition results for MIPS processor

Property (tests)	zChaff [16] (s)	Cluster #	Refined #	Spatial (s)	Speedup [16] versus spatial
p_1	127.52	3	2	49.41	2.58
p_2	49.24	3	2	15.73	3.13
p_3	9.18	2	1	4.99	1.84
p_4	13.78	2	1	7.28	1.89
p_5	31.63	3	2	12.74	2.48
p_6	120.72	3	2	54.21	2.23

A set of 20 properties were generated based on various interaction faults. Since it is hard to figure out the temporal relation between events, six properties of them cannot be handled by temporal decomposition. Table 9.1 shows the test generation results using the spatial decomposition approach for such properties. The first column indicates the selected properties. The second column gives the test generation time using zChaff. The third and fourth columns present the number of subproperty clusters and the number of refined subproperties for deriving learnings. The last two columns show test generation time using the spatial decomposition method (including the overhead of subproperty checking) and its improvement over the method using zChaff. Compared with the method without any learnings (column 2), the spatial decomposition-based learning method can drastically reduce the test generation time.

For the remaining 14 properties, both spatial and temporal decomposition are applied individually. Figure 9.5 illustrates the performance improvement over the method using zChaff. It shows that the temporal method can drastically reduce test generation time (2–4 times). Although spatial decomposition outperforms temporal decomposition in this case study, comparing with the method using zChaff, temporal decomposition can still have significant improvement (2.5 times).

9.7.2 A Stock Exchange System

The online stock exchange system (OSES) is a software which mainly deals with stock order transactions. The specification of OSES is described by a UML activity diagram which contains 27 activities and 29 transitions. We extract the formal model from the UML specification and transform it into a NuSMV description. We generate 18 complex properties to check various stock transactions. Each transaction is a sequence of activities (events). The test generation for a transaction using only one complex property is time-consuming. So we temporally decomposed the transaction into several stages which specify the branch activities along the path, and for each stage we create a subproperty.

Table 9.2 Decomposition results for OSES

Property (tests)	zChaff [16] (s)	Bound	Decomposed #	Temporal (s)	Speedup [16] versus temporal
p_1	25.99	8	3	0.78	33.32
p_2	48.99	10	4	2.69	18.21
p_3	39.67	11	5	3.45	11.50
p_4	247.26	11	5	22.46	11.01
p_5	160.73	11	5	15.68	10.25
p_6	97.54	11	4	1.56	62.53
p_7	31.39	10	4	12.31	2.55
p_8	161.74	11	4	12.62	12.82
p_9	142.91	10	4	17.57	8.13
p_{10}	33.77	10	4	1.76	19.19

Among the 18 complex properties, 10 of them are time-consuming (more than 10 s without using the proposed method). Table 9.2 shows the test generation results for these ten properties using temporal decomposition. The first column indicates the property. The second column indicates the test generation time using zChaff without any decomposition and learning techniques. The third column presents the bound of the complex property. The fourth column indicates the number of temporal subproperties decomposed along the stock transaction flow. The last two columns indicate the test generation time (using temporal decomposition) and its speedup over zChaff. In this case study, the proposed approach can produce around 3–63 times improvement compared to with the method using zChaff.

9.8 Chapter Summary

To address the test generation complexity of a single complex property using SAT-based BMC, this chapter presented a novel method which combines the property decomposition and learning techniques. By decomposing a complex property spatially and temporally, we can get a set of subproperties whose counterexamples can be used to predict the decision ordering for the complex property. Because of the learning from the simple subproperties to the complex property, the overall test generation effort can be reduced. The case studies demonstrated the effectiveness of the proposed method using both hardware and software designs that generated significant savings (2–63 times) in test generation time.

References

1. Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y (2003) Bounded model checking. Adv Comput 58:117–148
2. Case ML, Mony H, Baumgartner J, Kanzelman R (2009) Enhanced verification by temporal decomposition. In: Proceedings of international conference on formal methods in computer-aided design (FMCAD), pp 17–24
3. Lin H, Jiang J, Lee R (2008) To SAT or not to SAT: Ashenhurst decomposition in a large scale. In: Proceedings of international conference on computer-aided design (ICCAD), pp 32–37
4. Koo H, Mishra P (2009) Functional test generation using design and property decomposition techniques. ACM Trans Embed Comput Syst 8(4):32:1–32:33
5. Strichman O (2001) Pruning techniques for the SAT-based bounded model checking problem. In: Proceedings of correct hardware design and verification methods (CHARME), pp 58–70
6. Jin H, Somenzi F (2004) An incremental algorithm to check satisfiability for bounded model checking. In: Proceedings of BMC, pp 51–65
7. Chen M, Mishra P (2010) Functional test generation using efficient property clustering and learning techniques. IEEE Trans Comput Aided Des Integr Circuits Syst 29(3):396–404
8. Marques-Silva JP, Sakallah KA (1999) The impact of branching heuristics in propositional satisfiability. In: Proceedings of the 9th portuguese conference on artificial intelligence, pp 62–74
9. Shtrichman O (2000) Tuning SAT checkers for bounded model checking. In: Proceedings of the The international conference on computer aided verification (CAV), pp 480–494
10. Wang C, Jin H, Hachtel GD, Somenzi F (2004) Refining the SAT decision ordering for bounded model checking. In: Proceedings of design automation conference (DAC), pp 535–538
11. Chen M, Qin X, Mishra P (2010) Efficient decision ordering techniques for SAT-based test generation. In: Proceedings of design, automation and test in Europe (DATE), pp 490–495
12. Chen M, Mishra P (2011) Decision ordering based property decomposition for functional test generation. In: Proceedings of design, automation and test in Europe (DATE), pp 167–172
13. Clarke E, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge
14. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numerische mathematik 1:269–271
15. Moskewicz M, Madigan C, Zhao Y, Zhang L, Malik S (2001) Chaff: engineering an efficient SAT solver. In: Proceedings of design automation conference (DAC), pp 530–535
16. zChaff (2004) http://www.princeton.edu/~chaff/zchaff.html
17. Hennessy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann Publishers, San Francisco
18. NuSMV. http://nusmv.irst.itc.it

Chapter 10
Directed Test Generation for Multicore Architectures

10.1 Introduction

When SAT-based BMC is applied to generate directed tests for multicore architectures, there are two different categories of symmetry in the corresponding SAT instances. The first category is the "temporal" symmetry. It occurs because the SAT instance is encoded by unrolling the same architecture multiple times. This regularity has already been exploited by existing research [19] to accelerate the SAT solving process. On the other hand, the structural similarity of multiple cores also introduces a second category of symmetry or "spatial" symmetry. This symmetry appears among the CNF clauses for different cores at the same time step. Intuitively, the spatial symmetry can also be exploited by reusing the knowledge obtained from one core to other cores. Unfortunately, this intuitive reasoning is hard to implement because it is very difficult to reconstruct the symmetry from the CNF formula. The high-level information is lost during CNF synthesis, and it is inefficient as well as computationally expensive to recover through "reverse engineering" methods.

This chapter addresses the directed test generation for multicore architectures by developing a novel BMC-based test generation technique, which enables the reuse of learned knowledge from one core to the remaining cores in the multicore architecture. Instead of direct synthesis of the CNF for the multicore design, the CNF description of the entire design is composed using CNF formulae for cores and the memory subsystem. Since the CNF representation of cores are generated by performing variable substitution of the CNF for one of them, the correct mapping information is easily obtained. In this way, it is possible to translate and reuse the conflict clauses learned on any core to other cores. It can be proved that the composed CNF description has the same satisfiability as the original methods. The experimental results demonstrate that this approach can remarkably reduce the overall test generation time.

The rest of the chapter is organized as follows. Sect. 10.2 describes related test generation techniques. Section 10.3 describes the test generation methodology for multicore architectures. Sect. 10.4 presents the experimental results. Finally, Sect. 10.5 summarizes the chapter.

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_10,

© Springer Science+Business Media New York 2013

10.2 Related Work

A great deal of work has been done to reduce the SAT solving time during BMC [12, 15, 19, 21]. The basic idea is to exploit the regularity of the SAT instances between different bounds. For example, incremental SAT solvers [12, 21] reduce the solving time by employing the previously learned conflict clauses. Generated conflict clauses are kept in the database as long as the clauses which led to the conflicts are not removed. Strichman [19] observed that if a conflict clause is deduced only from the transition part of a SAT instance, it can be safely forwarded to all instances with larger bounds. This is because the transition part of the design will still be in the SAT instance when the design is unrolled. Besides, the learned conflict clause can also be replicated across different time steps. However, the existing approaches did not exploit the symmetric structure within the same time step. In directed test generation for multicore architectures, the same knowledge about the core structure needs to be rediscovered for each core independently, which can lead to significant wastage of computational power.

When BMC is applied in circuits, Kuehlmann [13] proposed that the unfolded transition relation can be simplified by merging vertices that are functionally equivalent under given input constraints. In this way, the complexity of transition relation is greatly reduced. However, since this technique was developed based on the And-inverter graph (AIG) representation of logic designs, it is difficult to use it to accelerate the solving process of CNF instances that are directly created from high-level specifications.

Some researchers have shown that validation based on high-level specification is effective. For example, Bhadra et al. [4] used executable specification to validate multiprocessor systems-on-chip designs. Mishra et al. [15] proposed directed test generation based on high-level specification. To accelerate the test generation process, conflict clauses learned during checking of one property are forwarded to speedup the SAT solving process of other related properties, although the bound is required as an input. When a SAT instance contains symmetric structure, symmetry breaking predicate [1, 3, 9, 14, 20] can be used to speedup the SAT solving by confining the search to non-symmetric regions of the space. By adding symmetry breaking predicates to the SAT instance, the SAT solver is restricted to find the satisfying assignments of only one representative member in a symmetric set. However, this approach cannot effectively accelerate the directed test generation for multicore processors, because the properties for test generation are usually not symmetric with respect to each core. Thus, the symmetric regions in the entire space are usually small despite the fact that the structure of each core is identical. On the other hand, in component analysis for SAT solving, Biere et al. [6] proposed that each component can be solved individually to accelerate the solving process. However, the symmetric structure is not used at the same time for further speedup.

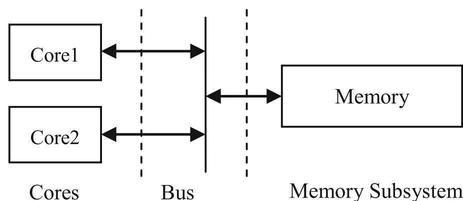


Fig. 10.1 Abstracted architecture of a two-core system

10.3 Test Generation for Multicore Architectures

The basic idea of the test generation for multicore architecture (TGMA) is motivated by previous works on incremental SAT-based BMC [19]. Based on the temporal symmetry between different bounds, these methods accelerate the SAT solving process by passing the knowledge (deduced conflict clauses) in the temporal direction. Nevertheless, the SAT instances generated by multicore designs also exhibit remarkable spatial symmetry. Figure 10.1 depicts the high-level structure of a system with 2 cores. Both cores are identical[1] and connected to memory subsystem with a bus. Figure 10.2 shows the SAT solving process when performing BMC for bounds 0, 1, 2, and 3 on this multicore architecture using the technique proposed in [19]. We use solid dots to represent different SAT instances and lines to indicate the conflict clause forwarding paths. Although different cores have identical structures, this spatial symmetry is not exploited.

Intuitively, it should be beneficial if the knowledge or conflict clauses can also be shared "vertically" among different cores as shown in Fig. 10.3, because the solving effort spent on a single core can be reused by other cores to save overall time consumption. Unfortunately, the spatial symmetry is difficult to recover from the CNF representation of the SAT instance. The reason is that most clauses contain auxiliary variables introduced during the CNF encoding process. Since these auxiliary variables are unlabeled, the correspondence between clauses from different cores cannot be established directly. Although the spatial symmetry can be partially recovered by solving a graph automorphism problem [1, 3, 9], it may require impractical time for large designs, because no polynomial time solution is found for graph automorphism problem. The underlying reason for this dilemma is that the high-level information is lost after the CNF encoding. In other words, a single flattened CNF SAT instance is not suitable to exploit the spatial symmetry.

Instead of using a monolithic CNF as input, TGMA solves this problem by composing the CNF description of the system using CNF formulae for one core, bus, and the memory subsystem. Since the cores are identical, their CNF representations are identical as well. We only need to perform variable name substitution to obtain the CNF for all other cores. As shown in Theorem 10.1, when the state variables are substituted by the correct names, the system CNF composed of these replicated CNF

[1] We first discuss TGMA in the context of homogeneous cores. The application of this approach on heterogeneous cores will be presented in Sect. 10.3.3. .

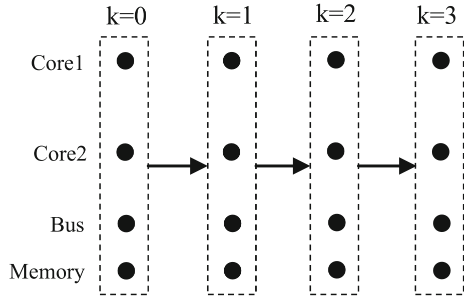


Fig. 10.2 Incremental SAT solving technique [19]

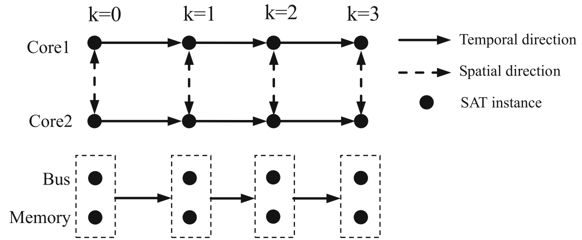


Fig. 10.3 Test generation for multicore architectures

for cores, bus, as well as memory subsystem will have the same satisfiability behavior as the original monolithic CNF representation. Since both the state variables and auxiliary variables in replicated cores are assigned by the TGMA algorithm, it is easy to obtain the correct mapping between variables and clauses in different cores. The spatial symmetry can then be effectively exploited during the SAT solving process. Before we describe the TGMA algorithm in detail, we first introduce some notations.

Definition 10.3.1 *Symmetric Component (SC)*Symmetric Component is a set of identical finite state machines (FSM). For the *j*th FM within an SC, its initial condition and transitional constraints are denoted as $I(s^{\text{is}}_{0,j})$ and $R(s^{\text{is}}_{i,j}, s^{\text{in}}_{i,j}, s^{\text{is}}_{i+1,j}, s^{\text{out}}_{i+1,j})$ $(0 \leq i \leq k-1)$, where $s^{\text{in}}_{i,j}, s^{\text{out}}_{i+1,j}, s^{\text{is}}_{i,j}$ are its input variables, output variables, and internal state variables at the *i*th (*i*+1th) time step. It should be noted that a symmetric component itself can also be viewed as a top-level FSM, whose input and output variables are the collection of all the input and output variables of FSMs within the top-level FSM. ■

In a multicore system with N_S identical cores, the set of all cores is modeled as a symmetric component F^S. Other asymmetric components, such as bus and memory subsystem, are modeled as a single finite state machine F^A. We also map the input and output of F^A to the output and input of F^S so that different cores can perform communication through bus and memory subsystem. Formally, we denote the initial condition and transition constraints of F^A as $I(s^A_0)$ and $R(s^A_i, s^{Sout}_i, s^A_{i+1}, s^{Sin}_{i+1})$ $(0 \leq i \leq k-1)$, where s^A_i represents internal state variables in bus and memory subsystem at the *i*th time step. Moreover, $s^{Sin}_i = \{s^{\text{in}}_{i,j} | 1 \leq j \leq N_S\}$ and $s^{Sout}_i = \{s^{\text{out}}_{i,j} | 1 \leq j \leq N_S\}$

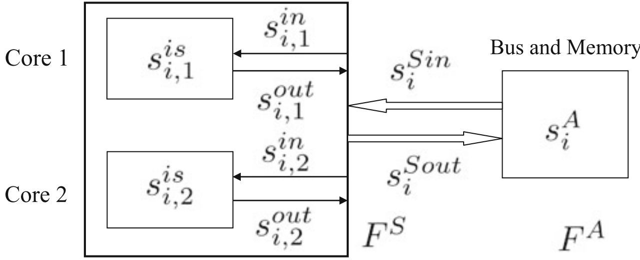


Fig. 10.4 FSM representation of Fig. 10.1 at time step i

are the input and output variables of the symmetric component F^S, which is the combination of the inputs and outputs of all cores. For example, Fig. 10.4 shows the FM representation of the system in Fig. 10.1. The symmetric component F^S is composed of core 1 and core 2. The rest of the system is represented by F^A. In the ith time step, the internal state variable of F^S are $\{s^{\text{is}}_{i,1}, s^{\text{is}}_{i,2}\}$ and s^A_i. The input and output variables of F^S (also the output and input variable of F^A) are $s^{\text{Sin}}_i = \{s^{\text{in}}_{i,1}, s^{\text{in}}_{i,2}\}$ and $s^{\text{Sout}}_i = \{s^{\text{out}}_{i,1}, s^{\text{out}}_{i,1}\}$, respectively.

The BMC formula of the multicore system can be expressed as

$$\begin{aligned}
\text{BMC}(M, p, k) &= I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \\
&= I(s^A_0) \wedge \bigwedge_{j=1}^{N_S} I(s^{\text{is}}_{0,j}) \wedge \bigwedge_{i=0}^{k-1} (R(s^{A_i, s^{\text{Sout}}_i, s^A_{i+1}, s^{\text{Sin}}_{i+1}}) \\
&\wedge \bigwedge_{j=1}^{N_S} R(s^{\text{is}}_{i,j}, s^{\text{in}}_{i,j}, s^{\text{is}}_{i+1,j}, s^{\text{out}}_{i+1,j})) \wedge \bigvee_{i=0}^{k} \neg p(s_i)
\end{aligned}$$

The basic idea of TGMA is to generate CNF formula

$$\begin{aligned}
\text{BMC}'(M, p, k) &= \text{CNF}^A_I \wedge \bigwedge_{j=1}^{N_S} \text{CNF}^S_I(j) \\
&\wedge \bigwedge_{i=0}^{k-1} (\text{CNF}^A_R(i) \wedge \bigwedge_{j=1}^{N_S} \text{CNF}^S_R(i, j)) \wedge \text{CNF}^p(k)
\end{aligned}$$

and perform SAT solving on $\text{BMC}'(M, p, k)$ instead of solving the CNF formula directly synthesized from $\text{BMC}(M, p, k)$, where CNF^A_I, $\text{CNF}^S_I(j)$, $\text{CNF}^A_R(i)$,

Algorithm 1: Test Generation for Multicore Architectures

Input: CNF formulae CNF_I^A, $CNF_I^S(1)$, $CNF_R^A(i)$, $CNF_R^S(i, 1)$, $CNF^p(k)$, Number of cores N_S, Maximum bound K_{max}
Output: Test $test_p$
Bound $k = 0$;
Initialize variable mapping table T;
Common Clause Set $CCS = \emptyset$;
Generate $CNF_I^S(j)$ using $CNF_I^S(1)$ for $1 < j \leq N_S$;
Add Clauses in $CNF_I^S(j)$ to CCS for $1 \leq j \leq N_S$;
Update T;
Add Clauses in CNF_I^A to CCS;
while $k \leq K_{max}$ **do**
  Generate $CNF_R^S(k, j)$ using $CNF_R^S(k, 1)$ for $1 < j \leq N_S$;
  Add Clauses in $CNF_R^S(k, j)$ to CCS $1 \leq j \leq N_S$;
  Update T;
  Add Clauses in $CNF_R^A(k)$ to CCS;
  Step1: $(ConflictC, test_p)$ = SAT($CCS \bigcup CNF^p(k)$,T);
  Step2: $CCS = CCS \bigcup$ Filter($ConflictC$);
  if $test_p \neq null$ **then** return $test_p$;
  $k = k + 1$;
end

$\text{CNF}_R^S(i, j)$ and $\text{CNF}^p(k)$ are the CNF representations of $I(s_0^A)$, $I(s_{0,j}^{\text{is}})$, $R(s_i^A, s_i^{S\text{out}}, s_{i+1}^A, s_{i+1}^{S\text{in}})$, $R(s_{i,j}^{\text{is}}, s_{i,j}^{\text{in}}, s_{i+1,j}^{\text{is}}, s_{i+1,j}^{\text{out}})$ and $\bigvee_{i=0}^{k} \neg p(s_i)$, respectively.

Algorithm 1 shows the test generation method for multicore architectures. It accepts the CNF representation of one core, bus, the memory subsystem as well as the properties at different time steps as inputs and produces corresponding directed tests. As indicated before, we first generate the CNF representations of the initial condition and transition constraints of all other FSMs in F^S based on the input CNF formulae $\text{CNF}_I^S(1)$ and $\text{CNF}_R^S(i, 1)$, which are the initial condition and transition constraints of the first FSM (Core 1). It is accomplished by replacing variable in $\text{CNF}_I^S(1)$ and $\text{CNF}_R^S(i, 1)$ with corresponding variables for other FSMs (cores). At the same time, a table T[2] is used to record the symmetric set of variables for both state variables and auxiliary variables. After that, the SAT solving process is invoked on the conjunction of clauses in CCS and $\text{CNF}^p(k)$, which is equivalent to $\text{BMC}'(M, p, k)$ defined above. Next, the following two steps are performed.

1. During SAT solving, analyze any conflict clause cls found by the SAT solver. If cls is purely deduced by the clauses which belong to a single FSM, replicate and forward cls to all other FSMs. This is implemented by substituting the variables in cls by their counterparts for each FSM in F^S based on table T. At the same time, the cls is also replicated in temporal direction, as discussed in [19].

[2] As discussed in Sect. 10.3.2, a physical table is not required, instead a mapping function is used in the framework.

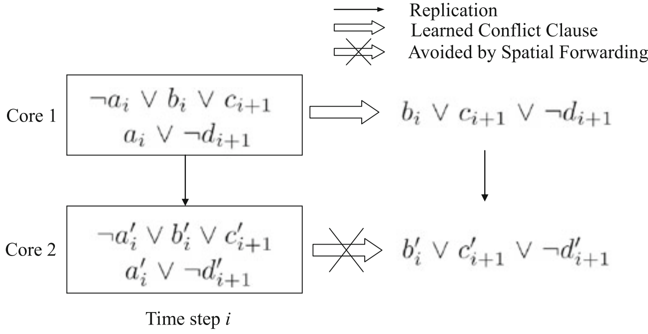


Fig. 10.5 Test generation for multicore architectures

2. After the solving process, only keep new conflict clauses that are deduced independent of $\text{CNF}^p(k)$, and merge them into CCS.

If the satisfying assignment, or a counterexample test_p is found in step 1, the algorithm returns it as a test. Otherwise, the algorithm repeats for each bound k until the maximum bound is reached.

We use the same example in Fig. 10.1 to illustrate the flow of Algorithm 1. The two different clause forwarding paths employed in TGMA are shown in Fig. 10.5. Suppose $(\neg a_i \vee b_i \vee c_{i+1})$ and $(a_i \vee \neg d_{i+1})$ are two clauses within $\text{CNF}_R^S(i, 1)$ (transition constraint of Core 1), in the first iteration for $k = 0$, two clauses $(\neg a'_i \vee b'_i \vee c'_{i+1})$ and $(a'_i \vee \neg d'_{i+1})$ will be produced during the generation of $\text{CNF}_R^S(i, 2)$ (transition constraint of Core 2). In the subsequent SAT solving process, suppose a conflict clause $(b_i \vee c_{i+1} \vee \neg d_{i+1})$ is deduced based on $(\neg a_i \vee b_i \vee c_{i+1})$ and $(a_i \vee \neg d_{i+1})$, it will be forwarded to Core 2, because its two parent clauses are all from the CNF formula for Core 1. Therefore, $(b'_i \vee c'_{i+1} \vee \neg d'_{i+1})$ can now be used by Core 2 to prevent the partial assignment $\{b'_i, c'_{i+1}, d'_{i+1}\} = \{0, 0, 1\}$, which will result in a conflict on a'_i. Such forwarding of conflict clauses is impossible using Strichman's approach [19], which only considers temporal symmetry but not spatial symmetry.

In the remainder of this section, we prove the correctness of TGMA and discuss the implementation details of the directed test generation algorithm for multicore architectures.

10.3.1 Correctness of TGMA

To prove the correctness of the TGMA approach, we need to ensure that the produced CNF formula $\text{BMC}'(M, p, k)$ in Algorithm 1 has the same satisfiability as $\text{BMC}(M, p, k)$.

Theorem 10.1 *$BMC(M, p, k)$ and $BMC'(M, p, k)$ have the same satisfiability.*

Proof Clearly, we have

$$
\begin{aligned}
\mathrm{BMC}(M, p, k) &= I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \\
&= I(s_0^A) \wedge \bigwedge_{j=1}^{N_S} I(s_{0,j}^{\mathrm{is}}) \wedge \bigwedge_{i=0}^{k-1} (R(s_i^A, s^{\mathrm{Sout}_i, s_{i+1}^A, s_{i+1}^{\mathrm{Sin}}}) \\
&\quad \wedge \bigwedge_{j=1}^{N_S} R(s_{i,j}^{\mathrm{is}}, s_{i,j}^{\mathrm{in}}, s_{i+1,j}^{\mathrm{is}}, s_{i+1,j}^{\mathrm{out}})) \wedge \bigvee_{i=0}^{k} \neg p(s_i)
\end{aligned}
$$

By their definitions, CNF formulae CNF_I^A, $\mathrm{CNF}_I^S(j)$, $\mathrm{CNF}_R^A(i)$, $\mathrm{CNF}_R^S(i, j)$ and $\mathrm{CNF}^p(k)$ are CNF representation of propositional formulae $I(s_0^A)$, $I(s_{0,j}^{is})$, $R(s_i^A, s_i^{\mathrm{Sout}}, s_{i+1}^A, s_{i+1}^{\mathrm{Sin}})$, $R(s_{i,j}^{\mathrm{is}}, s_{i,j}^{\mathrm{in}}, s_{i+1,j}^{\mathrm{is}}, s_{i+1,j}^{\mathrm{out}})$ and $\bigvee_{i=0}^{k} \neg p(s_i)$, where $0 \le i \le k-1$ and $1 \le j \le N_S$.

Therefore, $\mathrm{BMC}(M, p, k)$ has the same satisfiability as

$$
\begin{aligned}
BMC'(M, p, k) =& \mathrm{CNF}_I^A \wedge \bigwedge_{j=1}^{N_S} \mathrm{CNF}_I^S(j) \\
&\wedge \bigwedge_{i=0}^{k-1} (\mathrm{CNF}_R^A(i) \wedge \bigwedge_{j=1}^{N_S} \mathrm{CNF}_R^S(i, j)) \wedge \mathrm{CNF}^p(k)
\end{aligned}
$$

because the auxiliary variables introduced during CNF conversion do not change the satisfiability. In other words, $\mathrm{BMC}(M, p, k)$ and $\mathrm{BMC}'(M, p, k)$ have the same satisfiability. ■

In fact, the value of state variables in a satisfying assignment of $\mathrm{BMC}'(M, p, k)$ also satisfy $\mathrm{BMC}(M, p, k)$ and therefore can be used as a counterexample of the property p. The reason is that the value of the variables in a satisfying assignment of $\mathrm{BMC}'(M, p, k)$ will also satisfy all CNF formulae CNF_I^A, $\mathrm{CNF}_I^S(j)$, $\mathrm{CNF}_R^A(i)$, $\mathrm{CNF}_R^S(i, j)$ and $\mathrm{CNF}^p(k)$. Thus, the value of the state variables will satisfy corresponding propositional formulae $I(s_0^A)$, $I(s_0^j)$, $R(s_i^A, s_{i+1}^A)$, $R(s_i^j, s_{i+1}^j)$ and $\bigvee_{i=0}^{k} \neg p(s_i)$. Hence, they together will satisfy $\mathrm{BMC}(M, p, k)$, which is a conjunction of the above propositional formulae. Therefore, the correctness of TGMA is justified.

10.3.2 Implementation Details

The test generation algorithm for multicore architectures is built around NuSMV model checker [7] and zChaff SAT solver [10]. The system is modeled using SMV language, then NuSMV is used to generate the CNF formulae CNF_I^A, $\text{CNF}_I^S(1)$, $\text{CNF}_R^A(i)$, $\text{CNF}_R^S(i, 1)$ and $\text{CNF}^p(k)$ in DIMACS format as inputs of Algorithm 1. zChaff is employed as the internal SAT solver. In this section, we briefly explain CNF generation process and the implementation of Steps 1 and 2 in Algorithm 1.

The generation of CNF descriptions for a single core, bus, and memory subsystem using NuSMV is straightforward. The only practical consideration is that all variables are represented by their indices in CNF clauses. As a result, it is important to avoid the same index to be used by two different variables. Since NuSMV does not offer any external interface to control the index assignment, the source code is modified to make the index space suitable for the TGMA algorithm. The basic idea is to make the assignment of indices satisfy the following two constraints: (1) the indices of variables from the same core at the same time step are assigned continuously; (2) the indices of variables of the same time step across cores are assigned continuously as well. For example, in a 2-core system with each core having 100 variables, in time step 1 for core 1 we can use indices from 1 to 100 (controlled by the first constraint), whereas the second constraint indicates that the variables for core 2 at time step 1 should be 101–200. Therefore, 201–300 can be used to represent variables of core 1 in time step 2, and so on. Based on these two constraints, the computation of the indices of symmetric variables can be efficiently implemented as increasing or decreasing by a certain offset.

During SAT solving, the dependency of generated conflict clauses also needs to be tracked to determine whether they can be forwarded to other cores. This can be easily implemented within zChaff, which provides clause management scheme to support incremental SAT solving. For each clause in its clause database DB, zChaff uses a 32-bit group ID to track the dependency. Each bit identifies whether that clause belongs to a certain group. When a conflict clause is deduced based on clauses from multiple groups, its group ID is a "OR" product of the group ID of all its parent clauses, i.e., this clause belongs to multiple groups. zChaff also allows the user to add or remove clauses by group ID between successive solving processes. If one clause belongs to multiple groups, it is removed when any of these groups are removed.

With these mechanisms, steps 1 and 2 in Algorithm 1 can be implemented efficiently as follows:

1. Add clauses in $\text{CNF}_I^S(j)$ and $\text{CNF}_R^S(i, j)$ with group ID j, $1 \leq j \leq N_S$
2. Add clauses in CNF_I^A, $\text{CNF}_R^A(i)$with group ID $N_S + 1$.
3. Add clauses in $\text{CNF}^p(k)$ with group ID $N_S + 2$.

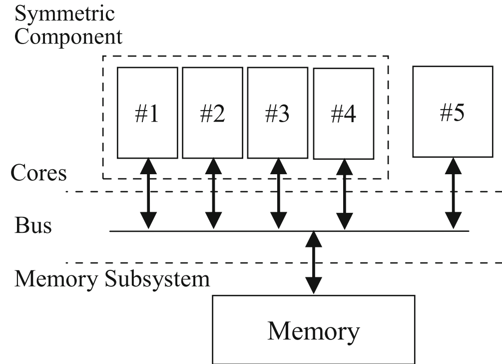


Fig. 10.6 Multicore system with different types of cores

4. When a new conflict clause is obtained during SAT solving, if it only belongs to a single group with ID smaller than $N_S + 1$, replicate this clause to all other cores with proper group ID.
5. After solving all clauses in DB with zChaff, remove clauses with group ID N_S+2.

The overhead introduced by dependency identification and tracking in the TGMA algorithm is negligible compared to the improvement in SAT solving time. At the same time, since the indices of variables in symmetric cores are carefully assigned, the mapping table T is not maintained explicitly, but implemented as a simple mapping function, which is used to generate forwarding clauses for different cores. In that way, the potential caching overhead can be avoided, which may deteriorate the performance of the SAT solver.

10.3.3 Heterogeneous Multicore Architectures

So far, we discussed the TGMA algorithm using homogeneous cores. This section describes the application of TGMA in the presence of heterogeneous cores. In a heterogeneous multicore system, if any two cores are completely different, it is impossible to reduce the test generation time by exploiting the symmetry. However, most real systems usually employ a cluster of identical cores for same computational purpose. In this case, they can be first grouped into symmetric components based on their types, then apply the TGMA algorithm to each symmetric component. For example, in the 5-core system shown in Fig. 10.6, core 5 is used for monitoring and core 1–4 are identical cores for computation. We can define core 1–4 as the symmetric component and apply TGMA on them. In general, TGMA can be applied on each cluster of identical cores in a system.

However, when the heterogeneous cores are not completely different, i.e., only some functional units in them are different, the TGMA algorithm can be employed in a more efficient way. Recall that the FSMs in a symmetric component are not

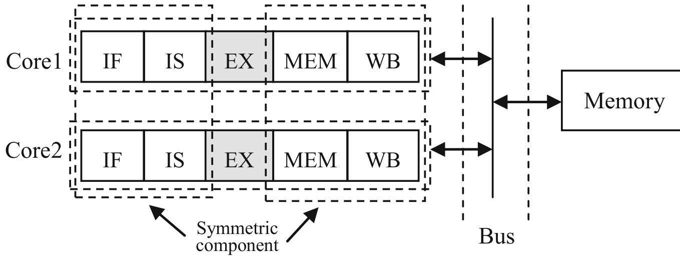


Fig. 10.7 Multicore system with different types of execution units

restricted to cores. We can actually define the symmetric component in such a way that it includes only the identical functional units in different cores. For example, Fig. 10.7 shows a system with heterogeneous cores. Both of the cores are pipelined with five stages: fetch, decode, execute, memory access, and writeback. The only difference is that they have different implementation in the execute stage EX. In this case, the symmetric component F^S can be defined as the set of all functional units in two cores except EX. These two execution stages as well as bus and memory subsystem are modeled in the asymmetric part F^A. Of course, the input and output of F^S here will include not only the input and output variable of the cores, but also all the interface variables between EX and other stages. In this way, the information learned on all other stages of one core can still be shared by the other core. Clearly, the correctness of TGMA is still guaranteed, because the selection of the symmetric component satisfies its definition.

10.4 Case Studies

We have evaluated the applicability and usefulness of the TGMA technique on different multicore architectures.

10.4.1 Experimental Setup

As described in Sect. 10.3.2, the designs and properties are described in SMV language and converted into required CNF formulae (DIMACS files) using modified NuSMV [7]. zChaff [10] is used as the SAT solver to implement the TGMA algorithm. Experiments were performed on a PC with 3.0 GHz AMD64 CPU and 4GB RAM.

First, we present results of TGMA using a multicore design that is composed of different number of identical cores, one bus, and memory subsystem. The pipeline inside each core has five stages: fetch, decode, execute, memory access, and writeback. Besides, each core has its own cache, which is connected with the memory

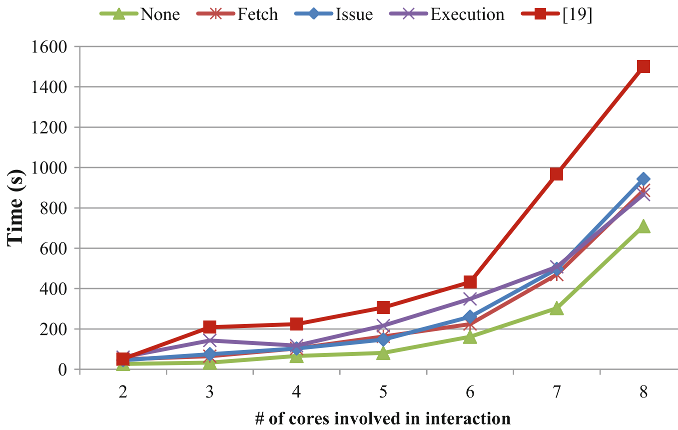


Fig. 10.8 Test generation time with heterogeneous cores

through the bus. Next, we will present (in Fig. 10.8) the applicability of TGMA on heterogeneous multicore architectures.

In order to activate the desired system behaviors, different number of properties are checked on designs with different complexities. For instance, 375 properties are used in case of 16-core design that triggers two simultaneous activities between cores. We have also used several properties that involve multicore interactions. For example, one test will activate the following scenario: "if the value in a memory location which is initialized as one by core 1, is increased by one by all other cores, it should be equal to the number of cores when it is readback by core 2". It should be noted that the corresponding property is not symmetric with respect to all cores.

10.4.2 Results

TGMA is compared with Strichman's approach [19] and original BMC [8]. Each approach was used to solve a sequence of SAT instances for the same property with varying bounds until a satisfiable instance is found. The input SAT instances for Strichman's approach and the original BMC were directly synthesized from $BMC(M, p, k)$ to improve their performance. When TGMA was applied, the SAT solving is performed on $BMC'(M, p, k)$ as indicated in Sect. 10.3.

Figure 10.9 presents the average test generation time for different number of cores. The original BMC failed to produce results within 3,000 s on several properties for the 16-core system. Therefore, its time is omitted. As expected, the time consumption increases with the number of cores. Both TGMA and Strichman's approach [19] are remarkably faster than original BMC [8]. By effective utilization of both spatial and temporal symmetry, TGMA outperforms [19] (which only considers temporal symmetry) by nearly 2 times.

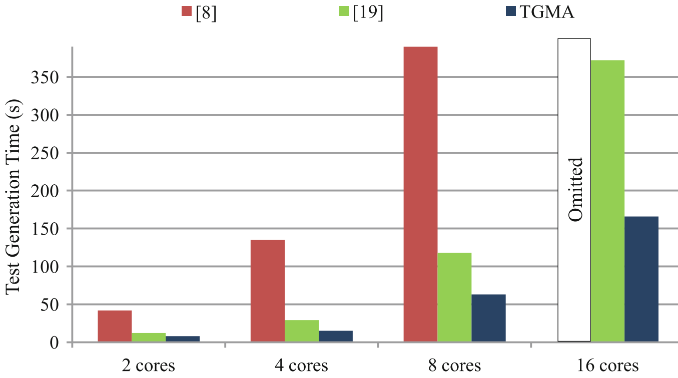


Fig. 10.9 Test generation time with different number of cores

Table 10.1 Test generation time for 8-core system

Property	Bound	[8] time (s)	[19] time (s)	TGMA time (s)	Speedup over [8]	Speedup over [19]
1	28	79	56	25	3.16	2.24
2	22	67	44	21	3.19	2.10
3	32	93	62	30	3.10	2.07
4	28	208	94	17	12.24	5.53
5	33	∗	342	148	–	2.31
6	20	413	124	47	8.79	2.64
7	20	∗	125	48	–	2.60
8	**23**	**883**	**140**	**63**	**14.02**	**2.22**
9	25	2106	157	128	16.45	1.23
10	25	1991	106	101	19.71	1.05
Total	–	5840	1250	628	9.30	1.99

∗ Represent run times exceeding 3,000 s

Table 10.1 shows a more detailed comparison of different approaches on the 8-core system for 10 most time-consuming properties. The first column represents the names of properties used. The second column shows the corresponding bounds or time steps to activate each property. The next three columns present the test generation time (in seconds) for each property using the original BMC [8], Strichman's approach [19], and TGMA, respectively. The time is calculated as the summation of the time to solve all the SAT instances from $k = 0$ to the bound of the property. The time calculation also includes the time consumed by non-SAT-solving steps in Algorithm 1. The last two columns indicate the speedup of TGMA over [8, 19]. Clearly, TGMA outperforms [19] by two times and [8] by an order of magnitude.

To inspect the reason why TGMA outperforms [19], the behavior of the SAT solver is analyzed. Table 10.2 shows details of the last five SAT instances immediately before the bound was found during the BMC of property 8 on the 8-core system (highlighted entry in Table 10.1). The first column in Table 10.2 is the time step of

Table 10.2 Detailed test generation information

k	[19]				TGMA			
	#Cls in DB	#Decision	#Fwd Cls	Time(s)	#Cls in DB	#Decision	#Fwd Cls	Time(s)
19	721427	40045	25608	2.4	756149	21231	4441	1.2
20	762855	71854	27329	3.6	857103	30049	26685	2.7
21	827272	56692	22824	3.4	900428	35687	24534	3.1
22	893382	203112	102202	15.4	965925	30873	6834	1.9
23	954998	2652411	142585	97.3	1029266	1228603	261989	52.8
Total	–	3024114	320548	122.1	–	1346443	324483	61.7

each SAT instance. The next four columns contain the real size of the clause database before the solving process, the number of decisions made by zChaff, the number of forwarded conflict clauses and the time consumption in [19]. Similar information of TGMA approach is represented in the last four columns. Compared to [19], the total number of decisions made by the SAT solver is much smaller when TGMA is applied. At the same time, the number of forwarded clauses are comparable. In other words, TGMA saves the time to rediscover the same knowledge for each core, without the overhead of forwarding too many conflict clauses.

We also investigated the impact of different number of cores involved in the interaction on the test generation time. In this experiment, a processor with eight 3-stage cores is used. Cores are connected to the memory subsystem using snoopy protocol. The desired test should trigger all cores perform read and write operation on the same shared memory variable in a certain order. The results are given in Figure 10.10. When the interaction involves only a small number of cores, the difference in test generation time of [8, 19], and TGMA is quite small. However, when more and more cores are involved, TGMA outperforms both [8, 19] remarkably, due to the usage of symmetry information.

Finally, to illustrate the effectiveness of TGMA in a more general scenario, the test generation time on a system with heterogeneous cores is measured by repeating the previous test generation experiment using cores with different implementations in their fetch, issue, execution stages. As discussed in Sect. 10.3.3, only learned conflict clauses within the symmetric components are replicated. Figure 10.8 shows the result. The "fetch" curve corresponds to a system where the 8 cores are identical except their fetch stages. Similarly, curves marked as "Issue" and "Execution" represent cores with different issue and execution stages, respectively. We also show the test generation time for homogeneous cores using TGMA ("None") and [19] as reference. It can be observed that, due to less scope of knowledge reuse, the time consumption of TGMA for heterogeneous cores are generally larger than homogeneous cores. Nevertheless, TGMA still outperforms [19] especially for complicated interactions involving many cores.

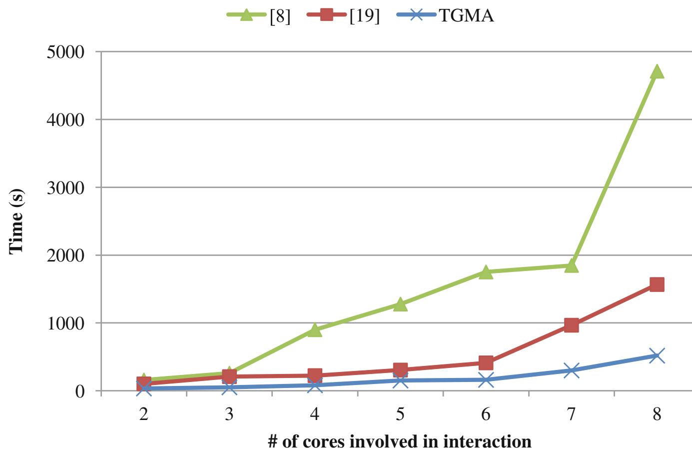


Fig. 10.10 Test generation time with different interactions

10.5 Chapter Summary

Functional verification of multicore architectures is challenging due to the increased design complexity and reduced time-to-market. Existing incremental SAT approaches have only exploited the symmetry in BMC across different time steps. This chapter presented a novel approach for directed test generation of multicore architectures that exploits both spatial and temporal symmetry in SAT-based BMC. The CNF description of the design is synthesized using CNF for cores, bus, and memory subsystem to preserve the mapping information between different cores. As a result, the symmetric high-level structure is well preserved and the knowledge learned from a single core can be effectively shared by other cores during the SAT solving process. The experimental results using homogeneous as well as heterogeneous multicore architectures demonstrated that the test generation time using TGMA is remarkably smaller (2–10 times) compared to existing methods.

References

1. Aloul FA, Markov IL, Sakallah K (2003) Shatter: efficient symmetry-breaking for boolean satisfiability. In: Proceedings of design automation conference, pp 836–839
2. Aloul FA, Markov IL, Sakallah KA (2003) Shatter. University of Michigan. http://www.aloul.net/Tools/shatter/
3. Aloul FA, Ramani A, Markov IL, Sakallah K (2002) Solving difficult SAT instances in the presence of symmetry. In: Proceedings of design automation conference, pp 731–736
4. Bhadra J, Trofimova E, Abadir M (2008) Validating power architecture technology-based mpsocs through executable specifications. IEEE Trans Very Larg Scale Integr Syst 16(4): 388–396

5. Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Proceedings of international conference on tools and algorithms for construction and analysis of systems, pp 193–207
6. Biere A, Sinz C (2006) Decomposing SAT problems into connected components. J Satisf Boolean Model Comput 2:191–198
7. Cavada R, Cimatti A, Jochim CA, Keighren G, Olivetti E, Pistore M, Roveri M, Tchaltse A (2010) NuSMV. ITC-Irst. http://nusmv.irst.itc.it/
8. Clarke E, Biere A, Raimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Formal Methods Syst Des 19(1):7–34
9. Darga PT, Liffiton MH, Sakallah KA, Markov IL (2004) Exploiting structure in symmetry detection for cnf. In: Proceedings of design automation conference, pp 530–534
10. Fu Z, Mahajan Y, Malik S (2001) zChaff. Princeton University. http://www.princeton.edu/chaff/zchaff.html
11. Gargantini A, Heitmeyer C (1999) Using model checking to generate tests from requirements specifications. In: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on foundations of, software engineering, vol 24, pp 146–162
12. Hooker JN (1993) Solving the incremental satisfiability problem. J Log Program 15(1–2): 177–186
13. Kuehlmann A (2004) Dynamic transition relation simplification for bounded property checking. In: Proceedings of IEEE/ACM international conference on computer-aided design, pp 50–57
14. Miller A, Donaldson A, Calder M (2006) Symmetry in temporal logic model checking. ACM Comput Surv 38(3):8
15. Mishra P, Chen M (2009) Efficient techniques for directed test generation using incremental satisfiability. In: Proceedings of international conference on VLSI design, pp 65–70
16. Mishra P, Dutt N (2004) Graph-based functional test program generation for pipelined processors. In: Proceedings of the conference on design, automation and test in Europe, pp 182–187
17. Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: engineering an efficient SAT solver. In: Proceedings of design automation conference, pp 530–535
18. Qin X, Mishra P (2011) Efficient directed test generation for validation of multicore architectures. In: Proceedings of international symposium on quality electronic design, pp 276–283
19. Strichman O (2004) Accelerating bounded model checking of safety properties. Formal Methods Syst Des 24(1):5–24
20. Tang D, Malik S, Gupta A, Ip CN (2005) Symmetry reduction in SAT-based model checking. In: Proceedings of international conference on computer aided verfication, pp 125–138
21. Whittemore J, Kim J, Sakallah K (2001) SATIRE: a new incremental satisfiability engine. In: Proceedings of design automation conference, pp 542–545

Chapter 11
Test Generation for Cache Coherence Validation

11.1 Introduction

Caching has been the most effective approach to reduce the memory access time for several decades. When the same data are cached by different processors, cache coherence protocols are employed to coordinate the accesses and guarantee that the most recent written data are returned. As the protocols are growing more and more complex, the verification teams are facing significant challenges to achieve the required coverage within tight time-to-market window.

Since all possible behaviors of the cache blocks in a system with n cores[1] can be defined by a global finite state machine (FSM), the entire state space is the product of n cache block level FSMs. Intuitively, full state or transition coverage can be achieved by performing a breadth first search (BFS) on this product FSM. The path that leads to each distinct state from the initial state can be used as a test case for that state. Unfortunately, since each test is used to activate only one transition, a large number of transitions may be unnecessarily repeated, if they are on the shortest path to many other transitions. Therefore, it is desirable to replace BFS with another efficient algorithm, which creates an input sequence that covers all transitions with minimum transition overhead. Since the number of directed tests can be quite large in many practical scenarios, it may be beneficial to generate the directed tests on-the-fly, so that the created tests can be directly fed to the simulator or the device under test without extra storage requirement. Clearly, the development of such algorithms requires a clear understanding of the state space of the complex global FSM. Although the FSM of each cache controller is easy to understand, the structure of the product FSM for modern cache coherence protocols can have quite obscure structure that can be hard to analyze.

This chapter presents an on-the-fly test generation for cache coherence protocols by analyzing the state space structure of their corresponding global FSMs [9]. Instead

[1] In this chapter, we use the term "**core**" to refer to each single processing units in multicore or multiprocessor systems.

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_11,
© Springer Science+Business Media New York 2013

of using structure-independent BFS to obtain the directed tests, the entire complex state space is decomposed into several components with simple structures. Since the activation of state and transition can be viewed as a path searching problem in the state space, these decomposed components with known structures can be exploited for efficient test generation. In the following sections, a graphical description of the state space structure is described for commonly used cache coherence protocols and it can be viewed as a composition of simple structures. Based on the Euler tour of hypercubes, we present an on-the-fly directed test generation algorithm that only requires linear space requirement with respect to the number of cores. The generated test forms a tour in the state space of corresponding global FSM, which activates all possible transitions of the global FSM with small overhead.

The rest of this chapter is organized as follows. Section 11.3 provides related background information. Section 11.4 describes the test generation approach in details. Experimental results are presented in Sect. 11.5. Finally, Sect. 11.6 concludes the chapter.

11.2 Related Work

Existing protocol validation techniques can be broadly classified into two categories: formal verification and simulation-based validation. Formal methods using model checking can prove mathematically whether the description of certain protocol violates the required property. For example, Murφ [5] was used to verify various cache coherence protocols based on explicit model checking. Counter-example guided refinement [4] is employed to verify complex protocols with multilevel caches. Symbolic model checking tools are also developed for coherence verification. For example, Emerson et al. [7] investigated the verification problem with parameterized cache coherence protocol using binary decision diagrams (BDD). Fractal coherence [12] enables the scalable verification of a family of properly designed coherence protocols. Although formal methods can guarantee the correctness of a design, they usually require that the design should be described in certain input languages. As a result, it is usually difficult to apply model checking on implementations directly.

Simulation-based approaches, on the other hand, are able to handle designs at different abstraction levels and therefore widely used in practice. For example, Wood et al. [11] used random tests to verify the memory subsystem of SPUR machine. Genesys Pro test generator [2] from IBM extended this direction with complex and sophisticated test templates. To reduce the search space, Abts et al. [1] introduced space pruning technique during their verification of the Cray processor. Wagner et al. [10] designed the MCjammer tool which can get higher state coverage than normal constrained random tests. Since an uncovered transition can only be visited by taking a unique action at a particular state, it may not be feasible for a random test generator to eventually cover all possible states and transitions. To address this problem, some random testers are equipped with small amount of memory, so that the future search can be guided to the uncovered regions. Unfortunately, unless the memory is large

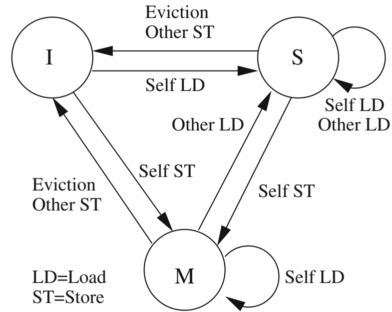


Fig. 11.1 State transitions for a cache block in MSI protocol

enough to hold the entire state space, it is still hard to achieve full coverage by such guided random testing.

11.3 Background and Motivation

In modern computer systems, since the latency to transfer data from the main memory to processing units is much larger than the time consumption for computation, each processing unit usually maintains its local copy of the main memory, or cache for fast access. One major problem of caching is that when the same data or memory block is cached in two or more different places, any future modification to it should be propagated to all the cached copies. Otherwise, it can lead to incorrect functional behaviors. Cache coherence protocols are therefore proposed to define the correct behavior of each cache controller, when different processing units issue loads and stores to the same memory location.

One of the simplest cache coherence protocol is the MSI snoopy protocol [8]. The behavior of the cache controller in a processing unit is modeled as an FSM. Figure 11.1 shows the state transition diagram of MSI protocol. The state of a cache block (line) can be either "Invalid" (I), "Modified" (M), or "Shared" (S). At the beginning, all cache blocks are in the invalid state. When a load request arrives from the core side (Self LD), the cache controller will request the data from the main memory and switch to shared state. When the core issues a store request (Self ST), the cache controller will first broadcast an invalidated request on the bus and then change to modified state. Such an invalidate request will inform all other cache controllers that are in shared or modified states to change to invalid state. A cache block may also change to invalid state, when it is evicted by another cache block which is mapped to the same location in the cache, or other cores issue store requests (Other ST).

Although MSI protocol is enough to guarantee the coherence of the cache system, it causes some unnecessary delay and traffic on the communication channels. Many variants of the MSI protocols are invented to further improve its performance.

For example, "Exclusive" (E) state is introduced in MESI protocol to avoid the traffic when a cache block is only used by one core. "Owned" (O) state is used in MOSI and MOESI protocol to reduce the delay when a modified block is loaded by other cores.

As cache coherence protocols are becoming more and more complex, it is getting harder to verify their implementations. From the validation perspective, it is always desirable to activate all possible state transitions of the entire multicore cache system. In other words, it is necessary to have a high state and transition coverage (100 %, if possible) in the global FSM of the entire memory (cache) subsystem.

11.4 Test Generation for Transition Coverage

The test generation for transition coverage (TGTC) of cache coherence protocols [9] is motivated by the basic BFS in the state space of a global FSM. Given the FSM description of any cache coherence protocol, it is possible to compose a test suite which can activate all states and transitions using two steps: (1) for each state, we find out the instruction sequences to reach it by performing a BFS on the global FSM; and (2) for each transition, we create the test by appending the required instructions after the instruction sequences to reach the initial state of this transition. However, such a naive approach has two problems. First, transitions close to the initial state are visited for many times. Thus, a large portion of the overall test time is wasted. Secondly, it is difficult to generate tests on-the-fly, because the memory requirement to run the BFS routine is quite large. Since all visited states in BFS process have to be recorded, its runtime memory requirement also grows exponentially.

To address these challenges, an efficient TGTC algorithm needs to satisfy two requirements: (1) the number of transitions should be reduced as much as possible without sacrificing the coverage goal; and (2) the space requirement for the test generation algorithm should be small. Fortunately, due to the highly symmetric and regular structure of the state space, it is possible to design a deterministic test generation algorithm, which can efficiently activate all states and transitions of popular cache coherence protocols. The basic idea is to divide the complex state space into several large hypercubes and other small components. Since hypercubes can be traversed with no extra overhead, a large number of unnecessary transitions can be avoided during activating all transitions.

In the rest of this section, we first describe how to generate tests to activate all transitions of a simplified cache coherence protocol: SI protocol. Next, we discuss the TGTC techniques for a variety of popular protocols including MSI, MESI, MOSI, and MOESI. This chapter focuses on the transition between two stable states. It assumes that the transition between stable states to transient state are correct.

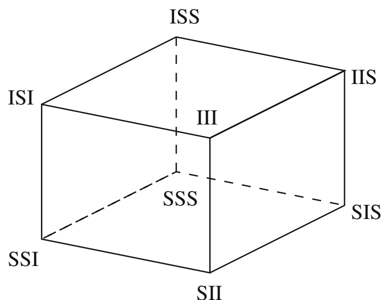


Fig. 11.2 Global FSM state space of SI protocol with 3 cores

11.4.1 SI Protocol

SI protocol is a trimmed version of MSI protocol, in which cores are not allowed to issue store operation. For a system with n cores, a valid global state of the system allows the cache blocks in any m cores in I state and cache blocks in the other $n - m$ cores in S state. Thus, there are 2^n valid global states. Besides, since any core in I (or S) state can be converted into S (or I) state within one transition, there are n outgoing and n incoming edges. It is easy to see that the entire state space of SI protocol with n cores is a n dimensional hypercube.[2] Figure 11.2 shows such a state space with three cores. *Since all edges are bidirectional for state transitions, we do not show transition directions explicitly.* For example, state III can be transformed into IIS when the first core loads the cache block. Similarly, state IIS can also be transformed into III, when the first core evicts this cache block.

To achieve full state and transition coverage of the state space, each edge of the hypercube needs to be traversed at least once in both directions. Since each global state has the same number of incoming and outgoing edges, it is possible to form a Euler tour [6] of the state space, which visits each edge exactly once in both directions.

Algorithm 1 shows the TGTC algorithm for SI protocol, which performs an Euler tour on an n dimensional hypercube. Here, *load(p)/evict(p) means the pth core performs a load/evict operation in a particular cycle, while all other cores remain idle*. We use the state space in Fig. 11.2 to show the execution of Algorithm 1. The algorithm starts by calling $CreateTestsSI(n)$. All cores are in I state at the beginning. In the first round of the *for* loop in line 2, the system first perform transition III-IIS by executing load(0). During $VisitHypercube$, it will first visit transition IIS-ISS and ISS-IIS for $i = 1$ and IIS-SIS for $i = 2$. Since $i > 1$, $VisitHypercube$ is invoked at line 6, which activates two transitions: SIS-SSS and SSS-SIS. Next, transition SIS-IIS is covered by executing evict(2) in line 7 of $VisitHypercube$.

[2] There are many transitions that start and end in the same states. For example, the global state will not change if a core in S state issues a load operation. Usually, these transitions are easier to cover, because they can be activated by appending one more operation at the end of existing tests, which are used to activate corresponding initial states. As a result, they are omitted in the state space structure description in this section. However, all possible transitions are considered in the actual implementation of TGTC as well as in the experimental results.

Algorithm 1: Test generation for SI protocol with n cores

```
CreateTestsSI(n)
1: for i = 0 to n − 1 do
2:    Output "load(i)"
3:    VisitHypercube(1, n − 1, i)
4:    Output "evict(i)"
5: end for

VisitHypercube(id, m, shift)
1: for i = 1 to m do
2:    newid = id + 2^i
3:    p = (i + shift) mod n
4:    Output "load(p)"
5:    if i > 1 then
6:       VisitHypercube(newid, i − 1, shift)
7:    end if
8:    Output "evict(p)"
9: end for
10: return
```

Finally, the global state goes back to III via transition IIS-III after evict(2) in line 5 of $CreateTestsSI$. In the next two rounds of the *for* loop in $CreateTestsSI$, we are essentially performing a "rotated" version of the previous traversal, which are going to cover all transitions in paths III-ISI-SSI-ISI-ISS-SSS-ISS-ISI-III and III-SII-SIS-SII-SSI-SSS-SSI-SII-III. Eventually, all transitions in the hypercube are covered by the generated test sequences.

Although the execution of Algorithm 1 seems to be complicated for larger n, the basic idea of this algorithm is quite easy: the hypercube is actually partitioned into n isomorphic trees with no overlapping edges. Once the hypercube is correctly partitioned, a Euler tour is performed on trees, because all edges are bidirectional. The space complexity of Algorithm 1 is linear with the number of cores n. The reason is that the function $VisitHypercube(id, m, shift)$ can be recursively called for at most $n - 1$ times. The algorithm therefore requires a stack that with at most $n - 1$ levels. As a result, the space complexity is $O(n)$.

11.4.2 MSI Protocol

The difference between MSI protocol and SI protocol is that a cache block can be changed to the modified (M) state, when it receives a store request. For the ease of discussion, we define the following terms.

Definition 11.1 **Global shared state** is a global state within which cores are in either shared or invalid states (e.g., IIS, ISI, ISS, SII, SIS, SSI, and SSS in Fig. 11.3). ■

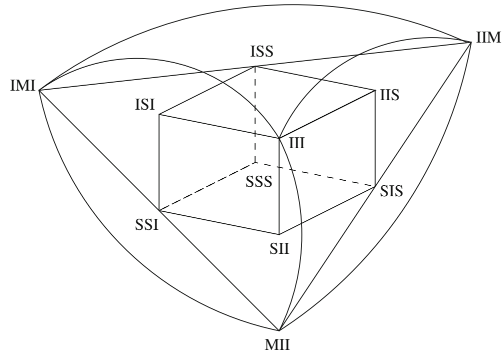


Fig. 11.3 State space of MSI protocol with 3 cores. For the clarity of presentation, the transitions to global modified states (IIM, IMI, MII) are omitted, if the transition in the opposite direction does not exist

Definition 11.2 Global invalid state is a global state within which all cores are in the invalid state (e.g., III in Fig. 11.3). ■

Definition 11.3 Global modified state is a global state within which one core is in the modified state (e.g., IIM, IMI, and MII in Fig. 11.3). ■

Figure 11.3 shows the state space of MSI protocol with three cores. Since only one core can be in the modified state for MSI protocol, there are n global modified states in the state space of a system with n cores. Global modified states are reachable from any other global states by store requests from corresponding cores. Besides, a global modified state can also be converted into the global invalid state or global shared states. For example, global modified state IMI can be converted to global invalid state III by evict(1), or global shared states ISS and SSI by load(0) or load(2), respectively. Clearly, all n global modified states form a clique, because there are two transitions with opposite directions between each pair of them. As a result, these transitions can be covered with a Euler tour. Unfortunately, *it is not possible to cover all transitions in the state space of MSI by a single Euler tour.* The reason is that for some global shared state like IIS, there are only outgoing transitions to global modified states, but no incoming transitions from them. Therefore, outgoing transitions are twice of incoming transitions. The similar scenario can also be observed for global modified states, which have more incoming transitions than outgoing transitions. To cover all transitions, some of them must be reused. In fact, the problem to minimize the number of reused transitions is called Chinese postman problem (CPP) [6], which can be solved by calculating the min-cost max-flow. Since the test generation has to be performed on-the-fly, it is not necessary to obtain the optimal solution by solving CPP, because the state space can be too large to fit into memory when there are many cores in the system. Instead, the uncovered transition to global modified state is visited one by one and the shortest path is used to link the end state of the previous transition and start state of the next transition.

Algorithm 2 presents the test generation algorithm for MSI protocol. We first invoke $CreateTestsSI(n)$ in Algorithm 1 to cover all transitions that also exist in SI protocol. Next, $VisitClique$ will recursively perform an Euler tour in the clique of all global modified states. For example, when we execute $VisitClique$ in the

Algorithm 2: Test generation for MSI protocol with n cores

```
CreateTestsMSI(n)
1: CreateTestsSI(n) /* Invoke Algorithm 1 */
2: VisitClique(0)
3: for each global shared state s do
4:    for i = 0 to n − 1 do
5:       Output "store(i)"
6:       Output the shortest path from current state to s
7:    end for
8: end for

VisitClique(p)
1: Output "store(p)"
2: Output operations to visit all bidirectionally reachable global shared states
3: for i = p + 1 to n − 1 do
4:    Output "store(i)"
5:    if i = p + 1 then
6:       VisitClique(i)
7:    end if
8:    Output "store(p)"
9: end for
10: return
```

state space shown in Fig. 11.3, we are first going to cover transition IIM-IMI. In the recursive call of $VisitClique$ in line 6, transition IMI-MII and MII-IMI are visited. After that, transition IMI-IIM is covered by execution of line 7. In the next round of iteration, IIM-MII and MII-IIM are visited. To improve the efficiency, we also traverse all global shared states that are bidirectionally reachable from current global modified state. Finally, in line 3–6 of $CreateTestsMSI(n)$ we are visiting all uncovered transitions from global shared states to global modified states. Notice that we do not need to run Dijkstra's algorithm to find shortest path in line 6, because we must be in a global modified state after executing the store operation in line 5. The target global shared state can be reached by issuing load and evict requests based on the position of "S" in its state vector.

11.4.3 MESI Protocol

In MESI protocol, a cache block goes to exclusive (E) state when it is the first one, which loads a memory address. In a system with n cores, there are n global exclusive states.[3] Figure 11.4 shows the state space with three cores. Unlike global modified states, global exclusive states cannot be converted to each other directly. Therefore, the test generation algorithm $CreateTestsMSI$ for MSI protocol needs to be modified to create tests for MESI protocol. We can add n groups of operations

[3] A **global exclusive state** is a global state with a cache block in exclusive state (e.g., IIE, IEI, and EII in Fig. 11.4).

Fig. 11.4 State space of MESI protocol with 3 cores

to cover transitions from the global invalid state to global exclusive states as well as transitions from global exclusive states to global modified states. Notice that the *CreateTestsSI* routine, which is used to visit all transitions between global shared states, also needs to be modified slightly. The reason is that in MESI protocol, the global invalid state will be converted into global exclusive states after any load request (III goes to IIE instead of IIS when the first core issues a load request).

11.4.4 MOSI Protocol

The MOSI protocol contains a new state "owned" (O), which can be used to avoid unnecessary writeback to memory. A cache block in the modified state is converted to the owned state, when other cores are trying to load the same cache block. The owned state can coexist with shared and invalid states. As a result, for a system with n cores, there are $n * 2^{n-1}$ global owned states.[4] Considering the fact that there are only $n + 2^n$ global states in MSI protocol with n cores, the state space of MOSI is much larger. Despite the large number of states, the state space structure of MOSI protocol is not complex. The entire space can be divided into three components. The first and second parts are the hypercube of global shared states and the clique of global modified states, respectively. They are identical to corresponding structures in MSI protocol. The third part is a set of n hypercubes with dimension $n-1$. Each of the $n-1$ dimensional hypercubes consists of 2^{n-1} global owned states, whose state vectors have "O" in the same position. For example, Fig. 11.5 shows the state space with 3 cores. It is easy to see that states (IOI,IOS,SOS,SOI) (IIO,SIO,SSO,ISO) and (OII,OSI,OSS,OIS) are composed of three 2-d hypercubes (squares).

One nice property of this state space structure is that there is no transition between the n hypercubes of global owned states. Therefore, a large number of transitions between global owned states can be efficiently covered. We can perform a Euler tour

[4] A **global owned state** is a global state with a cache block in owned state (e.g., IOI, IOS,…, OSS in Fig. 11.5).

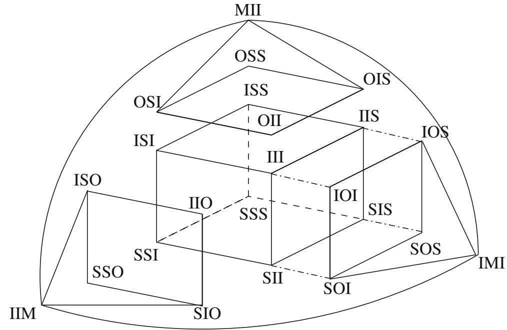


Fig. 11.5 State space of MOSI protocol with 3 cores

in each $n-1$ dimensional hypercube by invoking routine $CreateTestsSI$ on global owned states like IIO, IOI, and OII, where all but one core are in invalid state. In order to cover transitions from global owned states to global shared states, like IOS-IIS, we have to use a similar technique which was used in $CreateTestsMSI(n)$ to cover the store transitions.

11.5 Case Studies

To analyze the performance of the test generation framework, this section presents a number of experiments using M5 simulator [3]. M5 is a full system simulator, which implements a MOESI cache coherence protocol. In order to verify that the generated tests can achieve all transitions, the cache subsystem in M5 is slightly modified to allow different processes to access the same physical block. The load and store operations in the generated tests are translated into corresponding ALPHA instructions, while *evict* operation is achieved by loading a different memory address which is also mapped to the same location in the cache as the cache block under test. The *load-linked* and *store-conditional* instruction pairs are used to ensure the execution order of instructions in different cores.

Since M5 only supports MOESI cache coherence protocol, it is necessary to develop a protocol simulator which can be configured to simulate the state transition of a multicore system using MSI, MESI, and MOSI protocols. This simulator is used to validate the performance of TGTC on other protocols.

In the first experiment, TGTC is compared with the tests generated by performing BFS directly on the global FSM on different cache coherence protocols with various number of cores. Since tests generated by BFS are the shortest tests to drive the system from the global invalid state to the required transition, additional operations are used to reset the global state after execution of each test. Table 11.1 gives the results. Column "Total cost" presents the total number of transitions traversed to activate all transitions. Column "Average cost per transition" gives the average number of transitions needed to activate an uncovered transition. It can be observed that the

Table 11.1 Statistics of TGTC algorithm for different protocols

	# States	# Transitions	BFS		TGTC			
			Total cost (transition)	Average cost per transition	Total cost (transition)	Average cost per transition	Improv. factor (%)	Test generation time (s)
MSI 8 cores	264	5256	36896	7.0	14664	2.8	60.3	<1
MESI 8 cores	272	5392	37712	7.0	15312	2.8	59.4	<1
MOSI 8 cores	1288	26248	196400	7.5	100807	3.8	48.7	6.2
MOESI 8 cores	1296	26384	197216	7.5	101455	3.8	48.6	6.2
MSI 16 cores	65552	2621968	29100096	11.1	11567888	4.4	60.2	54.4
MESI 16 cores	65568	2622496	29103264	11.1	11570464	4.4	60.2	54.5
MOSI 16 cores	589840	23855632	275254368	11.5	131122063	5.5	52.4	586

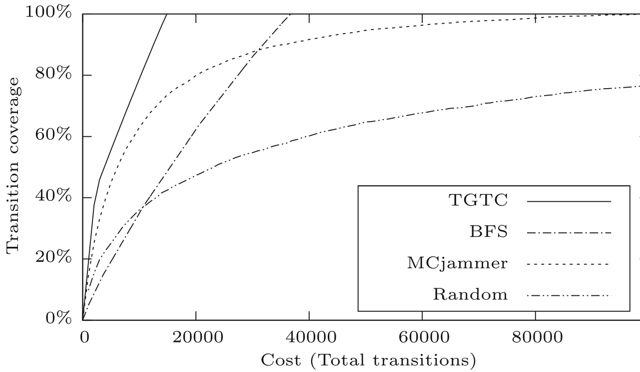


Fig. 11.6 Transition coverage versus cost for different test generation methods on MESI protocol with 8 cores

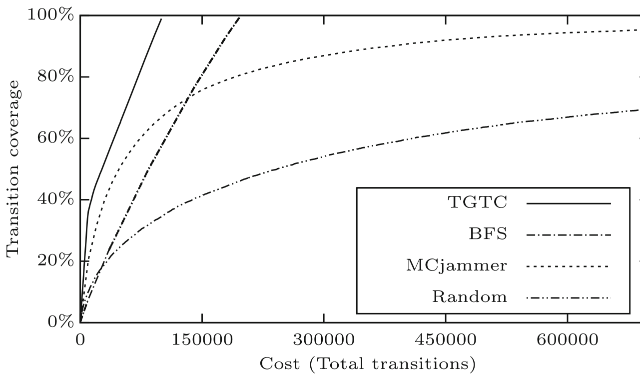


Fig. 11.7 Transition coverage versus cost for different test generation methods on MOSI protocol with 8 cores

total size of the tests generated by TGTC is 50–60 % smaller than the ones generated directly by BFS. This result can be explained by the fact that the Euler tour exploited in the TGTC algorithm typically covers load and evict transitions on global shared state. The store transitions on the other hand, are covered in a similar way as the BFS approach. Since the numbers of allowed load and evict transitions for any global state are equal, around half of the tests can be saved by exploiting the space structure.

The comparison of the state and transition coverage between TGTC and a directed random test generator MCjammer [10] is shown in Figs. 11.6 and 11.7. It can be seen that MCjammer is very efficient at the beginning. Actually, it is more efficient than BFS to achieve 70 % coverage. However, it becomes much slower to cover all transitions. The reason is that it is very unlikely for the algorithm with randomness to cover remaining uncovered transitions among all allowed transitions. On the other

hand, TGTC can always achieve 100 % state and transition coverage with stable higher coverage speed than the BFS-based tests.

Based on the experimental results, the overhead of TGTC can also be estimated. Although the TGTC algorithms are presented in recursive forms to simplify the presentation, they can also be implemented as iterative routines. As discussed in Sect. 11.4.1, TGTC have linear space complexity with the number of cores. Since the tests can be generated on-the-fly, its overall space requirement is very small. The test generation time in Table 11.1 suggests that the runtime of the TGTC algorithms is reasonable. For MOSI protocol with 23 million transitions, all the tests can be generated within 10 min, which indicates that TGTC is quite light-weighted for entire simulation based verification phase.

11.6 Chapter Summary

This chapter presented an efficient test generation approach for a wide variety of cache coherence protocols. Based on detailed analysis of the space structure, the on-the-fly test generation technique creates efficient test sequences for different parts of the global FSM state space to achieve 100 % state and transition coverage for each cache coherence protocol. Compared with existing approaches based on constrained-random test generation, this approach significantly increases the transition coverage metric with linear memory requirement. The experimental results on different cache coherence protocols demonstrated the effectiveness of this approach on systems with many cores, making it suitable for future multicore architectures.

References

1. Abts D, Scott S, Lilja D (2003) So many states, so little time: verifying memory coherence in the Cray X1. In: Proceedings of international parallel and distributed processing symposium
2. Adir A, Almog E, Fournier L, Marcus E, Rimon M, Vinov M, Ziv A (2004) Genesys-pro: innovations in test program generation for functional processor verification. IEEE Des Test Comput 21(2):84–93
3. Binkert N, Dreslinski R, Hsu L, Lim K, Saidi A, Reinhardt S (2006) The M5 simulator: modeling networked systems. IEEE Micro 26(4):52–60
4. Chen X, Yang Y, Delisi M, Gopalakrishnan G, Chou C-T (2007) Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In: Proceedings of HLVDT, pp 107–114
5. Dill D, Drexler A, Hu A, Yang C (1992) Protocol verification as a hardware design aid. In: Proceedings of international conference on computer design, pp 522–525
6. Edmonds J, Johnson EL (1973) Matching, Euler tours, and the Chinese postman. Math Program 5:88–124
7. Emerson E, Kahlon V (2003) Exact and efficient verification of parameterized cache coherence protocols. In: Proceedings of IFIP WG 10.5 advanced research working conference on correct hardware design and verification methods, vol 2860, pp 247–262

8. Hennessy J, Patterson D (2003) Computer Architecture: a quantitative approach. Morgan Kaufmann Publishers, San Francisco
9. Qin X, Mishra P (2012) Automated generation of directed tests for transition coverage in cache coherence protocols. In: Proceedings of the conference on design, automation and test in Europe, pp 3–8
10. Wagner I, Bertacco V (2008) Mcjammer: adaptive verification for multi-core designs. In: Proceedings of the conference on design, automation and test in Europe, pp 670–675
11. Wood D, Gibson G, Katz R (1990) Verifying a multiprocessor cache controller using random test generation. IEEE Des Test Comput 7(4):13–25
12. Zhange M, Lebeck A, Sorin D (2010) Fractal coherence: scalably verifiable cache coherence. In: Proceedings of MICRO, pp 471–482

Chapter 12
Reuse of System-Level Validation Efforts

12.1 Introduction

Increasing complexity of system-on-chip (SoC) architectures makes the demand of the high-level abstractions and analysis of SoC designs [1]. The functional errors of high-level specifications may result in inevitable malfunctions in low-level implementations. Therefore, it is a major challenge to guarantee the correctness of different abstractions [2, 3]. Validating each abstraction is necessary but time-consuming because it requires the profound understanding of the design. In addition, the inconsistency between different abstraction levels and the lack of automation techniques in each level aggravate the overall validation difficulty and workload. It is necessary to develop an approach that can automate the validation of high-level abstractions and reuse the validation effort among abstraction levels.

In SoC design, the top-down SoC design process starts from transaction-level modeling (TLM) [4] to register transfer level (RTL) implementation. As a system-level modeling specification, SystemC TLM [5] establishes a standard to enable fast simulation speed and easy model interoperability for hardware/software co-design. It mainly focuses on the communication between different functional components of a system and data processing in each component. Unlike TLM, RTL contains detailed information (such as interface and timing information) to describe the hardware behaviors. These differences limit the degree of validation reuse between TLM and RTL models. In the absence of significant reuse of design and validation efforts between different abstraction levels, the overall functional validation effort will increase since designers have to verify TLM as well as RTL models. Furthermore, the consistency between different abstraction levels should be guaranteed.

This chapter describes a top-down directed test generation methodology for both TLM and RTL designs. The basic idea is to use TLM specifications to perform coverage-based TLM test generation and generate RTL tests from TLM tests using a set of transformation rules. Since the generated TLM and RTL tests check the same functionality of the system, essentially they can ensure the consistency between TLM

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_12,

© Springer Science+Business Media New York 2013

specifications and RTL designs. This chapter also presents a prototype tool which enables automated directed RTL test generation from SystemC TLM specification. Since the RTL validation is based on the reuse of TLM validation effort, there is no extra cost (excludes defining the refinement rules) because it needs to be validated anyway.

The rest of this chapter is organized as follows. Section 12.2 presents related work on validation reuse and consistency checking between TLM and RTL models. Section 12.3 proposes our framework for TLM validation effort reuse. Section 12.4 presents the experimental results. Finally, Sect. 12.5 summarizes the chapter.

12.2 Related Work

TLM is promising to enable early design space exploration and hardware/software co-simulation. Hsiung et al. [6] adopted SystemC TLM models to enable rapid exploration of different reconfigurable design alternatives. In [7], Kogel et al. presented a SystemC-based methodology which provides sufficient performance, flexibility and cost efficiency as required by demanding applications. Shin et al. [8] proposed a method to automatically generate TLM models from virtual architecture models which can achieve significant productivity gains.

Simulation-based methods validate system using test vectors. They terminate when the required testing adequacy is achieved. Wang et al. [9] described a coverage directed method for transaction-level verification. The approach is based on random test generation and the coverage is increased by using fault insertion method. Although simulation using directed tests is fast, it is difficult to automate the directed test generation process. To enable automated analysis, various researchers have tried to extract formal representations [10–13] from SystemC TLM specifications. However, all these modeling techniques focus on the formal modeling and translation of SystemC specifications rather than directed test generation. It is hard to guarantee the functional coverage and correctness of the given specifications.

Reusing validation efforts between abstraction levels can reduce the overall validation time. There are various researches on validation reuse between TLM and RTL levels. Bombieri et al. [14] showed that transactor-based verification [15] is at least as efficient as a full RTL verification methodology which converts TLM assertions into RTL properties and creates new RTL testbenches. They also presented an incremental ABV methodology [16] to check the correctness of TLM-to-RTL refinement by reusing assertions. Jindal et al. [17] presented a method to reduce the verification time by reusing earlier RTL testbenches. Ara et al. [18] proposed an approach which combines transaction-level languages (e.g. SystemC) and the RTL level language (e.g. Verilog) based on Component Wrapper Language (CWL). By defining various test patterns using CWL, RTL verification suites from original specifications can be

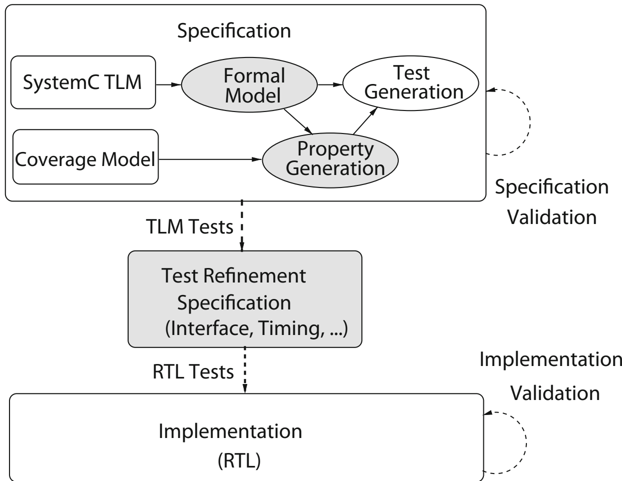


Fig. 12.1 Proposed RTL test generation methodology

quickly generated. Therefore, it can yield much shorter verification periods versus conventional methods.

12.3 RTL Test Generation from TLM Specifications

Figure 12.1 shows the framework of a novel RTL test generation methodology [19]. This methodology has three important steps: (i) translating SystemC TLMs to formal SMV specifications, (ii) deriving properties based on fault models to enable automated test generation, and (iii) refining TLM tests to RTL tests using the proposed test refinement specifications (TRSs). It is important to note that the test refinement is independent of how TLM tests are generated. In other words, test refinement can accept TLM tests generated by other approaches such as random test generation. The generated TLM tests can be used to validate TLM specifications. The refined RTL tests can be applied on RTL implementations for functional validation. Since Chaps. 2 and 3 have discussed the formal model extraction and directed test generation, this chapter focuses on the process of the TLM to RTL test refinement.

12.3.1 Automatic TLM Test Generation

This subsection first presents fault models to enable the automatic property generation. Then it introduces the TLM test generation method using model checking.

12.3.1.1 Property Generation Based on Fault Models

Test generation using model checking techniques requires that the automatically generated properties can cover as many desired scenarios in the design as possible. For test generation, properties are derived from a *fault model* which represents a complete set of specific errors. Each *fault* in the fault model indicates a potential "design error", which can be described by a temporal logic property. The test generated from such property can be applied on the design to check the specific scenario (negation of the fault). For example, when validating a desired scenario described by an LTL formula p, we use the negation $\neg p$ as a fault. By checking the property $\neg p$, we can derive a test to check the scenario where property p holds.

The properties generated from a fault model can guarantee the specific fault coverage in a design. Therefore, the testing coverage can be assured. In other words, a proper fault model with a good fault coverage determines the success of TLM test generation. The fault models for TLM presented here are inspired by the fault model based on *bit failures* and *condition failures* proposed in [20]. All such fault models can be easily obtained by analyzing the syntax of TLM models. For test generation, complex functional scenarios like "if communication C1 occurs before communication C2, then condition C3 will hold until communication C4 is asserted" are not considered. This is because the major concern of this chapter is the automation for directed test generation. By parsing the syntax of OSCI TLM models, it is difficult to figure out the complex dynamic semantics of a design automatically. It is important to note that the model checking based test generation does not exclude any properties written manually. The verification engineer can insert their properties after the SMV file generation and the corresponding TLM and RTL tests can be generated automatically as well.

In TLM, transaction data and transaction flow are recognized as two of the most important aspects. They indicate both the structure and behavior information. So as described in Sect. 3.4.3.2, this chapter focuses on such two fault models: transaction data fault model and transaction flow fault model .

Transaction data fault model deals with the possible variable assignments for each part of transaction data. However, in the property generation, due to the large size of value space, trying all the possible values of a data is time-consuming and impossible. In our experiment, we use the data bit fault model which checks each bit of a variable respectively. These model can not only partially guarantee the TLM data content coverage, but also increase the toggle coverage for the corresponding RTL designs. Since a transaction flow is a sequence of transactions, it can be used to reason the transaction ordering indirectly. Transaction flow fault model deals with the controls along the transaction flow. To ensure transaction flow coverage, all the branch conditions like *if–then-else*, *switch-case* statements along the transaction flow should be investigate. The goal is to check all possible transaction flows. Please note that the above models are not golden models. It is allowed that users can provide their own fault models to derive false properties for test generation. Based on the router example shown in Sect. 2.3.3, Fig. 12.2 presents two examples for these two fault models.

```
P1: For variable temp4, the first bit of parity cannot be 1.
  LTL formula: ˜ F (temp4.parity = 1 & temp4.to_chan != 0
        & temp4.payload_sz != 0);
P2: The condition "tmp_packet.to_chan=1" cannot be true.
  LTL formula: ˜ F(my_router.tmp_packet.to_chan = 1);
```

Fig. 12.2 Examples of two kinds of transaction faults and corresponding properties

```
// The property of a transaction flow fault
assert  ˜ F (my_router.tmp_packet.to_chan = 1);
// TLM Test
 p->to_chan = 1;
 p->payload_sz = 4;
 p->payload[0] = 128;
 p->payload[1] = 0;
 p->payload[2] = 0;
 p->payload[3] = 0;
 p->parity = 132;
```

Fig. 12.3 The TLM test for a transaction control fault

12.3.1.2 TLM Test Generation Using Model Checking

As described in Chap. 3, property falsification in model checking is promising for automated generation of directed test [21, 22]. The algorithm has two inputs: (i) model of the design in SMV specification and (ii) a set of properties derived from the specified fault models described in Sect. 3.4.3. During test generation, the model checker will generate one counterexample for each property. The generated counterexample is a sequence of variable assignments which can be transformed to a TLM test. Figure 12.3 shows an example of generated TLM test based on the transaction flow fault P2 shown in Fig. 12.2. It is derived from the condition of an "if–then-else" statement of the router example shown in Sect. 2.3.3. By applying this test on the TLM specification of the router example, the specified condition is assumed to be activated.

Clearly, model checking based approaches may be time-consuming in the presence of complex designs and properties. In these circumstances, various learning [23–26] and decomposition [27, 28] based optimization approaches described in Chaps. 5, 6, 7, 8 and 9 can be used to reduce the overall complexity of test generation.

12.3.2 Translation from TLM Tests to RTL Tests

A major challenge in test translation is how to bridge abstraction gap between TLM and RTL. For the same TLM specification, RTL designs may differ because of input/output definitions, timing details, programming styles and so on. So when converting TLM tests to RTL tests, it is required to provide necessary information

such as the input/output mappings between TLM and RTL as well as timing details of RTL input signals. For example, "$p \to to_chan$" in TLM is mapped to an input signal for "$DATA[1:0]$" in RTL.

To allow specifying rules for TLM to RTL test transformation, a mapping language TRS is developed. Since TLM tests only reflect the transaction data information, the TRS can analyze the transaction data in TLM tests and generate the corresponding RTL tests which are consistent to the interface protocol. One might argue that it may be easier to write RTL tests than writing TRSs. However, for the TLM tests which will be refined to the same RTL components, they share the same RTL input/output interface protocol. Generally, for each testing component, a large set of TLM tests will be generated. Most of them are only different with transaction data values. In other words, a large cluster of TLM tests can share one TRS. Therefore, it just needs to write several TRSs to cover all the testing scenarios which is time-efficient. In addition, the repeated sub-scenarios can be reused across TRSs. The overall automatic RTL test generation time can be significantly reduced. Generally, a TRS contains the following three parts:

- **Input/Output Mappings** specify the correspondence between TLM I/O variables and RTL I/O signals.
- **Patterns** are templates which define small segments of the test behavior. It can be used to compose various testing scenarios.
- **Timing Sequence** describes a complete scenario of input signals with timing information.

In this subsection, each part of the TRS will be discussed in details with illustrative examples. All these examples are based on the router example presented in Sect. 12.4.1.

12.3.2.1 Input/Output Mappings

During the TLM to RTL test translation, one important challenge is how to map TLM test data to its corresponding RTL test stimulus. Due to the difference between TLM data and RTL data, it is required to figure out the size information of each RTL signal as well as the bit correspondence between TLM data and RTL data.

In each mapping rule, the left hand side is the RTL data declaration, and the right hand side is the bit mapping from TLM data to RTL data. The TRS language allows the user to specify the RTL data using the concatenation of several TLM data. Also it supports the mapping from an array of TLM data to an array of RTL data. Figure 12.4 gives an example of the data mappings. In the example, $parity$ is an RTL data with 8 bits. It maps to the TLM variable $packet.parity$. The $header$ is an RTL data whose most significant six bits correspond to the TLM data $payload_sz$ and the least significant two bits correspond to the TLM data to_chan. The RTL data $payload$ is an array where the width of each element is 8 bits. The ith element $payload[i]$ corresponds to the ith element of the TLM data $packet.payload[i]$.

```
mapping_def:
   bit[7:0] parity=packet.parity;
   bit[7:0] header={packet.payload_sz[7:2], packet.to_chan[1:0]};
   bit[7:0] payload[0..packet.payload_sz-1]
             = packet.payload[0..packet.payload_sz-1];
end_mapping_def
```

Fig. 12.4 An example of mapping between TLM data RTL data

```
pattern reset()
    #5  RST = 1;
    #20 RST = 0;
end_pattern

pattern slave_read(int slave_no, int enable)
    #10 ENB%slave_no = %enable;
end_pattern
```

Fig. 12.5 Two examples of patterns

12.3.2.2 Patterns

When writing tests, some sub-scenarios may occur several times. To enable the reuse of scenario segments, TRS language introduces the construct *pattern* to group several statements together.

Essentially, the content of a pattern will substitute for the pattern statements in the timing sequence like a macro. Thus the usage of the pattern can reduce the programming time as well as increase the programming flexibility. In TRSs, a pattern can have parameters. During pattern text substitution, the tags defined in patterns will be replaced with the given value of parameters. Figure 12.5 presents two examples of patterns *reset* and *slave_read*. The pattern *reset* has no parameters. Its content will be directly embedded at the place of the pattern statement. The pattern *slave_read* has two parameters to indicate which slave will be enabled.

12.3.2.3 Timing Sequence

The timing sequence in TRS composes a sequence of statements and pattern instances to describe a testing scenario. According to the definition of input/output mappings and patterns, the compiler will translate testing scenarios described in timing sequence to corresponding RTL tests. Figure 12.6 presents an example of a timing sequence. It describes a testing scenario of the packet delivering for a router as follows: (i) a master sends a packet to a router, (ii) the router holds the packet and

```
SPEC router(packet)
    ......
    main:
     begin
       initialize();
       reset();
       -- the master sends a packet
       #5 PKT_VALID = 1'b 1;
       DATA = header;
       for(int i=0; i<packet.payload_sz; i++){
           #10 DATA = parity[i];
       }
       #10 PKT_VALID = 1'b 0;
       DATA = parity ;
       -- a slave receives the packet
       slave_read(packet.to_chan, 1);
       FINISH();
     end
END_SPEC
```

Fig. 12.6 An example of a timing sequence

notifies the corresponding slave to fetch the packet, and (iii) the slave receives the packet.

12.3.3 A Prototype Tool for TLM-to-RTL Validation Refinement

A prototype tool, called *Automatic RTL Test gEnerator from SystemC TLM* (*ARTEST*), is developed to incorporate the presented methods. Figure 12.7 shows both the structure and workflow of the tool. The following subsections will briefly introduce its three key components: (i) *TLM2SMV* for SMV model and property generation, (ii) TLM test generation using model checking, and (iii) *TLM2RTL* for RTL test generation.

12.3.3.1 TLM2SMV

Implemented based on the C++ parser Elsa [29], *TLM2SMV* can automatically translate the SystemC TLM to a SMV specification and derive properties based on the fault models. Due to the complex data type definition and complex constructs defined in SystemC TLM library files, direct translation to SMV will cause the state space explosion in the model checking stage. Instead, such definitions can be simplified and they can be predefined for SMV transformation. For example, we can restrict the queue size for TLM FIFO channels. In SystemC, an integer is 32-bit (with $2^{32} states$).

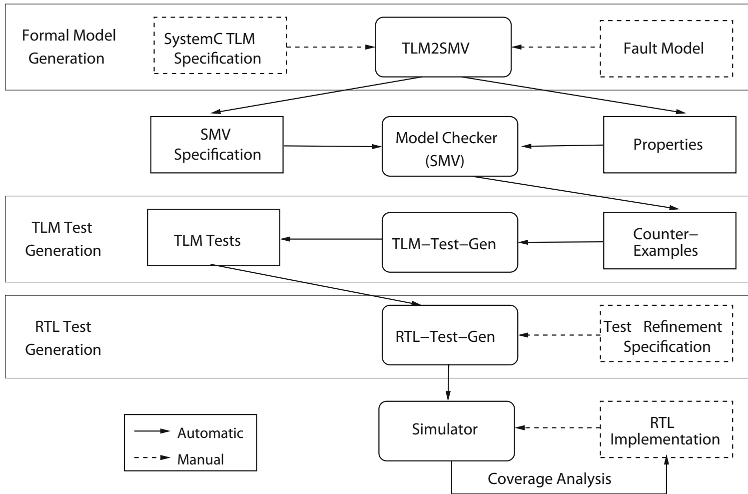


Fig. 12.7 The structure of the prototype tool ARTEST

However, we can reduce its size to 8 bits (with $2^8 states$) during the SMV transformation.

Before the TLM to SMV translation, preprocessing procedure of *TLM2SMV* will do the following three tasks: (i) eliminate the header files and the comments, (ii) add the necessary predefine constructs and (iii) convert the data type if necessary. Then *TLM2SMV* will start to transform the TLM specification. As described in Sect. 2.3.2, *TLM2SMV* will extract both static and dynamic information. In the meantime, it also explores the information such as transaction relevant data, branch conditions for the property generation. Finally, based on the collected information, a formal specification in SMV and properties derived by specified fault models can be both achieved. By using a suitable model checker (e.g., Cadence SMV checker [30]), a set of counterexamples are derived, and the TLM tests can be extracted from these counterexamples.

12.3.3.2 TLM Test Generation

When a specified safety property is false, SMV model checker will generate a counterexample to falsify it. A generated TLM counterexample is in the form of a sequence of state assignments. This sequence starts from first state (initial state) and ends at the error state which violates the property. If the cone of influence (COI) is enabled during the property checking, each state will only contain the variables which are relevant to the specified property. The generated counterexample is refined to produce the TLM test.

12.3.3.3 TLM2RTL

Since SystemC TLM focuses on the system-level modeling, the generate TLM tests lack the implementation-level knowledge. So the generated TLM tests are different with RTL tests and cannot be directly used to validate RTL implementation. For example, most loosely-timed TLM models are too abstract and assume that a transaction happened in one or a sequence of function calls. However, an RTL design has much more pins and it needs the detailed timing information for each signal. In our framework, the user should provide a TRS which provides the mapping rules for the TLM to RTL test translation. With the generated TLM tests and the TRS as inputs, the *TLM2RTL* can translate the TLM tests to RTL tests. Finally, the tool automatically reports the coverage of the TLM specification during the simulation of the generated RTL tests on the RTL implementation.

12.4 Case Studies

In this section, two case studies are presented to show the effectiveness of the presented method. The results are obtained while running *ARTEST* on a 2 GHz AMD Opteron Processor with 8G RAM using Linux operating system.

12.4.1 A Router Example

Section 2.3.3 has presented the details of the router example. Figure 2.8 shows the TLM structure of the router. The router consists of five modules: one master, one router and three slaves. It consists of four classes, eight functions, and 143 lines of code. The main function of the router is to analyze and distribute packets received from the master to target slaves.

At the beginning of a transaction, the master module creates a packet which is in the form as shown in Fig. 12.8a. To make the TLM packet description clear, Fig. 12.8b shows the organization of the corresponding RTL packet. The packet consists of three parts: header, payload and parity. The header has 8 bits, bit 0 and bit 1 are used as the address of output port. The other 6 bits indicate the size of the payload. So the maximum payload size is 63. The last byte of the packet is the parity of both header and payload. After that, the driver sends the packet to the router for package distribution. The router has one input port and three output ports. Each port is connected to a FIFO buffer (channel) which temporarily stores packets. The router has one process *route* which is implemented as a *SC_METHOD*. The *route* first collects a packet from the channel connected to the driver, decodes the packet header to get the target address of a slave, and then sends the packet to the channel connected to the target slave. Finally, the slave modules will read the packets when data is available in the respective FIFOs. The transaction data (i.e., packet) flows from

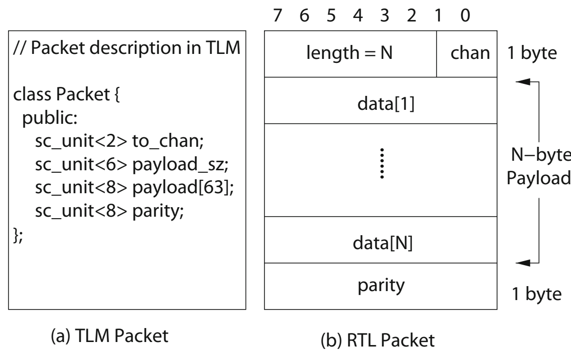
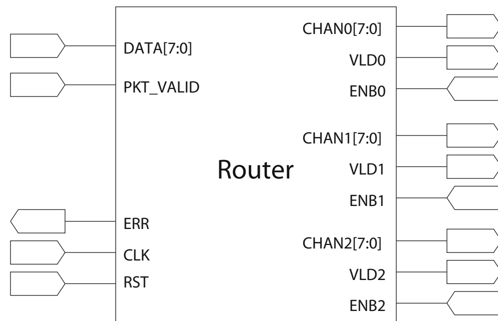


Fig. 12.8 The packet format of the router in **a** TLM and **b** RTL

Fig. 12.9 Block diagram of design under test



the master to its target slave via the router. By definition, this flow is determined by the address *to_chan* in the packet header.

The following subsections present the workflow of the RTL test generation and provide the validation result of the router implementation.

12.4.1.1 RTL Tests Generation

As described in described in Sect. 2.3.2, the tool *ARTEST* can derive SMV inputs from SystemC TLM specifications. Meanwhile, by applying different fault models defined in Sect. 3.4.3, a set of properties can be generated, where each property corresponds to a test generated using property falsification. As a high-level modeling language, SystemC TLM lacks the information of pins and timing in low-level implementation. The generated TLM tests are not appropriate as the inputs of RTL designs. Therefore, it is necessary to provide an interface mapping to enable TLM-to-RTL test translation.

Figure 12.9 shows the input/output interfaces of the router. This RTL information and other TLM details (such as the packet description in Fig. 12.8) are used to perform mapping between TLM variables and RTL signals. For example,

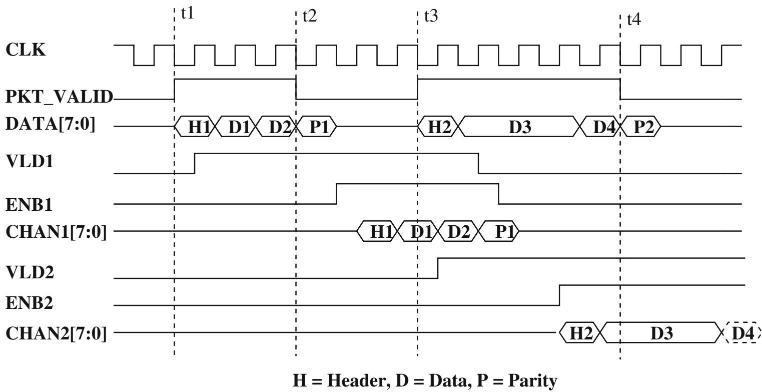


Fig. 12.10 Timing chart for the router example

"*packet.to_chan*" in TLM corresponds to the RTL data "*header*[1:0]" and "*packet.payload_sz*"corresponds to "*header*[7:2]". And the array of TLM data *packet.payload* will be mapped to RTL data "*payload*". Such information should be defined in the *mapping definition* of TRS as shown in Fig. 12.4.

The TRS of the RTL tests are derived from the timing specification of the RTL implementation. Figure 12.10 presents the timing chart of a testing scenario. From this chart, we can extract the timing specification for the router as follows. All input/output signals are active high and are synchronized by the falling edge of the clock. The *PKT_VALID* signal has to be asserted on the same clock when the first byte of the packet (the header byte) is driven onto the data bus. Each subsequent byte of data should be driven on the data bus with each new falling clock. After the last payload byte has been driven, on the next falling clock, the *PKT_VALID* signal must be deasserted (before the parity byte is driven). The packet parity byte should be driven on the next falling clock edge. The router asserts the *VLDx* ($x \in \{0, 1, 2\}$) signal when valid data appears on the *CHANx* output. The *ENBx* input signal must then be asserted on the falling clock edge in which data is read from the *CHANx* bus. As long as the *ENBx* signal remains active, the *CHANx* bus drives a valid byte on each rising clock edge. Such timing information needs to be extracted and described in the timing sequence of TRS.

Section 12.3.2 provides the details of the router TRS. The RTL tests can be obtained by using this specification. Figure 12.11 shows both TLM and RTL tests corresponding to the transaction flow fault shown in Fig. 12.2. The first part of the RTL test contains the initialization of the RTL input variables. The second part contains the reset sequence. The third part contains the assignment to *P*KT_VALID signal. The subsequent entries in the RTL test is generated by transforming corresponding TLM entry by using a combination of name mapping, delay insertion and composition of values (used in one case). Finally, the *PKT_VALID* signal needs to be low before sending the parity followed by assignment of read enable signals for

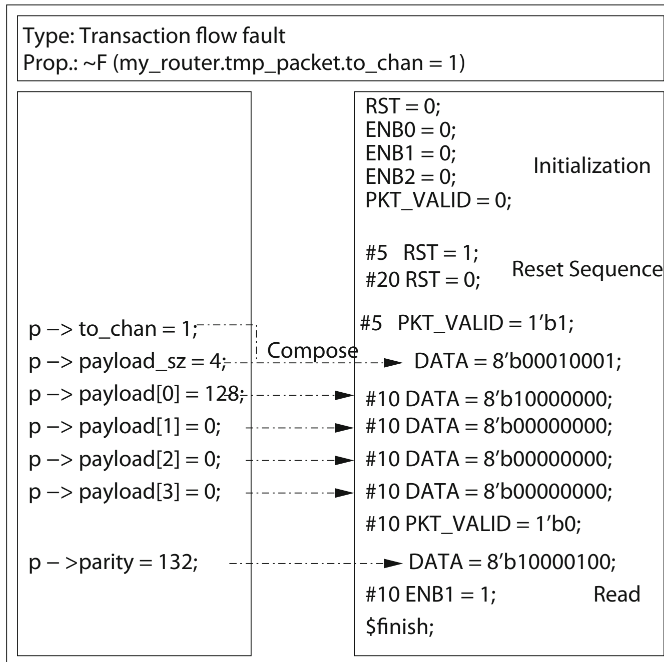


Fig. 12.11 TLM to RTL test transformation in the router example

four time steps (to read four entries: header, two data elements and parity) so that the slaves can read the packet.

To increase the RTL coverage, several RTL tests, which are not related to the proposed fault models, are manually generated. These tests are required to cover the additional functionalities in RTL that are not available in TLM. For example, TLM does not have notion of *reset* signal. Therefore, it needs to generate RTL tests related to reset check operations and so on.

12.4.1.2 RTL Validation and Analysis

A total of 92 TLM tests are generated: 4 based on transaction flow fault model and 88 for transaction data fault model. It is important to note that the TLM test generation and RTL test translation are independent. In other words, TLM tests can come from multiple sources. However, the tool will automatically convert TLM tests to RTL tests. As aforementioned, to increase the RTL coverage, 4 RTL tests are manually created based on FIFO overflow, reset check, and asynchronous read. Finally 92 TLM tests and 96 RTL tests are generated for validation.

To show the effectiveness of the directed tests, both random tests and directed tests are applied on the RTL implementation of the router. Various coverage metrics are measured using the Synopsys VCS cmView [31]. Table 12.1 shows the coverage

Table 12.1 RTL coverage results of the router example

Test type and number	Line (%)	Condition (%)	FSM (%) state/transition	Toggle (%) regs/nets	Path (%)	Time (min)
Rand_{100}	98.63	51.06	75.0/37.5	56.76/48.91	51.39	0.07
Rand_{1000}	99.92	53.19	100/62.5	56.76/57.61	56.94	1.23
Rand_{10000}	99.99	46.81	75/37.5	64.86/70.65	58.33	17.63
Directed	98.89	72.34	75.0/37.5	59.46/68.48	68.06	0.08
Rand_{100} + Directed	99.55	72.34	75.0/37.5	78.38/81.52	68.08	0.08
Rand_{1000} + Directed	99.97	78.72	100/62.5	78.38/84.78	73.61	1.35
Rand_{10000} + Directed	99.99	78.72	100/62.5	81.08/85.87	73.61	17.43
Extended + Directed	99.48	78.72	100/75	79.97/80.43	73.61	0.10

results. The first row indicates the RTL coverage metrics. The second to fourth rows show various coverage using 100, 1000 and 10000 random tests respectively. Although the number of random tests increases exponentially, there is no drastic improvement on the coverage ratio. The fifth row shows the coverage results using the generated directed tests. It shows that the method using directed tests can achieve better RTL coverage with significantly fewer tests. The sixth to eighth rows present the coverage result that combines both random tests and the directed tests. The results indicate that the presented method can activate the functional scenarios that are difficult to be activated by the random method. For example, in the third row and seventh row, it can be found that coverage using the random method can be further improved by adding the derived directed tests. This is because the directed tests are derived from TLM designs and carry the system-level information. To further improve the coverage result using directed method in the fifth row, the last row gives the coverage with four extended manual tests. It can achieve the best coverage (except the line coverage and toggle coverage) with much shorter simulation time.

Several fatal errors have been identified during validation of the RTL implementation using the generated directed tests. The first error is encountered when a FIFO buffer is empty and slave tries to read the corresponding channel, the empty FIFO buffer becomes full! This is due to the incorrect implementation of FIFO size which is always decremented without zero check. The second one occurred if the destination of packet is "channel 3". In this case the packet should be discarded, but in RTL the data is written to the "channel 0". Also, one of the directed tests identified an inconsistency between TLM and RTL FIFO implementations: the overflow in TLM level is 16 packets whereas the overflow in RTL is 16 bytes.

12.4.2 A Pipelined Processor Example

This subsection first presents the TLM model of a pipelined processor and associated TLM test generation. Next, it presents the TRS specification for RTL test generation. Finally, it discusses the results of the RTL design validation using generated tests.

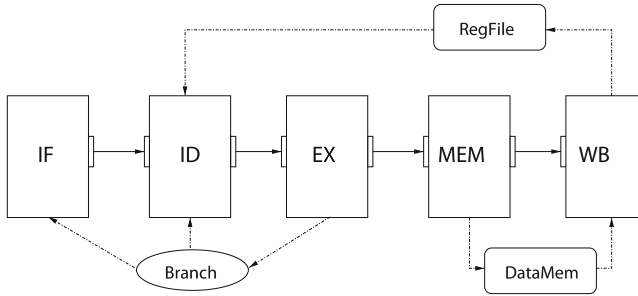


Fig. 12.12 The TLM graph model of the Alpha AXP processor

```
// Property derived from a transaction data fault
assert ~F(data_memory[3]=2);
//TLM Test
LDQ R0, 2(R0)       // register[0] = 2;
STQ R0, 3(R1)       // register[3] = 2;
```

Fig. 12.13 A TLM test for the Alpha AXP processor

12.4.2.1 TLM Test Generation

Figure 12.12 shows a simplified version of the Alpha AXP processor [32]. It consists of five stages: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM) and Write-back (WB). IF module fetches instructions from the instruction memory. ID module decodes instructions and reads the values of the operands if necessary. EX module does ALU operations, also it will notify whether the conditional or unconditional branch happens. Memory module reads and writes data to the data memory. Write-back module stores the result to specified registers. The communication between two modules uses the port binding associated with a blocking FIFO channel with one slot. For example, there is a binding from the *port* of IF module to the *export* of ID module, and the *export* of ID module binds to a blocking FIFO channel for holding incoming instructions. So each time, the IF module can only issue one instruction to ID module; otherwise it will be blocked. The whole TLM design contains 6 classes, 11 functions and 797 lines of code.

During the TLM to SMV translation, global data structures (e.g., register file, data memory) are defined in the SMV main function, and they are used as the input and output parameters by each modules. Initially, the program counter (PC) starts from 0 and the value of registers and memories are all 0. In this example, we use both transaction data fault model and transaction flow fault model to derive properties. Figure 12.13 presents an example of test generation for a transaction data fault.

```
SPEC Alpha_AXP(inst1, inst2)
mapping_def:
    bit[31:0] memory_1 =   {inst1.op[31:26], inst1.ra[25:21],
          inst1.rb[20:16], inst1.mem_disp[15:0]};
    bit[31:0] memory_2 =   {inst2.op[31:26], inst2.ra[25:21],
          inst2.rb[20:16], inst2.mem_disp[15:0]};
    ....
end_mapping_def

pattern initialize()
    Imem2proc_bus = 64'h 0000_0000_0000_0000;
    #2 RESET_=0;
end_pattern

pattern reset()
    RESET_ = 1;
    #2 RESET=0;
end_pattern

main: begin
    initialize();
    reset();
    if (inst1.class == MEMORY)
       Imem2proc_bus[31:0] = memory_1;
    ....
    if (inst2.class == MEMORY)
       #2 Imem2proc_bus[63:32] = memory_2;
    ....
    #2 FINISH();
end
END_SPEC
```

Fig. 12.14 A test refinement specification for the Alpha AXP processor

12.4.2.2 RTL Test Generation

In the Alpha AXP processor, there are four types of instructions: *CALL_PAL*, *OPERATE*, *BRANCH* and *MEMORY*. For *OPERATE* instructions, there are three different instruction formats. Figure 12.14 shows a partial TRS description that is used to translate the processor TLM tests to RTL tests. There are two input ports for the RTL design of the processor: *RESET* for resetting all five stages, and 64-bit signal *Imem2proc_bus* which contains two 32-bit instructions. In every two clock cycles, the processor fetches one 32-bit instruction from the instruction memory through the bus connected to the instruction memory. Since there are four different types of instructions, in the *mapping_def* part, it is necessary to list four different instruction formats. And in the timing sequence part, the input signals to *Imem2proc_bus* will be determined by the instruction class information included in TLM tests.

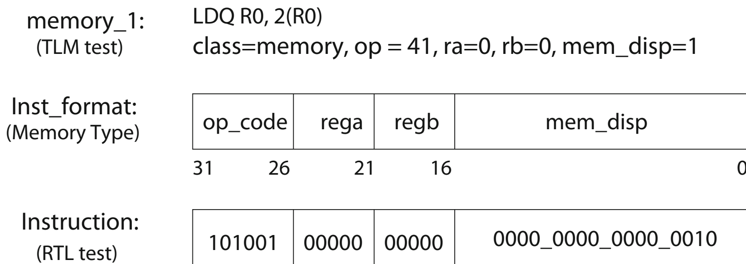


Fig. 12.15 The TLM to RTL instruction mapping of Alpha AXP processor

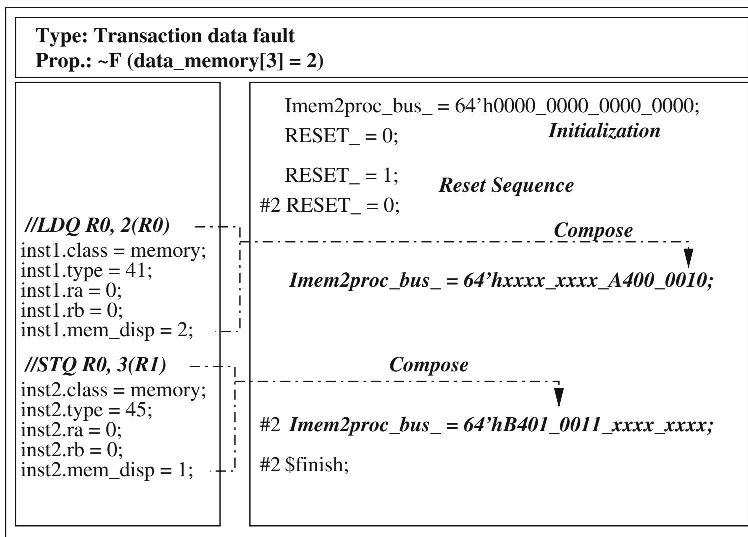


Fig. 12.16 A TLM to RTL test transformation in the Alpha AXP processor

Figure 12.15 shows the mapping from a TLM instruction to an RTL instruction. Since the given TLM test is of memory type, according to the TRS mapping information defined in Fig. 12.14, the 32-bit instruction contains four segments: opcode, register *rega*, register *regb* and memory address displacement. The mapping rules provide both value and place information for the transformation.

The *Alpha_AXP* TRS is applied on the TLM tests generated from the SMV counterexamples. Figure 12.16 shows an example of the transformation from a TLM test to an RTL test. The left part shows a TLM test with two TLM instructions, and the right part presents its corresponding RTL test. During the test transformation, each TLM instruction in the left part data will be composed and mapped to a 64-bit input RTL signal.

Table 12.2 RTL coverage results for the Alpha AXP Processor

Test type and number	Line (%)	Condition (%)	FSM (%) state/transition	Toggle regs/nets	Path (%)	Time (min)
$Random_{100}$	97.63	82.93	NA	67.69/65.89	60.27	0.20
$Random_{500}$	99.68	82.93	NA	69.23/66.36	72.60	0.35
$Random_{5000}$	99.98	82.93	NA	70.77/68.22	80.82	2.32
$Random_{50000}$	99.99	82.93	NA	70.77/68.22	80.82	23.18
Directed	98.94	95.73	NA	87.69/81.32	86.30	0.83
Directed + $Random_{100}$	99.92	96.34	NA	89.23/82.24	90.41	1.20
Directed + $Random_{500}$	99.98	96.34	NA	89.23/82.24	90.41	1.10
Directed + $Random_{5000}$	99.99	96.34	NA	89.23/82.24	90.41	3.10
Directed + $Random_{50000}$	99.99	97.56	NA	89.23/82.24	95.89	23.30

12.4.2.3 Validation Results

The test generation for the Alpha AXP processor is based on the transaction data and flow fault models. The transaction data faults mainly indicate the bit value change for each transaction variable and global variable such as data memory, register file, data forward and branch status. The transaction flow faults indicate the instruction category and instruction execution. Overall, there are 212 TLM tests generated, including 86 tests for condition faults and 126 tests for data bit faults. It costs 311.47 minutes to achieve all these tests using Cadence SMV verifier [30]. We also use the BMC tool NuSMV [33] to optimize the test generation time. By using NuSMV, the test generation time just needs 3.23 minutes.

Since some of the generated tests are redundant (same test) and can be removed, finally 112 TLM tests are generated, including 50 tests for transaction flow faults and 62 tests for transaction data faults. Both random tests and directed tests are applied on the RTL implementation to measure the effectiveness of the directed tests. In this example, 100, 500, 5000 and 50000 random RTL tests are derived respectively. The directed RTL tests are generated using the TRS presented in Sect. 12.4.2.2. The coverage results are shown in Table 12.2. For the condition coverage, there is no improvement with more random tests. It is important to note that, in this example, the test generation of 50000 random tests costs 23.18 minutes, while our 212 directed tests derived using bounded model checker just needs 3.23 minutes. Moreover, the proposed method using directed tests can achieve better coverage result (except line coverage) than the random method with less time. The result, which combines both random tests and directed tests, shows that the directed method can activate the functional scenarios that are difficult for random methods to explore. For the example in the fifth row, when applied 50000 random tests, the path coverage ratio is 80.82 %. However, by adding the directed tests incrementally in the tenth row, the path coverage ratio increases to 95.89 %.

12.5 Chapter Summary

Raising the abstraction level in SoC design flow can significantly reduce the overall design effort but introduce two challenges: (i) how to guarantee functional consistency between system-level designs and low-level implementations, and (ii) how to reuse validation effort between different abstraction levels. To address both problems, this chapter presented a methodology which reuses TLM validation effort to enable RTL validation as well as functional consistency checking between TLM and RTL models. By extracting formal models from TLM specifications, a set of TLM tests can be generated to validate all the specified TLM "faults". The TLM tests can be translated to their RTL counterparts using the presented test refinement specification. During the simulation, the TLM-to-RTL functional consistency can be verified based on the corresponding outputs. The case studies demonstrated that the RTL tests generated by the presented method can achieve the intended functional coverage.

References

1. Abrar S, Thimmapuram A (2010) Functional refinement: a generic methodology for managing ESL abstractions. In: Proceedings of international conference on VLSI design (VLSID), pp 122–127
2. Bombieri N, Fummi F, Pravadelli G, Marques-Silva J (2007) Towards equivalence checking between TLM and RTL models. In: Proceedings of international conference on formal methods and models for co-design (MEMOCODE), pp 113–122
3. Bruce A, Hashmi M, Nightingale A, Beavis S, Romdhane N, Lennard C (2006) Maintaining consistency between SystemC and RTL system designs. In: Proceedings of design automation conference (DAC), pp 85–89
4. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of international conference on hardware/software codesign and system, synthesis (CODES+ISSS), pp 19–24
5. Ghenassia F (2005) Transaction-level modeling with SystemC: TLM concepts and applications for embedded systems. Springer, Dordrecht
6. Hsiung P, Lin C, Liao C (2008) Perfecto: a SystemC-based design-space exploration framework for dynamically reconfigurable architectures. ACM Tran Reconfigurable Technol Syst (TRETS) 3(1):69–81
7. Kogel T, Doerper M, Kempf T, Wieferink A, Leupers R, Meyr H (2008) Virtual architecture mapping: a SystemC based methodology for architectural exploration of system-on-chips. Int J Embed Syst (IJES) 3(3):150–159
8. Shin D, Gerstlauer A, Peng J, Dömer R, Gajski D (2006) Automatic generation of transaction level models for rapid design space exploration. In: Proceedings of international conference on hardware/software codesign and system, synthesis (CODES+ISSS), pp 64–69
9. Wang Z, Ye Y (2005) The improvement for transaction level verification functional coverage. In: Proceedings of international symposium on circuits and systems (ISCAS), pp 5850–5853
10. Abdi S, Gajski D (2005) A formalism for functionality preserving system level transformations. In Proceedings of Asia and South Pacific design automation conference (ASPDAC), pp 139–144
11. Moy M, Maraninchi F, Maillet-Contoz L (2005) Lussy: a toolbox for the analysis of systems-on-a-chip at the transactional level. In: Proceedings of the international conference on application of concurrency to system design, pp 26–35

12. Karlsson D, Eles P, Peng Z (2006) Formal verification of systemc designs using a petri-net based representation. In: Proceedings of design, automation, and test in Europe (DATE), pp 1228–1233
13. Habibi A, Tahar S (2006) Design and verification of SystemC transaction-level models. IEEE Trans Very Large Scale Integr Syst (TVLSI) 14(1):57–68
14. Bombieri N, Fummi F, Pravadelli G (2006) On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL. In: Proceedings of design, automation, and test in Europe (DATE), pp 1–6
15. Balarin F, Passerone R (2006) Functional verification methodology based on formal interface specification and transactor generation. In: Proceedings of design, automation, and test in Europe (DATE), pp 1013–1018
16. Bombieri N, Fummi F, Pravadelli G (2007) Incremental ABV for functional validation of TL-to-RTL design refinement. In: Proceedings of design automation and test in Europe (DATE), pp 882–887
17. Jindal R, Jain K (2003) Verification of transaction-level SystemC models using RTL testbenches. In: Proceedings of the international conference on formal methods and models for co-design (MEMOCODE), pp 199–203
18. Ara K, Suzuki K (2003) A proposal for transaction-level verification with component wrapper language. In: Proceedings of design automation and test in Europe (DATE): designers' forum, p 20082
19. Chen M, Mishra P, Kalita D (2012) Automatic RTL test generation from systemC TLM specifications. Accepted to appear in ACM Transactions on Embedded, Computing Systems (TECS)
20. Ferrandi F, Fummi F, Gerli L, Sciuto D (1999) Symbolic functional vector generation for VHDL specifications. In: Proceedings of design automation and test in Europe (DATE), pp 442–446
21. Kupferman O, Vardi M (1999) Vacuity detection in temporal model checking. In: Proceedings of correct hardware design and verification methods (CHARME), pp 82–96
22. Mishra P, Dutt N (2008) Specification-driven directed test generation for validation of pipelined processors. ACM Trans Des Autom Electron Syst (TODAES) 13(3):1–36
23. Strichman O (2001) Pruning techniques for the SAT-based bounded model checking problem. In: Proceedings of correct hardware design and verification methods (CHARME), pp 58–70
24. Chen M, Mishra P (2010) Functional test generation using efficient property clustering and learning techniques. IEEE Trans Comput-Aided Des Integr Circuits Syst (TCAD) 29(3):396–404
25. Chen M, Mishra P (2011) Property learning techniques for efficient generation of directed tests. IEEE Trans Comput (TC) 60(6):852–864
26. Chen M, Qin X, Mishra P (2010) Efficient decision ordering techniques for SAT-based test generation. In: Proceedings of design, automation and test in Europe (DATE), pp 490–495
27. Chen M, Mishra P (2011) Decision ordering based property decomposition for functional test generation. In: Proceedings of design, automation and test in Europe (DATE), pp 167–172
28. Koo H, Mishra P (2009) Functional test generation using design and property decomposition techniques. ACM Trans Embed Comput Syst (TECS) 8(4): 32:1–32:33
29. McPeak S (1999) Elsa. http://www.eecs.berkeley.edu/smcpeak
30. McMillan K (2002) SMV model checker, cadence berkeley laboratory. http://embedded.eecs.berkeley.edu/Alumni/kenmcmil/smv
31. SYNOPSYS (2007) VCS verification library. http://www.synopsys.com
32. Sites R (1992) Alpha AXP architecture. Digit Tech J 4(4):51–65
33. FBK-irst, CMU (2006) NUSMV. http://nusmv.irst.itc.it/

Chapter 13
Conclusions

13.1 Summary

Existing SoC validation techniques employ a combination of simulation-based techniques and formal methods. Simulation is the most widely used approach for SoC validation using random or constrained-random test vectors. Although certain heuristics are used to generate constrained-random tests, due to the bottom-up nature and localized view of these heuristics, the generated tests may not yield a good coverage. Simulation using directed tests is promising since it can achieve the same coverage goal using orders-of-magnitude less directed tests compared to random or constrained-random tests, and therefore can drastically reduce the overall validation effort. However, directed test generation is mostly performed by human intervention. Handwritten tests entail laborious and time-consuming effort of verification engineers who have deep knowledge of the design under verification. For a complex design, it is infeasible to manually generate all directed tests to achieve a comprehensive coverage goal. Therefore, it is necessary to develop tools and techniques for automated generation of directed tests. This book presented efficient techniques to address various challenges associated with automated generation and reuse of directed tests across different abstraction levels.

A major challenge to enable directed test generation is to automatically extract a formal representation from system-level specification and develop an efficient coverage metric that allows coverage-driven directed test generation. Chapter 2 described how to automatically extract formal models from the high-level specifications including SystemC TLM models and UML activity diagrams. Based on various high-level design and fault models, Chap. 3 described efficient techniques for automated generation of directed tests. The basic idea of this approach is to generate one property for each fault in the fault model and then invoke a model checker using the generated property and the formal model of the design. The model checker produces a counterexample that can be used as a directed test to activate the intended fault in the design. Most automated test generation methods, especially for model checking based techniques, face the capacity restrictions of corresponding tools. In other

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2_13,
© Springer Science+Business Media New York 2013

words, test generation may be infeasible for complex SoCs since model checking can lead to state space explosion. This book presented five efficient techniques to reduce validation (test generation) complexity.

- *Functional Test Compaction*. Although directed tests are more effective compared to random tests to reduce the validation effort, the number of directed tests can still be prohibitively large. Chapter 4 presented a compaction technique by removing the redundant properties/tests to drastically reduce the number of directed tests to achieve the required coverage goal.
- *Property Clustering*. Directed test generation can be viewed as a searching process in a complex state machine. It is possible to waste searching effort by traversing paths that are not useful. Existing techniques exploit this fact by adding conflict clauses (in SAT-based BMC) to avoid making the same mistake again for the same property. Since test generation for SoC validation involves a lot of properties, it is likely that a set of similar properties may involve similar search paths. As a result, it would be promising to share learning (conflict clauses) between similar properties. Chapter 5 described four efficient techniques to cluster a set of similar properties to enable knowledge sharing across test generation instances.
- *Learning Techniques*. Efficient learning techniques can significantly reduce the test generation time by adopting conflict clauses or decision ordering based learning methods. Chapter 5 described efficient learning techniques through identification and reuse of common conflict clauses to reduce the overall test generation time for a cluster of similar properties. Similarly, Chap. 6 presented efficient decision-ordering techniques to improve the overall test generation time for a single property as well as for a cluster of similar properties. The assumption here is that only one property is solved at a time. As a result, it allows either learning during solving of one property with different bounds, or solving multiple properties with known bounds. Chapter 7 presented a framework to simultaneously solve all the similar properties to ensure that the knowledge obtained in previous solving iterations be shared across different bounds as well as between different properties.
- *Design and Property Decompositions*. It is promising to decompose the design and properties to convert a complex property checking scenario into several simple property checking instances. Chapter 8 described various design and property decomposition techniques to reduce the counterexample search space. This method produces several local counterexamples due to property checking involving decomposed design and properties. Chapter 8 presented an algorithm for merging local counterexamples to generate the final test (global counterexample). Composition of local counterexamples can be hard in some scenarios. Instead of using local counterexamples, Chap. 9 uses learning from solved subproperties to reduce the solving time of the original property.

Multicore architectures are widely used in today's desktop and embedded computing systems to circumvent the power wall and memory wall encountered by single core architectures. While more and more cores are integrated into SoCs to boost the throughput, their increasing complexity also introduces significant verification

challenges. This book presented two efficient techniques to reduce the test generation complexity in the presence of multicore SoCs.

- *Test Generation for Multicore Architectures*. When SAT-based BMC is applied to generate directed tests for multicore architectures, there are three different categories of symmetry in the corresponding SAT instances. The first category is the temporal symmetry. It occurs because the SAT instance is encoded by unrolling the same architecture for multiple times. The temporal and spatial symmetry are exploited by existing techniques by sharing knowledge between bounds and between similar properties, respectively. The structural similarity of multiple cores also introduces another category of symmetry or structural symmetry. Chapter 10 presented an efficient framework to share learning across increasing bounds, among different cores as well as between related properties at the same time to drastically reduce the overall test generation time.
- *Validation of Cache Coherence Protocols*. Processors with multiple cores and complex cache coherence protocols are widely employed to improve the overall performance. It is a major challenge to verify the correctness of a cache coherence protocol since the number of reachable states grows exponentially with the number of cores. Chapter 11 presented an on-the-fly test generation technique for cache coherence protocols by analyzing the state space structure of their corresponding global FSMs. Instead of using structure-independent breadth-first-search (BFS) to obtain the directed tests, the complex state spaces of cache coherence protocols are decomposed into several components with simple structures. Since the activation of states and transitions can be viewed as a path searching problem in the state space, these decomposed components with known structures can be exploited for efficient test generation.

Finally, Chap. 12 described a framework for automated reuse of high-level tests and assertions for validation of implementation. In this framework, SoC design employs TLM specification and RTL implementation, and the overall validation effort consists of validation of both specification and implementation. It is observed that the overall effort can be significantly reduced if TLM-level tests and assertions can be reused for RTL validation. The basic idea is to use TLM specification to perform coverage-based TLM test generation, and convert TLM tests to RTL tests using a set of transformation rules. Development of efficient test generation and reuse techniques can drastically reduce the overall SoC validation effort and will lead to cost-effective and reliable systems.

13.2 Future Directions

Functional validation is a major challenge in SoC design. This book presented various efficient system-level validation techniques, which can effectively reduce the overall validation effort. The work presented in this book can be extended in the following directions.

The coverage-driven property generation may generate a large set of properties for complex SoCs, and many of them may activate the same set of scenarios. Consequently, there exists a lot of redundancy in the derived tests. Therefore, property compaction can be employed before the automated test generation to reduce the required number of properties. To further reduce the number of directed tests, we can consider various test compaction techniques from the manufacturing testing domain. When optimal test compaction is not feasible, greedy approaches can be explored by identifying the most effective tests.

This book considered the scenarios where the design is same and the properties are different. The approaches presented in this book can also be extended to efficiently generate tests for different designs but using the same set of properties. A special case can be when a design is slightly modified according to new requirements. Thus, we need to regenerate the new tests for the properties of the previous design. It may be possible to reuse some of the previous tests. Moreover, since most of the functionality remains the same, prior learning can be reused to generate the new tests.

Currently, most assertion-based validation methods use simulation of both specification and implementation. Generally for a large design, there can be thousands of assertions that need to be checked at the same time. Checking them independently strongly affects the simulation performance. In the worst case, activating one assertion needs one directed test. Therefore, it is necessary to design a methodology that can investigate the dependence between assertions, and generate a small set of directed tests that can achieve the same assertion coverage.

This book demonstrated that the conflict clause forwarding and decision ordering based learning techniques are promising for system level test generation. It can be extended to other abstraction levels including RTL and gate-level models. For example, by incorporating our learning techniques, we believe that the performance of current SAT-based automatic test pattern generation (ATPG) approaches can be drastically improved.

Chapter 11 has shown that the state space of many cache coherence protocols in modern multicore architectures have quite regular structure. We believe that the proposed techniques can be further extended to effectively analyze the protocol implementation with large number of cores. Although the full transition coverage may become infeasible for too many cores, the knowledge about the space structure can be used to effectively distribute the test vectors within the state space, so that complex bugs can be detected.

A typical SoC design methodology consists of three important validation/testing phases: pre-silicon validation, manufacturing testing and post-silicon validation. This book is focused on pre-silicon validation. Manufacturing testing is primarily used to detect physical (structural) defects in each of the manufactured ICs. On the other hand, the focus of post-silicon validation is to detect design flaws that have escaped from pre-silicon validation. The directed test generation and automated reuse techniques can be extended to make them applicable in both post-silicon validation and manufacturing testing domains. Significant reuse between these domains will lead to drastic reduction in overall validation and testing effort.

Appendix A
Acknowledgments of Copyrighted Materials

Chapter 2 used some materials that are copyrighted by ACM. The definitive versions appeared in the ACM Transactions on Embedded Computing Systems [1] and in the proceedings of ACM Great Lakes Symposium on VLSI [2]. The work is produced under permission granted by ACM as stated by the policy: *under the ACM copyright transfer agreement,the original copyright holder retains the right to reuse any portion of the work,without fee,in future works of the author's own,including books,lectures and presentations in all media,provided that the ACM citation and notice of the Copyright are included*. Chapter 2 also used some materials that are copyrighted by Springer. The definitive version appeared in the Springer Design Automation for Embedded Systems [3]. The work is produced under automatic permission granted by Springer since the book will be published by Springer. Chapter 2 also used some materials that are copyrighted by IEEE. The definitive version appeared in the proceedings of IEEE International High Level Design Validation and Test Workshop [4]. The work is produced under permission granted by IEEE as we are the authors of the original work and by the policy: *as a member of the IEEE,you are permitted to reuse this content for free except for a* $3.50 *handling charge in order to obtain a copyright compliance license through RightsLink*.

Chapter 3 used some materials that are copyrighted by Springer. The definitive version appeared in the Springer Design Automation for Embedded Systems [3]. The work is produced under automatic permission granted by Springer.

Chapter 4 used some materials that are copyrighted by ACM. The definitive version appeared in the proceedings of International Symposium on Hardware/ Software Codesign and System Synthesis [5]. The work is produced under permission granted by ACM as stated above.

Chapter 5 used some materials that are copyrighted by IEEE. The definitive versions appeared in the proceedings of International Conference on VLSI Design [6] and in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems [7]. The work is produced under permission granted by IEEE as stated above.

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2,

© Springer Science+Business Media New York 2013

Chapter 6 used some materials that are copyrighted by EDAA. The definitive version appeared in the proceedings of Design Automation and Test in Europe [8]. The work is produced under permission granted by EDAA as stated by the policy: *Authors and employers retain all proprietary rights in any process, procedure or article of manufacture described in the work,and similarly retain unconstrained rights to publish, print or copy the above work in any form*. Chapter 6 also used some materials that are copyrighted by IEEE. The definitive versions appeared in the IEEE Transactions on Computers [9]. The work is produced under permission granted by IEEE as stated above.

Chapter 7 used some materials that are copyrighted by IEEE. The definitive version appeared in the proceedings of International Conference on VLSI Design [10]. The work is produced under permission granted by IEEE as stated above.

Chapter 8 used some materials that are copyrighted by ACM. The definitive version appeared in the ACM Transactions on Embedded Computing Systems [11]. The work is produced under permission granted by ACM as stated above. Chapter 8 also used some materials that are copyrighted by EDAA. The definitive version appeared in the proceedings of Design Automation and Test in Europe [12]. The work is produced under permission granted by EDAA as stated above.

Chapter 9 used some materials that are copyrighted by EDAA. The definitive version appeared in the proceedings of Design Automation and Test in Europe [13]. The work is produced under permission granted by EDAA as stated above.

Chapter 10 used some materials that are copyrighted by IEEE. The definitive version appeared in the proceedings of International Symposium on Quality Electronic Design [14]. The work is produced under permission granted by IEEE as stated above. Chapter 10 also used some materials that are copyrighted by ACM. The definitive version appeared in the ACM Transactions on Design Automation of Electronic Systems [15]. The work is produced under permission granted by ACM as stated above.

Chapter 11 used some materials that are copyrighted by EDAA. The definitive version appeared in the proceedings of Design Automation and Test in Europe [16]. The work is produced under permission granted by EDAA as stated above.

Chapter 12 used some materials that are copyrighted by ACM. The definitive version appeared in the ACM Transactions on Embedded Computing Systems [1]. The work is produced under permission granted by ACM as stated above. Chapter 12 also used some materials that are copyrighted by IEEE. The definitive version appeared in the proceedings of IEEE International High Level Design Validation and Test Workshop [4]. The work is produced under permission granted by IEEE as stated above.

References

1. Chen M, Mishra P, Kalita D (2012) Automatic RTL test generation from SystemC TLM specifications. ACM Trans Embed Comput Syst (Accepted to appear)
2. Chen M, Mishra P, Kalita D (2008) Coverage-driven automatic test generation for UML activity diagrams. In: Proceedings of ACM Great Lakes symposium on VLSI (GLSVLSI), pp 139–142
3. Chen M, Mishra P, Kalita D (2010) Efficient test case generation for validation of UML activity diagrams. Des Autom Embed Syst 14(2):105–130
4. Chen M, Mishra P, Kalita D (2007) Towards RTL test generation from SystemC TLM specifications. In: Proceedings of IEEE international high level design validation and test workshop (HLDVT), pp 91–96
5. Koo H, Mishra P (2008) Specification-based compaction of directed tests for functional validation of pipelined processors. In: Proceedings of international symposium on hardware/software codesign and system synthesis (CODES + ISSS), pp 137–142
6. Mishra P, Chen M (2009) Efficient techniques for directed test generation using incremental satisfiability. In: Proceedings of international conference on VLSI design, pp 65–70
7. Chen M, Mishra P (2010) Functional test generation using efficient property clustering and learning techniques. IEEE Trans Comput Aided Des Integr Circuits Syst 29(3):396–404
8. Chen M, Qin X, Mishra P (2010) Efficient decision ordering techniques for SAT-based test generation. In: Proceedings of design automation and test in Europe (DATE), pp 490–495
9. Chen M, Mishra P (2011) Property learning techniques for efficient generation of directed tests. IEEE Trans Comput 60(6):852–864
10. Qin X, Chen M, Mishra P (2010) Synchronized generation of directed tests using satisfiability solving. In: Proceedings of international conference on VLSI design, pp 351–356
11. Koo H, Mishra P (2009) Functional test generation using design and property decomposition techniques. ACM Trans Embed Comput Syst 8(4):32:1–32:33
12. Koo H, Mishra P (2006) Functional test generation using property decompositions for validation of pipelined processors. In: Proceedings of design automation and test in Europe (DATE), pp 1240–1245
13. Chen M, Mishra P (2011) Decision ordering based property decomposition for functional test generation. In: Proceedings of design automation and test in Europe (DATE), pp 167–172
14. Qin X, Mishra P (2011) Efficient directed test generation for validation of multicore architectures. In: Proceedings of international symposium on quality electronic design (ISQED), pp 276–283
15. Qin X, Mishra P (2012) Directed test generation for validation of multicore architectures. ACM Trans Des Autom Electron Syst (Accepted to appear)
16. Qin X, Mishra P (2012) Automated generation of directed tests for transition coverage in cache coherence protocols. In: Proceedings of design automation and test in Europe (DATE), pp 3–8

Index

2-interaction, 35

A
abstraction gap, 219
abstraction levels, 216
ABV, 9
Action nodes, 30
activity edges, 31
activity fault model, 50
ADL, 4
AHB, 1
Alpha AXP processor, 230
Always *p*, 47
antecedent clause, 82
APB, 1
approximately timed models, 5
architecture description language, 20
architecture description languages, 4
architecture manual, 20
ARTEST, 222
assertion based validation, 9
associative law, 173
ATE, 62
ATM, 31
atomic subproperties, 173
ATPG, 62
automated teller machine, 31
automated test generation, 22

B
backtrack, 109
backtrack, 81
base property, 88
Base time, 101
Bayesian networks, 44
BCP, 81, 109
BFS, 201
bias, 176
binary decision diagrams, 44
BIST, 62
bit correspondence, 220
bit failures, 218
bit priority, 110
bit-value ordering, 110, 111
block diagram, 20
BMC, 52
Boolean Constraint Propagation, 81
Boolean constraints, 108
boolean satisfiability, 44
bound effect, 174
bounded model checking, 44
branch events, 176
branches, 175

C
C++ parser, 222
cache coherence protocols, 201
cache coherence, 1
Cadence SMV checker, 223
cause-effect relations, 87
cause–effect, 174
channel, 25
clause forwarding, 131, 134
clustering rules, 173
Clustering time, 101
CNF clauses, 83
CNF, 108

M. Chen et al., *System-Level Validation*, DOI: 10.1007/978-1-4614-1359-2,
© Springer Science+Business Media New York 2013

C (*cont.*)
COI, 169
commutative law, 173
completeness, 10
completion transition, 34
Component Wrapper Language, 216
composition, 169
condition failures, 218
conditional events, 175
cone of influence, 169, 223
conflict analysis, 82
conflict clause forwarding, 89, 104, 112, 171
conflict clause, 81, 83, 134
conflict side, 83
conflict, 82
Conflict-driven backtracking, 82
Conflict-driven learning, 82
conjunctive form, 172
conjunctive normal form, 108
consequent events, 174
constrained-random test generation, 8
constrained-random testing, 43
constraint satisfaction problem, 44
Control nodes, 30
co-simulation, 216
counterexample, 9, 43, 45, 52, 53
coverage metrics, 48
coverage metrics, 7
coverage-oriented verification, 44
CSP, 44
cut, 83
CWL, 216

D
DAG, 175
data store, 31
data tokens, 22
data transfer edges, 20
data-transfer path, 20
De Morgan's laws, 47
Decision nodes, 31
decision ordering heuristics, 171
Decision ordering, 107
decision tree, 113
decomposed properties, 170
decomposition framework, 169
deep bounds, 169
delay, 175
design automation, 19
design decomposition, 145, 149
design graph, 84
design space exploration, 4
design under validation, 8, 43
diameter, 53
Dijkstra's algorithm, 176
DIMACS file, 178
DIMACS format, 178
DIMACS, 91
directed acyclic graph, 82, 175
directed edge, 175
directed testing, 43
disjunctive form, 172
DPLL implementation, 81
DPLL, 81
DSE, 4
DUV, 43
dynamic compaction, 64
dynamic frequency/voltage scaling, 1

E
EDA, 4
electronic design automation, 4
Elsa, 222
equivalence checking, 8
Euler tour, 202, 205
event implications, 180
events, 174
Eventually *p*, 47
expert knowledge, 169

F
false property, 172
fault coverage, 218
fault model, 48, 218
fault simulation, 64
final node, 34
finite state machine, 20
flow edge, 34
fork, 30
formal model, 19, 43, 45
Formal verification, 2, 8
forwarding, 87
FSM coverage, 147
FSM state dominance, 66
FSM, 20, 201
functional components, 19
functional correctness, 2
functional coverage, 43, 51
functional errors, 215
functional scenarios, 11, 173

functional unit, 179
functional validation, 2, 61

G
Gate/logic level, 5
generic fault models, 50
global counterexample, 145
global variable, 85, 155
golden models, 218
golden reference model, 5
graph model, 20
GRASP, 81
group id, 93

H
happen before, 174
hardware description language, 5
HDL, 5
Heterogeneous Multicore Architectures, 194
heuristic methods, 107
high-level abstraction, 20
hypercube, 205

I
implementation bugs, 2
implication graph, 82, 84, 93
implication relation, 174
implication vertex, 82
inconsistency, 215
incremental SAT, 80, 171
inevitable state, 73
inevitable transitions, 73
influence sets, 87
influence-based clustering, 97, 102
initial event, 176
initial node, 34
initial process, 23
initial state, 175
instruction memory, 230
interaction fault model, 51
interaction fault, 154
interaction, 35
interconnect pipelining, 1
interface protocol, 220
inter-property learning, 107
intersection calculation time, 96
Intersection-based clustering, 99
intra-property learning, 107, 119
IP cores, 12
ISCAS circuits, 80

J
Join, 30

K
key path fault model, 51
key path, 35
Kripke structure, 43, 52

L
learned knowledge, 83
learning-oriented decomposition, 169
line coverage, 228
linear temporal logic, 171
literal score, 108, 177, 118
literal, 108
local learning, 83
local variable, 85, 155
logic synthesis, 5, 170
logic/functional errors, 2
long distance backtracking, 109
loosely timed models, 5
loosely-timed, 224
LTL, 46

M
malfunctions, 215
manufacturing test, 61
mapping rules, 224
memory blocks, 1
Merge nodes, 31
merge, 30
MESI, 204, 208
microprocessors, 1
MIPS architecture, 93, 181
MIPS processor, 20
Model Algebra, 22
model checker, 9, 51
model checking, 8, 43, 145
model interoperability, 19
MOESI, 204
MOSI, 204, 209
MSI, 203, 206
multicore, 185, 188

N
Name substitution, 91
Next *p*, 47
non-chronological backtracking, 82
non-deterministic, 23
NuSMV, 181, 126

O

Object nodes, 30
online stock exchange system, 182
original property, 172
Original time, 101
OSCI, 218
OSES, 40, 182
overall performance, 171
overall system functional scenarios, 11
overhead, 96
overlap, 107, 171

P

partial counterexamples, 145
partial tests, 170
path coverage ratio, 232
path, 35
pattern, 221
Pentium FDIV bug, 2
Periodic function, 81
periodic score decaying, 177
periodically decaying time, 118
peripherals, 1
Petri-net, 29
pipeline edges, 20
pipeline interactions, 8, 151
place, 23
postcondition, 172
post-silicon validation, 2
precondition, 172
preprocessing, 92, 223
pre-silicon validation, 2
procedure call, 27
processes, 25
processors, 12
property clustering, 84
property compaction, 48, 67
property decomposition, 145, 149
property falsification, 169
property graph, 84
property negation, 47
property sequence, 176
property specification language, 9
propositional formulas, 108
prototype tool, 216
prune, 113
pseudorandom tests, 8
PSL, 9

R

random RTL tests, 232
reachable, 35
reason side, 83
reconfigurable design, 216
Reduction, 36
refined subproperties, 173
refinement rules, 216
register file, 20
register transfer level, 4
repeated validation efforts, 171
resolution, 113
reuse, 12, 171
RISC processors, 1
RTL data, 220
RTL simulation, 4
RTL test stimulus, 220
RTL, 4

S

safety property, 46
SAT instance, 107
SAT solvers, 148
SAT, 8, 44
SAT-based BMC, 52, 145
satisfiable, 84
satisfying assignment, 52, 107
Scaling, 36
search path, 110
segment, 114
semi-formal specification, 32
semiformal validation, 9
set covering, 65
SI, 205
silicon respin, 5
similarity threshold, 85
similarity, 85
simulation-based validation, 7
sink nodes, 30
SMV, 51
SoC, 1
spatial symmetry, 185, 187
spatially decomposable, 172
state and data transitions, 38
state space explosion problem, 147
state space explosion, 9, 44, 145, 169
state space, 201
static compaction, 64
static information extraction, 37
statistics, 116
stepwise refinement, 5
Structural overlap, 85
structural similarity, 94, 96
sub-functions, 170
subproperties, 155, 169
successful decision ratio, 109

SVA, 9
Symbolic model verifier, 51
Symmetric Component, 188
synchronize, 31
System/architectural level, 5
SystemC, 19
system-level modeling, 224
system-level specification, 19
System-on-Chip, 1
SystemVerilog assertion, 9

T
target state, 175
temporal logic, 46
temporal property, 43
temporal symmetry, 187
temporally decomposable, 174
test compaction, 63
test compression, 63
test decompression, 63
test dominance, 66
test essentiality, 66
test matrix, 65
Test Refinement Specification, 220
test refinement specification, 233
test vectors, 7
Textual clustering, 86
Textual overlap, 86
theorem proving, 8
time frames, 170
time-to-market, 43
Timing chart, 226
timing sequence, 221
TLM components, 23
TLM data content coverage, 218
TLM data, 220
TLM modules, 25
TLM, 5
TLM2RTL, 222
TLM2SMV, 222
toggle coverage, 218, 228
Transaction data fault model, 50, 218
transaction data, 50, 218
Transaction flow fault model, 50, 218
transaction flow, 50, 126, 218
transaction level modeling, 5, 19
transaction, 175
transactor-based verification, 216
Transistor/physical level, 5
Transition Coverage, 204
transition fault model, 51
translation, 33

U
UBMC, 52
UIP, 93
UML activity diagram, 29, 182
UML Profile, 19
UML, 5, 19
Unbounded Model Checking, 52
unified modeling language, 5
Unique Implication Point, 93
unrolling depth, 108
unsatisfiable core, 108, 171
untestable path, 80
Until *p*, 47

V
validation effort, 11
validation target, 8
variable ordering, 111
variable state independent decaying sum, 108, 176
verification, 2
VSIDS, 108, 176

X
XMI, 40
XML Metadata Interchange, 40

Z
zChaff, 81, 136