Design Verification for Sequential Systems at Various Abstraction Levels

Liang Zhang

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Dr. Michael S. Hsiao, Chair
Dr. Joseph G. Tront
Dr. Thomas L. Martin
Dr. G. Q. Lu
Dr. Mark Shimozono

January 27, 2005
Blacksburg, Virginia

Keywords: Formal Verification, Simulation, ATPG, SAT, Bounded Model Checking
©Copyright 2005, Liang Zhang

Design Verification for Sequential Systems at Various Abstraction Levels

Liang Zhang

(abstract)

With the ever increasing complexity of digital systems, functional verification has become a daunting task to circuit designers. Functional verification alone often surpasses 70% of the total development cost and the situation has been projected to continue to worsen. The most critical limitations of existing techniques are the capacity issue and the run-time issue.

This dissertation addresses the functional verification problem using a unified approach, which utilizes different core algorithms at various abstraction levels.

At the logic level, we focus on incorporating a set of novel ideas to existing formal verification approaches. First, we present a number of powerful optimizations to improve the performance and capacity of a typical SAT-based bounded model checking framework. Secondly, we present a novel method for performing dynamic abstraction within a framework for abstraction-refinement based model checking. Experiments on a wide range of industrial designs have shown that the proposed optimizations consistently provide between 1-2 orders of magnitude speedup and can be extremely useful in enhancing the efficacy of existing formal verification algorithms.

At the register transfer level, where the formal verification is less likely to succeed, we developed an efficient ATPG-based validation framework, which leverages the high level circuit information and an improved observability-enhanced coverage to generate high quality validation sequences. Experiments show that our approach is able to generate high quality validation vectors, which achieve both high tag coverage and high bug coverage with extremely low computational cost.

To my parents and Xiao Yan

Acknowledgments

First and foremost, I thank my advisor, Dr. Michael Hsiao, for his constant guidance and encouragement on my research. It is him who encouraged me to move on when I hit the dead-end on my research. Without his help, I would not dare to imagine of finishing my Ph.D.

I would like to thank Dr. Joseph G. Tront, Dr. Thomas L. Martin, Dr. G. Q. Lu and Dr. Mark Shimozono for taking their precious time to serve on my committee.

I am extremely grateful to Dr. Mukul R. Prasad and Dr. Indradeep Ghosh of Fujitsu Laboratories of America, who mentored me during my internship there. Significant portion of this dissertation is coming from the result of research projects done at my internship.

I thank Proactive alumni and current members for their valuable comments and suggestion on my research. To name a few, I am especially thankful to Dr. Shuo Sheng, Dr. Xiao Liu, Qingwei Wu Manan Syal, Rajat Arora, Puneet Gupta, Bin Li, Kameshwar Chandrasekar, Ronnie Lajaunie, and Vishnu Vimjam.

My discussions with Dr. Christian Stangier of Fujitsu Labs were always enjoyable and I learned a lot from him, especially on BDD related issues, which I am really

thankful. Debashish Sahoo of Stanford University, who was also an intern at Fujitsu Labs, gave me many constructive comments on my research.

Last but not least, I would like to express my deepest gratitude to my dear family for their constant support and love.

This work received support from Fujitsu Labotories of America Inc. and National Science Foundation.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Preliminaries 3

1.2.1 Boolean Decision Diagram 3

1.2.2 Boolean Satisfiability 6

1.3 Contributions 8

1.4 Outline of the Dissertation 11

2 Bounded Model Checking for Design Verification at Logic Level **12**

2.1 Introduction 12

2.2 Our Contributions 13

2.3 Background & Notation 15

2.4 Related Work 17

2.5 Incremental Deductive & Inductive Reasoning for SAT-based BMC . 19

2.5.1 Incremental Reasoning 19

2.5.2 Induction-based Learning 27

2.5.3 Variable Ordering 33

2.6 Logic Optimization for BMC 37

2.7 Experimental Results 38

2.8 Summary 43

3 Abstraction and Refinement for Design Verification at Logic Level 46

3.1 Introduction 47

3.2 Related Work 50

3.3 Background 52

3.3.1 Computational Tree Logic & Model Checking 52

3.3.2 BDD-based Symbolic Reachability Analysis 53

3.4 Dynamic Abstraction 58

3.4.1 Latch Selection Heuristic 63

3.4.2 Optimizations 66

3.4.3 Handling Counter-examples on Abstract Model 67

3.5 Experimental Results 68

3.6 Summary 72

4 Design Validation at Register Transfer Level via ATPG Technologies 74

4.1 Introduction 75

4.2 Preliminaries 79

4.2.1 Assignment Decision Diagrams 79

4.2.2 RTL Algebra Based Testing Approach 80

4.2.3 Validation Metrics 82

4.2.4 Modification of OBCC transformation Tables 83

4.3 Design Validation Framework 84

4.4 ATPG-based Validation Vector Generation Algorithm 84

4.4.1 Test Environment Generation 87

4.4.2 Test Environment Translation 90

4.4.3 Test Environment Relaxation 91

4.4.4 Augmenting the Algebra with Value Range 92

4.4.5 Techniques to Maximize the Error Detection 94

4.4.6 Static Learning to Aid Value Range Analysis 95

4.4.7 Identification of reset signals 98

4.4.8 STG Extraction 99

4.4.9 Counter and Iteration Loop Identification 100
4.4.10 Simulation-based Procedure for Special Case 103
4.5 Experimental Results 105
4.6 Summary 109
5 Conclusion **112**

List of Figures

1.1 BDD with ordering $b < d < a < c$ 4

1.2 BDD with ordering $a < c < b < d$ 5

1.3 The formulas for the basic gates 7

1.4 Translation of a Simple Circuit to CNF formula 7

2.1 ILA of Current BMC Iteration 15

2.2 Pre-Saturation Region (PSR) of BCOI 17

2.3 Example: Incorrect Use of Incremental BMC 21

2.4 Incremental Reasoning for BMC 23

2.5 Induction-based Learning 28

2.6 Profiling of SAT Solver for Two Different Variable Ordering Schemes 35

3.1 Conventional Latch Based Abstraction 55

3.2 Latch-based Unsatisfiability Analysis 59

3.3 Redundancy Index as a function of BMC-depth 63

4.1 Assignment Decision Diagram 80

4.2 Overview of the design validation flow 85

4.3 Validation Vector Generation Flow 86

4.4 Sample VHDL Code and Corresponding Structural RTL 88

4.5 Generated test environment 89

4.6 Augmenting the algebra with value range 92

4.7 Inaccurate value range selects wrong signal value. 96

4.8 Direct implication of RTL algebra 98

4.9 ADD for state variable & STG 100

4.10 State Traversal Sequence for Embedded Counter 101

4.11 Bus split and re-joint 103

4.12 Tag Cov. for Our and Random Approach with the same number of vectors 108

4.13 Bug Cov. for Our and Random Approach with the same number of vectors 109

4.14 Experimental Result on GPIO 110

List of Tables

2.1 Improvements to Incremental BMC 26

2.2 Effect of Induction-based Learning on BMC 32

2.3 Effects of Variable Ordering on BMC 36

2.4 Effects of Logic Optimization on BMC 38

2.5 Benchmark Characteristics 39

2.6 Experimental Results on UNSAT Cases 44

2.7 Experimental Results on SAT Cases 45

3.1 Problem Complexity on Concrete Model 69

3.2 Results: Static Abstraction and Proposed Dynamic Abstraction . . . 70

3.3 BDD Sizes of Different Abstraction Methods 72

4.1 Δ calculus for an adder 82

4.2 Modified Error Propagation Calculus for an Adder 83

4.3 Test Environment translation 90

4.4 Experiment results: Our Approach vs. Random Approach 107

4.5 Coverage Results of Our Approach on Different Metrics 111

Chapter 1

Introduction

Functional verification of sequential systems is rapidly becoming one of the most crucial and resource-intensive components of the product design cycle. It often surpasses 70% of the total development cost. With the ever increasing sizes and complexity of designs, it has been projected that the reality of functional verification continues to worsen. Simulation-based verification has traditionally been used as the primary approach for design verification from system level to component level. However, with complex design, especially with system on chip (SOC), traditional simulation-based verification has become ineffective in finding subtle design bugs. On the other hand, formal verification has gained increased attention in recent years, since it can mathematically prove or disprove the correctness of a design. The advances in boolean decision diagram (BDD) and boolean satisfiability (SAT) have greatly increased the applicability of formal verification to a larger scale. However, fundamentally, these problems belong to the NP-complete problem. It is widely believed that the worst case complexity of NP-complete problem is exponential. It is clear that no single

method will rule over all cases. We believe that the most amiable approach is a unified scheme, which relies on formal verification at logic level in exhaustively validating the correctness of each component, and resorts to simulation-based approach at higher abstraction level and system level in assuring a satisfactory validation coverage. This dissertation presents new ideas of extending the capacity and applicability of formal verification at the logic level as well as that of simulation-based validation at higher abstraction level.

1.1 Motivation

Design errors or bugs, especially the corner cases in a complex system, are expensive to discover and fix. However, it would be even more expensive to let bugs slip through the design phase and reach the final products. One well known story is the floating point bug in Intel's Pentium processor, which caused not only substantial financial damage but also reputational damage to the company. On the other hand, the reality of functional verification continues to worsen. The capacities of verification algorithms to handle large designs have been increased dramatically in recently years. However, these increases are often surpassed by the complexity increases of the design. With recent advances of formal verification techniques, especially BDD and SAT, formal verification has been accepted and applied in many real projects. It has been shown that formal verification can be very cost-effective in finding hard bugs. However, for very complex systems, formal verification often suffers from either state explosion problem or requires exponentially long time to finish. Thus, it is of significant importance to have a unified approach, which combines the merits of

different verification algorithms.

1.2 Preliminaries

In this section, we review two important techniques, which will be used throughout the dissertation.

1.2.1 Boolean Decision Diagram

Boolean decision diagrams (BDDs) have recently emerged as the representation of choice for many electronic design automation (EDA) applications, such as synthesis, verification, and testing. Although, the original concept of BDDs can be traced to the late 50's [Lee58, Ake78], it was the pioneering work of Bryant [Bry86] in late 80's that brought dramatic impact of BDDs on many EDA algorithms. The key attributes of BDDs are as follows:

- Given a practical variable ordering, the BDDs are canonical. In other words, suppose that we have correctly built the BDDs for two circuits, the two circuits are equivalent if and only if the BDDs are identical.
- BDDs can be very efficient at representing combinatorially large sets.
- Highly efficient graph manipulation algorithms exist that allow boolean operations to be performed directly on BDDs.

Many heuristics and optimizations for manipulating BDDs have been proposed that dramatically improved the performance and robustness of BDD algorithms. Figure 1.1

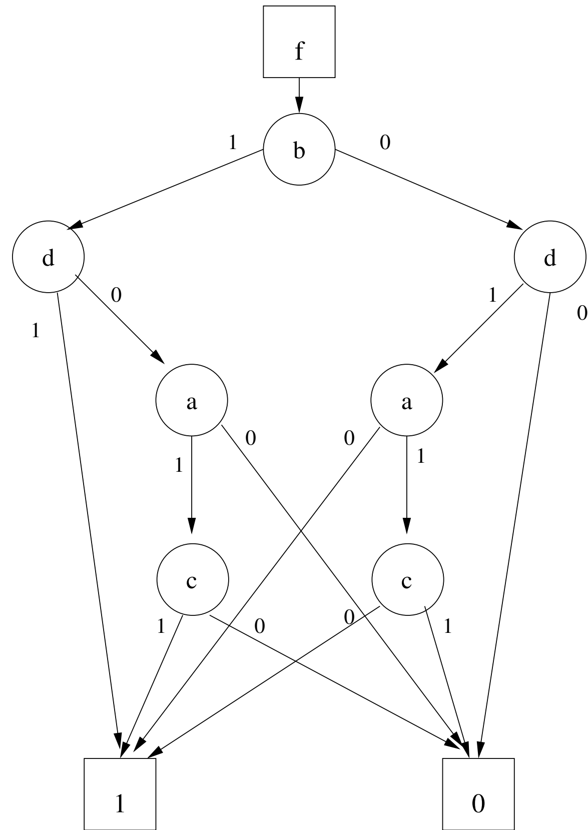


Figure 1.1: BDD with ordering $b < d < a < c$

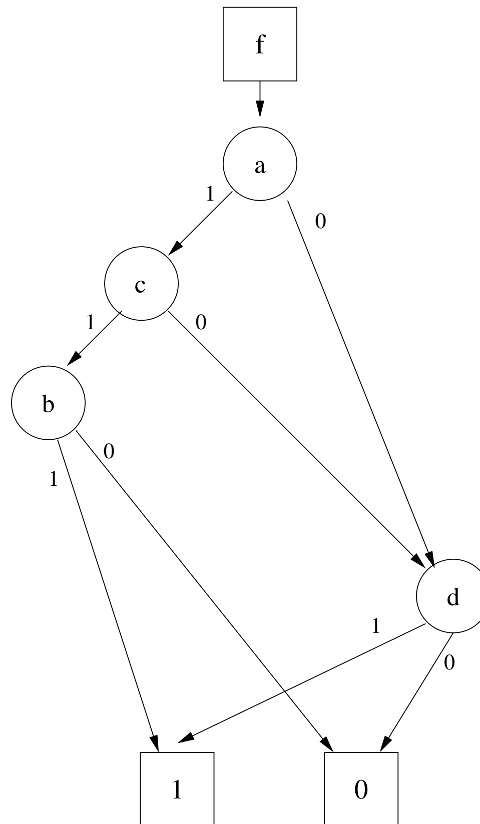


Figure 1.2: BDD with ordering $a < c < b < d$

and 1.2 shows the reduced ordered BDDs (ROBDD) of the function $f = abc + a'd + c'd$ with two different variable orderings.

We can see that the Figure 1.2 is a minimal BDDs, since there is exactly one node for each variable. The corresponding variable ordering is the optimal ordering.

Generally speaking, the underlying variable ordering can have a surprisingly dramatic impact on the size of the BDDs (in terms of nodes in the graph). Unfortunately, finding an optimal variable ordering is a very hard problem. Furthermore, it has

been proved that for some circuits, for example multiplier, any variable ordering will cause the size of BDDs to grow exponentially with the number of state bits. This is often called *state explosion* problem.

1.2.2 Boolean Satisfiability

The boolean satisfiability problem (SAT) is a decision problem which answers if the value of a given boolean formula can ever be true with some assignment of TRUE of FALSE to the variables inside the formula. If the formula is always FALSE, then the formula is said to be unsatisfiable. Otherwise, the formula is satisfiable. The SAT problem (except 2-SAT problem) is a well known NP-complete problem.

The most widely used format for the SAT formula is conjunctive normal form (CNF), which is a boolean expression of the conjunction of one or more disjunctive clauses. A disjunctive clause is a collection of one or more (positive or negative) literals connected by an OR symbol($\vee$). Apparently, for a CNF formula to be satisfiable, all of the disjunctive clauses must be true simultaneously, *i.e.*, no variable is forced to take conflicting values.

Here is an example of CNF formula:

$$f = (a) \wedge (\neg a \vee b \vee c \vee \neg d) \wedge (c \vee d) \wedge (d)$$

Formula f is satisfiable. And one of the truth assignment is:

$$a = true, b = false, c = true, d = true$$

In order to utilize generic SAT solvers for circuit verification purpose, circuits, which are generally represented in logic gates, must be transformed to the CNF formula.

Figure 1.3 shows the translation for a few typical gates.

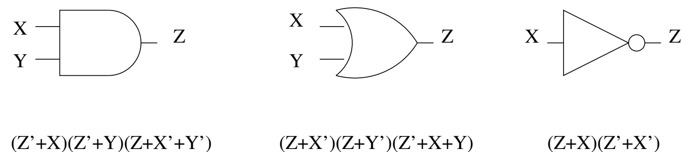


Figure 1.3: The formulas for the basic gates

Figure 1.4 shows the CNF formula for a simple circuit. We can see that the number of the clauses in the translated CNF formula is linear to the number of the gates in the circuit, and the translation can be done very efficiently.

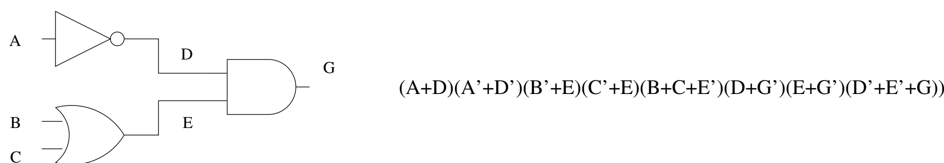


Figure 1.4: Translation of a Simple Circuit to CNF formula

Most of the state-of-the-art complete SAT solvers [MSS99, Zha97, zch03, GN02] are based on the branch and backtracking algorithm, also called the Davis Putnam Logemann Loveland (DPLL) search algorithm[DLL62].

The most significant improvement of the state-of-the-art SAT solvers comes from the learning techniques and non-chronological backtracking schemes, which facilitates the pruning of the search space. Additionally, next decision variable selection scheme as well as a highly efficient implementation of boolean constraint propagation (BCP) also contribute to the dramatic improvement of current SAT solver.

DPLL_With_Learning

```
1: while (TRUE) do
2:   if DECIDE_NEXT_BRANCH_VARIABLE() then
3:     while (DEDUCE() == CONFLICT) do
4:       level = CONFLICT_ANALYSIS();
5:       if (level == 0) then
6:         return UNSATISFIABLE;
7:       else
8:         BACK_TRACK(level);
9:       end if
10:    end while
11:  else
12:    return SATISFIABLE;
13:  end if
14: end while
```

Algorithm 1.2.1: Basic DPLL Algorithm with Learning Technique

Algorithm 1.2.1 shows the pseudo code for typical DPLL algorithm with learning techniques.

1.3 Contributions

This dissertation presents results on two different levels of abstraction. At the lower end, designs in logic level descriptions are verified exhaustively using improved formal verification techniques. We present novel ideas that significantly improve the capacity

and the run-time speed of SAT-based bounded property checking and BDD-based model checking. Specifically, the contributions of logic level formal verification are as follows:

- We offer novel concepts in maximizing the *relevant* shared information between successive iterations coupled with efficacious variable ordering for incremental reasoning in a typical iterative SAT-BMC set-up.
- We symbiotically integrate SAT-based inductive reasoning and SAT-based BMC, in that clauses learned during the induction phase can be used to accelerate the SAT search in subsequent BMC iterations.
- We enhance the default variable order of the SAT solver with structural information from BMC, without tampering with the internal details of the SAT solver.
- We introduce and develop the notion of dynamic abstraction, which is orthogonal to static abstractions that have traditionally been used in abstraction refinement frameworks, yet fully compatible with most such frameworks.
- We propose two possible heuristic techniques for implementing dynamic abstraction in the context of a SAT-BMC based abstraction framework that uses a BDD-based model checker. While we found this specific framework to be a good fit for our initial implementation of the idea, it is important to note that the notion of dynamic abstraction goes far beyond the specific implementations of it reported in this dissertation.
- We present experimental results on several large industrial benchmarks to demonstrate that dynamic abstraction can provide up to an order of magnitude of im-

provement over and above the performance improvement afforded by traditional static abstraction refinement techniques.

Experiments on a wide range of industrial designs have shown that the proposed optimizations consistently provide between 1-2 orders of magnitude speedup and can be extremely useful in enhancing the efficacy of typical SAT-BMC tools.

At the higher end, we rely on the simulation-based approach in validating designs in register transfer level descriptions. Our contributions are as follows:

- We propose a set of heuristics to utilize the high level circuit information for generating validation vectors.
- We improved upon an existing framework with value analysis capability to better handle control-intensive circuits.
- We improved the observability-enhanced statement coverage to better gauge the validation coverage.
- Our framework intelligently combines a deterministic and genetic algorithm based engines.

Experiments show that our approach is able to generate high quality validation vectors, which achieve both high tag coverage[FDK01] and high bug coverage with extremely low computational cost. Orders of magnitude improvement in coverage over conventional approach has been achieved.

1.4 Outline of the Dissertation

Part I is about the formal verification at the logic level. Chapter 2 introduces our ideas of extending the conventional SAT-based bounded model checking. Chapter 3 presents our new approach of doing abstraction-refinement for BDD-based formal verification.

Part II is about the design validation at the register transfer level. Chapter 4 covers two main focuses of simulation-based validation approach. First, it presents our highly efficient way of generating validation vectors directly from register transfer level descriptions. Secondly, it examines how to improve the accuracy of measured validation coverage.

In Chapter 5, we conclude the dissertation and discuss some ideas for future research in each area. Related work, when it exists, is also discussed in each chapter.

Chapter 2

Bounded Model Checking for Design Verification at Logic Level

2.1 Introduction

Functional verification of digital hardware designs is rapidly becoming one of the most crucial and resource-intensive components of the product design cycle. With the increasing size and complexity of designs, simulation-based validation is no longer sufficient to provide the requisite design coverage needed to expose subtle bugs. Formal verification techniques such as symbolic model checking based on binary decision diagrams (BDDs) [CGP99, McM93] offer the potential of exhaustive coverage and have met with some success. However, in practice, their application has been limited by the state explosion problem.

Bounded Model Checking (BMC) based on SAT methods [CBRZ01] is rapidly gaining

popularity as a complementary technique to BDD-based verification methods. Given a temporal logic property p to be verified on a finite transition system $\mathcal{M}$, the essential idea is to search for counter-examples to p in the space of all executions of $\mathcal{M}$ whose length is bounded by some integer k. This problem is translated into a Boolean formula which is satisfiable if and only if a counter-example exists for the given value of k. This check is performed by a conventional SAT solver, and it is typically iterated with increasing values of k until a counter-example to the property is found or some resource constraints are exceeded. Dramatic improvements in SAT solvers [GN02, MSS99, MMZ$^+$01, Zha97] have shown that SAT-based BMC can often reason about systems well beyond the capacity limit of BDD-based methods [AKMM03, CFF$^+$01].

Recently, a number of papers have proposed improvements over the original BMC method [CNQ03, Fal02, GA02, GGW$^+$03b, Sht04]. This chapter presents a number of optimizations to further strengthen a SAT-BMC implementation. Typically, the SAT solver performance dominates the overall runtime resources of a SAT-BMC solver. Therefore, all of the proposed optimizations focus on improving the performance of the SAT solver, in the context of the BMC problem, by directly or indirectly influencing two features that have been proved to be the key to the performance of modern SAT solvers, namely: 1.) Use of learned clauses, and 2.) variable ordering.

2.2 Our Contributions

We propose a number of conceptually simple but extremely effective optimizations to improve the performance of SAT-based BMC. An important unifying feature of our contributions is that the performance improvements are achieved by changing the

formulation and methodology of BMC, leaving the internal working of the SAT solver largely untouched. This is significant since it makes the proposed optimizations at least partly independent of the particular SAT solver and provides the flexibility to potentially harness the benefits offered by newer and better SAT solvers. Specifically, the contributions of this work are as follows:

- We offer novel concepts in maximizing the *relevant* shared information between successive iterations coupled with efficacious variable ordering for incremental reasoning in a typical iterative SAT-BMC set-up.
- We symbiotically integrate SAT-based inductive reasoning and SAT-based BMC, in that clauses learned during the induction phase can be used to accelerate the SAT search in subsequent BMC iterations.
- We enhance the default variable order of the SAT solver with structural information from BMC, without tampering with the internal details of the SAT solver.

Our experiments, conducted on a wide range of industrial designs, show that the proposed optimizations consistently provide between 1-2 orders of magnitude speedup and can be extremely useful in enhancing the efficacy of typical SAT-BMC tools.

The rest of the chapter is organized as follows. Section 2.3 describe the background and notation for BMC. Section 2.4 describes the related work in SAT-BMC. Section 2.5 presents the new techniques we have developed to enhance SAT-BMC. Section 2.6 shows how a simple procedure can dramatically improve the run-time of SAT-BMC.

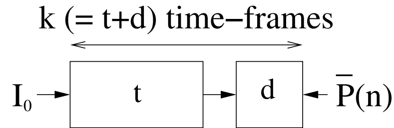


Figure 2.1: ILA of Current BMC Iteration

Section 2.7 discusses the experimental results, and Section 2.8 summarizes the chapter.

2.3 Background & Notation

This work is based upon the following SAT-BMC framework. We are given a temporal logic property p to be verified on a finite transition system $\mathcal{M}$. For simplicity of exposition let us assume that $\mathcal{M}$ has a single initial state I_0 and that the property p is an *invariant* [1]. The BMC problem is posed on a k-timeframe unrolled *iterative logic array (ILA)* of $\mathcal{M}$, as shown in Figure 2.1. The module $\overline{P}(n)$ is a monitor that checks for the violation of property p in *any* of the last (rightmost) n time-frames. We assert $p = 1$ in the first (leftmost) $k - n$ time-frames as these time-frames would have already been checked for violation of p in earlier BMC iterations. As observed in [CFF$^+$01] and confirmed by our experience, these assertions enhance the efficiency of the BMC. This feature is not explicitly shown in Figure 2.1 but will be assumed for the rest of the exposition. The ILA of Figure 2.1 is translated into a CNF formula and decided by a conventional SAT solver (in our case the `zchaff` solver [zch03]). This

[1] The discussion can be easily generalized to multiple initial states and arbitrary LTL properties [CBRZ01].

process starts with $k = 1$ and is iterated with increasing values of k till a violation of property p is detected or a user specified bound on k or some resource constraints are exceeded. In each successive iteration k is increased by n, *i.e.*, n new time-frames are added to the ILA from the current iteration. n is also known as the *step size* of BMC problem.

Our implementation incorporates the *bounded cone of influence (BCOI)* reduction proposed in the original BMC work [CBRZ01]. We define the notion of a *Pre-Saturation Region* (*PSR*) for the given BCOI cone, as illustrated in Figure 2.2. The outer dotted rectangle represents the unrolling of the circuit for j time-frames, while each small dotted rectangle represents one time-frame of the circuit. The solid polygon shows the reduced circuit based on the BCOI reduction. The shaded region denotes the *PSR* for the given BCOI. Simply put, it is the part of the BCOI where the BCOI is growing with each successive time-frame, *i.e.*, for time-frames of the BCOI outside the PSR, the BCOI and *cone of influence (COI)* are identical. The depth of the PSR (in terms of number of time-frames it encompasses) is denoted δ_{PSR}. *e.g.*, $\delta_{PSR} = j - i + 1$ in Figure 2.2. Further we define $d = max(n, \delta_{PSR})$. Thus the k-length ILA of Figure 2.1 is composed of two parts: (1) an ILA of d time-frames encompassing the PSR and the monitor block $\overline{P}(n)$ implemented on the last (right-most) n of these d time-frames and (2) a *tail* ILA of the remaining t time-frames where $t = k - d$. The first few iterations of BMC will have no tail block and only a part of the PSR.

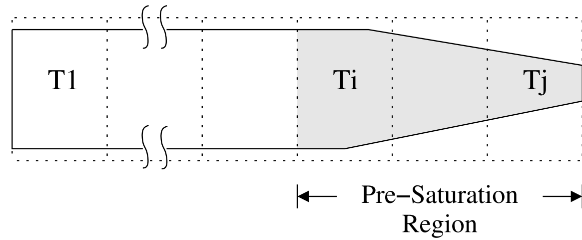


Figure 2.2: Pre-Saturation Region (PSR) of BCOI

2.4 Related Work

The application of incremental reasoning to the BMC problem was originally done within the SATIRE framework [WKS01] and later by Strichman [Sht04], under the name of *constraints sharing*. However, the typical speed-up observed in these works was about 2×. We have enhanced the basic incremental reasoning technique used in [Sht04, WKS01] to consistently achieve speed-ups of an order of magnitude or more. Recently, Een *et al.* [ES03] have proposed a new implementation for incremental SAT which can be used for incremental BMC. Our ideas for improving incremental BMC can potentially be applied in that framework as well.

Our idea of using learned clauses from a SAT-based induction run to enhance SAT-BMC bears some similarity to [CNQ03] where a BDD-based over-approximate reachability analysis is performed to generate learned clauses for BMC or [GGW+03b] where BDD-based analysis is performed on the unrolled circuit to generate clauses for BMC. However, the key difference in our approach is that the learned information is not generated from a separate analysis engine (BDDs in the above case), nor is our analysis performed solely for the purpose of improving the BMC runs. The SAT-

based induction run used in our approach is a complete approach in itself and can potentially prove the correctness of the property. Hence it nicely complements the BMC approach. The work of Een *et al.* [ES03] also performs SAT-BMC and induction in a single framework which potentially shares learned information between different SAT problems. However, the focus in that work is to realize a complete and efficient SAT method based on induction whereas our approach uses a *single* run of inductive reasoning, primarily as a learning mechanism to accelerate subsequent BMC runs.

SAT solver variable orderings customized for BMC were first explored by Strichman in [Sht04]. However the static variable ordering scheme proposed there was developed as an alternative to the DLIS ordering used in the GRASP [MSS99] SAT solver and its pre-decessors. As shown in [SZ03], this scheme is easily surpassed by the default conflict-based ordering heuristics of recent SAT solvers such as **zchaff** [MMZ$^+$01] and **BerkMin** [GN02]. Our approach, on the other hand, does not attempt to replace the SAT solver's default ordering scheme. Rather, the attempt is to complement it with useful information derived from the BMC problem, in a non-intrusive manner. Another recent work on SAT variable orderings for BMC [WJHS04] uses information from the *unsatisfiable cores* [ZM03] of previous SAT-BMC iterations to refine the variable order. We believe this idea can be effectively combined with our ideas for variable ordering to further improve its performance.

2.5 Incremental Deductive & Inductive Reasoning for SAT-based BMC

2.5.1 Incremental Reasoning

The philosophy behind incremental reasoning is very simple. When two CNF formulas share many common clauses, it may be possible to pass the knowledge learned during the SAT solution of the first formula to the second one, so that the solving of the second formula becomes easier. The learned knowledge is normally expressed in terms of conflict clauses. The following theorem, which is a formal re-statement of the understanding gained from past works on incremental SAT, *e.g.*, [WKS01], captures the essence of incremental reasoning.

Theorem 2.5.1 *Let $\varphi_1, \varphi_2, \psi, \zeta_1, \zeta_2$ and ϕ be CNF formulas, and let $\varphi_1 = \psi \wedge \zeta_1$, $\varphi_2 = \psi \wedge \zeta_2$. If solving $SAT(\varphi_1)$ produces the clauses of ϕ as conflict clauses such that each clause of ϕ is an implicate of ψ, then $SAT(\varphi_2) \equiv SAT(\varphi_2 \wedge \phi)$.*

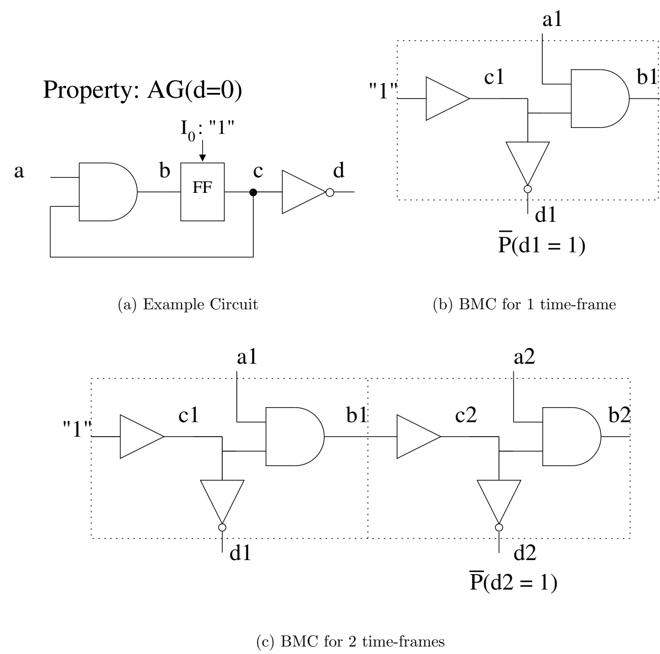
Proof: Since *every* clause of ϕ is an implicate of ψ, $\psi \Rightarrow \phi$. Hence, $\psi \equiv (\psi \wedge \phi)$. Therefore, $(\psi \wedge \zeta_2) \equiv (\psi \wedge \zeta_2 \wedge \phi)$. ■

Thus, during the process of solving CNF φ_1 if the SAT solver can analyze and gather the conflict clauses derived *solely* from the common portion of the CNFs, *i.e.*, ψ in the above, then these clauses, ϕ can be pre-preemptively added to the CNF φ_2 and will hopefully accelerate its subsequent SAT solution. SAT solvers such as SATIRE [WKS01] and zchaff [zch03] support this form of incremental reasoning.

Intuitively, SAT-BMC problems seem to be an ideal candidate for incremental reasoning since a large portion of the problem is common between successive BMC iterations. However, past attempts at incremental BMC [Sht04, WKS01] have met with only moderate success, with speed-ups of the order of $2\times$. In the following we offer some key insights to improve upon the basic incremental BMC formulation of [Sht04, WKS01] that allows us to consistently achieve speed-ups of an order of magnitude or more.

Identifying the shared CNF ψ: As discussed in earlier works [WKS01], the key to successfully using incremental SAT in solving two CNFs φ_1 and φ_2 is to identify a *significant common* portion ψ between them. Suppose we are given two successive iterations, I_j and I_{j+1} of the BMC problem, to be solved on k and $k+n$ length unrollings of the system, respectively. At first glance it may appear that the problem of I_j is a *proper subset* of the problem being solved in iteration I_{j+1}. So one might be tempted to mark the entire CNF of iteration I_j as the common portion ψ, *i.e.*, transfer *all* conflict clauses generated while solving I_j to the BMC problem of iteration I_{j+1}. However, as shown in the following example, this could lead to an incorrect result.

Example 2.5.1 *Given the circuit shown in Figure 2.3(a), let us assume that the initial value of the flip-flop c is 1, and that the property to be verified is* $AG(d = 0)$*, that is an invariant of* $d = 0$*. Figure 2.3(b) and 2.3(c) correspond to the first and the second iteration of BMC, which check for a counter-example in the first and second time-frame respectively. Apparently,* $(d = 0)$ *holds only for the first time-frame. In other words, the CNF formula for Figure 2.3(b) should be UNSAT, while the formula for Figure 2.3(c) should be SAT. It is easy to check that it is possible to learn the clause* $(\overline{c_1})$ *as a conflict clause while solving the first iteration of BMC (this clause can be*



(a) Example Circuit

(b) BMC for 1 time-frame

(c) BMC for 2 time-frames

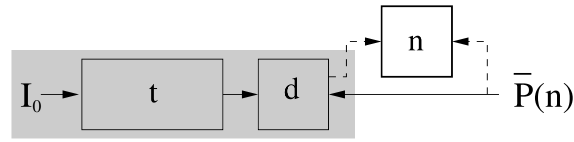
Figure 2.3: Example: Incorrect Use of Incremental BMC

derived from the assertion clause (d_1)*). If we allow all learned conflict clauses to be carried over directly to the second iteration, the second iteration would be rendered UNSAT by the addition of clause* $(\overline{c_1})$*. This is obviously incorrect.*

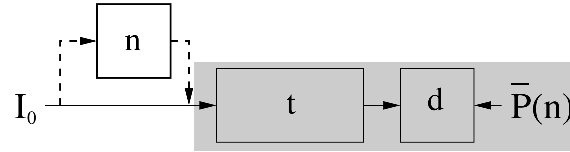
Note that the fundamental change between two successive BMC iterations is the insertion of a block of n additional time-frames. We make the key observation that the point of insertion of the new time-frames with respect to the ILA of the current iteration (Figure 2.1) can significantly impact the efficacy of the resulting incremental BMC scheme. Essentially, this decision determines what elements of the BMC problem can be included in the shared portion ψ, and hence which conflict clauses are transferable to the next iteration.

Figures 2.4(a)-2.4(c) show three different ways to unroll the circuit for the next iteration, where a new block of n time-frames is inserted into the ILA of Figure 2.1. The first option, shown in Figure 2.4(a), is to add the new block to the right of ILA. In the second alternative, shown in Figure 2.4(b), the new block of n frames is added to the left of the current ILA. Lastly, the new time-frames can be inserted somewhere in the middle of current ILA. The CNF formula for a typical BMC iteration (Figure 2.1) has four basic components:

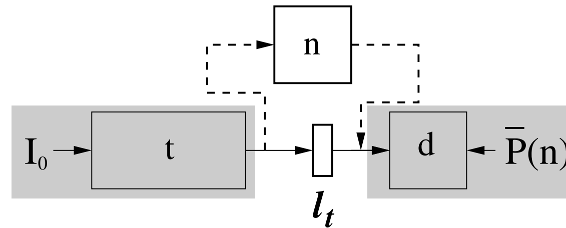
- $\phi_{\mathbf{I_0}}$: Clauses for asserting the initial state, I_0.
- $\phi_{\overline{\mathbf{P}}}$: Clauses for representing the monitor circuit $\overline{P}(n)$.
- $\phi_{\mathbf{CL}}$: Clauses for the logic gates implementing the functionality of each time-frame.
- $\phi_{\mathbf{l_i}}$: Clauses for the latches l_i of each time-frame i, which appear as buffers in



(a) New block added to the right



(b) New block added to the left



(c) **Proposed**: New block added in the middle

Figure 2.4: Incremental Reasoning for BMC

the unrolled circuit and connect the next-state outputs of one time-frame to the present-state inputs of the next time-frame.

For the right-hand-side insertion scheme (Figure 2.4(a)) the monitor circuit needs to shifted right, *i.e.*, re-posed on the newly inserted block of n time-frames. Therefore the clauses $\phi_{\overline{\mathbf{P}}}$ cannot be included in ψ, the shared clauses, for incremental BMC. Consequently any conflict clauses derived from the monitor circuit and/or linking the monitor circuit to the internal signals of the ILA cannot be used for incremental BMC

in the next iteration. Further, it is easy to see that the BCOI reduction will need to be recomputed in each iteration.

On the other hand, the left-hand-side insertion scheme (Figure 2.4(b)) necessitates re-posing the initial state clauses ($\phi_{\mathbf{I_0}}$) on the present-state variables of the left most time-frame in the newly inserted block. Thus, the clauses $\phi_{\mathbf{I_0}}$ cannot be included in the shared set ψ and consequently any conflict clauses expressing the implications of the given initial state on the problem cannot be used in incremental BMC of the next iteration.

Finally, our proposed scheme, shown in Figure 2.4(c) inserts the new block in the middle of the current ILA, specifically right before the PSR block. As shown in Figure 2.4(c) the next BMC iteration can be constructed from the current one by simply replacing the latch-buffer clauses $\phi_{\mathbf{l_t}}$, which link the PSR block to the tail block with the clauses for the newly added n time-frames. The entire shaded portion of Figure 2.4(c) forms the shared portion for incremental BMC of subsequent iterations. Consequently, many useful conflict clauses derived from both I_0 and $\overline{P}(n)$ can be transferred to subsequent BMC iterations. Note that, clauses $\phi_{\mathbf{l_t}}$ and any clauses derived from them are not carried over to future BMC iterations. Further, since the new block is inserted in the saturated portion of the ILA, the BCOI need not be re-computed in each iteration. Using the above notation we can formally define this scheme as follows.

Definition 2.5.1 (Middle Insertion Scheme) *Given the CNF φ_i for the BMC problem of the current iteration i, where $\varphi_i = \phi_{\mathbf{I_0}} \wedge \phi_{\overline{\mathbf{P}}} \wedge \phi_{\mathbf{CL}} \wedge \bigwedge_{j=1}^{k} \phi_{\mathbf{l_j}}$, ψ can be constructed as $\psi = \phi_{\mathbf{I_0}} \wedge \phi_{\overline{\mathbf{P}}} \wedge \phi_{\mathbf{CL}} \wedge \bigwedge_{\{j:1 \leq j \leq k, j \neq t\}} \phi_{\mathbf{l_j}}$. If ϕ be the set of conflict clauses derived solely from ψ, during the solution of $SAT(\varphi_i)$, then the incremental*

BMC CNF for iteration $i+1$ can be constructed as $\varphi_{i+1} = \psi \wedge \phi \wedge \phi_{\mathbf{CL_n}} \wedge \bigwedge_{j=k+1}^{k+n} \phi_{\mathbf{l_j}}$, where $\phi_{\mathbf{CL_n}}$ are clauses for the newly inserted time-frames and $\bigwedge_{j=k+1}^{k+n} \phi_{\mathbf{l_j}}$ denote the latch clauses connecting these time-frames.

Theorem 2.5.2 *The middle insertion scheme of Definition 2.5.1 preserves the correctness in incremental BMC.*

Proof: The proof follows directly from applying Theorem 2.5.1 to the ψ of Definition 2.5.1, as every conflict clause included in iteration $i+1$ is an implicate of the common CNF formula ψ for both iterations i and $i+1$. ■

The key to the correctness of this scheme is to observe that the BMC problem for the next iteration can indeed be constructed from the previous iteration as shown in Figure 2.4(c), *i.e.*, by inserting the new block in place of the latches l_t.

Note: The important difference between the three schemes is in the component of the current ILA that is excluded from the common portion ψ for incremental reasoning. For instance, in theory, one may have a right-insertion scheme (Figure 2.4(a) *mimic* middle insertion by excluding clauses of latches l_t from ψ and then shifting the conflict clauses in ϕ, derived from block d, by n time-frames to the right. However, closer inspection will reveal that this is essentially a more complicated way of realizing the middle insertion scheme. Further, the analysis of what clauses to shift and the shifting itself is not part of the incremental SAT framework available in SAT solvers and would need to be implemented externally.

Topologically driven variable ordering: Our framework uses a hybrid SAT variable ordering scheme. It combines an initial order derived from the topology of the

Instance	Bound	Default Incremental	Improved Incremental
C1(U)	100	2259	1368
C12(U)	130	1155	30
C22(S)	60	>7200	618
C32(S)	40	884	207

Table 2.1: Improvements to Incremental BMC

BMC circuit, (which provides a high level structure to the SAT search) with a conflict driven dynamic variable ordering scheme similar to that used in modern SAT solvers like zchaff [MMZ+01] and BerkMin [GN02]. Although equally applicable to non-incremental BMC, it nicely complements the other enhancements discussed in this section and is an important component of the overall gains realized through incremental BMC. Further details on the variable ordering algorithm and associated performance gains are discussed in Section 2.5.3.

Role of Incrementally learned clauses in variable ordering: By default, clauses incrementally inherited from previous BMC iterations contribute to the initial variable order just like original clauses of the CNF. This can have some harmful side-effects. For example, these clauses could cause the search to be biased toward areas of the problem with higher concentration of inherited conflict clauses and away from the newly added time-frames, which do not inherit any conflict clauses from previous time-frames. This would disrupt the topology driven structure imposed on the search by our variable ordering scheme. Therefore, in our incremental BMC implementation, incrementally inherited conflict clauses are not allowed to influence the variable order

of the SAT solver. Instead, they are only allowed to participate in search-space pruning. However, future conflict clauses derived from these clauses, during the SAT search, are allowed to contribute to the variable order according to the dynamics of the variable ordering scheme. This relatively simple idea, has a significant impact on the stability and robustness of the BMC flow.

Table 2.1 demonstrates the potential speedups that the middle insertion scheme described above can offer, compared to a default incremental run. The default configuration is the one shown in Figure 2.4(b). Both configurations use the same variable ordering as described above. For each instance, the number of time-frames is first reported under the column **Bound**, followed by the execution times with the default configuration and with our method, respectively. The letter 'U' denotes the instance is UNSAT and 'S' for SAT. In several cases, such as *C12*, the speed-up can be fairly dramatic. A more comprehensive comparison of our approach with respect to a *non-incremental* iterative BMC is made in Section 2.7.

2.5.2 Induction-based Learning

SAT-based inductive reasoning for property checking was proposed by the original proponents of BMC [CBRZ01] and stronger variants of induction such as *induction with depth* and *unique states induction* were later developed and reported by other authors [BC00, GGW$^+$03a, SSS00]. These works observed that for certain design-property combinations SAT-based inductive reasoning was significantly superior to state-space traversal, for proving the property.

The main criticism of SAT-based inductive reasoning has been that it can prove only

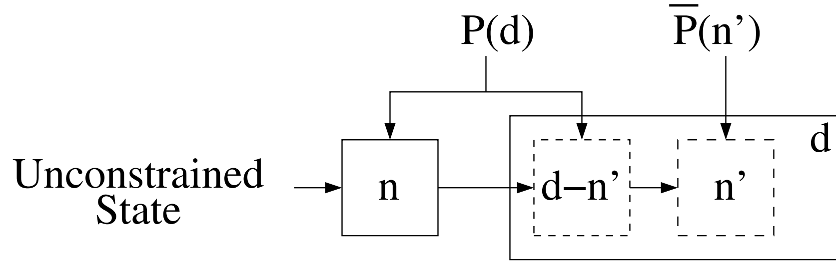


Figure 2.5: Induction-based Learning

a very small percentage of properties [2]. For the vast majority of cases the induction run proves to be an overhead. We have developed a novel use of inductive analysis whereby the SAT-based inductive analysis can be used to prove the property where possible and in the remaining cases conflict clauses learned during the induction run can be used to accelerate future iterations of SAT-BMC.

Our method is based on a variant of *induction with depth* [SSS00]. Suppose previous SAT-BMC iterations have established the absence of a counter-example (to the property being verified) in the first d time-frames from the initial state. The construction for the induction step is shown in Figure 2.5. Each block is named by the number of time-frames it contains. The size of block n is equal to the *step size* of SAT-BMC, and the size of the block d is $max(n, \delta_{PSR})$ (as defined in Section 2.3). Block d can be further decomposed into block n' and block $d - n'$, where block n' contains n timeframes (equal to block n), and block $d - n'$ contains all remaining time-frames of block d. The correctness of the property p is asserted on all frames in block n and block $(d - n')$. The counter-examples to p are checked on all time-frames of block n'.

[2]Variants such as unique states induction [SSS00] are complete but too expensive to apply in practice.

The initial state lines are left open. This construction is decided by a SAT solver. If the SAT solver returns UNSAT, the property can be shown to be inductively true for all depths [SSS00].

However, if the above run returns SAT (which is usually the case), we can intelligently apply the learned conflict clauses to subsequent BMC runs. In fact the construction of Figure 2.5 has been made identical to right-hand portion of a typical BMC iteration step (Figure 2.4(c)) to facilitate precisely this. Conflict clauses learned during the SAT solution of Figure 2.5 can be divided into three parts:

- H clauses: clauses that only involve variables in block d.
- T clauses: clauses that only involve variables in block n.
- HT clauses: clauses that involve variables in both block d and n.

These learned clauses will be applied to subsequent BMC iterations as outlined in the following algorithm.

1: Add H clauses **once** and **permanently** to the clause database.
2: For **each** new BMC iteration, clone T clauses to T' and add T' clauses **permanently** to the clause database.
3: For **each** new BMC iteration, clone HT clauses to HT' and add HT' clauses **temporarily** to the clause database.
4: After each iteration terminates, remove HT' clauses as well as any new conflict clauses derived from HT' clauses from the clause database. *i.e.*, HT' clauses or their derivatives are **not** incrementally carried over to the next iteration.

Algorithm 2.5.1: Application of Inductively Learned Clauses

The cloning in Step 2 is to conform to the variable names in the block n in Fig-

ure 2.4(c). This can be implemented by adding a displacement $\delta(i)$, which can be easily computed for each iteration i, to each literal in the T clauses. On the other hand, the cloning in Step 3 can be implemented by adding the $\delta(i)$ **only** to those literals in the HT clauses that belong to block n.

Clauses are added permanently to the database only if they are guaranteed not to alter the result of all subsequent BMC iterations. Each BMC iteration may derive new clauses from these permanently added clauses. Apparently, newly derived clauses will preserve the satisfiability of BMC problem as well. Our incremental SAT-BMC automatically transfers all permanent clauses as well as conflict clauses derived from them to subsequent iterations.

The correctness of the above scheme can be established by following theorems, where φ_i stands for the CNF for the ith iteration of BMC, n_i represents the newly added block in i^{th} iteration, and $\phi_{\overline{\mathbf{P}}}$, $\phi_{\mathbf{CL}}$, and $\phi_{\mathbf{l_i}}$ are as defined in Section 2.5.1.

Theorem 2.5.3 *It is sound to add H clauses permanently to BMC clause database.*

Proof: The CNF for the block d can be constructed as $\psi = \phi_{\mathbf{CL(d)}} \wedge \phi_{\overline{\mathbf{P}}}$. Because for any i, $\psi \subset \varphi_i$, and because H is derived solely form ψ, it follows that for all i, $SAT(\varphi_i \wedge H) \equiv SAT(\varphi_i)$ ■

Theorem 2.5.4 *It is sound for each BMC iteration to add T' clauses permanently to the clause database.*

Proof: For each BMC iteration i, n_i is structurally identical to block n in Figure 2.5. Since T clauses are solely derived from $\phi_{\mathbf{CL(n)}}$, it is apparent that T' clauses are solely derivable from $\phi_{\mathbf{CL(n_i)}}$ as well.

Since $\phi_{\mathbf{CL(n_i)}} \subset \varphi_{\mathbf{j}}$, for any $j \geq i$, it follows that for any $j \geq i$, $SAT(\varphi_{\mathbf{j}}) \equiv SAT(\varphi_{\mathbf{j}} \wedge \phi_{\mathbf{CL(n_i)}}) \equiv SAT(\varphi_{\mathbf{j}} \wedge T')$. ■

Theorem 2.5.5 *It is sound for each iteration to add HT' clauses to the clause database. However, HT' clauses should be removed upon the completion of current iteration.*

Proof: The CNF for the block nd, the merger of block n and block d in Figure 2.5, can be expressed as:

$$\psi = \phi_{\mathbf{CL(n)}} \wedge \phi_{\mathbf{CL(d)}} \wedge \phi_{\mathbf{l_{nd}}} \wedge \phi_{\mathbf{P(n)}} \wedge \phi_{\mathbf{P(d-n')}} \wedge \phi_{\mathbf{\overline{P}(n')}}.$$

For each BMC iteration i, the CNF for the block $n_i d$, the merger of the block n_i and the block d, can be expressed as:

$$\psi_{\mathbf{n_i}} = \phi_{\mathbf{CL(n_i)}} \wedge \phi_{\mathbf{CL(d)}} \wedge \phi_{\mathbf{l_{n_i d}}} \wedge \phi_{\mathbf{P(n_i)}} \wedge \phi_{\mathbf{P(d-n')}} \wedge \phi_{\mathbf{\overline{P}(n')}}.$$

Because the block $n_i d$ and the block nd are structurally identical, and because HT clauses are solely derived on ψ, HT' clauses should be solely derivable from $\psi_{\mathbf{n_i}}$.

Since we also have $\psi_{\mathbf{n_i}} \subset \varphi_i$, we can derive $SAT(\varphi_i) \equiv SAT(\varphi_i \wedge \psi_{\mathbf{n_i}} \wedge HT')) \equiv SAT(\varphi_i \wedge HT'))$

However, since $\phi_{\mathbf{l_{n_i d}}} \notin \varphi_{i+1}$, then $\psi_{\mathbf{n_i}} \notin \varphi_{i+1}$ thus the following may be true $SAT(\varphi_{i+1}) \neq SAT(\varphi_{i+1} \wedge \psi_{\mathbf{n_i}}) \equiv SAT(\varphi_{\mathbf{i+1}} \wedge HT')$ ■

Similar to the scenario described in Section 2.5.1, the conflict clauses learned from the induction-based reasoning are not allowed to influence the variable order of the SAT solver. Instead, they only contribute to search-space pruning. It is interesting to note that while the incremental formulation of Section 2.5.1 does not contribute any

Instance	Bound	No Learning	Learning
C2(U)	130	5052	2908
C3(U)	130	2812	351
C5(U)	130	14274	1140
C7(U)	130	1749	844
C36(U)	130	0.36	0.2*
C37(U)	130	0.7	0.2*

*property proved *true* by induction.

Table 2.2: Effect of Induction-based Learning on BMC

learned clauses to the newly added block n (Figure 2.4(c)) the above induction-based learning scheme contributes learned information both for the block n (T' clauses) as well its interaction with adjoining time-frames (HT' clauses). Thus, it nicely complements and bolsters the incremental learning of Section 2.5.1.

Table 2.2 shows the runtime for a few industrial instances. We can see that the induction-based learning can be very powerful, especially for hard UNSAT cases. In the last two instances the induction is actually able to prove the property, obviating the need for any further BMC. Note that the results of the last column include the time for the learning, which is typically in a fraction of a second for easy cases, and up to a few seconds for hard ones. Occasionally, the learned information can slow down the SAT solver, but we have found such cases to be rare. Usually the learned information provides significant performance improvements.

2.5.3 Variable Ordering

Variable ordering has long been recognized as a key determinant of the performance of SAT solvers. Often domain-specific knowledge can be used to enhance the SAT solver's performance for a given application. Since our SAT-BMC framework uses a CNF-based SAT solver, our aim was to improve upon the default ordering heuristics used in modern SAT solvers (such as the VSIDS heuristic of zchaff [MMZ+01]) by using domain specific knowledge from the BMC problem. Although not a major contribution of this work, our investigations revealed a few simple but interesting insights on variable ordering that can provide significant speed-ups over the default variable ordering of the SAT solver.

1: Compute initial variable scores using **Metric 1**
2: Sort variables by scores and break ties using **Criterion 2**
3: At each conflict increment scores of variables *involved* in the conflict
4: Periodically decay all scores and update variable order

Algorithm 2.5.2: Generic Hybrid Variable Ordering

Algorithm 2.5.2 shows a generic set-up for realizing a variety of variable ordering schemes that combine the default conflict driven variable ordering used in modern SAT solvers like zchaff and BerkMin with a minimal amount of external information, possibly derived from the problem domain. For example, a fully static, topological order can be realized by removing steps 3 and 4, choosing *Metric 1* in step 1 to assign the same score to all variables and setting *Criterion 2* to prioritize variables based on a topological order. In contrast, the default order that would be followed by a solver like zchaff would be to set *Metric 1* to assign an initial score to each variable equal to the number of clauses it appears in and *Criterion 2* would break ties based

on the variable number that was assigned to each variable during CNF generation. A whole slew of different schemes can be realized by different choices for *Metric 1* and *Criterion 2*. We experimented with several fully static as well as hybrid schemes. Our results concur with the observation of [SZ03] that *fully* static schemes, including those custom-generated for a BMC problem (*e.g.*, the one proposed in [Sht04]) are easily surpassed by the default conflict-driven ordering heuristics of modern SAT solvers. In addition, we made the following observation which is key to our particular choice of the variable ordering scheme:

• **Observation:** In the scheme of Algorithm 2.5.2 the choice of the initial order and variable scores (steps 1 and 2) can have a significant impact on the performance of the SAT solver.

The reason for this observation is that even though the conflict-driven dynamic variable ordering of the SAT solver (steps 3 and 4) is free to alter the initial order, in practice such changes are gradual and usually not dramatic. Therefore, the initial order determines the high-level structure of the search tree, and hence the efficiency of the SAT search. Specifically, the scheme that worked fairly well for us can be realized by the following choices in Algorithm 2.5.2:

• **Metric 1:** Initial score for each variable equal to the number of clauses it appears in, in the BMC CNF.

• **Criterion 2:** Break ties by prioritizing variables based on a topological ordering of them from primary inputs toward the property monitor in the unrolled BMC circuit.

As discussed above, the default ordering scheme of the SAT solver (zchaff in our case) is to use the same choice of Metric 1 as above but to break ties (Criterion 2) based on

the variable numbers assigned to variables during CNF generation. While this could differ from one BMC implementation to another, a typical method is to generate the BMC CNF by traversing and unrolling the circuit backward from the property monitor toward the primary inputs (since this yields the BCOI as a by-product). This is the default scheme used in our experiments. Interestingly, this choice of *Criterion 2* is exactly the opposite of our proposed scheme.

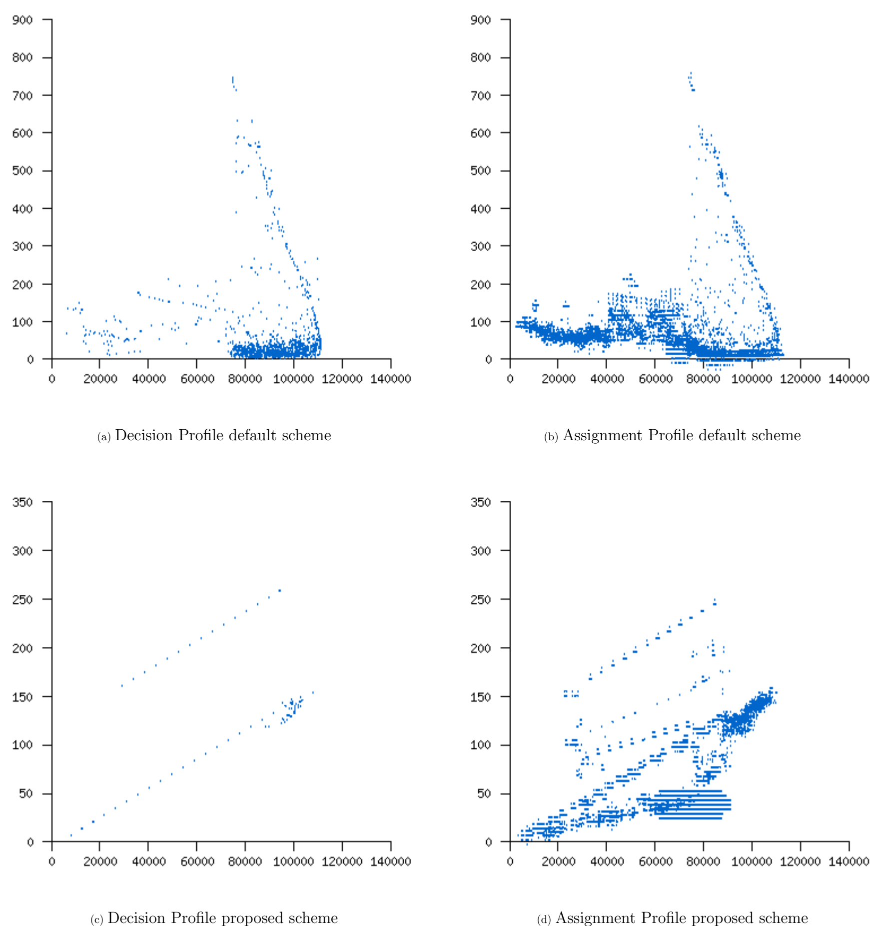


Figure 2.6: Profiling of SAT Solver for Two Different Variable Ordering Schemes

Instance	Bound	Default Order	New Scheme
C1(U)	100	3242	2259
C14(U)	190	1298	151
C23(U)	50	952	126
C26(S)	50	2553	165

Table 2.3: Effects of Variable Ordering on BMC

The experimental results for a subset of our benchmarks, in Table 2.3, show that this simple insight can potentially produce significant speedups of up to an order of magnitude compared to the default ordering scheme. While different speed-ups are observed on different benchmarks and in rare cases the default scheme even outperforms our scheme, we found our scheme to be significantly better on most of the instances we encountered. Most importantly, this scheme integrated very well with the other improvements described earlier.

We experimented with several other variants of Algorithm 2.5.2 including a slight variation on the default scheme above, where the ties were broken randomly in *Criterion 2*. This scheme proved both unpredictable and significantly worse than even the default scheme. Another interesting variant is one where *Metric 1* in step 1 assigns the same score to all variables and *Criterion 2* prioritizes variables based on a topological ordering based on the BMC circuit. This essentially sets the initial variable order to a *pure* topological order (inputs to outputs or the reverse). We found this scheme also to be significantly worse than both our scheme and the default one. Figure 2.6 shows the profiling of decision levels of each branching variable, and assignment made during SAT solution.

While a detailed discussion on the above findings is beyond the scope of this chapter, a couple of pertinent observations are in order. First, having the initial order as a purely topological (such as the last scheme discussed above) is usually worse than the "two-step" scheme followed in our scheme as well as the default order, which prioritize assignment to high-weight variables across time-frames. Second, our scheme which essentially makes assignments from inputs to outputs generally performs better than the default scheme which assigns from outputs to inputs. Interestingly the latter is more akin to the justification-based ordering schemes which have worked very well for circuit-based SAT solvers [IPC03, KPKG02]. The reason for this could be that the default scheme is at best a semi-static justification scheme, which could be fairly poor at "guessing" the right assignments to justify and hence nearly not as effective as the dynamic justification algorithms implemented in [IPC03, KPKG02].

2.6 Logic Optimization for BMC

We also observed that the BMC runtime can be substantially improved by applying a lightweight single pass of combinational logic optimization to simplify the circuit, before performing BMC. This is partly due to the fact that for model checking RT-level specifications they need to be synthesized down to logic level where engines such as SAT or BDDs can be applied to them. Verification front-ends typically perform a low cost synthesis which may produce fairly unoptimized logic. Moreover, since a SAT solver operates on a particular representation of a logic function (rather than BDDs which build and work on a canonical form) the efficiency of SAT-BMC can be significantly influenced by optimizing the logic representation.

Instance	Bound	Baseline	Base+Logic optim.	Compression
C11(U)	130	1980	170	25%
C15(U)	80	1844	163	24%
C17(U)	90	5676	360	22%
C18(S)	60	483	37	30%
C34(S)	30	2474	242	14%

Table 2.4: Effects of Logic Optimization on BMC

We have used a simple script based on fanin don't care optimization and constant propagation which we found to give a good trade-off between optimization effort and BMC runtime reduction. The average runtime of the optimization is only on the order of a few seconds for all our benchmarks. However, we have found that even though it reduces the CNF size by only 20-30%, it can give up to an order of magnitude speedup in BMC runtime. Table 2.4 shows the effect of running logic optimization on a few representative instances from our benchmark suite. Column 3 and 4 show the run time of our baseline implementation of the BMC without and with logic optimization. Column 5 shows the compression ratio of the size of CNF due to logic optimization.

2.7 Experimental Results

Our BMC framework: We have implemented a SAT-based BMC package in C++ within the VIS framework [The96]. Our package uses the 2003 release of the zchaff SAT solver [zch03]. Further, our base implementation (column 4-5) includes many of the previously published optimizations for BMC including the BCOI reduction, the

Inst.	PI	FF	Gate	Inst.	PI	FF	Gate	Inst.	PI	FF	Gate
C1	69	118	640	C2	69	122	641	C3	208	868	4880
C4	236	778	4256	C5	236	778	4257	C6	85	161	1385
C7	85	161	1385	C8	118	367	1436	C9	118	367	1489
C10	118	375	1496	C11	118	375	1495	C12	15	53	419
C13	15	53	418	C14	15	53	416	C15	63	159	1094
C16	330	1158	5155	C17	405	1951	8965	C18	405	1951	8965
C19	114	125	531	C20	103	122	482	C21	53	70	460
C22	308	746	3837	C23	308	746	3840	C24	393	1895	8514
C25	393	1895	8514	C26	393	1895	8514	C27	393	1895	8514
C28	58	66	612	C29	58	66	612	C30	289	654	4822
C31	289	654	4823	C32	289	654	4826	C33	289	655	4837
s38417.1	29	1562	13826	s38417.2	29	1563	13829	C34	289	657	4856
s38584.1	13	755	6667	s38584.2	13	614	4972	C35	289	654	4822

Table 2.5: Benchmark Characteristics

AND-INVERT graph (AIG) for representing the circuit (albeit without 2-level hashing), incremental reasoning etc. In addition, we employ a lightweight single pass of logic optimization as a pre-processor to simplify the circuit for the subsequent BMC. We use a combination of fanin don't care optimization (similar to the `simplify` command of the SIS tool [S$^+$98]) and constant propagation in our optimization script. Typically, the script takes of the order of a few seconds even for the largest circuit but usually speeds-up the downstream BMC by an order of magnitude or more. This is in agreement with the observation of Kuehlmann *et al.* [KPKG02] who have also em-

ployed logic optimization techniques, directly implemented on the AND-INVERTER graph, to speed up BMC. The ideas discussed in Sections 2.5.1 - 2.5.3 have been implemented on this base package.

We have tested our tool for safety properties extracted from several modules of 4 industrial designs and two large ISCAS'89 benchmarks. The four industrial designs contain 3356, 10463, 40055, and 16414 lines of RTL verilog code respectively. The cone of influence of the extracted properties contains 53 to 1951 latches and 416 to 8965 gates (after logic optimization). The properties for the ISCAS benchmarks correspond to justification problems for randomly generated states. The detailed characteristics of the testbenches are reported in Table 2.5. All experiments were run on a 2.4 GHz Pentium 4 machine with 1G RAM, running Redhat Linux.

We have run VIS-BMC along with three configurations of our BMC tool on above benchmarks. The results are shown in Table 2.6 and Table 2.7. Our base implementation (column 4-5) is much faster than the VIS BMC package on all benchmarks, often by 1-2 orders of magnitude. Thus we feel it represents a reasonable baseline for demonstrating our proposed optimizations. The base version uses the default variable ordering, basic incremental BMC and left-addition method (Figure 2.4(b)) in formulating the BMC. No induction-based static learning is enabled either. The *non-incremental* BMC (column 6-7) is our implementation of the most common way of doing the BMC. Essentially, it is the base method with incremental reasoning disabled but the proposed variable ordering enabled. The last configuration (column 8 -9) is our proposed method incorporating all the features discussed in Sections 2.5.1 - 2.5.3.

The timeout limit for each instance is set to two hours except for large ones, for

which up to 4 hours are allow. The upper bound is set to 260 for C12 - C15, 130 for all remaining ones. All configurations are run with step size of 10. The run-times reported are the cumulative time over all iterations.

Table 2.6 shows unsatisfiable instances (up to the limit of the upper bound). Table 2.7 shows the satisfiable instances, the length of the shortest counter-example is reported following the instance name. For every aborted instance, we report the largest bound at which a definite conclusion was drawn. For example, for C4, within the 2 hours limit, the VIS-BMC, our base implementation and non-incremental BMC can only reason up to 30, 50 and 110 frames respectively (that the problem is UNSAT), while our proposed BMC is able to finish all 130 frames with only 493 seconds. Similarly, for C17, the first three configurations are only able to finish up to 30, 80, and 90 time-frames respectively within the 4 hours limit, while our proposed BMC finishes all 130 frames with just 8922 seconds.

We can see that the proposed BMC has achieved orders of magnitude speedups, especially for hard UNSAT cases. Each individual technique may not work well independently for every benchmark, but collectively they improve on all benchmarks. For most of SAT cases, our BMC is much faster than non-incremental and base BMC, and is only marginally slower for a few SAT cases. Note that, the non-incremental BMC also uses proposed variable ordering scheme. This further illustrates the power of our incremental reasoning and the induction-based learning.

For comparison we have also run above benchmarks with the recently published SATORI [IPC03] tool. The default configuration of SATORI was used. The results are reported in columns 10-11 of Table 2.6 and Table 2.7. SATORI is a sequential Boolean reasoning engine that integrates circuit-driven search algorithms typi-

cally used in sequential ATPG with certain powerful techniques used in modern SAT solvers. We must point out that SATORI implements a *complete* search algorithm, *i.e.* given enough time it can conclusively prove or disprove a property. In that sense its comparison with SAT-BMC is not intended to be fair since SAT-BMC is designed to give bounded proofs (or counter-examples). However, the comparison is intended to put our work in the context of this somewhat orthogonal, yet powerful technology that is designed to work on similar kinds of problems and shares several algorithmic features with SAT-BMC engines. The results shown in Table 2.6 and Table 2.7 provide an interesting comparison of the two technologies and a clear proof that they are in fact orthogonal in performance. Specifically, in some instances SATORI was able to provide unbounded proofs of correctness for the property fairly quickly (*e.g.*, $C12 - C14$), or find really deep counter-examples where our SAT-BMC could only prove bounded correctness of the property (*e.g.*, $C4, C5$). Similarly in many cases the SATORI tool timed out without any partial answer but the SAT-BMC was able to provide a bounded proof of correctness of the property or discover a counter-example fairly quickly. Additionally, the length of the counter-example generated by SAT-BMC is always much shorter than that of SATORI, which is highly desired for error diagnosis purposes. There are interesting points of contrast between the algorithmic features of SAT-BMC and SATORI, that result in the orthogonal behavior of the two engines. The interested reader is referred to [IPC03] for details.

2.8 Summary

In this chapter we have presented a number of optimizations to improve the performance and capacity of a typical SAT-based bounded model checking framework. The proposed ideas include clever orchestration of variable ordering and learned information in an incremental framework for BMC, a novel idea to combine SAT-based inductive reasoning and BMC and BMC-specific ordering strategies for the SAT solver.

A key distinguishing feature of these techniques is that from an implementation standpoint they are all external to the SAT solver. However, they improve the performance of the SAT solver, in the context of the BMC problem, by influencing the use of learned information in the SAT solver and the variable ordering; the two features that are key to the performance of modern SAT solvers. Experiments on a wide range of industrial designs have shown that the proposed optimizations consistently provide between 1-2 orders of magnitude speedup and can be extremely useful in enhancing the efficacy of typical SAT-BMC tools.

There are several avenues for improving the above SAT-BMC framework. One possibility is to integrate some of the promising techniques published earlier in the literature, into our framework. These include the "on-the-fly" graph compression technique of [GA02] as well as the BDD-based clause learning techniques of [GGW+03b, CNQ03]. This and other issues will be the subject of our future investigations and development.

Instance	VIS-BMC		Base		Non Incr.		Our BMC		SATORI	
	Bnd	Time	Bnd	Time(s)	Bnd	Time	Bnd	Time(s)	Leng.	Time(s)
C1	70	>2h	110	>2h	120	>2h	130	**3697**	-	>2h
C2	70	>2h	100	>2h	110	>2h	130	**2908**	-	>2h
C3	30	>2h	50	>2h	110	>2h	130	**351**	-	>2h
C4	30	>2h	50	>2h	110	>2h	130	493	1311	**63**
C5	30	>2h	50	>2h	130	>2h	130	1140	2827	**392**
C6	40	>2h	130	**1530**	130	5300	130	2168	-	>2h
C7	70	>2h	130	**338**	130	>2h	130	844	-	>2h
C8	60	>2h	90	>2h	130	4657	130	**461**	-	>2h
C9	60	>2h	70	>2h	130	3945	130	**539**	-	>2h
C10	60	>2h	80	>2h	130	3649	130	**1328**	-	>2h
C11	60	>2h	80	>2h	130	3747	130	**474**	-	>2h
C12	70	>2h	150	4631*	260	3989	260	171	UNSAT	66
C13	80	>2h	190	6769*	260	4641	260	363	UNSAT	3955
C14	70	>2h	180	4719*	250	>2h	260	334	UNSAT	618
C15	60	>2h	140	4290*	190	>2h	260	**2950**	-	>2h
C16	40	>4h	110	>4h	130	12946	130	**3567**	-	>4h
C17	30	>4h	80	>4h	90	>4h	130	**8922**	-	>4h
C18	40	>4h	80	>4h	100	>4h	130	**12946**	-	>4h
s38417.1	80	>4h	110	>4h	130	9805	130	**2601**	-	>4h
s38417.2	70	>4h	100	>4h	130	9267	130	**2844**	-	>4h

*Time when memory usage exceeds 1G Byte

Table 2.6: Experimental Results on UNSAT Cases

Instance	VIS-BMC		Base		Non Incr.		Our BMC		SATORI	
	Bnd	Time	Bnd	Time	Bnd	Time	Bnd	Time	Length	Time
C19(59)	60	1228	60	40	60	**30**	60	41	-	>2h
C20(35)	40	16	40	4	40	4	40	**2.6**	1324	6138
C21(85)	80	>2h	80	>2h	80	>2h	90	**1611**	-	>2h
C22(60)	30	>2h	40	>2h	60	1247	60	**396**	-	>2h
C23(60)	30	>2h	40	>2h	60	3157	60	**1102**	-	>2h
C24(45)	40	>2h	50	4264	50	**565**	50	754	-	>2h
C25(45)	30	>2h	50	2380	50	1036	50	**545**	-	>2h
C26(45)	30	>2h	50	**500**	50	572	50	694	-	>2h
C27(45)	30	>2h	50	620	50	**363**	50	723	-	>2h
C28(23)	30	24	30	**3.3**	30	3.5	30	4.4	2143	664
C29(59)	60	1437	60	57	60	414	60	**20**	464	1218
C30(29)	30	6789	30	325	30	48	30	**32**	224	229
C31(29)	30	5254	30	370	30	**31**	30	40	-	>2h
C32(36)	30	>2h	40	5232	40	205	40	**152**	-	>2h
C33(34)	30	>2h	40	3157	40	127	40	**91**	-	>2h
C34(16)	20	255	20	3.9	20	2.5	20	**2.3**	-	>2h
C35(25)	20	>2h	30	242	30	21	30	35	120	**15**
s38584.1 (100)	70	>4h	90	>4h	100	6604	100	**3426**	-	>4h
s38584.2 (75)	70	>4h	80	**853**	80	1584	80	1039	-	>4h

Table 2.7: Experimental Results on SAT Cases

Chapter 3

Abstraction and Refinement for Design Verification at Logic Level

Abstraction refinement has recently emerged as an enabling technology for applying model checking techniques to large real-life designs. Previous techniques for abstraction refinement work on static abstractions, in that the abstract model produced by the abstraction algorithm is not modified by the downstream model checking. We propose a new, dynamic method of abstraction, which can be applied during successive steps of the model checking algorithm to further abstract the model produced by traditional static abstraction methods. This is facilitated by information gathered from an analysis of the proof of unsatisfiability of SAT-based bounded model checking problems, solved on the concrete model, and passed to the model checker. It effectively allows the model checker to work with smaller abstract models. Experiments on several industrial benchmarks demonstrate that dynamic abstraction significantly improves the performance of the abstraction refinement flow and also

enables the successful application of abstraction refinement-based model checking to larger designs.

3.1 Introduction

The application of formal verification techniques, such as model checking, to real-life industrial designs, has traditionally been hampered by what is commonly known as the *state explosion* problem. Dramatic increases in the size of digital systems and components and the corresponding exponential increase in the size of their state space have kept industrial-size designs well beyond the capacity of current model checkers. Abstraction refinement has recently emerged as a promising technology that has the potential to bridge this verification gap.

The basic idea behind *abstraction refinement* [Kur95] is to attempt to verify the property at hand on a simplified version of the given design. This simplified version, or *abstraction*, is generated by removing elements from the original design that are not relevant to the proof of the given property. If the property passes on the abstract model it is guaranteed to be true on the original design as well. However, if the property fails, counter-examples produced on the abstract model must be validated against the original design. If this is not possible, the process is iterated with another abstract model which approximates the original model more closely. The new abstract model can be obtained by embellishing the current abstraction with more details from the original design [WHL+01, CCK+02, WLJ+03] or by re-generating a more complete abstract model from the original design [MA03, GGYA03]. Usually the challenge in abstraction refinement is to construct as small an abstract model as possible, to

facilitate the model checking, while retaining sufficient detail in the abstraction to decide the property. Thus, the ideal technique for abstraction refinement is one which achieves a good balance between the *size* and *accuracy* of the abstract model.

Previous work on abstraction refinement-based model checking uses *static abstractions* in that the abstract model produced by the abstraction step is not modified in any way by the downstream model checking. In this work we make the key observation that when a property is checked on a circuit model using say a typical BDD-based symbolic model checker, there may be state elements that are *partially abstractable*,*i.e.*, while the state element is necessary in the proof of the property, it may actually be required only in certain image computation steps. For example, there are latches in the design that are solely present for initialization purposes, *i.e.*, for the purpose of proving the given property they effectively become redundant after a few steps of image computation.

We use this observation to develop a new method of *dynamic abstraction* whereby the design under verification is abstracted during successive image computation steps of the model checking phase. This abstraction can be performed *in addition to* and after the abstraction through a traditional static abstraction algorithm. It provides for a more aggressive yet accurate abstraction methodology, effectively allowing the core model checking algorithm to work on smaller abstract models. Information regarding the dynamic abstractability of different latches is deduced from an analysis of the unsatisfiable core of SAT-based bounded model checking problems, solved on the concrete model, and passed to the model checker. An important benefit of dynamic abstraction is that since it is tightly coupled to the model checker, feedback from the model checking process can be effectively used to guide the model checking process.

In fact we exploit this feature in one of our techniques for dynamic abstraction.

The main contributions of our approach are the following:

- We introduce and develop the notion of dynamic abstraction, which is orthogonal to static abstractions that have traditionally been used in abstraction refinement frameworks, yet fully compatible with most such frameworks.
- We propose two possible heuristic techniques for implementing dynamic abstraction in the context of a SAT-BMC based abstraction framework that uses a BDD-based model checker. While we found this specific framework to be a good fit for our initial implementation of the idea, it is important to note that the notion of dynamic abstraction goes far beyond the specific implementations of it reported here.
- We present experimental results on several large industrial benchmarks to demonstrate that dynamic abstraction can provide up to an order of magnitude of improvement over and above the performance improvement afforded by traditional static abstraction refinement techniques.

This chapter is organized as follows. Section 3.2 reviews related work on abstraction refinement followed by some background material in Section 3.3. We develop our dynamic abstraction algorithm in Section 3.4. In Section 3.5 we present an experimental evaluation of the proposed techniques. We summarize this chapter in Section 3.6.

3.2 Related Work

Abstraction refinement was first introduced by Kurshan [Kur95] for verifying linear time properties. The last few years have seen a lot of research activities on this topic. Abstraction refinement methods can be broadly classified into two kinds of methods:

1. counter-example driven approach.
2. counter-example independent approach.

Counter-example driven methods for abstraction refinement [WHL+01, CCK+02, CGKS02, BK04, MH04] typically work by iteratively refining the current abstraction so as to block *a particular* (false) counter-example encountered in model checking the previous abstract model. The refinement algorithm could use a combination of structural heuristics or functional analysis based on SAT or BDDs or some combination of these. A recent paper [GKMH+03] enlarges the scope of the refinement by using multiple counter-examples from the previous abstract model. This notion is further generalized by Wang *et al.* in [WLJ+03]. The GRAB tool described there uses a BDD representation of the set of all shortest counter-examples in the previous abstract model, called *synchronous onion rings (SORs)*. Each iteration (referred to as a *generation* in [WLJ+03]) performs a series of micro-refinements to eliminate all counter-examples represented in the SORs.

Counter-example independent abstraction refinement was introduced by McMillan and Amla in [MA03] and later developed by Gupta *et al.* [GGYA03]. The basic idea is to perform a SAT-based bounded model check (BMC) [CBRZ01], of the property, up to some depth k, on the original design and generate the abstract model through

an analysis of the *proof of unsatisfiability* [GN03, ZM03] of the BMC problem. Essentially, the abstraction excludes latches and/or gates that are not included in the proof of unsatisfiability of the BMC problem and thereby guarantees that the abstract model also does not have any counter-examples up to depth k. Successive abstract models are similarly generated by solving BMC problems of increasing depth. Our current implementation uses this framework of abstraction refinement although our contribution would be meaningful in counter-example driven frameworks as well.

Recent papers have also proposed improvements to other aspects of abstraction refinement-based model checking, most notably the concretization test and the granularity of abstraction. The use of BMC to concretize abstract counter-examples was first proposed in [WHL$^+$01] and most current abstraction refinement frameworks use some variant of SAT or ATPG-based BMC for this task. Bjesse and Kukula [BK04] proposed an enhancement whereby the concrete error trace need not be the same length as the abstract counter-example. Information from the abstract counter-example is merely used as a guide for the concretization algorithm. Other recent works [GKMH$^+$03, WHS04] have generalized the granularity of the abstraction. Here cut-points inserted at arbitrary points in the fanin cones of latches are also used as abstraction points in addition to latch outputs.

All of the above works share two common features, namely 1.) the abstraction step is algorithmically distinct from the model checking, *i.e.*, the abstraction is performed outside the model checker, and 2.) The abstraction is purely structural in nature and has no temporal component, *i.e.* the same structural abstraction is used for each image computation step in BDD-based model checking (or each unrolled time-frame in SAT-based BMC). This work can be distinguished from all previous work in

abstraction refinement based on these two aspects. Our proposed method analyzes the temporal behavior of various latches and based on this analysis, dynamically abstracts sets of latches *during* the course of the model checking. For example, in the case of a BDD-based model checker, successively more abstracted versions of the transition relation are used for successive image computation steps. A key point to note is that the dynamic abstraction can be applied in addition to any abstraction performed by any of the traditional static abstraction methods. In that sense our contribution is complementary to previous research on abstraction refinement.

3.3 Background

3.3.1 Computational Tree Logic & Model Checking

Computation tree logic (CTL) is a temporal logic system that is often used to express properties of a finite state machine (FSM) system. CTL requires that each of the future operators X, G, F and U be immediately preceded by a path quantifier, which is either existential quantifier **E** or universal quantifier **A**.

With a FSM, it is possible to describe all computation paths as a tree with root s_0, which is the initial state of FSM. If we know the present node in the tree, we know the history and the presence, since the path back to the initial state is unique.

The most widely used properties are *safety* and *liveness* properties. The safety properties describes that something bad will never happen, while liveness says that something good will eventually happen. Safety properties can be modeled as: $\mathbf{A}\mathrm{G}\ \neg f$ and liveness can be represented as $\mathbf{E}\mathrm{G}\ g$.

Model checking (MC) is the process of determining if a given property is true in a given model. Sequential systems are often represented as finite state machines (FSM). Although the number of states of a FSM is finite, it is almost impossible to explicitly enumerate all of them to verify that a certain property holds on all reachable states. In general, the state set is represented by BDDs symbolically. The model checking with symbolically represented state sets and transition relations is called *symbolic* model checking.

3.3.2 BDD-based Symbolic Reachability Analysis

Apparently, when we consider certain properties about an FSM, we are only interested in reachable states. Whatever behavior the FSM has in non-reachable states does not matter. Indeed, we may arbitrarily change the output function and transition relations in non-reachable states. BDDs are used for reachability analysis for the following reasons:

1. A set of states can be represented by a monolithic BDD, so that we do not need to enumerate explicitly every single state to check for property violation.

2. The transition relationships (or characteristic functions) between successive states can also be modeled as BDDs either monolithically or in partitions. Therefore, the computation of next states from current states can be done with one step of computation. This is often called *image* computation.

3. The fix-point computation is very easy. Because BDDs are canonical, we can check in constant time whether the reachable states have saturated by comparing two BDDs at successive time steps.

BDD-based symbolic reachability analysis is widely used for model checking purposes. The circuit under verification can be modeled as a sequential circuit M with primary inputs $W = \{w_1, w_2, \ldots, w_n\}$, present state variables $X = \{x_1, x_2, \ldots, x_m\}$ and the corresponding next state variables $Y = \{y_1, y_2, \ldots, y_m\}$. Thus, M can be represented as $M = \langle T(X, Y, W), I(X) \rangle$, where $T(X, Y, W)$ is the transition relation and $I(X)$ is the set of initial states (more precisely the characteristic function of the set of initial states). M has a set of latches (state elements) $L = \{l_1, l_2, \ldots, l_m\}$. Thus, x_i and y_i would be the present state and next state variables corresponding to latch l_i. The transition relation T is a conjunction of the transition-relations of the individual latches. Thus,

$$T(X,\, Y,\, W) = \bigwedge_{i \in \{1 \ldots m\}} T_i(X,\, y_i,\, W) \tag{3.1}$$

Here, $T_i(X, y_i, W) = y_i \leftrightarrow \Delta_i(X, W)$ is the transition relation of latch l_i and $\Delta_i(X, W)$ is its transition function in terms of primary inputs and present state variables.

The core model checking operation is the *Image* computation, as defined in Equation 3.2. It computes the states reachable in one step, via the transition relation T, from the current states S_C.

$$\mathit{Img}(Y) \equiv \exists X, W.\ S_C(X) \wedge T(X, Y, W) \tag{3.2}$$

Whenever the BDDs representing Img(Y) is identical to the BDDs representing the S_C(X), the reachability analysis of FSM has reached fix-point, *i.e.*, the reachable states have been exhaustively traversed.

For the purpose of this dissertation we will only consider model checking of invariants *i.e.*, CTL properties of the form $\mathbf{AG}p$ where p is a Boolean expression on the variables

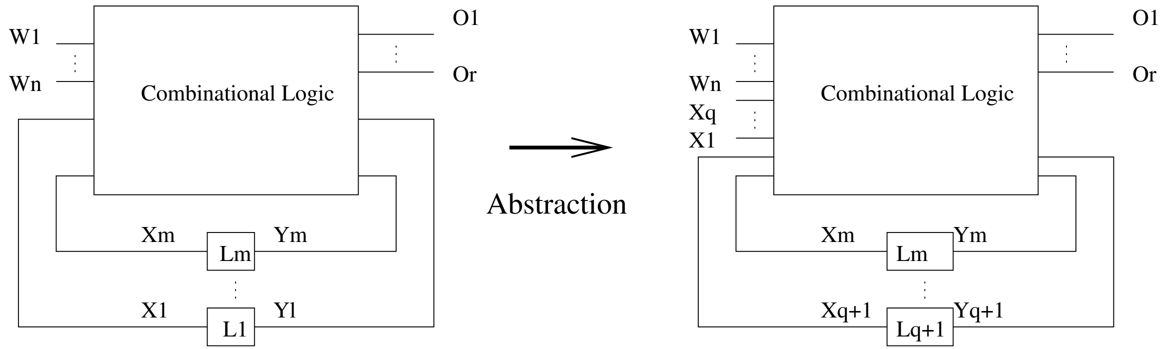


Figure 3.1: Conventional Latch Based Abstraction

of the given circuit model.

Given a subset of latches $L_{abs} = \{l_1, l_2, \ldots, l_q\}$, $L_{abs} \subseteq L$, that we would like to abstract away from the design, Figure 3.3.2 shows the most common way of doing abstraction by cutting open the feedback loop of latches L_{abs} at their present-state variables X_{abs}, *i.e.* making the variables X_{abs} primary inputs and removing the logic cones of the transition functions of latches L_{abs} from the circuit model. Functionally, the abstracted transition relation $\widehat{T}$ can be defined as

$$\widehat{T}(\widehat{X}, \widehat{Y}, \widehat{W}) = \bigwedge_{i:l_i \in \widehat{L}} \widehat{T_i}(\widehat{X}, y_i, \widehat{W}) \tag{3.3}$$

where $\widehat{W} = W \cup X_{abs}$, $\widehat{X} = X - X_{abs}$, $\widehat{Y} = \{y_i : x_i \in \widehat{X}\}$, and for all i such that $y_i \in \widehat{Y}$, $\widehat{T_i}(\widehat{X}, y_i, \widehat{W}) = T(X, y_i, W)$.

Apparently, this way of doing abstraction is very easy to implement. It can also be easily proved that the abstract circuit is an over-approximation of the original circuit,

in that whenever a property passes on the abstract circuit, it is guaranteed that the property passes on the original one as well. This approach is also called latch-based abstraction.

It is possible that the abstraction scheme not only work on the latches but also on the combinational logic [WHS04]. However, for the purpose of simplicity of this dissertation, we are focusing on the latch-based scheme only.

A concept that will be frequently used in the following treatment is that of the *proof of unsatisfiability (POU)* of a CNF SAT formula. As reported in [GN03, ZM03] modern SAT solvers such as `zchaff` [zch03] can be modified to produce a proof of unsatisfiability when the CNF formula being solved is found to be unsatisfiable. The proof of unsatisfiability (denoted by $\mathcal{P}$ in the sequel) of a SAT CNF formula can be represented as a directed acyclic graph, the nodes of which are clauses and each node (other than the leaves) has precisely two children. The root of this graph is the empty clause and the leaves are clauses from the original CNF. All other nodes (including the root) such that they can be derived through a resolution operation on their two children clauses.

The basic framework for abstraction refinement in our current implementation is similar to the one developed in [MA03] and [GGYA03]. A simplified version of the algorithm used in [MA03, GGYA03] is shown in Algorithm 3.3.1.

The abstraction is based on the use of SAT-based bounded model checking (BMC) [CBRZ01]. Let v be a variable in the representation of the transition relation T. A k-step BMC problem is generated by unrolling and replicating T, k times. Let $v^1, v^2, \ldots, v^k$ denote the k instantiations of v in the unrolled BMC problem. The idea of the abstraction method is to solve k-step SAT-BMC problem on the original

```
1:  k = InitValue
2:  if SAT-BMC(M, p, k) is SAT then
3:      return "found error trace"
4:  else
5:      Extract proof of unsatisfiability, P of SAT-BMC
6:      M' = ABSTRACT(M, P)
7:  end if
8:  if MODEL_CHECK(M', p) returns PASS then
9:      return "passing property"
10: else
11:     Increase bound k
12:     goto Step 2
13: end if
```

Algorithm 3.3.1: Abstraction Refinement Using SAT-BMC

design and analyze the POU returned by the SAT solver to generate the abstraction. The POU is scanned to identify a set of latches L_{abs} such that for each $l \in L_{abs}$ the variables $l^1, l^2, \ldots, l^k$ do not appear in any of the clauses of the POU. These latches can then be abstracted using Equation 3.3 (in the ABSTRACT operation in Step 6 of Algorithm 3.3.1). The rationale is that since these latches provably do not contribute to the property check in the first k time-frames, they might be irrelevant from the point of view of deciding this property for unbounded behaviors as well.

The model checking algorithm employed in Algorithm 3.3.1 (Step 8) may use a variety of methods. For simplicity of exposition we will assume a symbolic model checker using BDDs [McM93]. However, our ideas can be applied to other methods, such as SAT-based induction or BMC, in a fairly straightforward manner. Algorithm 3.3.2 shows the pseudo-code for a symbolic invariant checking algorithm that could use BDDs. Here B denotes the "bad states" *i.e.*, states that violate p and S_C denotes the set of currently reached states. Basically, the algorithm computes the set of states reachable from the initial states checking intersection of S_C with B after each step. The core operation is the image operation *Img*, as defined in Equation 3.2.

3.4 Dynamic Abstraction

In the following we develop our technique for dynamic abstraction which is based on an analysis of the proof of unsatisfiability of a SAT-BMC problem solved on the concrete design. As will be shown in the following, this approach can be seen as a generalization of the approach presented in [MA03, GGYA03].

Given the original circuit M and the property $P = \mathbf{AG}p$ let us assume that a SAT-

Invariant Check($M\langle T, I\rangle$, B)

1: $S_C = \emptyset$; $S_N = I$;
2: **while** $S_C \neq S_N$ **do**
3: $S_C = S_N$;
4: **if** $B \cap S_C \neq \emptyset$ **then**
5: **return** "found error trace";
6: **end if**
7: $S_N = S_C \cup Img(S_C)$;
8: **end while**
9: **return** "no bad state reachable";

Algorithm 3.3.2: Symbolic Invariant Checking

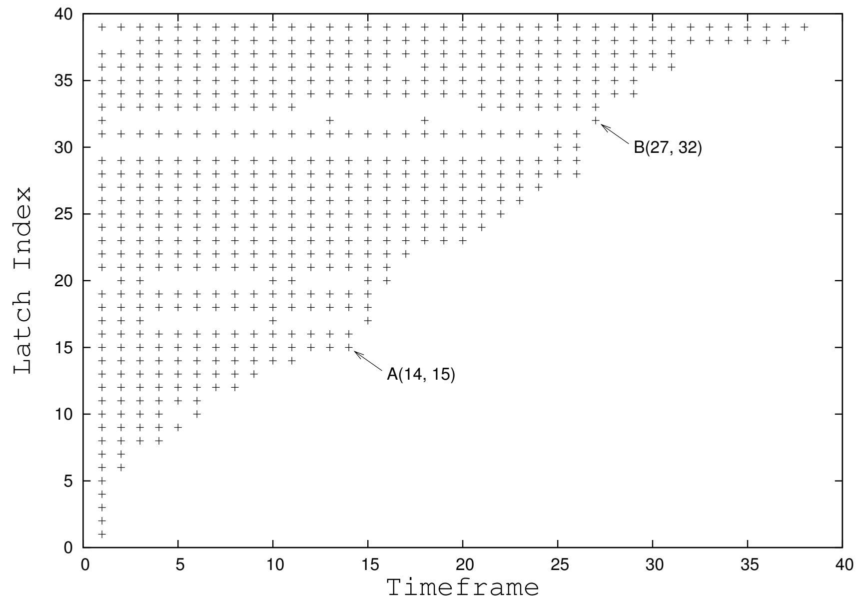


Figure 3.2: Latch-based Unsatisfiability Analysis

BMC problem of depth k has been solved on M and found no counter-example. Further, suppose that the SAT solver generates a proof of unsatisfiability $\mathcal{P}$ for this problem as described in [GN03, ZM03]. Figure 3.2 is a graphical representation of the POU from a 40-step SAT-BMC problem on a real circuit example. For each latch (plotted for 40 representative latches on the y-axis) the plot shows the time-frames for which the corresponding instantiation of the latch variable appeared in the POU of the SAT-BMC problem. The latches have been sorted on the y-axis for better readability of the data. Given a latch variable $l \in L$, we can define the *redundancy index*, $\rho(l)$ of l, with respect to the proof $\mathcal{P}$, as follows:

Definition 3.4.1 (Redundancy Index) *The redundancy index $\rho(l)$ of latch l with respect to the proof of unsatisfiability $\mathcal{P}$ is the smallest time-frame index such that for all time-frames j, $\rho(l) \leq j \leq k$, there does not exist a clause with variable l^j in $\mathcal{P}$.*

For example, in Figure 3.2 the points marked **A** and **B** show that latch number 15 has a redundancy index of 15 and latch 32 has a redundancy index of 28. Simply put, the redundancy index is the earliest time-frame after which the given latch stops participating in the POU of the *current* BMC problem. The situation depicted in Figure 3.2 is quite typical of a large variety of benchmarks we have experimented with. Most latches are not used in all time-frames of the POU. Moreover, there are several latches that are *only* used in the first few time-frames.

In the following we develop a result which shows that we can dynamically abstract away latch l from the transition relation, anytime after $\rho(l)$ image computation steps, in the invariant checking algorithm of Algorithm 3.3.2, and still have the model checking result match that of the BMC *up to* the first k steps.

At each step of image computation we can define a *candidate set* of latches which is essentially the set of latches whose redundancy index is less than or equal to the index of the current image computation step.

Definition 3.4.2 (Candidate set) *The candidate set of latches for iteration j of image computation in Algorithm 3.3.2 is denoted $\mathcal{C}_j$ and is defined as $\mathcal{C}_j = \{l_i : l_i \in L, \rho(l_i) \leq j\}$*

For example, in Figure 3.2 the candidate set at time-frame 15 consists of the first 15 latches, *i.e.*, $\mathcal{C}_{15} = \{l_1, l_2, \ldots, l_{15}\}$. A modified version of Algorithm 3.3.2, incorporating dynamic abstraction is given below.

In Algorithm 3.4.1 X_{abs} are the present state variables corresponding to the latches L_{abs} chosen for abstraction, ABSTRACT_TR abstracts the chosen latches from the transition relation, as per Equation 3.3 and the operation CHOOSE_ABSTRACTION_LATCHES is a heuristic to choose a subset of the *candidate latches* of the current iteration (Definition 3.4.2), for abstraction.

Since the dynamic abstraction is developed from an analysis of the POU of a k-step SAT-BMC problem the following result can be stated about the correctness of Algorithm 3.4.1

Theorem 3.4.1 *Algorithm 3.4.1 will not find a counter-example to the given property in the first k steps of image computation.*

Proof outline: The proof of this result is along the lines of the main result in [MA03]. It is based on the observation that the above dynamic abstraction implementation is

Invariant Check_DynamicAbstract ($M\langle T, I\rangle$, B)

```
1:  S_C = ∅; S_N = I;
2:  while S_C ≠ S_N do
3:     L_abs = CHOOSE_ABSTRACTION_LATCHES(L);
4:     T = ABSTRACT_TR(L_abs, T);
5:     B = ∃X_abs . B;
6:     S_C = S_N;
7:     S_C = ∃X_abs . S_C;
8:     if  B ∩ S_C ≠ ∅ then
9:        return "found error trace";
10:    end if
11:    S_N = S_C ∪ Img(S_C);
12: end while
13: return "no bad state reachable";
```

Algorithm 3.4.1: Symbolic Invariant Checking with Dynamic Abstraction

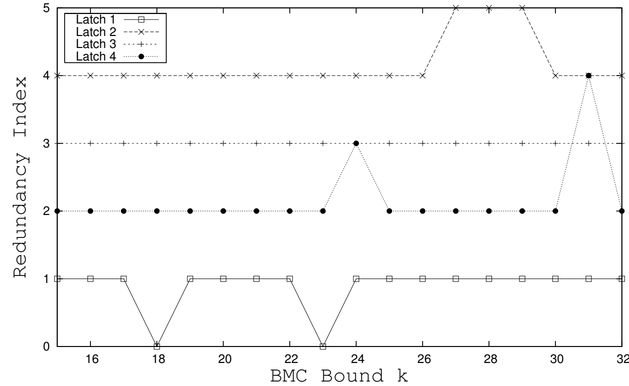


Figure 3.3: Redundancy Index as a function of BMC-depth

equivalently implemented as SAT-BMC by unrolling T for k time-frames and simply cutting open the the unrolled latches in the requisite time-frames that were abstracted by the dynamic abstraction algorithm during the image computation. The original POU $\mathcal{P}$ still applies to this abstracted BMC problem since the abstracted variables did not appear in it to begin with. ∎

Theorem 3.4.1 is a generalization of the main result of [MA03] which is also used in [GGYA03]. Indeed, if we restrict our abstraction to latches with redundancy index '0' this special case is exactly the abstraction algorithm of [MA03, GGYA03].

3.4.1 Latch Selection Heuristic

A key component of Algorithm 3.4.1 is the latch selection heuristic CHOOSE_ABSTRACTION_LATCHES. This decides which latches, out of the current candidate set, should be abstracted at a given image computation step. This heuristic can have a significant bearing on the overall performance of the algorithm.

The most aggressive approach would be to perform dynamic abstraction for *all* latches in the current candidate set and at the earliest possible time as indicated by the redundancy index (RI) of each latch. However, this approach suffers from several drawbacks. The following issues drive the choice of this heuristic:

1. **How often to abstract latches:** In the aggressive version of the heuristic described above, abstraction may need to be performed very frequently, in the worst case at every image computation step. Since the dynamic abstraction is implemented via quantification of next-state variables, from the transition relation (TR), the overhead can be significant. Thus, a good heuristic should limit the abstraction to a few image computation steps.

2. **Extrapolating unbounded behavior from $\rho(l)$:** In abstraction based on POU of a k-step SAT-BMC we are trying to extrapolate unbounded behavior of a latch, with respect to the given property, based on its bounded behavior. Intuitively, a latch that was active only in the first few steps of the BMC (*i.e.* has a small RI), *e.g.*, latch l_2 in Figure 3.2, is more likely to be inactive beyond k time-steps than one which was active up to $k-1$ or $k-2$ steps (*i.e.* has a large RI), *e.g.*, latch l_{38} in Figure 3.2. This intuition is supported by the data shown in Figure 3.3. The graph plots the redundancy indices for 4 latches from a design, generated from the POUs of different BMC problems with varying depth k. Apart from minor fluctuations, the RI values of each latch are remarkably consistent, despite having been derived from *independent* BMC runs of *different* depth. Of course this is not the case for every latch, an artifact of the intrinsic approximate nature of any abstraction refinement framework. However, interestingly enough, such consistency in the RI values is observed for

a large percentage of the latches actually abstracted by our algorithm.

3. **Size and depth of the reachable state space:** Abstraction of latches comes at the cost of enlarging the set of permissible behaviors of the circuit. This can potentially enlarge the reachable state space, result in larger BDDs for the reached state representation and/or increase the depth of the reachability computation. This factor should be considered by the latch selection heuristic if possible.

With the above criteria in mind, we have developed and tested several heuristics for CHOOSE_ABSTRACTION_LATCHES and found the following two to give a reasonable trade-off between overheads and abstraction power. Several other richer variants of these are possible and could be the subject of future research.

Heuristic 1: *Dynamically abstract just once at* $\lceil \delta \cdot k \rceil$ *time-steps, (where* $0 < \delta < 1$*), and abstract all latches in the candidate set at this point.*

The philosophy behind this heuristic to minimize the overheads of abstraction by doing it only once (issue 1 above) and being aggressive by choosing all candidates for abstraction. δ is kept fairly low to increase the likelihood of the latches being redundant for future image computations (in agreement with issue 2 above). We used $\delta = 0.2$ in our experiments but any other small value should be equally effective.

Heuristic 2: *Before the start of model checking analyze the proof* $\mathcal{P}$ *and gather a set of latches* $\mathcal{S} = \{l : l \in L, \rho(l) \leq \delta \cdot k\}$*. Every* r *steps of image computation, compute the set of latches* $\mathcal{N}$ not *in the support set of current reached state set BDD. If* $|\mathcal{S} \cap \mathcal{N}| \geq \tau$ *abstract all latches in the set* $\mathcal{S} \cap \mathcal{N}$*. Repeat every* r *image computation steps.*

The intuition behind using the set $\mathcal{N}$ is that the removal of such latches is less likely to cause a blow-up in the current step of image computation. This ties in with issue 3 discussed above. We used parameter settings of $\delta = 0.2$, $r = 2$, $\tau = 10$ for our experiments but the heuristic is not sensitive to these particular settings.

Qualitatively, *Heuristic 1* is based on an aggressive one time application of dynamic abstraction whereas *Heuristic 2* is a more conservative, and controlled application of dynamic abstraction. This distinction is borne out by the experiments discussed in Section 2.7.

3.4.2 Optimizations

In the following we describe some inexpensive optimizations that can be used to further improve the performance of a model checking algorithm that performs dynamic abstraction as described above.

Bypassing the Error-State Check: A simple corollary of Theorem 3.4.1 is the following.

Corollary 3.4.1 *The error state check $B \cap S_C \neq \emptyset$ (line 8) in Algorithm 3.4.1 will always yield* `false` *in the first k iterations of the algorithm.*

This simple result obviates the need to perform the error state intersection check (line 8 of Algorithm 3.4.1) in the first k iterations of image computation. This check can be fairly expensive at deeper image computation steps and/or when the target states are not simply the negation of the property but an enlarged target computed through a few pre-image computation steps. For example, in case of benchmark $P14$

in Table 3.2 the error-state check at depth 11 cost more than a 1600 secs. Also, note that this result is equally applicable to frameworks that only use static abstraction.

Cone of Influence Reduction: As proposed in [MA03], an abstraction of some latches can create opportunities for further abstraction by applying the standard *cone of influence (COI)* reduction on the abstracted model. While the optimization was proposed in the context of their static abstraction procedure the same can be done after each abstraction step in a dynamic abstraction algorithm, such as ours. The key point is that any subsequent abstraction due to the COI reduction *does not* increase the space of allowable behaviors of the design. Thus, the quality of the abstraction is not diminished in any way but the design becomes smaller and more tractable for the model checking.

3.4.3 Handling Counter-examples on Abstract Model

Our current implementation uses the counter-example independent refinement. Whenever a counter-example is found in the abstract model as reported in line 9 of Algorithm 3.4.1, a SAT-BMC is used to check if it is a true counter-example on the concrete model. If it is not a true counter-example, then we repeat the abstraction process with a deeper unrolling for SAT-BMC as shown in Algorithm 3.3.1, which serves as the refinement scheme of current implementation. Otherwise, an error trace is returned to the user. Note that the effectiveness of proposed dynamic abstraction is unaffected by the underlying refinement scheme.

3.5 Experimental Results

We have implemented the proposed dynamic abstraction algorithm as well as the static abstraction algorithm of [MA03, GGYA03] in C++ within the VIS framework [The96]. The POU extraction is based on the algorithm of [ZM03] and it has been extended to report the redundancy index (RI) for each latch. The downstream model checker has been modified to take RIs of latches as inputs and can further abstract the statically abstracted model on the fly using proposed latch selection heuristics. We use CUDD for the BDD-based computation, and ZCHAFF [zch03] as the SAT solver for BMC. The image computation uses the IWLS95 method [RAP+95]. The *sift* method of dynamic variable reordering is used for all BDD operations, and the image cluster size is set to 5000.

We have tested our tool for safety properties on different modules from three real-life industrial designs, which contain 10463, 40055, and 16414 lines of RTL verilog code respectively. A commercial front-end parser was used to synthesize the verilog code and to extract the cone of influence (COI) for each property. All experiments were run on 1.5 GHz Pentium 4 machines with 1G RAM, running SuSe Linux. The problem characteristics are reported in Table 3.1. The time-out limit is set to 24 hours for all problems. The results are reported in the Table 3.2. The second column shows if the property (except P14) is a passing property or the length of the shortest counterexample if it is a failing property. A question mark was shown for P14, since none of the methods can finish within 24 hours. Column 3 shows the number of latches in the statically abstracted model. Column 4 is the cumulative CPU time, which includes both abstraction time and model checking time (including BMC). Columns 5 and 7 report the number of latches in the final dynamically abstracted

Problem	Concrete Model		
	# PIs	# FFs	# Gates
P1	330	1158	5155
P2	401	1896	8910
P3	405	1951	8577
P4	671	2735	11381
P5	1015	2971	10044
P6	1020	3039	10060
P7	1981	5407	18193
P8	1950	5468	19161
P9	1943	5644	19189
P10	3490	8998	27297
P11	308	746	3837
P12	289	654	4823
P13	289	654	4826
P14	356	1644	7408

Table 3.1: Problem Complexity on Concrete Model

Prob.	Pass /Fail	Static Abstraction		Dynamic Heuristic I		Dynamic Heuristic II	
		# FFs	Time(s)	# FFs (diff.)	Time(s)	# FFs (diff.)	Time(s)
P1	Pass	264	133	204 (-60)	**97**	221 (-43)	114
P2	Pass	257	936	218 (-39)	**171**	240 (-17)	550
P3	Pass	266	187	243 (-23)	311	252 (-14)	**172**
P4	Pass	271	**68**	211 (-60)	145	249 (-22)	**68**
P5	Pass	286	474	216 (-70)	**258**	276 (-10)	441
P6	Pass	277	360	224 (-53)	**90**	254 (-23)	141
P7	Pass	245	322	222 (-23)	390	227 (-18)	**219**
P8	Pass	224	1198	184 (-40)	**305**	201 (-23)	769
P9	Pass	268	89	234 (-34)	**77**	247 (-21)	79
P10	Pass	293	563	221 (-72)	**85**	269 (-24)	270
P11	60	123	687	81 (-42)	**658**	87 (-36)	690
P12	36	170	30704	160 (-10)	**1880**	170 (0)	30632
P13	29	201	>24h (27)	168 (-33)	**23781**	196 (-5)	>24h (27)
P14	?	115	>24h (13)	93 (-22)	>24h (**26**)	109 (-6)	>24h (13)

Table 3.2: Results: Static Abstraction and Proposed Dynamic Abstraction

model for Heuristics 1 and 2 respectively (Section 3.4.1) as well as the additional latches abstracted with respect to the static abstraction method. For example, the static abstraction is able to abstract a model with only 224 latches for property P8, for which the concrete model has 5468 latches. Dynamic heuristic 1 is able to further abstract 40 more latches away, and the final model has 184 latches. Similarly, dynamic heuristic 2 is able to abstract 23 more latches, and the final size of the model is 201. Again, the reported time in column 6 and 8 are cumulative CPU times. For all time-out cases, we also report the number of completed image computation steps. For example, for P14, dynamic heuristic 1 is able to compute 26 steps of images within 24 hours (without hitting bad states nor reaching fix-point), while static abstraction and heuristic 2 can only finish 13 steps. Therefore, a better partial result has been achieved by the dynamic approach 1 over static approach and heuristic 2.

We can see that proposed heuristic 1 is extremely powerful in reducing the overall runtime, even though sometimes the number of additional abstracted latches is not too significant. For example, with only 10 additional latches abstracted away for P12, it achieves over an order of magnitude speed-up compared to the pure static abstraction approach. Also for P8, our approach is able to reduce the overall time from 1198 seconds down to 305 seconds. However, as discussed in previous section, aggressive abstraction may potentially slow down the subsequent model checking. In our experiment, dynamic heuristic 1 does experience occasional slow-down. For example, it spent 145 seconds on P4, in which it took 174 steps of image computation to reach fix-point. On the other hand, both static abstraction and dynamic heuristic 2 need only 51 steps of image computation to reach fixed-point in 68 seconds. As explained in the previous section, heuristic 2 is a more conservative and controlled application of dynamic abstraction. It consistently outperforms the pure static abstraction for

Problem	Static		Dynamic I		Dynamic II	
	Peak	Final	Peak	Final	Peak	Final
P2	253K	57K	77K	18K	158K	49K
P4	61K	650	81K	13K	54K	763
P8	331K	22K	183K	13K	252K	14K
P11	132K	10K	117K	8160	130K	11K
P12	4360K	628K	852K	120K	4360K	628K

Table 3.3: BDD Sizes of Different Abstraction Methods

all cases and has achieved significant speed-up on many of them. For example, it reduces the runtime for P10 from 563 seconds down to 270 seconds.

Table 3.3 shows more detailed statistics of the BDD-based model checking process for some representative benchmarks from Table 3.2. The columns represent the peak and final BDD sizes of the model checking, for the each of the three methods. The results show that the two dynamic abstraction methods are able to significantly reduce the BDD sizes which is consistent with the overall runtime gains reported in Table 3.2. For example, for P11 dynamic 1 is able to reduce the number of peak nodes by over an order of magnitude and the number of final nodes by more than 5 times.

3.6 Summary

In this chapter we have presented a method for performing dynamic abstraction within a framework for abstraction-refinement based model checking. The dynamic

abstraction is applied during successive image computation steps of the model checking algorithm and can be applied after and in addition to the abstraction performed by traditional static abstraction methods. It is facilitated by information gathered from an analysis of the proof of unsatisfiability of SAT-based bounded model checking problems, solved on the concrete model, and passed to the model checker. It effectively allows the model checker to work with smaller abstract models. We have proposed two strategies for realizing the dynamic abstraction. Our experiments on several large industrial benchmarks demonstrate that the proposed dynamic abstraction strategies improves the performance of a model checking flow based on abstraction refinement by up to an order of magnitude over what can be achieved with state-of-the-art static abstraction refinement methods and and also enables the successful application of abstraction refinement based model checking to larger designs.

Although the presented framework is a complete method for abstraction refinement based model checking the concept of dynamic abstraction goes beyond the specific implementation presented in this chapter. There are several avenues for furthering this research including development of better and more sophisticated latch selection strategies, based on the treatment presented in Section 3.4.1. Another avenue would be to apply the notion of dynamic abstraction to other abstraction refinement frameworks incorporating a variety of model checking techniques and other heuristics for static abstraction refinement.

Chapter 4

Design Validation at Register Transfer Level via ATPG Technologies

We present a framework for high-level design validation using an efficient register-transfer level (RTL) automatic test pattern generator (ATPG). The RTL ATPG generates the test environment for each validation target, which includes variable assignments, conditional statements, and arithmetic expressions in the HDL description. A test environment is a set of conditions that allow for full controllability and observability of the validation target. After the ATPG terminates, the back-end translator intelligently translates the test environments into the validation vectors by filling in the necessary values. Since the observability of error effect is naturally handled by our ATPG algorithm, our approach is superior to existing validation methods, which only focus on the excitation of HDL descriptions. We propose a set of heuristics to

utilize the high level circuit information to enhance RTL ATPG and to maximize the validation efficiency. The enhanced RTL ATPG is able to generate high quality validation vectors in very low CPU time compared to previous methods. The experimental results on ITC99 benchmark circuits and an industrial circuit demonstrate that very high design error coverage can be obtained in a small amount of CPU time.

4.1 Introduction

Despite of the advances of formal verification methods, such as equivalence checking, model checking, theorem proving, and etc., in recent years, simulation remains the primary approach for design validation, especially at the high abstraction levels of the circuit or when the design under validation is very complex.

The keys to simulation-based validation approach are the metrics for determining the validation coverage and the algorithms for generating validation vectors. The most common coverage metrics are statement, branch and condition coverage adopted from software testing [Bei90]. The vector generation algorithms can be broadly categorized into deterministic method, simulation-based generation method or the combination of the two. In [CRS+00b], the authors proposed a simulation-based ATPG approach for design validation, in which the input VHDL code is first instrumented so that the execution trace can be captured. The instrumented VHDL is then simulated using a commercial simulator. The underlying procedure interacts with the simulator via the trace files and tries to generate vectors to maximize the statement coverage. The approach was applied to an industrial circuit and was able to improve the coverage from designer-provided test vectors. Another algorithm [HR96] proposes a mutant

analysis and tries to generate vectors to capture every injected mutant. The authors in [PNS00] propose the analysis of paths for HDL code. Each path starts from a variable definition and ends at the usage of that variable. The constraints are formed from the path and provided to constraint solver to generate each validation test. The authors in [CCRS03] proposed a genetic algorithm (GA) based approach to automatically generate test programs for microprocessor cores. A gate-level fault simulator is used to evaluate the fitnesses of generated test programs and to guide the GA process. The gate-level implementation must be available to apply this approach. In [FAD99], the authors use integer linear programming and boolean satisfiability methods to generate vectors that can achieve high tag (an observability-enhanced statement) coverage [FDK01]. The tag coverage is a better coverage metric than traditional software coverages in that the propagation of error effects to an observation point is considered. In [GF01], authors proposed a deterministic RTL-ATPG algorithm, which is able to efficiently generate logic-level stuck-at tests from a RTL HDL description. The algorithm utilizes a set of 9-valued algebra to perform symbolic justification and propagation to derive the test environment. Once a test environment is obtained, the precomputed test vectors can be plugged into the test environment to derive the complete test vectors.

In [GR03], the authors established through an extensive empirical study that a good logic-level stuck-at test set is also an excellent RTL validation vector set. The authors also proposed the enhancements over [GF01] such as replacing the test set library with universal test set for arithmetic modules, and using the combination of gate-level and RTL ATPG tool. The gate-level ATPG is used to derive the stuck-at test sets for random logic, and the RTL ATPG is used to justify and propagate them.

The constrained functional verification (CFV) [YAPA04, YPAA03, YSP+99] focuses on the generation of random vectors based on the input biases, and the set of constraints imposed on the state variable and inputs. Generally speaking, the CFV is much more effective than the pure random simulation in reaching more functional state space in a short amount of time. The CFV can be easily applied to complex designs and high level designs. It is also suitable for verifying the blocks within a complete system environment. However, it requires the user to provide the constraints and input bias, which is generally a non-trivial task itself. Additionally, the completeness of the verification is not guaranteed even with a large number constraints. Shimizu *et al* [SD02] proposed a unified framework of CFV targeting the interface protocols. An important application of CFV is for compositional verification [McM97] , which performs the system level verification by separately verifying each of the blocks with certain constraints.

Our work focuses on the validation of RTL design models. We are assuming that the RTL implementation of the design is available and any combination of input values is legal. Our validation vector generation algorithm is an intelligent combination of a deterministic engine and a GA-based engine. The deterministic engine is derived from [GF01]. While the RTL ATPG algorithm was originally targeted for manufacturing tests, we have discovered that with some modifications, the ATPG algorithm can be extended to generate validation vectors efficiently. The original ATPG targets every construct that is synthesized to the structural RTL netlist. However, for design validation purposes, we only need to target constructs that directly map to variable assignment, arithmetic expressions, and conditional constructs at the behavioral level. We also added the technique of test environment relaxation to the ATPG, which is able to help the validation of additional portions of the design that may be hard

to test. In order to more effectively handle control-intensive circuits, we extended the 9-valued algebra of [GF01] to 10-valued algebra. We also propose several high-level heuristics to efficiently use high-level circuit information, such as finite state machine (FSM) information, implication-based learning, and embedded counters, to guide the ATPG search process [1]. For the cases, which deterministic engine breaks down, our GA-based engine is invoked to generate the validation vectors to maximize the tag coverage. If there are any partially justified test environments produced by the deterministic engine, our GA-based engine will automatically use them as the starting GA population.

Our approach is very different from the previous approaches. Most of the previous techniques do not consider the observability issues; instead, they focus on maximizing the excitation of the potential error sites. Compared to the approach described in [FAD99], which targets the tag coverage, our test generation method improves upon the tag simulation process and is able to handle much larger designs. Lastly, the back-end test environment translation has been enhanced with new techniques to maximize the design error detection. As the result, the generated vectors are more compact and more powerful for design validation.

Note that while automatically deriving validation vectors at the RTL, we are validating on the implementation circuit, which may be a buggy circuit. Nevertheless, this fact can add advantages to our work in the following scenarios:
(i) If a golden RTL model is present, then from the model the test vectors may be derived and used to compare output responses from an implementation.
(ii) The test vectors obtained from an implementation can be applied to its executable

[1]The preliminary result was published in [ZGH03]

behavioral specification (if available) and will produce different outputs when the bugs are excited and propagated in the implementation. (We assume this scenario in this framework)
(iii) The vectors may be used to validate the logic-level circuit derived from an RTL description if formal equivalence checking fails. Formal equivalence checking can fail if the RTL description and the gate level implementation are structurally quite different. Thus, automatically generated validation test benches at the RTL can aid the verification process to a large extent.

The rest of chapter is organized as follows: Section 4.2 discusses the preliminaries of our work. Section 4.3 discusses the overview of our design validation framework. Section 4.4 describes our validation vector generation algorithm as well as the details of new enhancement heuristics. Section 4.5 reports the experimental results, and finally Section 4.6 summarizes the chapter.

4.2 Preliminaries

4.2.1 Assignment Decision Diagrams

First introduced in [CGR93] for high-level synthesis, an assignment decision diagram (ADD) is a compact representation for functional RTL circuits. ADDs can be viewed as a set of control switches, which select the proper inputs to feed target nodes. ADDs also provide us a structural view of the circuit under test (CUT), which makes ADD a good candidate representation for testing purposes. Figure 4.1 shows a segment of VHDL code, and part of the associated ADD. The VHDL code contains a counter, and

only the part of ADD associated with the counter is shown. The triangular-shaped symbol is an Assignment Decision Node (ADN). The binary control signals (CTL1 and CTL2) feeding into the ADN are mutually exclusive. In this case, depending on the values of CTL1 and CTL2 signals, the ADN will select the corresponding value to be routed to the output. The other circular nodes are simply arithmetic or logical operators that comprise the computing paths in the circuit.

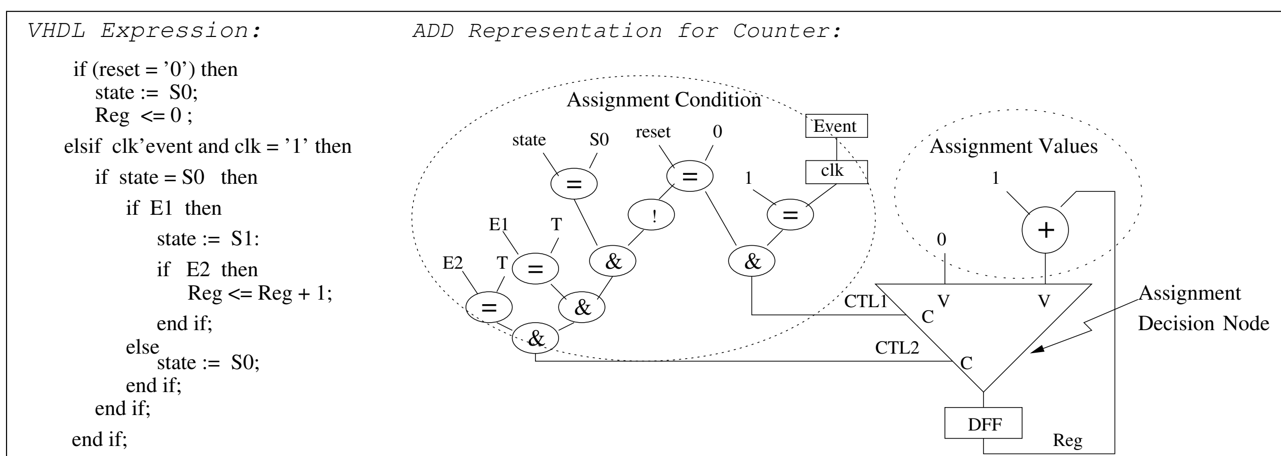


Figure 4.1: Assignment Decision Diagram

4.2.2 RTL Algebra Based Testing Approach

In [GF01], the authors defined following 9 symbols for RTL justification and propagation purposes.

- **Cg** (general controllability) is the ability to control its value to arbitrary value.

- **C0** (controllability to zero) is the ability to control the variable to the value 0.

- **C1** (controllability to one) is the ability to control the variable t0 1; i.e., ”000...01”.
- **Ca1** (controllability to all ones) is the ability to control the variable to all ones; i.e., ”111...11”.
- **Cq** (controllability to a constant) is the ability to control the variable to any fixed constant.
- **Cz** (controllability to the Z value) is the ability to control the variable to high-impedance Z.
- **Cs** (controllability to a state) is the ability to control the state variable to a particular state.
- **O** (observability) is the ability to observe a fault at a variable.
- **O’** (complement observability) is defined for single-bit variables only. It signifies the zero/one fault.

During the testing of each RTL primitive, symbolic RTL justification and propagation are performed using a branch-and-bound search procedure to trace out the path, which starts from the primary inputs (PIs), traverses through the target site, and continues to one or more primary outputs (POs). The obtained symbolic path, which may span across multiple clock cycles, is termed a *test environment*. Then each pre-computed test vector from the test set library for the MUT can be readily plugged into the test environment to obtain the system-level test set for the RTL module.

c = a + b	b	b - Δ	b + Δ	b + ?
a	a + b	a + b - Δ	a + b + Δ	a + b + ?
a - Δ	a + b - Δ	a + b - Δ	a + b + ?	a + b + ?
a + Δ	a + b + Δ	a + b + ?	a + b + Δ	a + b + ?
a + ?	a + b + ?	a + b + ?	a + b + ?	a + b + ?

Table 4.1: Δ calculus for an adder

4.2.3 Validation Metrics

In order to measure the thoroughness of the validation, we adopted into our framework the Observability-Based Code Coverage (OBCC or tag coverage)[FDK01]. The OBCC is superior to classical software testing coverages (such as statement and branch coverage) in that it incorporates observability as well as controllability information into the simulation. The basic strategy of OBCC is the efficient computation of tag coverage. The tag is introduced in OBCC as a mechanism to extend standard coverage metrics so that the observability can be computed. A tag is defined as a symbol placed at a given location, which can be used to represent the presence of an incorrect value. First, the tags are injected at variable assignment statements and branch conditions, then the tags are propagated based on a set of calculus rules for supported primitives.

Table 4.1 shows the tag calculus for an adder, $c = a+b$. The $+\Delta$ represents a positive tag, while -Δ denotes a negative one. The question mark ? means that tag may be killed as the result of operation during execution. A tag is declared to be observed at a PO only when either $+\Delta$ or -Δ has been successfully propagated to at least one PO.

c = a + b	b	b - Δ	b + Δ	b + U	b + ?
a	a + b	a + b - Δ	a + b + Δ	a + b + U	a + b + U
a - Δ	a + b - Δ	a + b - Δ	a + b + U	a + b + U	a + b + U
a + Δ	a + b + Δ	a + b + U	a + b + Δ	a + b + U	a + b + U
a + U	a + b + U	a + b + U	a + b + U	a + b + U	a + b + U
a + ?	a + b + ?	a + b + U	a + b + U	a + b + U	a + b + ?

Table 4.2: Modified Error Propagation Calculus for an Adder

4.2.4 Modification of OBCC transformation Tables

In order to have a more realistic analysis of validation coverage, especially for arithmetic modules, we added a new symbol of U to represent the error with an unknown sign. All OBCC tag propagation transformation tables have to be modified to accommodate the new calculus. We are assuming the aliasing probability to be 0. We will show later that we have incorporated in our framework the techniques to reduce the aliasing probability. Table 4.2 shows the modified propagation table for the adder. Tables for other modules can be easily derived. We can see that our new error propagation is more optimistic than the original OBCC scheme. For example, when two errors with opposite signs are propagated to operand a and b, with original OBCC scheme, only a ? is placed on the output c. Our experiences show that almost all ? symbols were easily blocked (non-propagatable) before reaching any observable points. This results in the over conservativeness of OBCC when applied to circuits with arithmetic modules. On the other hand, our modified scheme provides a more realistic and accurate analysis.

4.3 Design Validation Framework

Our design validation algorithm follows the similar approach of [GF01]. Figure 4.2 shows the overview of our design validation framework. First, the input HDL description (either VHDL or Verilog) is compiled into an internal structural RTL representation based on Assignment Decision Diagrams (ADD) [CGR93]. Then, a ATPG-based procedure generates the validation vectors for the design. The HDL test bench wrapper is generated to facilitate the simulation. Next, a commercial HDL simulator is invoked to simulate the validation vectors and to measure the traditional coverages. The tag coverage is calculated by an in-house simulator. Finally, the simulation responses are captured and compared against the responses from the specification by a checker program.

4.4 ATPG-based Validation Vector Generation Algorithm

The core of our validation framework is an efficiently RTL ATPG, which includes a deterministic engine and a simulation-based generation. The RTL ATPG directly generates validation vectors for a given RTL HDL description.

The deterministic part of our RTL ATPG is based on the work of [GF01], which uses a set of 9-valued algebra, as shown in section 4.2, to represent the controllability and observability objectives for a given test target, and uses a branch and bound deterministic algorithm to justify these objectives symbolically. To better suit our validation purpose, we extend the the algebra with a new symbol **Cp** and value range

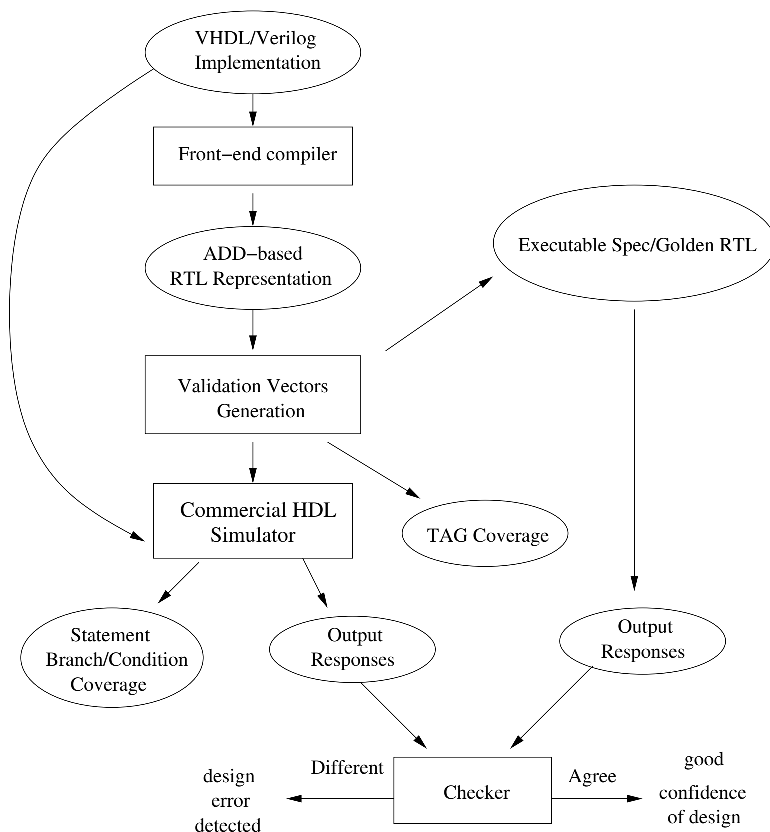


Figure 4.2: Overview of the design validation flow

analysis capacity. The symbolic justification the propagation is now conducted via a branch and bound algorithm using 10-valued RTL algebra.

Figure 4.3 shows the deterministic validation vector generation flow. First, a preprocessor builds a validation target list for the circuit, which includes all condition, arithmetic, and assignment constructs. Next, the ATPG iterates through the list and generates the test environment for each target. If the test environment cannot be obtained for a given validation target, the ATPG tries to generate the relaxed

test environment for it. After the test environment generation stops, the back-end translator is invoked to generate the validation vectors from the test environments.

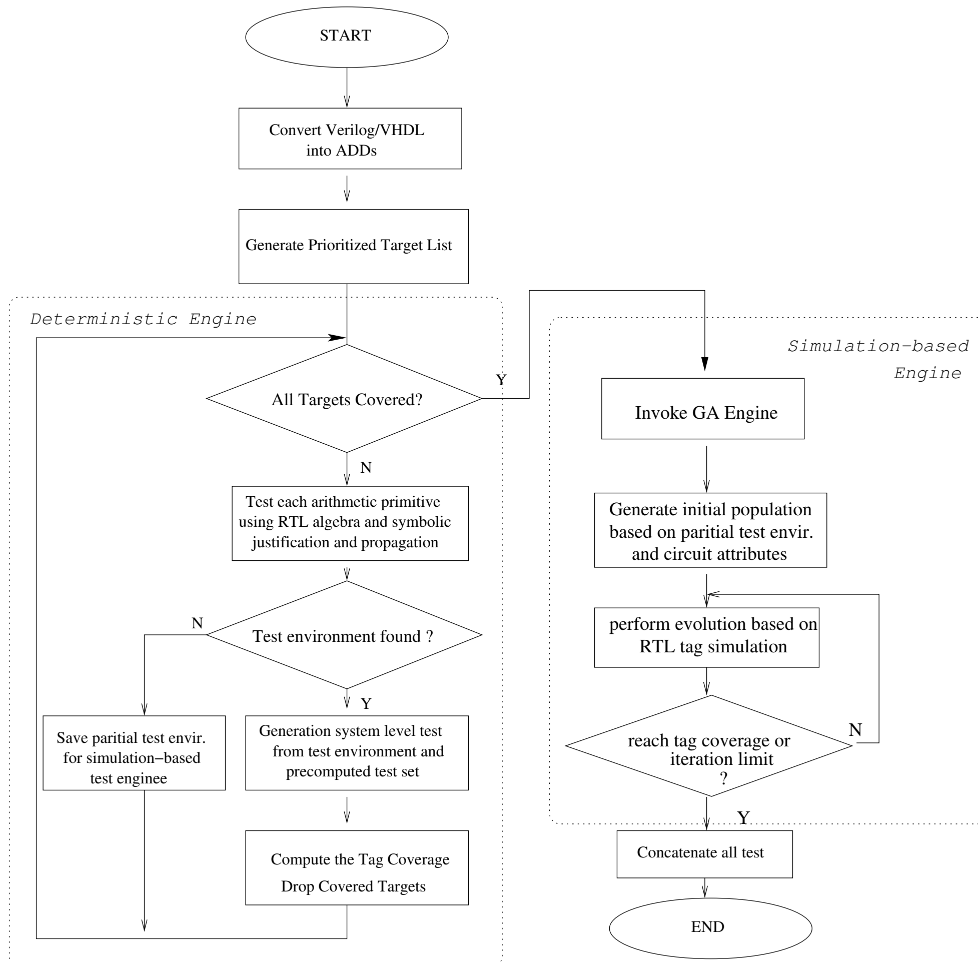


Figure 4.3: Validation Vector Generation Flow

4.4.1 Test Environment Generation

The test environment is a set of conditions that allow controllability and observability of the validation target. Each test environment can be viewed as a symbolic path which starts from the PIs, traversing through the target site, and reaches at one or more POs or observable variables. The test environment generation process is essentially searching for a sufficient symbolic path, through which the excitation objectives can be delivered to the target site, and error effect can be propagated to the PO.

The algorithm 4.4.1 is the pseudo code of our deterministic test environment generation.

Test_Environment_Generation(T)

1: S_{path} = Set of symbolic propagation paths from PI to PO through T;
2: **while** $S_{path} \neq \emptyset$ **do**
3:    p = CHOOSE_A_PATH_FROM(S_{path});
4:    S_c = EXCITATION_OBJECTIVES(T) $\cup$ PROPAGATION_CONSTRAINTS(p);
5:    **if** JustifyBranchBound(S_c) = TRUE **then**
6:       **return** "test environment found";
7:    **end if**
8:    S_{path} = S_{path} - p;
9: **end while**
10: **return** "No test environment found";

Algorithm 4.4.1: Test Environment Generation for a given validation target T

Consider the VHDL description and its structural RTL in Figure 4.4 as an example to illustrate the Algorithm 4.4.1. Assume that RST, ina and inb are PIs, and out is

the only PO for the circuit. Suppose that the multiplier M3 is our current validation target. Our algorithm executes as follows:

```
If RST = '1' then
      a := 1;
      b := 1;
      c := 0;
      i  := 0;
      out  := 0;
elsif clk'event and clk = '1' then
      i := i + 1;
      if ( i=1) then
            a := ina + 3;
            b := inb;
      elsif (i=2) then
            c := a * b;
      elsif (i=3) then
            out := c;
      endif;
endif;
```

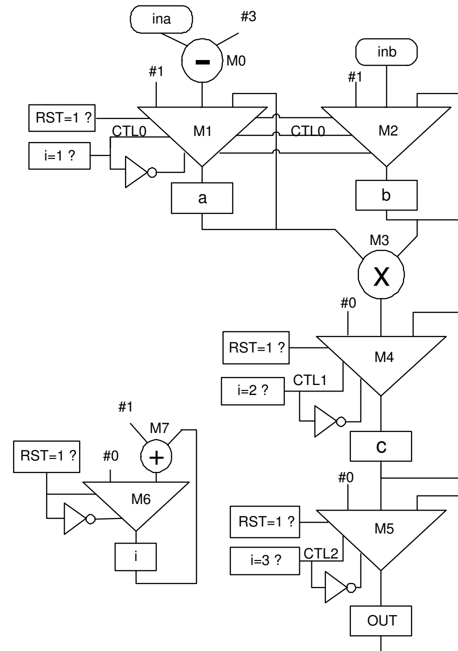


Figure 4.4: Sample VHDL Code and Corresponding Structural RTL

1. Find the shortest propagation path $M3 \rightarrow M4 \rightarrow C \rightarrow M5 \rightarrow OUT$.
2. Propagation constraints of (CTL1,1,C1) and (CTL2,2,C1) are injected. The first objective means a C1 algebra at time frame 1 is needed on signal CTL1.
3. Excitation objectives of (a,0,Cg) and (b,0,Cg) are injected at the inputs of M1. The objectives mean algebra Cg is needed on both signal a and b at time frame 0.
4. Justify all objectives via a branch-and-bound search.

5. All objectives are justified. The test environment is generated as shown below. Note that the test environment contains all justified algebra on the PIs and Cgs on internal nodes. The fan-in nodes are also included in test environment for the internal nodes.

The generated test environment can be visualized as in Figure 4.5.

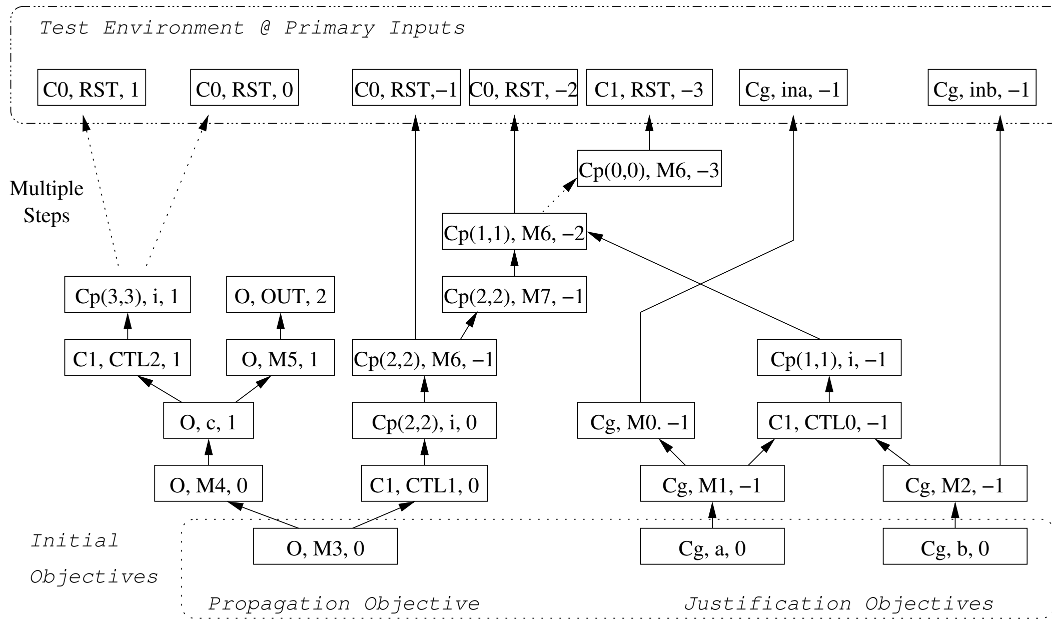


Figure 4.5: Generated test environment

Next subsection will show how to plug precomputed validation vectors into test environment to form the validation vectors.

4.4.2 Test Environment Translation

A generated test environment must be translated into validation vector(s) to be applicable to the design.

Time	STEP1			STEP2			STEP3		
Frame	RST	ina	inb	RST	ina	inb	RST	ina	inb
-2	1	x	x	1	x	x	1	2	3
-1	0	x	x	0	8	10	0	8	10
0	0	x	x	0	x	x	0	1	0
1	0	x	x	0	x	x	0	4	7
2	0	x	x	0	x	x	0	5	5

Table 4.3: Test Environment translation

Table 4.3 highlights partial results as the procedure proceeds. Suppose we need to apply 5 and 10 at the two inputs of the multiplier. First, all algebra except the C_g at PIs are translated. For the above test environment, only the value of RST is determined at this step, as shown from the columns under the heading "STEP 1". Secondly, the value 5 is plugged into (a,0), then following the trace $(a, 0, Cg) \rightarrow (M1, 0, Cg) \rightarrow (M0, -1, Cg) \rightarrow (M0, -1, Cg) \rightarrow (ina, -1, Cg)$ the value can be propagation backward to the PI. The value may need to be adjusted when propagated through certain types of RTL constructs. For example, the value 5 is propagated from (a,0) through (M0,-1) without any adjustment. However the 5 on (M0,-1) implies the 8 at (ina,-1), since M0 is a subtracter, and the other operand is constant 3. Similarly the 10 can be plugged in (b,0,Cg) and value can be propagated to the (inb,-1). The translation results are recorded in columns under "STEP 2". Finally, all unspecified

PIs are filled with the random numbers to form the fully specified test vectors. The last three columns show the final test vectors.

4.4.3 Test Environment Relaxation

If the complete test environment cannot be derived using the Algorithm 4.4.1, the ATPG relaxes the controllability condition and repeats the algorithm (no excitation objectives are added in step 4) to generate a relaxed test environment. The benefit of test environment relaxation can be illustrated by the following example.

1. $c := a + b$;

2. $c := a + d$;

3. $c := a - b$;

Suppose statement 1 is the correct implementation, while statements 2 and 3 are erroneous versions. With complete test environment for statement 1, we can fully control the values of a and b. In other words, by enforcing the values of a and b to be different from any other signals/variables, the erroneous values computed for c in statements 2 and 3 can be guaranteed to be different from correct one. The errors will be propagated to PO by the test environment and the design error will be captured.

Now suppose, the ATPG cannot find a complete test environment for statement 1, it produces a relaxed version instead. With this relaxed test environment, even though we do not have full controllability over the operand a the detection of statement 3 can still be guaranteed by controlling the value of b. However, the relaxed test environment should not always replace the complete test environment, since the relaxed

test environment is not as powerful as the complete one in terms of error detection. For example, the it cannot detect the substitution error of statement 2 if the values of b and d happen to be same.

4.4.4 Augmenting the Algebra with Value Range

As demonstrated in the previous example, once we generate a test environment, we can easily deliver test vectors to the MUT. However, the inadequacy of the original 9-valued algebra may hinder the test environment generation process. Figure 4.6

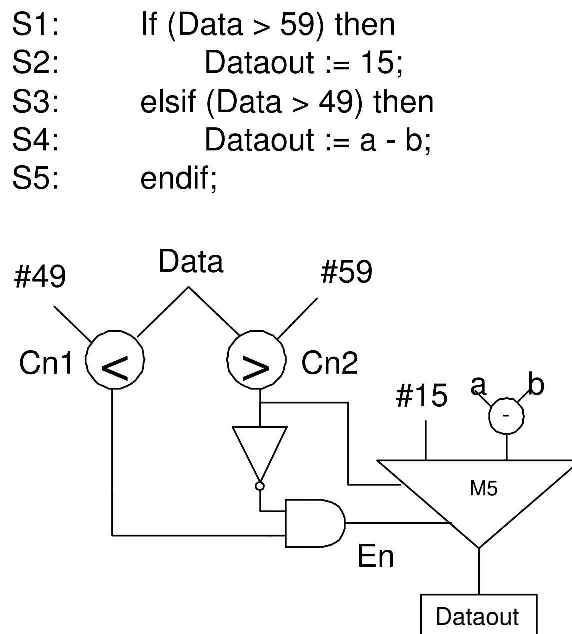


Figure 4.6: Augmenting the algebra with value range

shows a segment of VHDL code and its corresponding ADD representation. Suppose signal a is on the propagation path of test environment. Then the signal *En* must be

asserted to C1 to select the output of subtracter to *Dataout*. To justify the C1 on *En* , the *Cn1* needs to be C1, which leads to the the general controllability of Cg on signal *Data*. At the same time, the *Cn2* needs to be C0, which requires another Cg on the *Data*. Since no signal can be arbitrarily set to two different values simultaneously, two Cgs lead to the unresolvable conflict. With the original 9-valued algebra, there is no other way to justify the propagation conditions. As a result, the ATPG fails to generate corresponding test environment and subsequently aborts. However, we can see that (En,C1) is actually justifiable. With any value between 50 and 59 of *Data*, the *En* is justified to C1.

The reason that the previous ATPG does not find the solution is that in justifying the (Cn1, C1), *Cg* on *Data* is too strong of a condition. However, with the original 9-valued algebra, *Cg* is the only possible value.

To overcome this inadequacy, we introduce a new symbol **Cp** with a value range to the RTL algebra set. A justified Cp together with a specified value range, say [30;0], on a signal X means that we can control the X to be a value between 30 and 0. But we do not have to control which particular value X should be.

In the above example, to justify the (Cn1, C1), we only need to justify (Data, Cp[MAX;50]), since we do not care the exact value of Data as long as it is greater or equal to 50. (MAX is the largest number representable by Data.) Similarly, the (Data, Cp[59;0]) satisfies the (Cn2,C0). Therefore, we are able to relax the previously strong condition to a much weaker yet sufficient condition. To simultaneously satisfy both Cps, the combined justification objective becomes (Data, Cp[59;50]).

We can see that the resultant justification objectives are exactly the conditions expressed at VHDL statement *S3* and *S1*. Note that two Cps on the same signal are

not compatible if and only if the two Cps have non-overlapping value ranges.

The augmentation of the new symbol and the related value range analysis increase the computation overhead; however, they are indispensable in modeling the justification objective more accurately. As a result, they are used sparingly only when the initial algebra fails to generate a test environment.

4.4.5 Techniques to Maximize the Error Detection

With random filling of the unspecified PIs, as discussed in the previous section, the values set for b may not uncover the error if it is accidentally set equal to the value for d. To remedy this problem, while translating the test environment, the ATPG enforces b to be different from all other signals in the circuit so that the signal substitution errors on b will be detected. The exclusiveness of signal values have been enforced as much as possible during the test environment translation process.

For the relational operations, we maximize the error detection by plugging 3 properly selected values. The validation vectors can be selected as follows:

1. If one operand of relational operation is constant, say k, then 3 values of k+1, k, and k-1 are applied to the other input of relational operation.
2. If both inputs are variables, first assign unique value, say, k to one input, and then apply k+1, k, and k-1 for the other input.

For example, in validating the $(a > 3)$, we need to check that it is correct to use "greater-than" operation here, and that the value range boundary 3 is correct. In other words, we need to differentiate it from the following implementations:

$$(a \geq 3), (a \neq 3), (a < 3), (a \leq 3), (a > 5), (a < 4), \text{ and } (a > 2)$$

Following the above mentioned rules, we know that 4,3,and 2 need to be plugged into the generated test environment to get 3 different validation sequences. Applying these 3 sequences on the HDL description, the 4, 3, and 2 will be delivered to signal a at target site. It is obvious that each of the vector sequence is able to detect some errors but none of them can detect all bugs. However, collectively they can capture all of the bugs. Note that, from the tag coverage [FAD99] point of view, requiring three validation sequences on $(a > 3)$ is an overkill. Apparently, the algorithms in [FAD99] will not derive those "redundant" validation vectors. As the result, the [FAD99] cannot guarantee the error detection associated with operational operators.

4.4.6 Static Learning to Aid Value Range Analysis

In the previous section, we demonstrated the necessity of inclusion of value range analysis to enhance the symbolic justification. However, there is a related performance issue.

In that example, to justify the (En, C1), both (Cn1, C1) and (Cn2, C0) are needed. Suppose ATPG picks (Cn2,C0) first, which leads to the (Data, Cp[59;0]). Since the justification is conducted in a depth-first fashion, the ATPG goes on to justify (Data, Cp[59;0]) immediately. Suppose after a certain number of steps, the (Data, Cp[59;0]) is successfully justified. Then, ATPG moves on to justify (Cn1, C1), which leads to a tighter value rang of [59;50] on *Data*. This tighter value range needs to be justified as well. Therefore, the ATPG has to justify the two related Cps on the *Data* respectively. For some circuits, this only increases the computation moderately if the two Cps are

both easy to be justified. However, for some other circuits, the first unnecessary justification can cause large number of backtracks. For example, the value of Data is fed from the memory as shown in Figure 4.7. With the first objective Cp[59;0], the

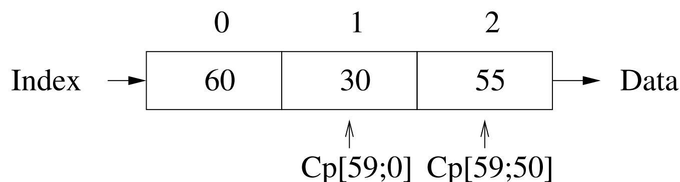


Figure 4.7: Inaccurate value range selects wrong signal value.

value 30 may be selected, and accordingly the *index* is justified to 1. However, the selected value of 30 conflicts with the second objective Cp[59;50]. If the justification of *index* is not trivial, a large number of backtracks will be needed before ATPG eventually picks correct value 50 if given enough time.

In order to quickly determine the desirable value range for each encountered objective, we use static learning process to construct the RTL implications. It is desirable to have a mechanism that can determine the value range as tight as possible and as quickly as possible.

We augmented validation vector generation algorithm with following static learning steps to find RTL implications.

1. For each comparator type RTL primitive (=, /=, >, <, and etc), inject two implication constructs as in Figure 4.8. The upper bound of value range is determined by the signal bit width.

2. Recursively forward propagate the implication constructs using the rules shown

in Figure 4.8. We only propagate through binary logic gates, such as BUF, INV, AND, NAND, OR, NOR. Implication propagation stops at all other gates.

3. Update the implication list for reached RTL primitives.

By only injecting the implications at comparator primitives and by only propagating through binary logic gates, our implication process focuses on the control logic of circuit, since our algebra-based ATPG has been proved to be effective for data-path circuits.

Following the same example in Figure 4.6, after the static learning, we have two implications on *En*, which are C1→(Data,Cp[MAX;50]), and C1→(Data,Cp[59;0]). When justifying algebra C1 on En, checking the implication lists, ATPG knows that two implications are required. Immediately, the ATPG can derive correct value range of [59;50] for the signal *Data*. Thus, unnecessary steps and backtracks are avoided.

Our seemingly simple implication procedure is actually very powerful in finding implications for the control logic and in avoiding conflicts. First, the control inputs of ADN are mutually exclusive; that is, any control signal being C1 implies all other control signals feeding the same ADN are C0. Therefore C1 on any control input will have many implications. Therefore using statically computed implication can speed up the justification process. Secondly, asserting any control input of ADN is functionally equivalent to specifying a particular basic block in the HDL code to be executed. Even though the execution path reaching that basic block may not be unique, it is very likely that a particular code segment is guaranteed to be executed. Therefore, the related condition variables are implied by the single assertion of control input on ADN. Thirdly, the nested conditional statements in VHDL/Verilog always induce

the re-convergent predicate signals. Therefore, there are many opportunities to apply this implication-based techniques.

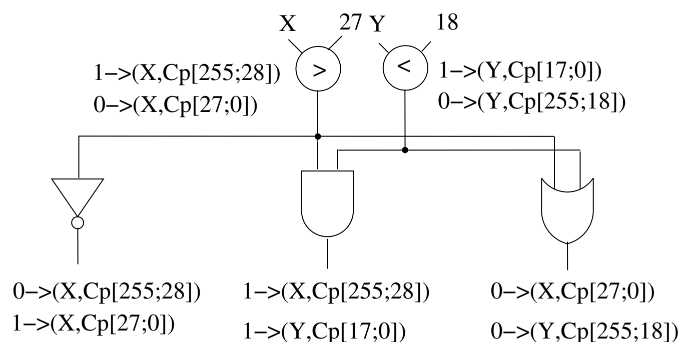


Figure 4.8: Direct implication of RTL algebra

4.4.7 Identification of reset signals

It is much easier to extract some circuit characteristics at the RT level than at the gate level. Application of those knowledge can be very beneficial for generating validation vectors. Reset input identification is one of the improvements we implemented in our framework.

In [STH02], the authors used *Reset Signal Masking* technique to compute the *Reset Power* for each primary input. This information is later used to bias the subsequent GA process to avoid repeatedly resetting the circuit.

Our framework identifies the reset input by performing a structural analysis. Figure 4.1 shows the expression for a reset in VHDL and corresponding functional RTL ADD representation. Note that synthesizable VHDL/Verilog requires reset input to all registers to always be expressed in a fixed style, making our identification easy. From

this figure, the reset input relates to a special structure in functional RTL description. The structural analysis of the circuit in our preprocessing step quickly identifies the reset input. When our framework translates test environments to system level test vectors, if any reset input in a time frame is not specified, the non-reset value will be favored for the reset input in that time frame. This is useful because by avoiding frequent reset of the circuit, we often can accidentally detect more faults. Another application of the reset knowledge is in the later simulation-based test engine. The first vector can use this knowledge to initially reset the circuit, with the subsequent vectors encoded in such a way to avoid reset.

4.4.8 STG Extraction

State justification is known as one of the most difficult problems in sequential ATPG. Unlike gate-level techniques, which cannot view the global relationships among flip-flops, our framework can readily extract the STG directly from the RTL netlist. For each FSM inside the circuit, there is one ADD associated with the state variable of the FSM. At RT level, all states of a FSM are explicitly declared. Thus, the size of states is always manageable. The complete STG of FSM can be easily constructed.

Figure 4.9 shows the ADD for the state variable *STATEO*, and extracted STG. The corresponding VHDL code is shown in Figure 4.1. Note that, we do not draw out the complete control logic for the ADD. Each control input except the *RESET* is associated with a single FSM state. For example, the control inputs CTL0, CTL1, and CTL2 are all controlled by *state S0*. The extracted STG is complete in the sense that it contains all the states of the FSM, and captures all transition arches of the FSM. However, it does not contain the transition conditions associated with

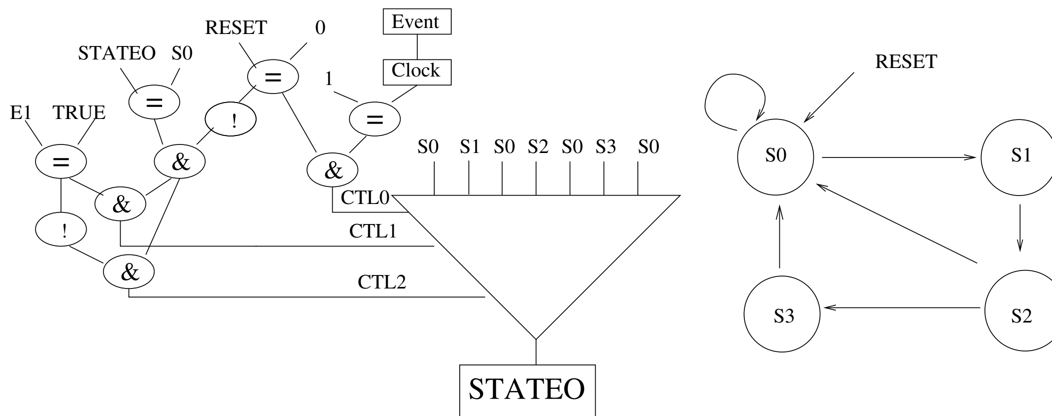


Figure 4.9: ADD for state variable & STG

each arch. The transition conditions are implicitly embedded in the circuit. Every FSM has a reset state, which is S0 in this example. Conceptually, every state is the predecessor state of reset state. However, we do not store resetting arches in STG, since the resetting arches are not favored for test generation purposes. Instead, we just label the reset state. Implicitly, every state can go to reset state in one step.

4.4.9 Counter and Iteration Loop Identification

Counters are difficult for ATPG, especially when the counters require long sequences. In some circuits, counters may even be the bottleneck for testing. For example, the output of a counter may be used to control the input of another module. With counters frequently used in modern digital systems, having an engine that can automatically extract and exploit this knowledge would be of immense aid. The algorithm in [GF01] proposed a heuristic for counter identification. The approach dynamically checks for the repetitive appearances of the same justification objective on the same

variable across a number of time frames. One drawback of that approach is high computation overhead. It needs to check the condition frequently throughout justification process. Secondly, the counter identification can be a false positive. In other words, it may mistakenly declare a non-counter structure to be a counter. The counter identification in our framework is performed statically. In the structural RTL representation, the counter structure contains an adder, a feed-back loop through some flip-flops and ADN, and a constant input to the adder, as shown in Figure 4.1. During the preprocessing step, our algorithm marks all adder modules that fit in the counter structure, then perform static learning for each counter.

Consider again the counter example in Figure 4.1 to illustrate how our ATPG identifies the FSM iteration loop for the counter. The FSM iteration loop is the shortest state sequence loop that increments the counter continuously. Note that, the ADD for state variable and the STG are shown in Figure 4.9. For example, inspecting the VHDL code and STG we can see that

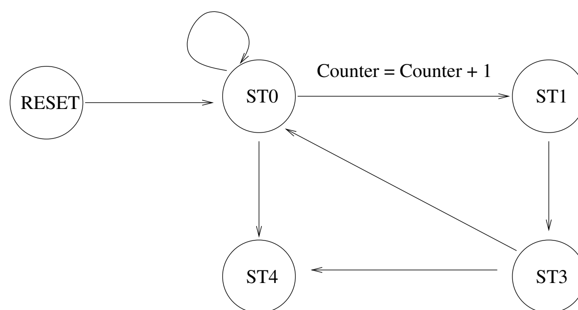


Figure 4.10: State Traversal Sequence for Embedded Counter

$S0 \rightarrow S1 \rightarrow S2 \rightarrow S0$ is the shortest FSM loop as shown in Figure 4.10, which increments the counter continuously. Suppose during the justification, we need to

justify the counter to be 31. Then, the shortest state sequence is $(S0 \rightarrow S1 \rightarrow S2)$x31, a total 62 states!. Without this knowledge, the branch-and-bound method often searches blindly in the large search space.

Our ATPG uses the following steps to identify the above sequence loop.

1. Perform breath-first search on circuit netlist at the control input of counter ADN to find out the controlling state. That state is the operation state for the counter. In this example, S0 is controlling the CTL2. Thus S0 is the operation state of counter.

2. Assert the control input of ADN and compute the implication of it. For example, the assertion of CTL2 implies E2 is true, E1 is true, and etc.

3. Check state variable ADN (in Figure 4.9) to see if any control input is implied to C1. The state at the corresponding data-input is the next state. In this example, CTL1 is implied to C1, therefore the next state is S1.

4. Do breath-first search on extracted STG to find the shortest path from counter next state to counter operation state. Note that, the reset arches are not allowed. Here we get $S1- > S2- > S0$.

5. The final iteration loop contains 3 states and 3 transition arches found in previous steps. $S0 \rightarrow S1 \rightarrow S2 \rightarrow S0$.

The above implication-based approach is performed statically during the preprocessing step. The identified FSM iteration loop for each counter is stored in a data structure and will be used during the justification process to quickly guide the ATPG to the correct solution.

4.4.10 Simulation-based Procedure for Special Case

Figure 4.11 shows one of the cases that RTL algebra cannot effectively handle, even with the the added value analysis extension. In this example, signal b needs to be

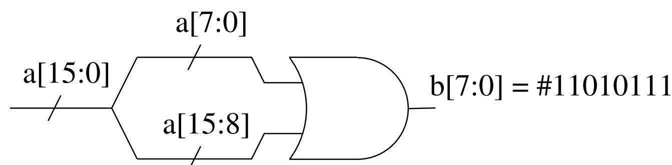


Figure 4.11: Bus split and re-joint

justified to "11010111". Even though we can model the justification objective as (Cp,[215;215]), this justification objective cannot be justified successfully through signal a due to the bus re-joint. It is difficult to represent the justification objective collectively with just one algebraic symbol. There are too many possible combinations for a[15:8] and a[7:0] to justify the b. As a result, the possible values of a[15:0] are scattered and cannot be efficiently represented by RTL algebra. Therefore backward justification aborts on this. For this particular case, gate-level ATPG will potentially be more effective and efficient than RTL ATPG. In general, when word-level entities are combined through Boolean operators, the resulting value ranges will be difficult to obtain. If such situation arises, our validation framework will likely be unable to effectively generate test environment for the affected RTL primitives. As the result, the validation quality may suffer.

On the other hand, the simulation-based approach performs the forward simulation only. It may be able derive the vectors satisfying the conditions on b. In our framework, we incorporated a genetic algorithm (GA) based ATPG procedure to boost the

the tag coverage when deterministic test generation aborts on the primitives similar to the one discussed above.

As shown in the [KS98], each of the deterministic and simulation-based test generators has its own merits. In order to leverage the power of simulation-based approach for those MUTs where only partial test environments were derived, we implemented a genetic-algorithm (GA)-based functional RTL test engine. Our approach is different from previous high-level GA-based test generators, such as ARTIST in following ways. First, we are not targeting any software testing metrics, such as statement or branch coverage. Instead, we chose the tag coverage for the fitness function of GA, which takes into account observability as well as controllability of possible faulty effects. Second, we do not start with pure random trial sequences. Instead, initial individuals are generated based on the partial test environment information collected during the deterministic test phase. These informations include state traversal sequences and counter sequences. Reset signal knowledge is also used here. The reset input is biased to be non-reset value unless the partial test environment guides us not to do so. Third, we restrict the GA evolution process with a small number of iterations. This is due to the fact that tag simulation can be expensive (no parallel simulation available). It would not be advantageous to allow the GA to pursue a few hard tags for too long. Our experience tells us that with good initial individuals, the GA often converges on a solution in a small number of iterations. Other detailed information about GA algorithm can be found in [HRP97]. It should be noted that the set of 10-valued RTL algebra, used by deterministic engine for symbolic justification and propagation is not used for GA engine. Instead, our GA uses integer values plus tags to represent fault-free and faulty effects of RTL circuits.

This extension of simulation-based engine only needs to be applied to very few circuits. We use the observability-based code coverage (OBCC) [FDK01] as the coverage metrics for the GA process. OBCC coverage is superior to the classical software testing coverage (such as statement and branch coverage) in that it incorporates observability as well as controllability information into the simulation. For the aborted RTL primitives, the GA process is invoked to generate vectors, that excite the primitives and bring the output primitives to the observable points.

4.5 Experimental Results

We applied our framework to 10 ITC99[CRS00a] benchmark circuits and an industrial general purpose input /output bus controller, *GPIO*. For each VHDL description, we manually injected 15 to 30 bugs, which include the most typical design errors, such as missing case statement, missing signals, wrong signals, wrong variable values, wrong ordering of nested if statements, wrong operation types, and etc. Table 4.4 shows circuit characteristic and validation results.

We report both tag coverage, and bug coverage for each circuit, where the bug coverage is defined as the ratio of the number of detected bugs to the total number of injected bugs. The reported TGen is the execution time (in seconds) spent on generating the validation vectors. Note that, our framework works on a given HDL description, and no gate-level implementations are needed. However, we include the gate-level characteristics in the table to show the complexities of each design. For each circuit, the total number of VHDL lines is first reported, followed by the number of logic gates corresponding to the VHDL and the number of flip-flops. Then, the

number of validation vectors generated using our RTL ATPG is reported. Next, the execution time, bug coverage, and tag coverage are reported. For example, circuit B11 has 118 lines of VHDL code and can be synthesized to 397 gates at the gate level. It has 30 FFs. Using the VHDL alone for our RTL ATPG, 193 validation vectors are generated in only 5.8 seconds. The tag and bug coverages are 66.7% and 70.0% respectively. For each circuit, we also applied 5,000 random vectors. The achieved tag coverages are reported in the last column of the table. Note that the test generation times of our method are orders of magnitude smaller than the ones reported in [FAD99] for similar sized circuits. A direct comparison is not possible as the circuits used in [FAD99] are not publicly available.

When our results are compared with random generation of validation vectors, Figures 4.12 and 4.13 show the tag coverage and bug coverage, respectively. For each circuit, the same number of vectors for our approach and random approach were applied. Note that in all circuits, our RTL ATPG outperforms random generation of validation vectors. In most of the circuits (except B01, B02 and B06), random generation cannot achieve very high tag coverage or bug coverage. In fact, in the five largest circuits, our method achieves orders of magnitude better results in both tag and bug coverages. Note also that since the number of validation vectors we generated was significantly fewer than 5000, the validation time would be reduced by the same ratio. For example, in circuit B08, with only 100 vectors, we are able to achieve 93.3% tag coverage in under 8 seconds of computation, while with 5000 random vectors, only 2.2% tag coverage was achieved. In terms of simulation time for validation, 50-fold is achieved. In fact, increasing the random vectors beyond 5000 vectors still would not be helpful. In this regard, several orders of magnitude lower validation time can be achieved with our method.

Circuit	Characteristics			Our Approach				Random	
	#line	#Gate	#FF	#Vec	TGen(s)	Bug Cov.	Tag Cov.	#Vec	Tag Cov.
B01	110	56	5	192	0.03	100%	58.3%	5000	54.2%
B02	70	30	4	77	0.03	100%	53.8%	5000	53.8%
B03	141	181	30	653	3.51	100%	90.9%	5000	80.0%
B04	102	547	66	165	0.12	100%	83.8%	5000	74.2%
B05	332	643	34	255	19.00	45.0%	41.1%	5000	15.0%
B06	128	76	9	49	0.06	86.7%	72.1%	5000	73.5%
B07	92	434	51	104	5.97	100%	91.2%	5000	1.8%
B08	89	190	21	100	7.32	100%	93.3%	5000	2.2%
B10	167	190	17	337	17.5	85.0%	100%	5000	33.1%
B11	118	397	30	193	5.8	70.0%	66.7%	5000	20.8%
GPIO	1002	1720	148	652	96.0	90.0%	76.6%	5000	6.8%

Table 4.4: Experiment results: Our Approach vs. Random Approach

Figure 4.14 shows the detailed validation results on the industrial benchmark circuit *GPIO*. The X-axis is the number of validation vectors applied, while the Y-axis is the coverage of respective metrics. For *GPIO*, 30 design errors were injected. Note that this circuit is not randomly validatable, as shown by the curves for random vectors. Our approach is able to obtain both high TAG coverage as well as high bug coverage. We can also clearly observe the close correlation between the achieved tag coverage and bug coverage for both random test vectors and vectors generated by our approach. However, the tag coverage is slightly more pessimistic than the bug coverage in that the coverage reported is lower. In other words, the actual bug coverage generally can

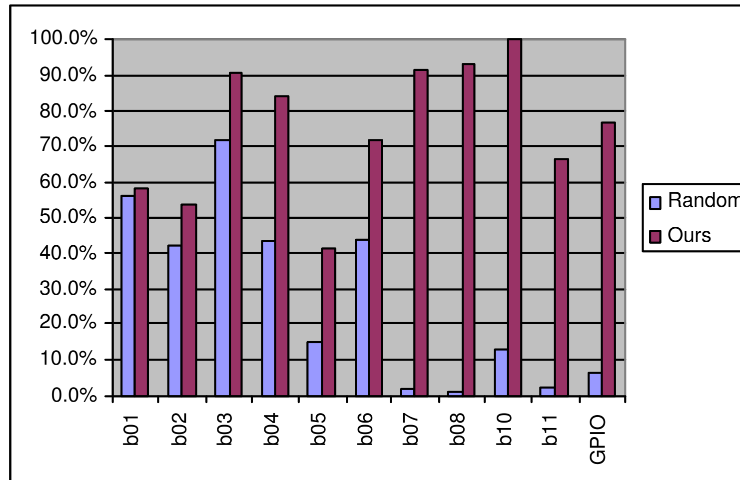


Figure 4.12: Tag Cov. for Our and Random Approach with the same number of vectors

be higher than the tag coverage. Nevertheless, these two coverage metrics track very well with each other.

We also run the vectors generated by our approach on TRansEDA[VN-] coverage analysis tool. The results are shown in Table 4.5. We can see that tag coverage is superior and the vectors generated targeting this coverage metric already gives close to 100% coverage for the traditional metrics (B05 has large redundancy in both VHDL and gate-level implementation). Note that, those traditional coverages are currently used in the industry to measure the design validation and a number of commercial tools exist in this effect.

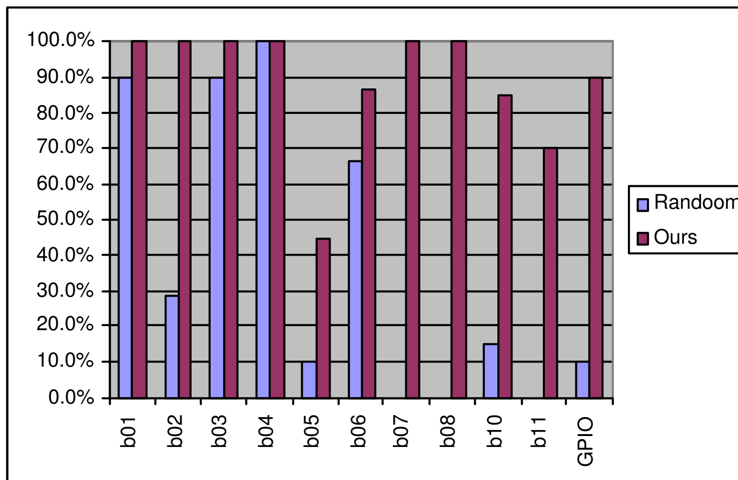


Figure 4.13: Bug Cov. for Our and Random Approach with the same number of vectors

4.6 Summary

Although logic level formal verification techniques are very powerful in finding corner case bugs and in proving a design free of any bugs at the module level, however, generally speaking, these logic level techniques do not scale well to be applied at the higher levels of abstraction nor at the system level. Additionally, some high level circuit constructs, such as tri-state buffer and bi-directional gates, cannot be easily represented by the logic level boolean models. Therefore it is of tremendous importance to have a high-level design validation framework which handles the system level design validation tasks directly. The most commonly used system-level design validation approach is random simulation or bias random simulation. In this chapter, we have presented an automatic design validation framework for HDL descriptions

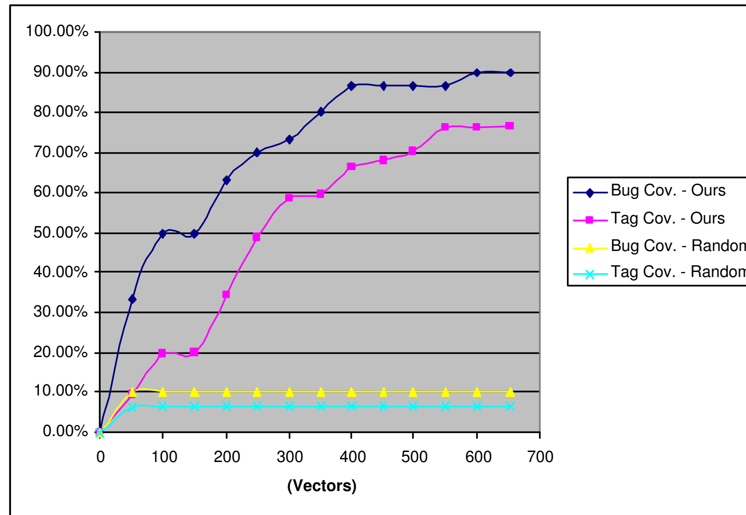


Figure 4.14: Experimental Result on GPIO

at Register Transfer Level, which features a set of novel improvements on a unified deterministic and simulation-based validation framework.

The core of our framework is modified RTL ATPG, which efficiently generates the validation vectors. Our approach is superior to existing approaches, which target only the excitation but not propagation of the bugs in the design. We also allow for relaxation of test environments such that additional hard errors may be detected. Experiments show that our approach is able to generate high quality validation vectors, which achieve both high tag coverage and high bug coverage with extremely low computational cost. Orders of magnitude improvement in coverage over random patterns can be achieved.

Circuit	Statement Cov.(%)	Branch Cov.(%)	Condition Cov.(%)	Tag Cov.(%)
B01	100	100	98.4	58.3
B02	100	100	100	53.8
B03	100	100	100	90.9
B04	100	100	100	83.8
B05	65.4	61	n/a	41.1
B06	100	100	100	72.1
B07	97.9	97.2	n/a	91.2
B08	100	100	n/a	93.3
B10	94.7	95.3	90	100
B11	85.7	87.9	100	66.7
GPIO	100	100	100	76.6

Table 4.5: Coverage Results of Our Approach on Different Metrics

Chapter 5

Conclusion

This dissertation has developed new approaches for the design verification of sequential systems at different abstraction levels.

At the logic level, we focuses on incorporating a set of novel ideas to the existing formal verification approaches. First, we presented a number of optimizations to improve the performance and capacity of a typical SAT-based bounded model checking framework. The proposed ideas include clever orchestration of variable ordering and learned information in an incremental framework for BMC, a novel idea to combine SAT-based inductive reasoning and BMC and BMC-specific ordering strategies for the SAT solver. Experiments on a wide range of industrial designs have shown that the proposed optimizations consistently provide between 1-2 orders of magnitude speedup and can be extremely useful in enhancing the efficacy of typical SAT-BMC tools. Secondly, we have presented a method for performing dynamic abstraction within a framework for abstraction-refinement based model checking. The dynamic

abstraction is applied during successive image computation steps of the model checking algorithm and can be applied after and in addition to the abstraction performed by traditional static abstraction methods. It is facilitated by information gathered from an analysis of the proof of unsatisfiability of SAT-based bounded model checking problems, solved on the concrete model, and passed to the model checker. It effectively allows the model checker to work with smaller abstract models. We have proposed two strategies for realizing the dynamic abstraction. Our experiments on several large industrial benchmarks demonstrate that the proposed dynamic abstraction strategies improves the performance of a model checking flow based on abstraction refinement by up to an order of magnitude over what can be achieved with state-of-the-art static abstraction refinement methods and and also enables the successful application of abstraction refinement based model checking to larger designs.

At the register transfer level, where the formal verification is less likely to succeed, we rely on the simulation-based validation approach. To be precise, we proposed a set of heuristics to leverage the high level information to guide our ATPG-based symbolic search engine to generate high quality validation sequence. We also proposed improvements over observability-enhanced coverage metrics to better gauge the validation coverage. Experiments show that our approach is able to generate high quality validation vectors, which achieve both high tag coverage and high bug coverage with extremely low computational cost. Orders of magnitude improvement in coverage over traditional simulation-based approach has been achieved.

There are several avenues for further improving our SAT-BMC framework. One possibility is to integrate some of the promising techniques published earlier in the literature, into our framework. These include the "on-the-fly" graph compression tech-

nique of [GA02] as well as the BDD-based clause learning techniques of [GGW+03b, CNQ03]. This and other issues will be the subject of our future investigations and development. Regarding the dynamic abstraction-based verification, the possible future research includes development of better and more sophisticated latch selection strategies, based on the treatment presented in Section 3.4.1. Another avenue would be to apply the notion of dynamic abstraction to other abstraction refinement frameworks incorporating a variety of model checking techniques and other heuristics for static abstraction refinement.

For the high-level validation framework, it would be interesting to investigate and incorporate the constrained functional verification and compositional verification approach [YAPA04, YPAA03, YSP+99].

Bibliography

[Ake78] S. B. Akers. binary decison diagrams. *IEEE Transaction on Computers*, 27, July 1978.

[AKMM03] Nina Amla, Robert Kurshan, Kenneth McMillan, and Ricardo Medel. Experimental Analysis of Different Techniques for Bounded Model Checking. In Hubert Garavel and John Hatcliff, editors, *Proc. of the* 9th *TACAS*, volume 2619 of *LNCS*, pages 34–48. Springer, April 2003.

[BC00] Per Bjesse and Koen Claessen. SAT-based Verification without State Space Traversal. In Warren A. Hunt and Steven D. Johnson, editors, *Proc. of the* 3rd *Intl. Conf. on Formal Methods in CAD*, volume 1954 of *LNCS*, pages 372–389. Springer, Nov. 2000.

[Bei90] B. Beizer. *Software Testing Techniques (2nd ed.)*. Van Nostrand Rheinold, New York, 1990.

[BK04] Per Bjesse and James Kukula. Using Counter Example Guided Abstraction Refinement to Find Complex Bugs. In *Proc. of the Design Automation and Test in Europe Conf.*, pages 156–161, Feb. 2004.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transations on Computers*, C-35:677–691, June 1986.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, July 2001. Kluwer.

[CCK+02] Pankaj Chauhan, Edmund M. Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis. In *Proc. of the* 4th *Intl. Conf. on Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 33–51, Nov. 2002.

[CCRS03] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero. Fully automatic test program generation for microprocessor cores. In *Proc. Design, Automation and Test in Europe*, pages 1006–1011, 2003.

[CFF+01] Fady Copti, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of Bounded Model Checking in an Industrial Setting. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proc. of the* 13th *Intl. Conf. on Computer Aided Verification*, volume 2102 of *LNCS*, pages 436–453. Springer, July 2001.

[CGKS02] Edmund M. Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT-based Abstraction Refinement Using ILP and Machine Learning Techniques. In *Proc. of the* 14th *Intl. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 265–279. Springer, July 2002.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CGR93] V. Chaiyakul, D. D. Gajski, and L. Ramachandran. High-level transformations for minimizing syntactic variances. In *Proc. Design Automation Conference*, pages 413–418, 1993.

[CNQ03] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals. In *Proc. of the Design Automation and Test in Europe*, pages 898–903, March 2003.

[CRS00a] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *IEEE Design & Test of Computers*, 17(3):44–53, July-September 2000.

[CRS+00b] F. Corno, M.S. Reorda, G. Squillero, A. Manzone, and A. Pincetti. Automatic test bench generation for validation of rt-level descriptions: an industrial experience. In *Proc. DATE*, pages 385–389, 2000.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[ES03] Niklas Eén and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. In *Proc. of the* 1st *International Workshop on Bounded Model Checking*, July 2003.

[FAD99] F. Fallah, P. Ashar, and S. Devadas. Simulation vector generation from hdl descriptions for observability-enhanced statement coverage. In *Proc. Design Automation Conference*, pages 666–671, 1999.

[Fal02] Farzan Fallah. Binary Time-Frame Expansion. In *Proc. of the Intl. Conf. on CAD*, pages 458–464, Nov. 2002.

[FDK01] Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. Occom – efficient computation of observability-based code coverage metrics for functional verification. *IEEE Trans. Computer-Aided Design*, 20(8):1003–1015, August 2001.

[GA02] Malay K. Ganai and Adnan Aziz. Improved SAT-based Bounded Reachability Analysis. In *Proc. of the* 15th *Intl. Conf. on VLSI Design*, pages 729–734, January 2002.

[GF01] Indradeep Ghosh and Masahiro Fujita. Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams. *IEEE Trans. Computer-Aided Design*, 20(3):402–415, March 2001.

[GGW+03a] Aarti Gupta, Malay Ganai, Chao Wang, Zijiang Yang, and Pranav Ashar. Abstraction and BDDs Complement SAT-based BMC in Di*Ver*. In Jr. Warren A. Hunt and Fabio Somenzi, editors, *Proc. of the* 15th *Conf. on Computer-Aided Verification*, volume 2725 of *LNCS*, pages 206–209. Springer, July 2003.

[GGW+03b] Aarti Gupta, Malay Ganai, Chao Wang, Zijiang Yang, and Pranav Ashar. Learning from BDDs in SAT-based Bounded Model Checking. In *Proc. of the* 40th *Design Automation Conf.*, pages 824–829, June 2003.

[GGYA03] Aarti Gupta, Malay Ganai, Zijiang Yang, and Pranav Ashar. Iterative Abstraction Using SAT-based BMC with Proof Analysis. In *Proc. of the Intl. Conf. on CAD*, pages 416–423, Nov. 2003.

[GKMH+03] Marcelo Glusman, Gila Kamhi, Sela Mador-Haim, Ranan Fraer, and Moshe Y. Vardi. Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation. In *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 176–191. Springer, April 2003.

[GN02] Evgueni Goldberg and Yakov Novikov. BerkMin: a Fast and Robust Sat-Solver. In *Proc. of Design Automation and Test in Europe*, pages 142–149, March 2002.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proc. of the Design Automation and Test in Europe*, pages 886–891, March 2003.

[GR03] Indradeep Ghosh and Srivaths Ravi. On automatic generation of rtl validation test benches using circuit testing techniques. In *Proc. Great Lakes Symposium on VLSI*, pages 289–294, 2003.

[HR96] G.A. Hayek and C. Robach. From specification validation to hardware testing: A unified method. In *Proc. International Test Conference*, pages 885–893, 1996.

[HRP97] M.S. Hsiao, E.M. Rudnick, and J.H. Patel. Sequential circuit test generation using dynamic state traversal. In *Proc. Eur. Design Test Conf.*, pages 22–28, Mar 1997.

[IPC03] M. K. Iyer, G. Parthasarathy, and K.-T. Cheng. SATORI-A Fast Sequential SAT Engine for Circuits. In *Proc. of the Intl. Conf. on CAD*, pages 320–325, Nov. 2003.

[KPKG02] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Trans. on CAD*, 21(12):1377–1394, Dec. 2002.

[KS98] Yong Chang Kim and Kewal K. Saluja. Sequential test generators: past, present and future. *INTEGRATION, the VLSI journal*, 26:41–54, 1998.

[Kur95] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.

[Lee58] C. Y. Lee. Binary decision programs. *Bell System Technical Journal*, 38:985–999, July 1958.

[MA03] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 2–17. Springer, April 2003.

[McM93] K. L. McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem*. Kluwer, 1993.

[McM97] K. L. McMillan. A compositional rule for hardware design refinement. In *Proc. 9th International Computer Aided Verification Conf.*, pages 24–35, 1997.

[MH04] Freddy Y. C. Mang and Pei-Hsin Ho. Abstraction Refinement by Controllability and Cooperativeness Analysis. In *Proc. of the* 41st *Design Automation Conf.*, pages 224–229, June 2004.

[MMZ+01] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the* 38th *Design Automation Conf.*, pages 530–535, June 2001.

[MSS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers*, 48(5):506–521, May 1999.

[PNS00] C. Paoli, M.L. Nivet, and J.F. Santucci. Use of constraint solving in order to generate test vectors for behavioral validation,. In *Proc. High Level Design Validation and Test Workshop*, pages 15–20, 2000.

[RAP+95] R.K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD algorithms for FSM synthesis and Verification. In *Proc. of Intl. Workshop on Logic Synthesis.*, May 1995.

[S+98] E. M. Sentovich et al. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, ERL, College of Engg., U. C. Berkeley, May 1998.

[SD02] Kanna Shimizu and David L. Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In *Proc. of Design Automation Conference*, pages 801–806, 2002.

[Sht04] Ofer Shtrichman. Accelerating Bounded Model Checking of Safety Formulas. *Formal Methods in System Design*, 24(1):5–24, Jan. 2004. Kluwer Academic Publishers.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt and Steven D. Johnson, editors, *Proc. of the* 3rd *Intl. Conf. on Formal Methods in CAD*, volume 1954 of *LNCS*, pages 108–125. Springer, Nov. 2000.

[STH02] Shuo Sheng, Koichiro Takayama, and Michael S. Hsiao. Effective safety property checking based on simulation-based atpg. In *Proc. IEEE Design Automation Conference*, pages 813–818, 2002.

[SZ03] Ohad Shacham and Emmanuel Zarpas. Tuning the vsids decision heuristic for bounded model checking. In *Proc. of the* 4th *Intl. Workshop on Microprocessor Test and Verification*, pages 75–79, May 2003.

[The96] The VIS Group. VIS: A system for Verification and Synthesis. In *Proc. of the* 8th *Intl. Conf. on Computer Aided Verification*, volume 1102 of *LNCS*, pages 428–432. Springer, July 1996.

[VN-] http://www.transeda.com.

[WHL+01] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines. In *Proc. of the* 38th *Design Automation Conf.*, pages 35–40, June 2001.

[WHS04] Chao Wang, Gary D. Hachtel, and Fabio Somenzi. Fine-Grain Abstraction and Sequential Don't Cares for Large Scale Model Checking. In *Proc. of the Intl. Conf. on Computer Design*, October 2004.

[WJHS04] Chao Wang, HoonSang Jin, Gary Hachtel, and Fabio Somenzi. Refining the SAT Decision Ordering for Bounded Model Checking. In *Proc. of the* 41st *Design Automation Conf.*, pages 535–538, June 2004.

[WKS01] Jesse P. Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Proc. of the* 38th *Design Automation Conf.*, pages 542–545, June 2001.

[WLJ+03] Chao Wang, Bing Li, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. Improving Ariadne's Bundle by Following Multiple Threads in Abstraction Refinement. In *Proc. of the Intl. Conf. on CAD*, pages 408–415, Nov. 2003.

[YAPA04] Jun Yuan, Adnan Aziz, Carl Pixley, and Ken Albin. Simplifying boolean constraint solving for random simulation-vector generation. *IEEE Trans. Computer-Aided Design*, 23(3):412–420, March 2004.

[YPAA03] Jun Yuan, Carl Pixley, Adnan Aziz, and Ken Albin. A framework for constrained functional verification. In *Proc. of Intl. Conf. Computer-Aided Design*, pages 142–145, 2003.

[YSP+99] Jun Yuan, K. Shultz, Carl Pixley, H. Miller, and Adnan Aziz. Modeling design constraints and biasing in simulation using bdds. In *Proc. of Intl. Conf. Computer-Aided Design*, pages 584–589, 1999.

[zch03] http://ee.princeton.edu/~chaff/zchaff.php, Dec. 2003.

[ZGH03] Liang Zhang, Indradeep Ghosh, and Michael Hsiao. Efficient sequential atpg for functional rtl circuits. In *Proc. International Test Conference*, pages 290–298, 2003.

[Zha97] Hantao Zhang. SATO: An Efficient Propositional Prover. In *Proc. of the Intl. Conf. on Automated Deduction*, pages 272–275, July 1997.

[ZM03] Lintao Zhang and Sharad Malik. Validating SAT Solvers using an Independent Resolution-based Checker: Practical Implementations and Other Applications. In *Proc. of the Design Automation and Test in Europe*, pages 880–885, March 2003.

Vitae

Liang Zhang was born and raised in Xi'an, a northern city of China known for its rich history . He received his B.E. in Electronic Engineering and B.A. in English from Tianjin University in 1996, and his M.S. in Electronic Engineering from Tsinghua University in 1999. He then spent two years at Rutgers University as a Ph.D student prior to transferring to Virginia Polytechnic Institute and State University with Dr. Michael Hsiao. Currently, he is a Ph.D candidate at the Bradley Department of Electrical and Computer Engineering of the same institute.

He will be joining Cadence Design System, San Jose, California, working on formal equivalence checking. His research interests include formal verification, semi-formal verification, and testing.