Fast Static Learning and Inductive Reasoning with Applications to ATPG Problems

Michael Dylan Dsouza

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

Master of Science
in
Computer Engineering

Michael S. Hsiao, Chair
Chao Wang
Patrick R. Schaumont

February 19, 2015
Blacksburg, Virginia

Keywords: Static learning, inductive reasoning,
multi-node invariants, logic implications, boolean constraint propagation

Copyright 2015, Michael Dylan Dsouza

Fast Static Learning and Inductive Reasoning with Applications to ATPG Problems

Michael Dylan Dsouza

(ABSTRACT)

Relations among various nodes in the circuit, as captured by static and inductive invariants, have shown to have a positive impact on a wide range of EDA applications. Techniques such as boolean constraint propagation for static learning and assume-then-verify approach to reason about inductive invariants have been possible due to efficient SAT solvers. Although a significant amount of research effort has been dedicated to the development of effective invariant learning techniques over the years, the computation time for deriving powerful multi-node invariants is still a bottleneck for large circuits. Fast computation of static and inductive invariants is the primary focus of this thesis. We present a novel technique to reduce the cost of static learning by intelligently identifying redundant computations that may not yield new invariants, thereby achieving significant speedup. The process of inductive invariant reasoning relies on the assume-then-verify framework, which requires multiple iterations to complete, making it infeasible for cases with a large set of multi-node invariants. We present filtering techniques that can be applied to a diverse set of multi-node invariants to achieve a significant boost in performance of the invariant checker. Mining and reasoning about all possible potential multi-node invariants is simply infeasible. To alleviate this problem, strategies that narrow down the focus on specific types of powerful multi-node invariants are also presented. Experimental results reflect the promise of these techniques. As a measure of quality, the invariants are utilized for untestable fault identification and to constrain ATPG for path delay fault testing, with positive results.

This work was supported in part by NSF Grant 1016675.

∼ To my parents and to the memory of my grandmother ∼

Acknowledgments

The work presented in this thesis is the result of contributions by many individuals. It gives me great pleasure to express my gratitude to all people who made it possible.

First and foremost, I consider myself extremely fortunate to have had the chance of working with Prof. Michael Hsiao, my advisor. This work would not have been possible without his guidance, support and his faith in me. His optimism towards research problems, knack for interesting ideas and his exceptional reasoning ability will always be a source of inspiration. Interactions with him have always been a great learning experience. I sincerely thank him for making my graduate school experience worthwhile.

I am honoured to have Prof. Chao Wang and Prof. Patrick Schaumont on my thesis committee. I am also grateful for all the interactions through courses and various CESCA events, which added significant value to my graduate school research and learning experience.

I would like to express my sincere gratitude to Dr. Aravind Dasu for giving me the chance to work at USC-ISI and Surendra Bommu for the internship opportunity at Synopsys. Both internships had a great impact on me and has shaped my perspective towards research and development in the area of EDA.

I am grateful to Vineeth for all the help during the entire duration of our master's degree program. I thank Sarvesh for all the ideas, guidance, motivation and Sharad for making

the time spent in the lab to be fun and memorable. It was great to have been part of the PROACTIVE research group and to have worked closely with the RSS research group. I would like to acknowledge the help and support from all the members of the two research groups, whose enthusiasm towards their work was a great motivation.

I would like to thank my roommates, Vineeth, Vivek, Shashank and friends, Sudarshan, Anupriya, Tejaswi, Chaitra, Mishell and Jose for all the good times and for regularly reminding me that graduate life does not have to be all about work.

Most importantly, I would like to thank my parents, my sister and the rest of my family for their constant support and for being there, every step of my life.

Michael Dylan Dsouza,

February 19, 2015

Contents

1 Introduction **1**

1.1 Invariants . . . 2

1.2 Contributions . . . 7

1.3 Organization of this thesis . . . 8

2 Background **10**

2.1 Notations . . . 10

2.1.1 Unrolling the circuit . . . 11

2.1.2 Conjunctive Normal Form (CNF) . . . 12

2.2 Static invariants . . . 14

2.2.1 Categories . . . 15

2.3 Boolean Constraint Propagation (BCP) . . . 16

2.3.1 Direct and indirect implications using BCP . . . 17

2.3.2 Extended backward learning with BCP . . . 18

2.4 Inductive invariants . . . 19

2.4.1 Generating the signal value database . . . 20

2.4.2 Potential invariants . . . 21

2.4.3 Reasoning about true invariants: assume-then-verify . . . 21

2.5 Untestable fault identification . . . 24

2.6 Non-robust test for path delay faults . . . 25

3 Fast static learning **27**

3.1 Redundant computations 28
3.2 Filter strategies 29
3.2.1 Static filter 30
3.2.2 Dynamic filter 31
3.2.3 Results of filters on single-node learning 32
3.2.4 Two-node learning 35
3.3 Application to untestable fault identification 36
3.4 Experimental results 36
3.4.1 Multi-node invariants 36
3.4.2 Untestable fault identification 37

4 Fast inductive reasoning **40**

4.1 Size of the potential invariant set 40
4.2 A simple static filter 41
4.3 Proving the inductive invariants 42
4.3.1 Dynamic filters for potential true invariants 43
4.3.2 Dynamic filters for false invariants 45
4.4 Potential invariants involving more than 3 nodes 47

5 Experimental results **51**

5.0.1 Multi-node inductive invariants 51
5.0.2 Untestable fault identification 55
5.0.3 Non-robust test for path delay faults 56

6 Conclusion **61**

Bibliography **63**

List of Figures

2.1 A circuit unrolled for three time-frames 12

2.2 Two node static implication 14

2.3 Invariants and the state space 20

2.4 The base case 22

2.5 Inductive step 23

4.1 A cut of size k in the union of two fan-in cones 49

List of Tables

2.1 CNF representation of basic logic gates 13

3.1 Num. RTCs that yield new implications 28

3.2 Effect of varying the static cutoff threshold K 30

3.3 A comparison of different fast learning techniques 33

3.4 Multi-node invariants and untestable faults identified 39

4.1 #Vectors vs size of potential invariants 41

4.2 Effect of filters on the proving process 47

5.1 Multi-node inductive invariants 54

5.2 A comparison of untestable faults identified with different sets of multi-node invariants 55

5.3 Testing path delay faults - ISCAS benchmarks 59

5.4 Testing path delay faults - ITC benchmarks 60

List of Algorithms

3.1 staticFilter(ID) 30
3.2 dynamicFilter(ID) 31
3.3 twoNodeFilter(ID) 35
5.1 Partition size 53

Chapter 1

Introduction

Electronic Design Automation (EDA) of integrated circuits (IC) and systems is an important area of computer engineering with a direct influence on information technology. In accordance with Moore's law [1], the complexity of ICs has been growing exponentially from a few thousand transistors in the initial ICs, a few decades ago, to billions of transistors in modern ICs. Due to tremendous research and development, EDA has been able to support such scaling that has enabled sophisticated, cost effective electronic devices that are part of our modern lives.

EDA, in its essence, provides algorithms and tools to transform a complex IC design from a high level description to its physical implementation. This implementation is then taken through fabrication and manufacturing test. In such a process, the design progresses through many levels of abstraction. These levels include various synthesis and verification stages. The performance of software tools that enable EDA is of extreme importance from a design productivity standpoint. For example, generating a set of test vectors through automatic test pattern generation (ATPG) for very large designs can take weeks or longer. IC manufacturers

are always under pressure to deliver highly complex products of high quality to market, in as less time as possible.

The focus of this thesis is on the basic concept of invariants and its efficient computation. Since invariants capture relations fundamental to the circuit, they have shown to be very useful in various applications such as ATPG, logic optimization, verification, etc., as discussed in the following sections. The knowledge of such non-trivial relations, as captured by invariants, has shown to significantly improve the performance of many EDA techniques.

1.1 Invariants

A digital circuit is an interconnection of logic gates, each of which performs a basic Boolean operation such as logical AND, logical OR, logical NOT, etc. This assembly of gates may also consist of storage elements known as registers, or flip-flops, that store state values at each clock cycle. Asserting logical values on one or more signals in the circuit and propagating these values in the circuit using techniques such as Boolean Constraint Propagation (BCP) or logic implication, facilitates the learning of logical relations between various signals. Such logical relations are known as invariants.

The set of all logic values of the flip-flops in a circuit, at any instance of time, is known as the state of the circuit. The initial state of a circuit in which flip-flops are not constrained can be any one of all possible states. States that are reachable from an unknown initial state constitute the reachable state space and the remaining states are categorized as unreachable. Unreachable states contain a subset of states, which cannot be reached from any initial state, known as invalid states.

Invariants can be classified as either static or inductive invariants. Static invariants are

relations that are always true, irrespective of the circuit state. That is, the relations captured by these invariants hold in both reachable and unreachable state spaces. Inductive invariants, on the other hand, are true only when the circuit is in the reachable state space. These relations may not hold in an unreachable state.

Static implications/invariants have shown to have a positive impact on a wide range of EDA applications such as Automatic Test Pattern Generation (ATPG) [2–4], logic and fault simulation [5], fault diagnosis [6] and silicon debug [7, 8], logic verification [9–12], logic optimization [13, 14], untestable fault identification [15–19], etc. The general idea behind these applications is that non-trivial invariants should be learned to capture relations that are often missed by existing methods.

Although new types of learning have been proposed over the years, the computation time for learning these powerful relations has also been increasing, which may hinder their applicability to large circuits. These new concepts such as indirect implications, extended backward learning, dynamic learning, 16-value logic, reduction lists, and recursive learning were proposed in [2, 20–24], furthering the computation efficiency of invariants. A compact graphical representation to store two-node invariants was proposed in [25] which enabled capturing multiple time-frame information in sequential circuits as well.

Advances made in satisfiability (SAT) solvers (e.g., chaff [26]) have fueled research related to its applications to different EDA problems [10, 27–30]. This line of research has also resulted in various learning strategies that utilize the SAT solver. SAT solvers generally operate on a Boolean formula that is represented in the Conjunctive Normal Form (CNF). Boolean constraint propagation (BCP), a technique used by SAT solvers to propagate Boolean values in a CNF formula, plays a key role in computing implications in a circuit. However, because non-trivial implications are not inherently present in the CNF formula that is directly converted from a circuit, BCP alone is not capable of finding them. Hence, static learn-

ing techniques are needed to learn new implications, which can subsequently be converted to clauses and added to the formula. A CNF formula that is augmented/extended in this manner contains learned relations that can also significantly constrain the search space of SAT solvers. This additional knowledge also enhances correlation between literals, further speeding up BCP and indirectly improving performance of SAT solvers. Static learning can be computationally tedious when it comes to learning multi-node invariants. This thesis presents techniques that can speed up this process.

SAT solvers have been proved to be an effective tool to reason about inductive invariants as well. The notion of assume-then-verify was proposed in [31] to efficiently identify equivalent flip-flop pairs, used in an ATPG based framework for sequential equivalence checking. This was further generalized in [32]. A model based on Binary Decision Diagrams (a data structure used to represent Boolean formulae) was also used in [33], based on the assume-then-verify two time-frame model. The notion of assume-then-verify is based on the principles of mathematical induction with greatest fixed-point iteration. A SAT-based technique was proposed in [34], which extended [33] for inductive equivalence checking when circuits were transformed due to re-timing and re-synthesis optimizations. The notion of kth invariant was introduced in [35], where a Bounded Model Checker (BMC) and an invariant checker, both based on SAT techniques, were jointly used to compute kth invariants. These invariants were then integrated with a SAT solver for sequential equivalence checking. SAT-based bounded model checking that utilized incremental deductive and inductive reasoning was presented in [36]. A technique to find targeted multi-node inductive invariants was introduced in [37] and filters to speedup the proving process were also presented. The invariants were utilized for sequential equivalence checking of hard instances.

Inductive invariants, unlike the static counterpart, can offer additional constraints to the circuit. Inductive invariants are extremely powerful in blocking away a portion of illegal

states, because relations captured by such invariants may not hold in unreachable states. When both static and inductive invariants are added to the CNF formula of the circuit, this additional knowledge enhances the deductive power of the CNF; as a result, not only can it speed up the BCP and improve performance of SAT solvers, but also block away a large portion of the search space due to illegal states.

The power of invariants and their applicability to a variety of EDA areas has triggered research targeting efficient computation of these relations. Inductive invariant reasoning still suffers from the computation time bottleneck, being both computation and resource intensive. The massive amount of potential invariants in case of inductive reasoning along with the limitations posed by SAT solvers are the current obstacles that hinder fast computation of such relations. This thesis aims to alleviate these problems through effective filter techniques that can speed up the fixed point iteration even when the set of potential invariants contain multi-node invariants of different sizes. A technique to focus on specific types of effective multi-node invariants is also presented.

Faults that cannot be detected by any vector or sequence of vectors are known as untestable faults. An ATPG engine would have to exhaust the entire search space in order to find all untestable faults, leading to a severe negative impact on its performance. Therefore, due to its computational complexity, ATPG based methods to identify untestable faults are not feasible for large circuits. Fault independent methods, such as analysis based on conflicts in the circuit [15, 16, 18], that avoid ATPG have been shown to be very effective in identifying untestable faults. STRATEGATE, a sequential ATPG tool, integrates a sequential test generator based on [38, 39] and a sequentially untestable fault identifier based on [18], a conflict based fault independent method that utilizes invariants to maximize impossibilities/conflicts.

An invariant describes an illegal/conflicting value assignment when the clause representing the invariant becomes false. Powerful invariants can help identify many more untestable

faults in any given circuit. As a measure of quality, we utilize the invariants obtained from our methods for untestable fault identification, using the identifier that is part of STRATEGATE. The knowledge of untestable faults in the circuit can in turn speedup the ATPG engine, by avoiding such faults during test pattern generation. Identifying untestable faults that are manifested due to redundancies in the circuit can help logic synthesis in reducing the circuit area.

Due to the complexity of sequential test pattern generation, design for testability (DFT) techniques are commonly used to enhance the testability of sequential designs. Among such techniques, SCAN is currently the most predominant. In a sequential circuit, based on its functionality, not all states in the circuit are reachable. Among such unreachable states, many of them can be illegal. A circuit, as it is, does not contain any structural information about such unreachable states, due to which structural testing for path delay faults in a traditional manner can result in over-testing as shown through experiments in [40]. This over-testing is due to the fact that a significant amount of delay faults are structurally testable, but due to the logical operations of the circuit, may be functionally untestable. Constraining ATPG for broadside transition testing was investigated in [41]. In order to restrict ATPG to generate patterns that are functionally valid, pseudo functional testing methodology was presented in [42]. Path delay fault ATPG by integrating static learning techniques into incremental SAT was presented in [43]. The impact of static implications to effectively constrain the ATPG for pseudo-functional scan tests was presented in [44]. In addition to static invariants, inductive invariants are much more effective in constraining the state space. We demonstrate the effectiveness of static and inductive invariants, considered together, as constraints on the non-robust test for path delay faults with positive results.

1.2 Contributions

As the size and complexity of digital circuits continue to increase, the computational cost of learning powerful relations, as described above, also rises. In this thesis, we propose a novel technique to reduce the computational cost without incurring a significant loss in the quality of learned invariants. The proposed technique is based on an intelligent identification of redundant computations in extended backward learning, achieved by filtering those signals that have little to no chance of yielding additional learning. In large circuits, the proposed filter can, in fact, increase the number of invariants learned within the same time limit, since the filter allows us to explore a much wider search space. Our experimental results demonstrate that this method can be used to find two-node and multi-node invariants for large circuits.

The number of potential invariants drastically increases with the circuit size with a negative impact on the proving time of the assume-then-verify framework. The more the number of invariants to be proved, the more SAT solver calls are needed. We present methods to reduce the number of potential invariants by eliminating those that represent relations that can be statically deduced. Simply limiting the number of potential invariants is not enough. We must also reduce the cost of inductively proving these invariants. To deal with this, we present effective filters that can significantly reduce the number of SAT solver calls and provide a substantial performance boost, even in cases with a diverse set of multi-node potential invariants.

Finally, while exhaustive mining and proving of potential invariants involving up to three node invariants may be feasible to some extent, the cost becomes prohibitive for invariants with more than three nodes. We present techniques to compute specific types of multi-node invariants, with sizes beyond three nodes, that prove to be effective in constraining the CNF.

Therefore, the final set of inductive invariants contain up to k node invariants; $k \leq 10$ in our implementation.

Static and inductive invariants from the techniques described in this thesis are applied for untestable fault identification. Eliminating untestable faults has benefits to both test generation and logic synthesis. It helps to reduce the circuit area and avoid generating vectors for faults that are not testable. This identifier utilizes impossible value combinations, deduced from the invariants, to reason about untestable faults and redundancies. Using experimental results, we show that the multi-node static invariants from our method are able to find the same or more number of untestable faults in comparison to the traditional approach, in a fixed execution time limit of three hours. Additionally, for many of the benchmarks, these invariants helped find many more untestable faults than previously reported results for the identifier based on the same principle. Adding inductive invariants to this set of invariants, further helped identify significantly more untestable faults, demonstrating the effectiveness of inductive invariants and a useful measure of quality for our methods of obtaining these invariants.

The thesis is concluded with experimental results showing the constraining power of inductive invariants in testing path delay faults. We compare the amount of over-testing when the ATPG is subjected to different constraints such as a higher unrolling depth, static invariants and finally static and inductive invariants combined. We show that inductive invariants identify many more path delay faults that are not functionally testable.

1.3 Organization of this thesis

The thesis is organized as follows: Chapter 2 provides an overview of ideas and methods that are essential to the contributions of this thesis. Chapter 3 deals with fast static learning.

It contains a description of our observations, filtering strategies, algorithms and results. The process of pruning the set of potential invariants, techniques to speed up the process of proving these invariants, including different observations and analyses are described in Chapter 4. Experimental results that show the effect of inductive invariants on untestable fault identification and path delay fault testing are presented in Chapter 5. Chapter 6 concludes the thesis.

Chapter 2

Background

The concepts necessary to understand the work presented in this thesis are described in this chapter. Techniques such as unrolling the circuit, converting it to CNF and propagating constraints in the CNF are explained in detail. Various notations and definitions are provided along with the detailed description of the methods used for static learning and inductive reasoning. Finally, ideas related to the ATPG applications such as untestable fault identification and non-robust test for path delay faults, with the use of invariants in such applications, are also explained.

2.1 Notations

The following notations are used hereafter. Logic gates are denoted using uppercase letters. Lower case letters such as v are chosen to represent Boolean values; $v \in \{0, 1\}$. A value assignment such as $G = v$ is represented as G_v. A node or a literal is an atomic formula that represents a Boolean value assignment to a logic gate. The positive phase of a gate, G_1, is represented by the literal G and the literal $\neg G$ denotes the negative phase of the signal, G_0.

Boolean operations on literals G_i, H_j, such as conjunction and disjunction are expressed as $(G_i \wedge H_j)$ and $(G_i \vee H_j)$, respectively. A similar notation is also used to represent invariants. The uppercase letter I is used to represent invariants followed by a lower case letter as subscript to differentiate between invariants (e.g., I_j represents the jth invariant). The set of potential invariants is denoted as P.

2.1.1 Unrolling the circuit

The work presented in this thesis has been implemented on sequential circuits. The set of all output values of flip-flops/ memory elements, at any time, is known as the state of the circuit. A change in at least one value of a memory element of the circuit causes a transition, taking the circuit to a new state. A sequential circuit that has been unrolled for n time-frames, consists of all gates in the circuit replicated n times, essentially a copy of the circuit in each time-frame. In such a circuit, the flip-flops are replaced with buffers connecting adjacent time-frames. With respect to each time-frame, flip-flops from the previous time-frame are treated as psuedo primary inputs(PPI) and signals that drive flip-flops in the next time-frame are treated as pseudo primary outputs (PPO). For the purpose of static learning, we unroll the circuit up to three time-frames as shown in 2.1 and perform static learning of the signals in the middle time-frame. This allows us to capture relations between nodes within the same time-frame as well as nodes that span time-frames. Proving inductive invariants requires single time-frame and two time-frame unrolling. We also utilize a higher unrolling depth in constraining the ATPG for path delay faults.

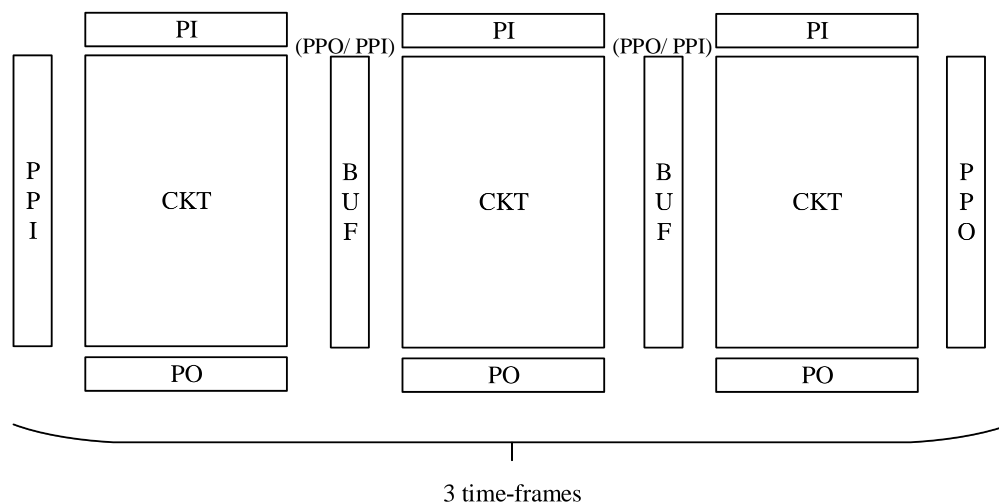

Figure 2.1: A circuit unrolled for three time-frames

2.1.2 Conjunctive Normal Form (CNF)

A logic circuit can be represented as a formula expressed in CNF: a conjunction of clauses, wherein each clause is a disjunction of literals. A clause imposes a constraint on the values of literals such that the clause is always true. Logic gate to CNF transformation is a process of expressing the characteristic function of the gate as a set of constraints. The CNF of the entire circuit is merely a conjunction of the constraints for individual gates. The implementation of the static learning engine and the invariant reasoning engine in this work is based on the CNF representation of digital circuits.

CNF representation of circuits

A logic implication $(G \rightarrow \neg K)$ in its clause form is $(\neg G \vee \neg K)$. This clause also represents a two node static invariant. A logic gate G performs a boolean operation on its inputs to produce an output; $G = f(I)$, where I represents inputs of G. This logic equality holds if both operands are true or both are false. $(G = f(I))$ is same as $(G \rightarrow f(I)) \wedge (f(I) \rightarrow G)$. Therefore, the CNF representation of a logic gate is $(\neg G \vee f(I)) \wedge (\neg f(I) \vee G)$, where *f(I)*

Table 2.1: CNF representation of basic logic gates

Gate Type	**CNF**
NOT	$(G \lor A) \land (\neg G \lor \neg A)$
BUFFER	$(\neg G \lor A) \land (\neg A \lor G)$
OR	$(\neg G \lor A \lor B) \land (G \lor \neg A) \land (G \lor \neg B)$
NOR	$(G \lor A \lor B) \land (\neg G \lor \neg A) \land (\neg G \lor \neg B)$
AND	$(G \lor \neg A \lor \neg B) \land (\neg G \lor A) \land (\neg G \lor B)$
NAND	$(\neg G \lor \neg A \lor \neg B) \land (G \lor A) \land (G \lor B)$
XOR	$(\neg G \lor A \lor B) \land (\neg G \lor \neg A \lor \neg B) \land (G \lor A \lor \neg B) \land (G \lor \neg A \lor B)$
XNOR	$(\neg G \lor A \lor \neg B) \land (\neg G \lor \neg A \lor B) \land (G \lor \neg A \lor \neg B) \land (G \lor A \lor B)$

is the actual boolean operation on the inputs. This formula is then simplified to obtain the disjunction of literals form using the double negation law, De Morgan's laws and the distributive law.

Consider an example of an AND gate G with two inputs $(A,\ B)$. The CNF representation of this AND gate $(G = A \land B)$ is obtained as follows:

$$\begin{aligned} CNF(G) &= (G \rightarrow (A \land B)) \land ((A \land B) \rightarrow G) \\ &= \neg G \lor (A \land B)) \land (\neg A \lor \neg B \lor G) \\ &= \neg G \lor A) \land (\neg G \lor B) \land (\neg A \lor \neg B \lor G) \\ \therefore CNF(G) &= (\neg G \lor A) \land (\neg G \lor B) \land (\neg A \lor \neg B \lor G) \end{aligned} \tag{2.1}$$

Similarly, the CNF representation of every logic gate type can be derived. Table 2.1 shows the final CNF representation for basic gate types, where G is the gate output and $A,\ B$ are the inputs. These CNFs can be extended for any gate with more than two inputs.

2.2 Static invariants

Static invariants, also known as static logic implications, capture the logical relations between various gates in the digital circuit. The terms static and dynamic are associated with logic implications to indicate when the implications were computed. Static implications are computed as a one-time process prior to using them in any application and the relations thus computed are always true for the circuit. On the other hand, dynamic implications are computed during run-time of the application. For example, during the ATPG process, some signals in the circuit may have different values, including the fault effect, which could imply other signals in the circuit.

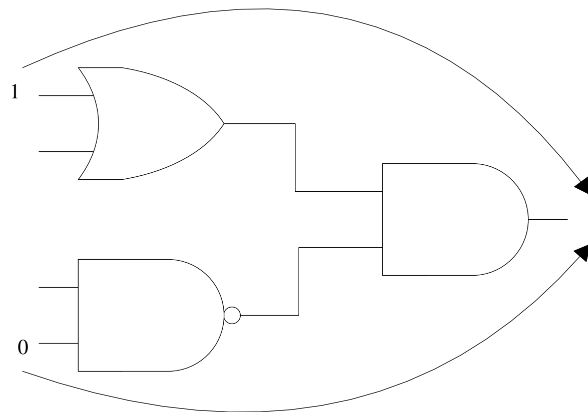

Figure 2.2: Two node static implication

Trivial as well as non-trivial relations can be captured via logic implications. To learn implications of asserting a value of logic zero or logic one on a gate in the circuit, the value is asserted on that gate and is propagated in the circuit. Thus, the implications of setting a logic value of zero or one on that specific gate can be learnt. There are various methods to propagate a value in the circuit, to learn and to store implications.

A single node implication is a relationship between a pair of nodes, with one node (G_i) logically implying another node (H_j), represented as $G_i \rightarrow H_j$. Similarly, relationships

among multiple nodes are captured using multi-node implications, where a set of nodes imply a single node, i.e., $((G_i \wedge H_j) \rightarrow I_k)$. A single node implication $(G_i \rightarrow H_j)$ in the clause form is represented as $(\neg G_i \vee H_j)$, which is a two-node invariant. A multi-node implication of $((G_i \wedge H_j) \rightarrow I_k)$ in the clause form is represented as $(\neg G_i \vee \neg H_j \vee I_k)$, a three node invariant. For example, Figure 2.2 shows a two node trivial implication, where the shown logic values on OR and NAND gate inputs together imply a logic one at the output AND gate, but individually they do not imply any value on the AND gate.

An implication engine that can compute powerful relations in the circuit can have a significant impact on various applications. The following subsections of this chapter will elaborate on the various concepts and ideas related to static logic implications in digital circuits.

2.2.1 Categories

As indicated, static logic implications capture both trivial as well as non-trivial relations in the circuit. Formally, static logic implications are grouped into three categories of direct, indirect and extended backward implications.

Direct implications

Direct implications capture the logical relationships immediately on a circuit gate. So, the direct implications of G_v are logical relations between G and all the fan-in and fan-out gates of G. Since all relations here involve gates directly connected to G and are very easy to compute, they are usually termed as trivial relations.

Indirect implications

Indirect implications extend direct implications. They help capture the logical relation of G_v with other logic gates that may not be directly connected to G. These relations are computed by propagating the set of direct implications in the circuit. Such computation is more involved in comparison with direct implications, but they are still easy to compute, hence included in the set of trivial relations.

Extended backward implications

Extended backward implications capture relations between gates in the circuit that may be seemingly unrelated by trivial inspection. It is based on the exploration of unjustified implied gates in the circuit and are not obtained directly by the propagation of direct or indirect implications. Such relations are non-trivial and have been proved to be extremely valuable through various applications. The technique used to learn such relations are described in the following sections.

2.3 Boolean Constraint Propagation (BCP)

BCP is a process that propagates constraints in the formula representing the circuit. G_v is treated as constraining the literal G to the value v. During this process, literals that are false are deleted from clauses, thus reducing the said clause by the number of false literals. BCP follows two simple rules for every clause of the formula:

- If all literals in a clause except one are set to *false*, and the remaining literal, l, is unassigned, then l is set to *true*. The clause, known as a *unit clause*, is satisfied (*true*).

- If all literals in the clause are assigned *false*, the clause is said to be unsatisfied (*false*).

2.3.1 Direct and indirect implications using BCP

The first step in BCP is placing the initial constraints on literals, which are then propagated. A constraint such as L_1 is propagated by removing every clause that contains L; satisfied clauses. The second part of propagation is to reduce every clause that contains $\neg L$ by deleting the literal $\neg L$. During propagation, all clauses that are reduced to just one literal are called *unit clauses*. The literal values of unit clauses are further propagated. BCP terminates if there are no more constraints to propagate, if any clause is unsatisfied (*false*), or if a literal needs to be assigned both 1 and 0 during BCP, known as a conflict.

Every unit clause that occurs during BCP is an implication of the initial constraints. That is, the set of literals in every unit clause that occurs during BCP of G_v is the implication set of G_v. Since BCP propagates the literal values of all unit clauses, the resulting implication set contains both direct and indirect implications. In comparison to logic simulation, BCP is an efficient method to compute implications since the set of implications from a single call to BCP contains both forward and backward set of direct and indirect implications.

$$(A\vee D)\wedge(B\vee D)\wedge(\neg D\vee\neg A\vee\neg B)\wedge(C\vee E)\wedge(\neg C\vee\neg E)\wedge(\neg F\vee E\vee D)\wedge(\neg E\vee F)\wedge(\neg D\vee F) \tag{2.2}$$

For example, consider the CNF representation of a very simple circuit as shown in Equation 2.2. To obtain the direct and indirect implications of $\neg A$, BCP($\neg A$) is performed. The constraint $\neg A = 1$ is applied to the CNF, which modifies the CNF such that all clauses that contain $\neg A$ are satisfied (true), hence removed and all clauses that contain A are reduced by deleting the literal A. The resulting reduced CNF is shown in Equation 2.3, which has a

unit clause D that needs to be propagated by performing BCP(D).

$$(D) \wedge (B \vee D) \wedge (C \vee E) \wedge (\neg C \vee \neg E) \wedge (\neg F \vee E \vee D) \wedge (\neg E \vee F) \wedge (\neg D \vee F) \tag{2.3}$$

The occurrence of the unit clause D means that $\neg A \rightarrow D$, so D is added to the implication set of $\neg A$. BCP(D) further reduces the CNF to Equation 2.4 with a unit clause F that is added to the implication list of $\neg A$.

$$(C \vee E) \wedge (\neg C \vee \neg E) \wedge (\neg E \vee F) \wedge (F) \tag{2.4}$$

BCP(F) as shown in Equation 2.5 further reduces the CNF, but there are no more unit clauses, hence the list of implications is $\neg A \rightarrow \{D, F\}$.

$$(C \vee E) \wedge (\neg C \vee \neg E) \tag{2.5}$$

2.3.2 Extended backward learning with BCP

Extended backward learning (or recursive learning of depth 1) involves learning new implications that are not obtained directly by the propagation of constraints. Such relations are non-trivial and have been proved to be extremely valuable through various applications. This learning process is more involved in comparison to the simple application of BCP; hence, it is computationally costly. It is based on the exploration of unjustified implied gates in the circuit, which in the CNF translates to reduced two-literal clauses (RTC) formed during BCP.

Consider a clause of the form $(A_i \vee \neg B_j \vee \cdots \vee F_s)$ which is reduced to a clause with only two literals $(A_i \vee \neg B_j)$ during the application of BCP(G_v). The following operations on the

sets of implications result in the set of extended backward implications:

$$\{\{BCP(G_v, A_i)\} \cap \{BCP(G_v, \neg B_j)\}\} - \{BCP(G_v)\} \tag{2.6}$$

The operations shown in Equation 2.6 have to be performed on each and every RTC obtained from BCP, which is clearly expensive. Every new learned implication is added as a new clause. That is, during extended backward learning, if G_v is found to imply Z_k, then the new clause $(\neg G_v \vee Z_k)$ is added to the formula. So, with every new implication learned, the CNF is augmented/extended to include this new information.

This method uses a single level of recursion (depth = 1); the reduced two literal clauses obtained during any BCP in 2.6 are ignored. Multiple levels of recursion can be performed by implementing extended backward learning on reduced two literal clauses obtained during each BCP in 2.6. Multi-level recursion is expensive in terms of computation time, hence it is limited to generally one or two levels. The final set of implications is the set union of direct, indirect and extended backward implications, which can be easily obtained from a single BCP call on the augmented CNF.

2.4 Inductive invariants

Inductive invariants are logical constraints that are always true in the reachable state space, but they may or may not be true if the circuit is in an unreachable state. Figure 2.3 illustrates the validity of invariants across the complete state space. $I_a,\ I_s,\ I_p,\ I_k,\ I_t$ are all inductive invariants. Among these five, I_a holds only in the reachable state space, whereas I_s, I_p, I_k, I_t hold true in the reachable space as well as some portions of the unreachable space. I_x and I_e are static invariants, and they are valid in the entire state space.

As the name suggests, the process of learning inductive constraints finds its basis in mathematical induction. A database of signal values is first generated by logic simulation, which is then searched to find the set of potential invariants. This set is then refined via an induction based fixed-point iteration to eliminate false candidates. At the end of this process, we obtain the set of true inductive invariants. The concepts and the process is explained in the following sections.

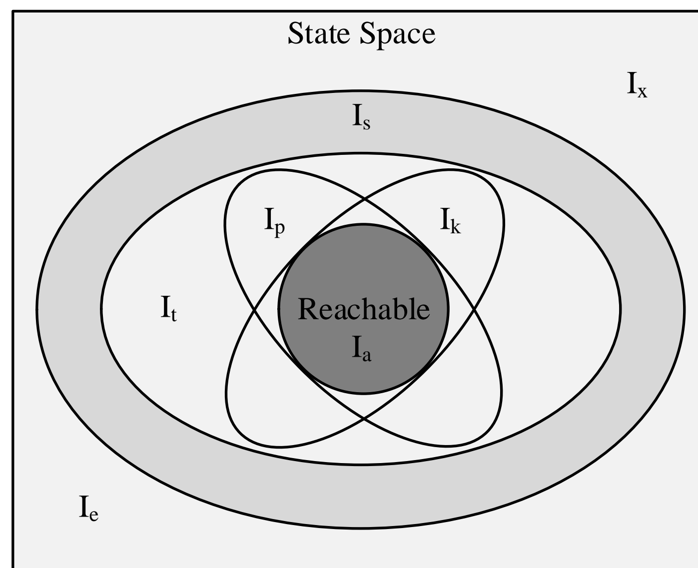

Figure 2.3: Invariants and the state space

2.4.1 Generating the signal value database

The first stage of obtaining inductive invariants is sequential logic simulation. The circuit starts at an unknown state (all flip-flops assigned X). Random vectors are generated and are applied to the circuit. Input vectors are applied till the circuit is fully specified (i.e., no unknown values). This is followed by continued simulation for a predetermined set of (random or user-provided) vectors. The Boolean values of each gate in the circuit for all the vectors is stored in a simulation database.

2.4.2 Potential invariants

Potential invariants are candidate constraints that need to be proved. True inductive invariants are a subset of these potential invariants. Finding the set of potential invariants involves traversing the database in search of missing patterns. For example, if a node is found to have a constant value, it is a single node potential invariant, e.g., if gate A is found to have a constant value of zero in the entire database, $\neg A$ is a potential invariant. Obtaining two node potential invariants requires a database search for missing patterns involving the two said nodes. Since a pair of two boolean nodes can have only four possible values (00, 01, 10, 11), a potential invariant can be formulated for each of these values if they are found to be missing. For instance, let us say that the pattern $M = 0$ and $N = 1$ is missing. The clause $(M \vee \neg N)$ satisfies this observation, since only M_0 and N_1 does not satisfy this clause. $(M \vee \neg N)$ is the corresponding potential invariant. Likewise, finding three or more node potential invariants requires searching for missing patterns that involve that number of signals.

2.4.3 Reasoning about true invariants: assume-then-verify

The process of reasoning about true invariants is based on the principle of mathematical induction, which involves a base case and the inductive step. The assume-then-verify method proves the truthfulness of potential invariants through an iterative fixed point approach. At the core of this approach is a SAT solver. For this implementation, the zChaff SAT solver based on [26] was used. The provision for incremental SAT-solving (ability to add and delete new clauses), a feature in zChaff, is essential for an efficient implementation. The proving process is as follows.

The base case

Let the set of potential invariants be denoted as P and let I_k be the individual potential invariant, where $k \in \mathbb{N}$ and $I_k \in P$. The first step of an inductive proof is the base case. Essentially, the base case aims to answer the following for each potential invariant: given a property ($\neg I_k$), are there any input value assignments such that $\neg I_k$ is true in a given reachable state, S_0? If the property is not true in this reachable state S_0, then it is a false invariant. This problem is formulated as a CNF instance and the underlying SAT solver is used to find a solution.

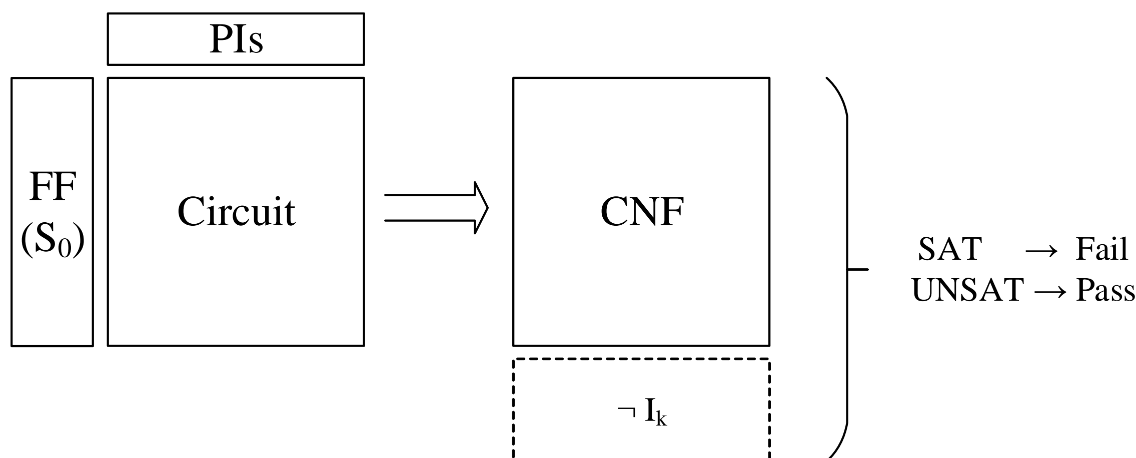

Figure 2.4: The base case

The CNF for a single time-frame of the circuit, constrained with a known reachable state, S_0, is constructed and provided to the SAT solver. S_0 can be taken from any of the states reached from the earlier simulation. For each potential invariant, the negation of the potential invariant ($\neg I_k$) is added to the CNF of the SAT solver as illustrated in Figure 2.4. If the CNF instance is determined to be unsatisfiable ($UNSAT$), I_k passes the base case, because no input combination can violate I_k in state S_0. On the other hand, if the SAT solver finds the instance to be satisfiable (SAT), it means that I_k can be made false in state S_0. Hence, I_k cannot be a true invariant.

Inductive step

The set of potential invariants that pass the base case are considered for the inductive step. The inputs and flip-flops are unconstrained. The circuit is unrolled for two time-frames and the CNF for this unrolled circuit is constructed. The SAT solver now operates on this CNF. All potential invariants (P) are assumed to be true in the first time-frame; the corresponding first time-frame clauses are added to the SAT solver CNF at the beginning of an iteration. Each potential invariant (I_f) is then verified by adding its negated second time-frame clause to the SAT solver CNF. Each second time-frame verification forms an instance, which is solved by the SAT solver, as shown in Figure 2.5. The goal is to eliminate any false invariants by finding a set of input assignments and an initial state such that this new CNF instance is satisfiable.

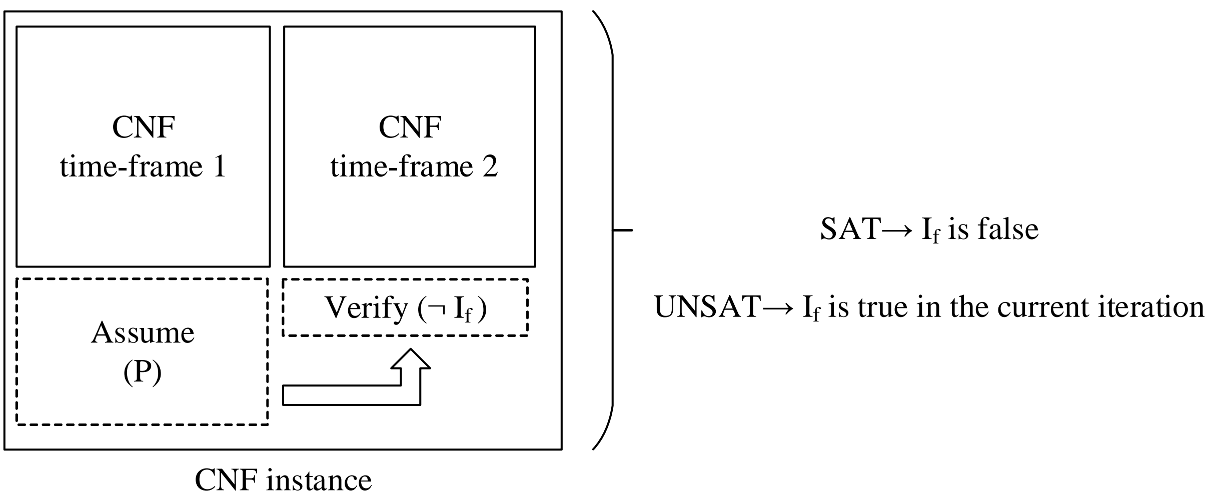

Figure 2.5: Inductive step

If the SAT solver returns *UNSAT*, the potential invariant is considered true for this iteration. *UNSAT* indicates that such a transition is impossible under the current set of constraints in the first time-frame. On the other hand, if the SAT solver finds a satisfiable assignment (*SAT*), then I_f is a false invariant since it is possible to invalidate this constraint from a

state that initially held all the potential invariants. In such a case, this false invariant can be immediately removed form the pool of potential invariants.

The constraints that remain in P, at the end of an iteration, are taken together and assumed in the first time-frame in the next iteration. The process is repeated by negating the second time-frame equivalent of each I_k and then solving that instance. Every subsequent iteration is repeated by refining P via deletion of false invariants. The fixed point is reached when either P becomes empty (no true inductive invariants), or the size of P remains the same between two consecutive iterations. We note that the removed invariants may not necessarily be false, but are not true with the available set of constraints. If more constraints were to be made available to the SAT solver (e.g., multi-node invariants), some of the invariants may be proved to be true.

Clearly, this process can be extremely expensive with respect to the computation time since large circuits can contain hundreds of millions of potential invariants. Techniques to reduced this size before beginning the process of proving them as well as techniques for faster reasoning are discussed in Chapter 4.

2.5 Untestable fault identification

Faults that cannot be detected by any vector or sequence of vectors are known as untestable faults. A fault can be untestable if it cannot be excited, cannot be propagated to a primary output or the conditions to excite the fault and to propagate it to any primary output cannot be achieved together by any set of input vectors. In sequential circuits, some states are not reachable starting from an unknown state. Such unreachable states can also be the cause of untestable faults.

ATPG based methods render untestable fault identification infeasible, due to which fault independent methods, such as conflict based analysis, have been well research and shown to be very effective. The central concept of such conflict based analysis was first introduced in [15] and was extended over the years. The idea is that if a fault necessarily requires conflicting values on some signals in the circuit in order to be detected, then that fault is untestable. [15] was based on conflicts on the same node. That is, faults that require a conflict such as G_0 and G_1 simultaneously for its detection are untestable. This concept can easily be extended to work for a general set of conflicting assignments on n nodes. Since no fault is specifically targeted, it is fault independent.

An invariant describes an illegal/conflicting value assignment when the clause representing the invariant becomes false. The central concept of the identifier used in this thesis is to reason about untestable faults in the sequential circuit using the conflicting assignments, as originally proposed in [16]. Here, we use the conflicting assignments deduced from the static and inductive invariants obtained from our methods.

2.6 Non-robust test for path delay faults

Path delay fault (PDF) is a fault model that aims to capture defects due to process variation. For example, the total delay of combinational paths in the circuit may exceed a certain specified amount. All paths that start from a primary input/flip-flop and end at a primary output/flip-flop should ideally be considered. All gates that are not on the path, but drive on-path gates are known as off-path inputs.

The non-robust test is a method to test path delay faults, wherein each path being tested is assumed to be the only faulty path. As the name suggests, the test is not robust enough because if there are other paths that are faulty, then the transition produced at the source of

the path may not be detected at the destination of the path. Nonetheless, the paths that are shown to be not testable using non-robust test are guaranteed to be non-testable; therefore, we use this as an application to demonstrate the constraining power of invariants.

The test is a two vector test that simulates a transition at the source of the path. The condition for the non-robust test is that all off-path inputs should necessarily take non-controlling values on application of the second vector. This test can be performed using a SAT solver. The circuit is unrolled for two time-frames. For each path being tested, the two vectors that are applied have opposing boolean values on the source node (G) of the path. There can be either a falling transition or a rising transition at G; two tests per path. This is the same as adding unit clauses of the form G and the negation of its second time-frame equivalent $\neg G$, or vice versa, to the SAT CNF instance. The non-controlling values on the off-path inputs of the second time-frame are also unit clause constraints added to the SAT solver instance. All these constraints together form a CNF instance that is solved by the SAT solver. If the solver determines the instance to be $UNSAT$, then the PDF is not testable because the constraints that were imposed on the CNF could not be satisfied. If the instance is SAT, the PDF is testable.

Due to the functionality of the circuit, the PDFs that are determined to be testable may in fact be not testable in the reachable state space, which can contribute to a non-trivial amount of over-testing as demonstrated through recent research findings. Effectively constraining the ATPG has been shown to find such paths that are not functionally testable. The experimental results in this thesis demonstrate the extent of such functionally untestable PDFs that can be determined through powerful inductive invariant constraints.

Chapter 3

Fast static learning

This chapter deals with efficient computation of static invariants. A novel technique to reduce computation cost of static learning is presented in the following sections. In its essence, the technique intelligently identifies redundant computations that may not yield new invariants, thereby achieving significant speedup as shown through experimental results. This chapter is concluded with the results of applying multi-node static invariants for untestable fault identification. In contrast to the traditional approach, results show that many benchmark circuits complete within the fixed time limit with additional new invariants. The remaining circuits, including large benchmarks, learn many more new invariants in the same time duration. Majority of the benchmarks show a variable but small loss in learning. The additional new invariants more than compensate for the loss, enabling us to identify the same or in many cases, more untestable faults.

Table 3.1: Num. RTCs that yield new implications

Circuit	#Gates	#RTCs	#Useful RTCs
s1488	686	82796	2034
s4863	2511	287055	1269
s5378	3042	101492	625
s9234	5866	862957	2998
s13207	8772	992599	3603
s15850	10470	2018185	5274

3.1 Redundant computations

Extended backward learning over multiple nodes, which is the core of this learning engine, involves multiple BCP calls and set operations for every Reduced Two-literal Clause (RTC), rendering exhaustive learning impractical for larger circuits. It is imperative to optimize this process in order to harness the power of implications in various applications. As a first step, we identify redundant computations in the learning process.

Extended backward learning technique is applied to every RTC at the end of BCP. Obtaining new implications of G_v from an RTC $(A_i \vee B_j)$ requires the following three BCP calls: BCP(G_v), BCP(G_v, A_i) and BCP(G_v, B_j). Adding to this cost are the following set operations on the implication lists obtained from the individual BCPs: $\{$BCP$(G_v, A_i) \cap$ BCP$(G_v, B_j)\}$ - $\{$BCP$(G_v)\}$. Since BCP has been extensively researched and refined in state-of-the-art SAT solvers, there is little to be optimized with respect to BCP. The aforementioned operations are all necessary, which leads to the question of how many out of the total RTCs yield new implications?

Table 3.1 shows the number of useful RTCs that occur in a single iteration through the middle time-frame of the unrolled 3-frame circuit with learning performed for each node in the middle frame. An RTC that yields at least one new implication is said to be a *useful*

RTC. For each circuit, the second column lists the total number of gates in the circuit, followed by the total number of RTCs during the entire learning process and the last column lists the total number of useful RTCs.

As shown in this table, the number of useful RTCs is significantly less than the total number of RTCs. This means that performing extended backward learning on all RTCs would result in a lot of wasted computation. Clearly, there is great scope for optimization if the computation on those RTCs that do not lead to new implications can be predicted and eliminated. To address this, we propose several techniques to filter out these redundant computations as described in the following sections.

3.2 Filter strategies

An ideal approach to achieve the desired optimization is to accurately predict if an RTC will yield a new implication and skip extended backward learning of that RTC if needed. Due to the lack of observable properties that can exactly distinguish useful RTCs from those less useful ones, we propose two kinds of filters that closely approximate this effect, based on an observation.

We observed that the same RTCs repeat during the entire duration of the learning procedure, that is, an RTC of the form $(G_i \vee H_j)$ may occur during multiple BCPs of different nodes. Not all of these occurrences yield new implications. Useful RTCs are observed to occur in clusters. That is, they are observed to yield new implications in their first few initial occurrences and the rest of useful occurrences are usually clustered into groups with small, closely spaced intervals between clusters. Based on these observations we propose two kinds of filters; a static fixed cutoff threshold filter and a dynamic filter. We also discuss the effect of combining the two for a better outcome.

Table 3.2: Effect of varying the static cutoff threshold K

Circuit	No Filter		Static (K=80)			Static (K=120)			Static (K=150)		
	#Lrnt	T (s)	%Loss	T (s)	S	%Loss	T (s)	S	%Loss	T (s)	S
s9234	2.95M	579.54	4.74	151.33	3.83	4.71	204.14	2.84	4.69	238.43	2.43
s13207	1.51M	562.53	1.48	186.25	3.02	0.46	248.65	2.26	0.39	290.11	1.94
s15850	2.20M	754.80	1.79	268.17	2.81	1.25	343.93	2.19	0.59	401.17	1.88
s35932	554K	6644.40	0.68	412.92	16.09	0.69	498.72	13.32	0.68	577.57	11.50
s38417	1.27M	1323.63	5.19	398.02	3.33	0.03	547.32	2.42	0.01	663.49	1.99
s38584	7.53M	32237.5	3.04	1946.67	16.56	2.80	2690.72	11.98	2.59	3065.99	10.51

T: Time, S: Speedup, K:1000, M:1000000

In all the proposed filters, every RTC is assigned an ID. Both, $(G_i \vee H_j)$ and $(H_j \vee G_i)$, are treated as the same RTC and are assigned the same ID.

3.2.1 Static filter

As observed, majority of the RTCs always yield new implications in their first few occurrences. So, the static filter processes only the initial K occurrences of an RTC. This approach keeps track of the occurrences of unique RTCs using a counter. On exceeding a fixed cutoff threshold value K, processing/learning for the specific RTC is skipped for the rest of its occurrences as shown in the simple Algorithm 3.1.

Algorithm 3.1 staticFilter(ID)

```
occurrence[ID]++;
if (occurrence[ID] > K) then
  return SKIP;
end if
return DO_NOT_SKIP;
```

However, skipping of RTCs would result in loss of some learned invariants. To mitigate this loss, we consider a dynamic approach towards eliminating wasteful computations as discussed next.

3.2.2 Dynamic filter

The usefulness pattern of RTCs differ from one to another. So, the dynamic filter aims to customize thresholds for every unique RTC during run time. In order to do so, this method keeps track of the most recent occurrence count at which the reduced two literal clause was useful. Every time the RTC occurs, the filter checks for the number of occurrences since the clause was useful. If this difference exceeds a fixed cutoff threshold value T, the processing of that RTC is skipped for the next K occurrences; a skip window. Values of T and K are gradually increased with the occurrences, per RTC.

Algorithm 3.2 dynamicFilter(ID)

```
occurrence[ID]++;
Update values of T and K based on occurrence[ID];
if ((lastLearnt[ID] < LIMIT1) &&
(occurrence[ID] > LIMIT2)) then
   return STOP;
end if
if (occurrence[ID] < threshold[ID]) then
   return SKIP;
else if (occurrence[ID] == threshold[ID]) then
   lastUseful[ID] = threshold[ID];
end if
if ((occurrence[ID] - lastUseful[ID]) >= T) then
   threshold[ID] = threshold[ID] + K;
   return SKIP;
end if
return DO_NOT_SKIP;
```

Algorithm 3.2 describes the steps involved in the dynamic filter. The threshold is ID specific.

The containers named *lastUseful* and *lastLearnt* are updated with the occurrence count value at the time the RTC yields a new implication, which is not shown in the algorithm. If the lastUseful[ID] value is less than the current threshold, it is updated to the threshold value i.e., lastUseful[ID] = max{lastLearnt[ID], threshold[ID]}. The constant LIMIT2 is set based on an observation of usefulness patterns, in that, a significant amount of reduced two literal clauses yield new implications only in their initial few occurrences (LIMIT1) and then are never useful for thousands of occurrences.

Combination of static and dynamic filters : The results, as discussed later, show significant speedup but benefit some circuits more than the others. So, we combine the previous two filters so that all circuits can benefit from a single filter. The combination is such that the static filter is applied first. On exceeding this fixed cutoff threshold, instead of skipping the RTC, we continue with the dynamic filter.

3.2.3 Results of filters on single-node learning

Static learning without the application of filters is used as the basis to analyze the effect of the filters. All the results reported in this thesis were obtained through experiments conducted using a 3.33GHz, Intel(R) Core(TM) i7 CPU with 6118MB of memory, running Linux as the operating system. The algorithms were implemented in C++. All results for static learning are reported for computations involving a single iteration through the middle time-frame of the unrolled circuit.

The efficacy of the results are measured in terms of the number of new implications that were found out of the ones that were learnt without the application of filters. This is listed as the percentage loss in the table. As a performance metric we also list Speedup values in

Table 3.3: A comparison of different fast learning techniques

Circuit	No Filter		Static (K=150)			Dynamic			Combined		
	#Lrnt	T (s)	%Loss	T (s)	S	%Loss	T (s)	S	%Loss	T (s)	S
s9234	2.95M	579.54	4.69	238.43	2.43	0.50	116.92	4.96	0.72	272.42	2.13
s13207	1.51M	562.53	0.39	290.11	1.94	3.15	121.67	4.62	0.38	311.61	1.80
s15850	2.20M	754.80	0.59	401.17	1.88	3.52	174.35	4.33	0.44	432.38	1.75
s35932	554K	6644.40	0.68	577.57	11.50	0	1044.75	6.36	0.68	1467.53	4.53
s38417	1.27M	1323.63	0.01	663.49	1.99	5.27	266.95	4.96	0.01	740.83	1.79
s38584	7.53M	32237.5	2.59	3065.99	10.51	2.59	3633.88	8.87	1.04	6999.80	4.61

T: Time, S: Speedup, K:1000, M:1000000

the table. These performance and quality metrics are computed as follows:

$$
\begin{aligned}
Speedup &= T_{old}/T_{new} \\
\%Loss &= (T - F) * 100/T
\end{aligned}
\tag{3.1}
$$

T_{old} = execution time for learning without filter.

T_{new} = execution time for learning with filter.

T = Total new implications without filter.

F = Total new implications with filter, that were present in the set of implications learnt without filter.

Tables 3.2, 3.3 contain the results of single-node learning using the larger circuits among the ISCAS '89 benchmarks. The second column, in both tables, lists the total number of new sequential implications from static learning followed by a column with the time required for this exhaustive computation. Next, the results of filters are reported. In addition to the

amount of loss in learned invariants as compared to the exhaustive run (no filter), the time and speedup are also reported.

For static filters, smaller values of K resulted in higher loss than shown in Table 3.2. It is clear that increasing the value of K results in a reduction of loss as expected, but the required computation time also increases. The loss in learning even with $K = 150$ is significant in some circuits, albeit reduction in run-time, especially for circuits s9234 and s38584.

The results for the dynamic filter, as shown in Table 3.3, are based on the following parameters: if occurrence[ID] $<$ 500, then T = 30, K = 200 and if occurrence[ID] >= 500, then T = 50, K = 500. The value of LIMIT1 was set to 5 and LIMIT2 to 100. The circuit s9234 shows a drastic reduction in %loss from 4.69% in static filter (K = 150) to just 0.50%, with a reduced computation time. Whereas, circuits s13207, s15850, and s38417 show an increase in %loss, but with a reduced computation time. The circuit s35932 shows no loss in learning compared to the exhaustive learning; although the computation time is higher than the static filter, it is still much faster than exhaustive learning. The performance for s38584 is approximately the same as the static filter with a slight difference in run time. The results from static and dynamic filters do not show a clear trend. Different circuits benefit from different types of filters.

The results for the combined filters are shown in the final columns of Table 3.3. The static cutoff threshold value was set to 150. When the occurrence[ID] count exceeded a value of 150, the dynamic filter was applied with T = 30 and K = 200. The parameters were experimented with an aim towards minimal %loss. The constraints of LIMIT1 and LIMIT2 were not included here due to the initial static fixed cutoff threshold. The results show consistent reduction in %loss and an increased run-time compared to previous results of static and dynamic filters considered individually. The run-time is still significantly less when compared to the exhaustive run.

3.2.4 Two-node learning

Multi-node learning involves asserting boolean values on multiple nodes and then propagating these values. We perform BCP for all possible unique node pairs in the middle time-frame of the unrolled circuit, thus learning three-node invariants. The filter used here, in Algorithm 3.3, is a slightly modified version of Algorithm 3.2 with altered parameters. The number of unique pairs increases quadratically with the size of the circuit, which also substantially increases the number of RTCs and their occurrences. In order to account for this, we gradually alter the parameters T, K and LOW_LIMIT with the increase in value of occurrence[ID]. The STOP_LIMIT and LOW_LIMIT are very helpful to counter the effect of increased redundant computations.

Algorithm 3.3 twoNodeFilter(ID)

```
occurrence[ID]++;
Update values of T, K, and LOW_LIMIT based on occurrence[ID];
if (occurrence[ID] > STOP_LIMIT) | |
(lastLearnt[ID] < LOW_LIMIT) then
   return STOP;
end if
if (occurrence[ID] < threshold[ID]) then
   return SKIP;
else if (occurrence[ID] == threshold[ID]) then
   lastUseful[ID] = threshold[ID];
end if
if ((occurrence[ID] - lastUseful[ID]) >= T) then
   threshold[ID] = threshold[ID] + K;
   return SKIP;
end if
return DO_NOT_SKIP;
```

3.3 Application to untestable fault identification

As a demonstration of the quality of invariants, we applied the learnt invariants to a sequential untestable fault identifier that uses invariants. An invariant describes an illegal/conflicting value assignment when the clause representing the invariant becomes false. The central concept of this identifier is to reason about the untestable faults in the sequential circuit using these conflicting assignments, as proposed initially in [16]. Two-node and three-node invariants were computed separately, but were combined to help identify untestable faults. The results are reported in Table 3.4, which compares the filtered and non-filtered invariants across different ISCAS '89 benchmarks. Additionally, the table compares our results with [18], which also identifies untestable faults using conflicting assignments. It should be noted that our implementation is based on unrolling the benchmarks for 3-time-frames, whereas the number of unrolled time-frames in [18] is listed under the column titled 'TF [18]'. Results are discussed in detail in the next section.

3.4 Experimental results

3.4.1 Multi-node invariants

The experimental results are reported in Table 3.4. Single-node learning was performed prior to two-node learning in the same experiment. The value of STOP_LIMIT was set to 100,000. The other parameters were modified for three different intervals of occurrence[ID] values as follows: $K = 200$, T = 30, LOW_LIMIT = 0, for the occurrence[ID] range of (0, 2000); $K =$ 500, T = 50, LOW_LIMIT = 20, for the occurrence[ID] range of (2000, 10000); and finally, $K = 700$, T = 70, LOW_LIMIT = 5000, for the occurrence[ID] values > 10000. Since the exhaustive multi-node learning is computationally tedious, we set the upper limit on the

execution time to 3 hours. The column titled #Lrnt lists the total number of implications in millions (M) and the column #New contains the total implications learnt from the filtered approach that were not found in the non-filtered, exhaustive approach.

In the non-filtered approach, the search completed only for two small circuits s344 and s349 within 3 hours. Whereas, 13 circuits finish within 3 hours in the filtered approach. The %Loss for majority of the circuits is small and new invariants are considerably large to compensate for the loss. The circuits with #Lrnt and %Loss marked as '#' have a very high number of implications, in the range of billions, which made it impossible to complete the respective computations in reasonable time. Based on the circuit size and the actual available results, approximate #New implications have been listed in the table, marked with an '*'.

3.4.2 Untestable fault identification

In Table 3.4, the column titled 'Unt' lists the total untestable faults that were identified. Both, trivial and learned invariants were provided to the untestable fault identifier. The three-node invariants are powerful enough to have identified more untestable faults than previously reported results, for some of the benchmarks. Adding to these results, fast learning helped identify many more untestable faults compared to the no-filter approach for some of the benchmarks due to the fact that it was able to find many more invariants.

Majority of the circuits, whose results were listed in [18], help identify a higher number of untestable faults with both the approaches. For example, in s526, with only two-node and local three-node invariants, only 11 untestable faults were found by [18]. On the other hand, exhaustive (no-filter) search found nearly 1 million non-trivial 3-node invariants in 3 hours. 22 untestable faults, which is 11 more than [18], were identified with these new invariants.

With the proposed filtering strategy, there is a 0.091% loss from those invariants found by the exhaustive search. However, it was able to find 125K additional invariants since the exhaustive method was limited to 3 hours. The filtering found all these invariants in just less than one hour. With all these invariants, 23 untestable faults were found, which is more than the exhaustive approach. Likewise, in circuits s820, s832, s953, s967, s1488 and s9234, more untestable faults were identified with the invariants found using the proposed filter. In all other circuits, the number of untestable faults were the same for both the no-filter and filter; however, the number of invariants found by the filtering technique is significantly larger. We note that the number of untestable faults found is equal or more than those reported in [18], except for s5378 which identifies 881 untestable faults as opposed to the 884 untestable faults from [18]. However, in [18], 5 time-frames (2 time-frames forward and 2 time-frames backward) were used in this circuit.

Table 3.4: Multi-node invariants and untestable faults identified

Circuit	No-filter			Filtered				Unt	TF
	#Lrnt(M)	Time(h)	Unt	%Loss	#New	Time(h)	Unt	[18]	[18]
s298	1.47	3	33	0.011	905	0.275	**33**	6	9
s344	0.574	0.767	5	0.507	39	0.054	**5**	4	5
s349	0.658	0.759	7	0.359	37	0.054	**7**	6	5
s382	0.275	3	4	0.668	4	0.206	4	-	-
s386	0.584	3	67	0.067	11684	0.387	**67**	63	3
s400	0.442	3	10	0.537	2695	0.247	10	10	3
s444	0.491	3	18	0.308	27384	0.382	18	18	3
s526	0.980	3	22	0.091	125K	0.847	**23**	11	9
s713	1.577	3	38	4.571	305K	0.259	38	38	3
s820	2.704	3	7	1.377	3.39M	3	**13**	-	-
s832	2.514	3	9	2.223	3.591M	3	**16**	4	3
s953	2.096	3	10	5.194	1.195M	3	**10**	-	-
s967	1.476	3	14	4.405	3.012M	3	**17**	-	-
s991	0.522	3	8	5.234	3.372M	0.254	8	-	-
s1196	2.403	3	1	1.201	542K	3	1	1	3
s1238	1.790	3	30	1.573	505K	3	**30**	21	3
s1269	0.607	3	2	0.641	350K	0.855	2	-	-
s1423	0.538	3	15	2.948	322K	1.267	**15**	14	3
s1488	11.589	3	0	17.759	11.01M	3	5	-	-
s1494	12.626	3	4	18.242	6.631M	3	23	-	-
s1512	1.743	3	7	3.741	988K	0.379	7	-	-
s3330	186.975	3	372	0.399	11.11M	3	372	-	-
s4863	13.634	3	126	4.137	10.99M	3	126	-	-
s5378	1485	3	881	0.225	45.6M	3	881	884	5
s9234	2909.8	3	526	#	50M*	3	**538**	-	-
s13207	32593.2	3	992	#	70M*	3	992	-	-
s15850	5785.4	3	502	#	60M*	3	502	-	-
s38417	#	3	832	#	100M*	3	**832**	466	5

* : The values are approximations based on circuit size and available data from other circuits. K : Kilo (1,000), M : Million (1,000,000)
- : Results were not listed in [18]. # : %Loss for these circuits were difficult to determine due to the size of the implications.

Chapter 4

Fast inductive reasoning

This chapter presents filters for fast inductive reasoning. As explained in section 2.4.2 the size of the set of potential invariants depends on the number of missing patterns in the logic simulaton database and increases with the size of the circuit. Techniques for fast inductive reasoning and methods to focus on specific types of multi-node invariants are presented. Their impact on untestable fault identification and effectively constraining the ATPG for path delay faults is demonstrated with positive experimental results.

4.1 Size of the potential invariant set

The number of potential invariants is based on the number of nodes considered and increases with the size of the circuit. The number of patterns that are missing from the database should decrease with an increase in the number of vectors used for simulation. This expected decrease is due to the fact that more states are explored with increase in vectors, leading to more patterns being observed. Table 4.1 shows variation in the number of total single, two and three node missing patterns with increase in database size for two small benchmark

Table 4.1: #Vectors vs size of potential invariants

#Vectors	#Potential invariants	
	s298	s526
500	2177918	12000550
1000	2006094	11218777
5000	1736989	11156290
10000	1676409	11138314
20000	1636886	11144744

circuits. As seen from the table, the reduction is not significant beyond 10000 vectors. s298 is a small circuit with only 14 flip-flops, and yet we already have more than one million potential invariants. Similarly, s526 has only 21 flip-flops. Larger circuits will have substantially larger sets of potential invariants.

The amount of potential invariants prohibit exhaustive mining and proving all such invariants. Many of these potential invariants can be eliminated through structural relations already contained in the CNF. Static learning can benefit this elimination process through the non-trivial relations added to the CNF. The process is as follows.

4.2 A simple static filter

Consider a three node potential invariant $I_k = (A \vee \neg G \vee P)$ corresponding to the missing pattern $AGP = 010$. The following checks based on two node static invariants can be performed on such a relation to know if it is already known: $\neg A \rightarrow \neg G$, $\neg A \rightarrow P$, $G \rightarrow A$, $G \rightarrow P$, $\neg P \rightarrow A$ and $\neg P \rightarrow \neg G$. Each of the implications can be obtained via BCP. The next set of checks involves three node static invariants as follows: $(\neg A \wedge G) \rightarrow P$, $(\neg A \wedge \neg P) \rightarrow \neg G$ and $(G \wedge \neg P) \rightarrow A$. If any of the implication relations is found to be true,

the invariant I_k is already known to be a static invariant and can be removed from the set of potential invariants.

In order to best utilize the filter, we perform static learning on single nodes first. Non-trivial two node static invariants are added to the CNF. This is followed by inductive reasoning on single and two-node invariants. The true inductive invariants are then added to the CNF before finding multi-node potential invariants. This aids the filter in removing additional unnecessary potential invariants from the set through the above mentioned static filters. The false invariants from the single and two node potential invariant reasoning are carried over to the set of multi-node potential invariants to be proven together. Since the set was not constrained enough, many true single and two node inductive invariants might have been proved to be false. So, carrying over these invariants to the set of multi-node invariants can result in some of them being proved true and can potentially speedup the process of reasoning about these multi-node invariants with the help of filters presented in the following sections.

4.3 Proving the inductive invariants

The process of proving an inductive invariant involves verifying invariants one at a time. This process iteratively removes false invariants until a fixed point is reached. The resultant set is the set of true inductive invariants as described in Section 2.4.3. Since the core of this process involves repeated calls to a SAT solver, we will want to reduce the number of calls to the SAT solver as much as possible. Although a faster SAT solver can greatly benefit the process, reducing the number of SAT solver calls can prove to be much more advantageous.

To do so, structural relations known through the static implications via BCP are utilized in a *dynamic* manner. Static learning performed prior to this step aids in the filtering process and also helps the SAT solver through the augmented CNF. Proving single and two node

invariants, followed by adding the true invariants to the CNF can help the proving process for multi-node invariants.

4.3.1 Dynamic filters for potential true invariants

The techniques discussed in this section are applied when a potential invariant is found to be true in any iteration prior to reaching the fixed point. Consider a set of potential invariants P with the following three node invariants as part of this set:

- $I_j : (W \vee \neg B \vee J)$
- $I_k : (X \vee \neg B \vee J)$
- $I_l : (E \vee \neg B \vee J)$

Let us suppose that during some iteration t, we find that I_j is true because the SAT solver returned *UNSAT*. If $(W \rightarrow X)$ and $(W \rightarrow E)$, then there is no need to check I_k and I_l with the SAT solver, since they will also be true as long as I_j is true. This is based on the following transitive relation: $(B \wedge \neg J) \rightarrow W$ and $(W \rightarrow X)$, $(W \rightarrow E)$, therefore $(B \wedge \neg J) \rightarrow X$ and $(B \wedge \neg J) \rightarrow E$. In this example, such a relation saved two SAT-solver calls. The same transitive relation check can be performed for invariants with any number of nodes. This simple filter, and the filter described in the next subsection for false invariants, were first presented in [37]. The described filter requires that in order to skip solving for an invariant, $n-1$ literals must be the same compared with I_j. This comparison can prove to be expensive for a diverse set of potential invariants with varying number of nodes, therefore restricting the use of structural relations.

This filter can be further strengthened in the following manner to fully utilize implications via BCP. Consider the same potential invariant set P as above. In addition to the three

invariants, let there be an invariant $I_r : (L \vee R \vee \neg Z)$. None of these nodes match with any of I_j, but if $(W \rightarrow R)$, $(\neg B \rightarrow L)$ and $(J \rightarrow \neg Z)$, then whenever I_j is true, I_r is true. I_r is also true if $(W \rightarrow \neg Z)$, $(\neg B \rightarrow \neg Z)$ and $(J \rightarrow \neg Z)$, as long as I_j is true. This is based on the fact that an invariant is true if at least one of the literals in the invariant is true. So, if each literal of a true clause I_a implies at least one literal, same or different, in another clause I_d, then I_d is also true as long as I_a holds true. This enables finding relations between true clauses with the same or different number of literals, which proves to be useful, especially when the total set of potential invariants contains clauses with variable number of nodes. This filter provides additional speedup by finding many more equivalent clauses and it covers the previous filter as well.

Dynamic filters are applied per iteration. That is, all clauses found to be true using the dynamic filter are true only in that iteration. But, since I_j being true means that I_k, I_l and I_r are true, these three potential invariants need not be checked using the SAT solver in the inductive step, for all subsequent iterations till I_j is proved to be false. This prevents unnecessary checks during the filtering process, further speeding up the iterative fixed point approach. Since I_j being false does not imply anything on I_k, I_l and I_r through the implication relation, they are still considered in the 'assume' phase of the process. I_j being false brings the three eliminated potential invariants back to the pool of invariants that need to be verified.

In case the set of potential invariants are partitioned, this dynamic filter can be used to find true invariants in other subgroups after the fixed point for a subgroup is reached. The fixed point contains all true invariants that belong to the subgroup. Consider an example with the same potential invariants I_j, I_k, I_l and I_r. Let us say that the set is partitioned such that I_j is in one of the subgroups (S_n) and I_k, I_l, I_r, are all in other subgroups $(S_p$, where $p > n)$. If I_j exists in the fixed point of S_n, then I_j is a true invariant. From the discussion

about the filters in this section, we know that I_k, I_l and I_r will always be true. I_k, I_l and I_r do not offer any additional constraints nor do they help the process in any way, so they can be removed from the pool of invariants. Once the fixed point is reached in a subgroup, the true invariants are permanently added to the CNF before proceeding with the iterations for the next subgroup.

4.3.2 Dynamic filters for false invariants

The SAT solver returns SATISFIABLE if an invariant is false, which suggests that the CNF instance with all invariants P in the first time-frame and $\neg I_q$ in the second time-frame is satisfiable. Every satisfiable instance has an assignment to all literals that make that instance satisfiable. This set of literal values is known as the *witness*. In case a potential invariant is found to be false, it is definitely false in all iterations and can be immediately removed from the pool of potential invariants. The witness for the satisfiable instance can be obtained from the SAT solver, which can be used to find other potential invariants that are false. Any other potential invariant I_n for which I_n and its second time-frame equivalent $\neg I_n$, that is satisfied by the witness is also a false invariant.

If the set of potential invariants is partitioned, then this filter can be used to eliminate false invariants not present in its subgroup. The first step involves verifying if the witness obtained is, in fact, a true witness. The following two cases are observed when dealing with partitions.

1. The witness violates at least one constraint in another subgroup.

2. The witness satisfies all constraints.

The witness is said to violate a constraint if it does not satisfy the first time-frame equivalent

of the constraint. This is due to the fact that the set of potential invariants was partitioned, so the witness was obtained for only a subset of the total constraints. If the witness satisfies case 2, it can be used to eliminate false invariants in all subgroups. If the witness belongs to case 1, it can only be used to eliminate false invariants within the subgroup.

Results of the dynamic filter

Table 4.2 compares the time required for the proving process with and without the application of dynamic filters. The potential invariants involve only single node and two node potential invariants. The two node potential invariants were limited to the first 100,000 on applying the static filter (Section 4.2). The first column lists the circuit, followed by the total potential invariants and the third column shows the number of invariants that were proved to be true out of the total potential invariants. The computation times, as listed in columns four and five, include the total time required for both the base case and the inductive step. The filters, when applied, speedup both the base case and the inductive step. As seen from the table, small circuits such as s382, s400, s444 and s526 require more than an hour to reason about the set of single and two node potential invariants, whereas all of these circuits complete the process in <= 66 seconds on application of the filters. The filtered approach required only 47 seconds in case of s1423 in contrast to 2347.76 seconds for the non-filtered approach. All circuits show tremendous improvement in performance.

s298, among the simplest ISCAS '89 circuits with 142 gates (14 flip-flops), required 420 seconds to complete the proving process for 424000 potential invariants, on application of the filters. This set included up to 10 node invariants, with the 3 node potential invariants limited to 400000. The attempt to prove this set of potential invariants without any filters did not complete in 3 hours and was terminated. In both cases, the set of potential invariants were partitioned into subgroups of 40,000 potential invariants.

Table 4.2: Effect of filters on the proving process

Circuit	#P	#T	Without filters	With filters
			Time(s)	Time(s)
s298	9024	1173	7.11	0.36
s344	25010	79	23.82	0.98
s382	39626	726	*TO1	**17.94**
s386	28514	0	30.26	1.056
s400	41623	1395	*TO1	**20.28**
s444	48400	1761	*TO1	**26.45**
s526	56552	3484	*TO1	**65.51**
s641	100179	23572	198.99	7.29
s713	100183	23764	201.87	7.35
s820	100239	193	622.89	13.38
s832	100236	186	654.65	13.96
s1196	100348	15	75.79	6.03
s1238	100353	16	77.42	6.16
s1423	100275	944	2347.76	**47.84**
s298*	424100	17608	*TO2	**420**

#P = total potential invariants, #T = total true inductive invariants
s298* : The case with multi-node invariants up to 10 nodes
Time includes both base case and inductive step
*TO1(time-out) = 1 hour, *TO2 = 3 hours

Clearly, without such filters, proving large sets of multi-node inductive invariants even for small circuits can be infeasible.

4.4 Potential invariants involving more than 3 nodes

Thus far, single node, two node and three node potential invariants have been mined from the simulation database and refined using static filters as described in Section 4.2. Mining

invariants with more than three nodes and reasoning about them is expensive. Since the number of true invariants that are found in a given set depends on the effectiveness of the available constraints, it is beneficial to constrain the set as strongly as possible with larger-sized multi-node invariants. The number of potential invariants involving more than three nodes can be significantly high, so we restrict the size of such potential invariants by narrowing down the search to focus on patterns involving only flip-flop nodes and nodes influenced by flip-flops.

As flip-flops control the state space of the circuit, some states may be reachable, while some may not. Among the reachable states, there could also be states that are hard to reach. Invariants involving flip-flops can be very effective in constraining the SAT solver instance, providing effective guidance towards reasoning about inductive invariants. In order to obtain potential invariants with flip-flops, we focus on the false two-node invariants from the first stage of inductive reasoning; the stage involving only single and two node inductive invariants, as described in Section 4.2.

False invariants capture relations through patterns that were not observed in the simulation database, but are actually achievable. It can be reasoned that such relations are not observed because the states in which such patterns are not missing are are either impossible or difficult to attain. Such states can be found by tracing the fan-in cone of the nodes present in false invariants [37]. The flip-flops present in the fan-in cones of such nodes directly control the values these nodes can take. Therefore, missing patterns involving these flip-flops are the states that are either hard-to-reach or unreachable. Invariants involving these flip-flops can be very effective in constraining the CNF instance given to the SAT solver and if proved to be true, capture powerful multi-node relations.

In order to obtain such large-sized multi-node invariants, the fan-in cone of each node in a two-node false invariant ($G_1 \vee G_2$) is computed. The union of the flip-flops present in

these two cones are considered as candidate set of flip-flops for which potential invariants are mined. To reduce the number of potential multi-node invariants and for a faster search, we restrict the maximum number of flip-flops in the union of the two fan-in cones to $k = 10$. Since this search is performed for all false two-node inductive invariants, the sets of flip-flops can contain redundant information such as repetitions or subsets, which are discarded.

We extend the technique of obtaining multi-node invariants, presented in [37], as follows. In case the number of flip-flops in the union of the two fan-in cones exceeds the threshold k, we obtain a cut of k nodes in the fan-in cone. Starting from the flip-flops, the fan-out gates present in the union of the cones are traced until the set of fan-outs is less than or equal to k, as illustrated in Figure 4.1. This ensures that information about the hard to reach/impossible states is not completely lost, and is captured via nodes that are influenced by these states. Finally, potential invariants involving all flip-flop nodes (size $<= k$) and nodes present in the cut of the union of the two fan-in cones are obtained. This means that there can be up to k node potential invariants in the set of multi-node potential invariants.

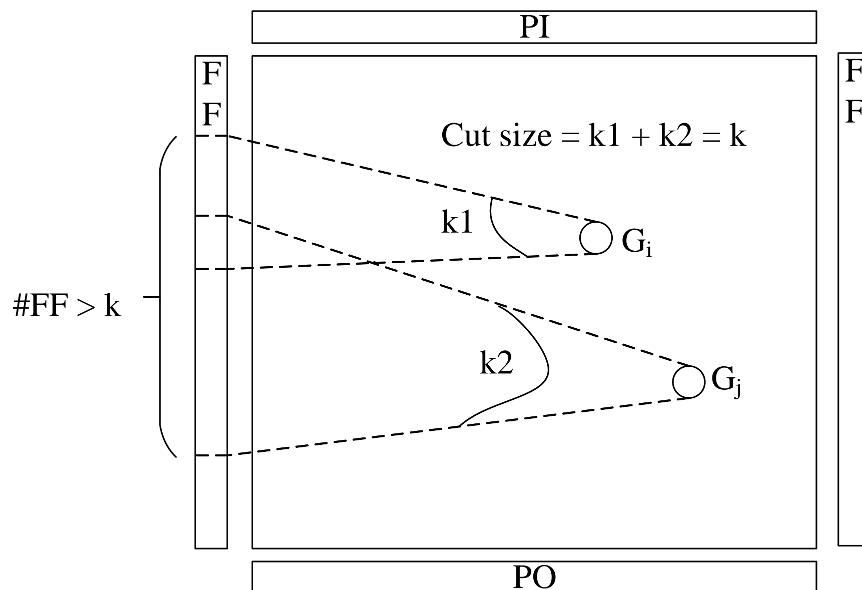

Figure 4.1: A cut of size k in the union of two fan-in cones

To benefit from the dynamic filters, we organize the set of potential invariants such that single node, two-node and three-node potential invariants are placed before other multi-node invariants in the set. Any invariant, when true, can potentially imply other invariants that are yet to be verified as per the dynamic filter discussed in Section 4.3.1. Invariants, when false, can be used to eliminate other invariants in the list using the witness obtained from the SAT solver as described in Section 4.3.2. True invariants are a small subset of the total potential invariants. Smaller invariants (≤ 3 nodes) that constitute majority of the potential invariants also contain a large portion of the total false invariants. If these invariants are proved false earlier, then the invariants yet to be verified can potentially be eliminated as false using the witness, as presented in Section 4.3.2.

The experimental results are presented in the following chapter.

Chapter 5

Experimental results

The results were obtained through experiments implemented in C++ on a 3.33GHz, Intel(R) Core(TM) i7 CPU with 6118MB of memory, running Linux as the operating system.

5.0.1 Multi-node inductive invariants

Due to the enormity of the size of two node and three node potential invariants, we limited them to a fixed size. Experiments were performed by varying the parameters such as size of the set of two node, three node potential invariants and size of the flip-flops, size of the cut in the union of fan-in cones, all based on the time required to complete the proving process. Since the two node and three node potential invariants were limited to a fixed size, we also experimented with different ways to obtain the fixed sized potential invariants. The approach towards obtaining the results are as follows.

Single node and two node static invariants were computed separately via extended backward learning using BCP. The newly learnt non-trivial invariants were added to the CNF, thus

upgrading it to contain this new information. Multi-node inductive invariants up to k nodes were computed in the following three phases using the techniques proposed in this thesis.

- Single node and two node invariants were proved first, with true inductive invariants added to the CNF.
- The false invariants from the previous phase + first N three node invariants obtained via *forward* search in the database + multi-node invariants are proved together. True invariants were added to the CNF.
- The false invariants from the previous phases + first N three node invariants obtained via *backward* search in the database are proved together and the true ones were added to the CNF.

Since the number of all possible two node and three node potential invariants are extremely high even for small circuits, we limited the size of this set to N statically filtered potential invariants. Since limiting the set to first N potential invariants presents a risk of missing out on interesting invariants that may be encountered later on through exhaustive search, we search in both the forward direction and backward direction to find potential invariants with a predetermined limit on the size (N).

Forward search involves starting from the node in the least level and the search progresses forward with nodes in higher levels. Similarly, backward search starts at the highest level and progresses towards nodes in lower levels. The value of N was varied based on the time required for the particular circuit. Circuits that complete the proving process faster were run with a higher limit of N. Adding true invariants after each phase benefits the static filter in eliminating invariants that can be statically deduced from the updated CNF.

Considering the amount of potential invariants, proving them all at once is not feasible. So,

we partition the set of potential invariants in the following manner before starting with the iterative proving process:

Algorithm 5.1 Partition size

```
if (# P ≥ 1000000) then
    PARTITION_SIZE = 100000;
else if (# P ≥ 500000) then
    PARTITION_SIZE = # P / 10;
else if (# P ≥ 100000) then
    PARTITION_SIZE = # P / 5;
else if (# P ≥ 50000) then
    PARTITION_SIZE = # P / 3;
else
    PARTITION_SIZE = # P;
end if
```

Note: #P = size of the potential invariant set.

Table 5.1 shows the results for several ISCAS '89 and ITC '99 circuits. The circuits with total potential invariants $\geq 1M$, except for s4863, were computed with $N = 1M$ set for both two-node and three-node potential invariants. Two node potential invariants did not exceed this limit. As seen in the table, proving time for such circuits is extremely fast considering the amount of potential invariants.

Many circuits, including a few smaller ones required a higher computation time when the size of the set was large. So, we impose a more strict limit on such circuits. The results for circuits s382, s400, s444, s526, s1423, s1488, s1494 and s4863 were computed with a search limit of 200K potential invariants in the forward direction and 200K in the backward direction. Circuits b07, b12, b13 and b14 were further limited to only 100K two node invariants and 20K three node potential invariants in the forward direction along with a limit of $k = 4$ on the number of flip-flops and size of the cut in the fan-in cone.

Circuit b15 was subjected to a limit of 100K potential two-node invariants and 20K three node potential invariants in the forward and backward directions each. A limit of 500K

Table 5.1: Multi-node inductive invariants

Circuit	#P	#True				Time	Circuit	#P	#True				Time
		1n	2n	3n	> 3n	(m)			1n	2n	3n	> 3n	(m)
s298	1M	1	1172	19787	12320	15	b01	59.1K	0	0	724	14	< 1
s344	2M	0	80	2160	1038	45	b02	7K	0	0	120	8	< 1
s349	2M	0	80	2172	1038	45	b03	2M	57	18609	12339	763	39
s382	440K	0	726	869	671	192	b04	2.2M	0	203	462	69730	51
s386	2M	0	0	19550	51	33	b05	301K	96	42295	31	530	239
s400	442K	2	1393	830	671	82	b06	183K	0	48	3441	321	0.5
s444	449K	2	1759	741	872	106	b07	201K	41	21554	360	0	192
s526	465K	5	3479	1885	7838	125	b08	2M	0	653	16874	23415	303
s641	2.1M	55	47382	216	0	27	b09	2M	9	3610	11000	2883	76
s713	2.1M	56	47859	243	0	26	b10	2M	0	1527	11867	20	55
s820	2.1M	0	217	3743	7	48	b11	2.1M	8	7419	6215	0	206
s832	2.1M	0	209	3884	7	49.8	b12	125K	2	1539	0	3705	331
s1196	2.3M	0	113	4322	23015	39	b13	120K	39	21400	0	0	86
s1238	2.3M	0	95	5012	21997	54	b14	148K	115	3910	0	23879	7
s1423	841K	4	8435	8	675	146	b15	127K	111	1770	2	18	163
s1488	763K	0	2089	3044	16	93	b20	906K	206	14773	16	0	259
s1494	758K	0	2015	3013	16	96	b21	906K	230	32821	2	0	234
s4863	1.4M	0	3592	36	156	178	b22	910K	300	14034	0	0	205
s13207	605K	3184	118K	29	231	162	s15850	607K	916	23757	24	1117	252
s38417	306K	21	294	53	4350	1205							

Time includes total time for finding invariants, the base case and the inductive step
K: 1000, M: 1000000, n: nodes, m: minutes, #P = total potential invariants

two node potential invariants and 200K potential invariants in the forward and backward directions each were set for circuits b20, b21 with an exception of circuit b22 where the three node potential invariants were limited to 200K in the forward direction only.

Table 5.1 clearly shows that the number of true invariants is a very small percentage of the total potential invariants. Many large circuits such as b15, b20, b21, b22 have an insignificant amount of true invariants. The reason for this is that the CNF was not constrained enough because the potential invariants were only a very small subset of the total possible multi-node

Table 5.2: A comparison of untestable faults identified with different sets of multi-node invariants

Circuit	Unt [18]	Static Unt	Static + inductive Unt	Circuit	Unt [18]	Static Unt	Static + inductive Unt
s298	6	33	33	b01	0	0	0
s344	4	5	5	b02	0	0	0
s349	6	7	7	b03	18	22	**116**
s382	-	4	**19**	b04	123	124	**127**
s386	63	67	67	b05	82	88	**366**
s400	10	10	**26**	b06	10	11	11
s444	18	18	**34**	b07	3	3	**83**
s526	11	23	**82**	b08	0	0	0
s641	-	0	**55**	b09	9	12	**30**
s713	38	38	**97**	b10	1	3	**11**
s820	-	13	**27**	b11	9	13	**31**
s832	4	16	**32**	b12	3	13	**17**
s1196	1	1	1	b13	16	18	**90**
s1238	21	30	30	b20	745	750	**1077**
s1423	14	15	**16**	b21	1086	1088	**1460**
s1488	-	5	**13**	b22	1060	1072	**1556**
s1494	-	23	**25**	s4863	-	126	126

- : Results were not listed in [18].

potential invariants. The following sections present the results for the two ATPG problems using the inductive invariants from Table 5.1.

5.0.2 Untestable fault identification

Table 5.2 compares the number of untestable faults identified with different cases. Columns two and three list the results from [18]. Although results for ITC '99 benchmarks were not presented in [18], the same identifier was used to generate the results for the ITC '99 benchmarks listed in the table.

As seen from Section 3.4.2, static invariants in many circuits identified a higher number of untestable faults through fast learning in comparison to the traditional approach. Here, we

compare the number of untestable faults identified via static invariants from fast learning with the case when inductive invariants are added to this set. All cases show improvement in results when compared with the results from [18]. Majority of the circuits help identify many more untestable faults when both inductive and static invariants are used together in comparison to using only static invariants. For example, consider the b05 benchmark. The case with inductive invariants helped identify 366 untestable faults in comparison to 88 untestable faults identified with only static invariants. Similarly, in s641, static invariants did not help identify any untestable faults, but with inductive invariants 55 untestable faults were identified. Large circuits such as b20, b21, and b22 also greatly benefit from inductive invariants. Clearly, inductive invariants capture relations of great value that can significantly impact applications such as untestable fault identification.

5.0.3 Non-robust test for path delay faults

The learnt invariants were applied to constrain the ATPG during the non-robust test for path delay faults as explained in Section 2.6. Experiments were performed by varying the additional constraints on the CNF apart from the path constraints required for the test. The following cases were considered.

1. Unconstrained CNF
2. Circuit is unrolled for k time-frames and non-robust test is performed in the last two time-frames.
3. Non-trivial multi-node static invariants as new constraints.
4. Non-trivial multi-node static invariants + multi-node inductive invariants as new constraints.

In case 2, we unroll the circuit for k time-frames. The additional k - 2 time-frames serve as constraints to the ATPG. We show the results for $k = 6$ in Tables 5.3, 5.4. Higher values of k require a higher ATPG computation time. In case 3, we performed static learning for up to 3 node invariants with an upper limit of 3 hours on the execution time. Case 4 combines these static invariants with the inductive invariants from Table 5.1. For small circuits, all paths were considered, but it is not feasible to do so in case of larger circuits. So, we limited the search for paths to a fixed amount of longest paths in the circuit. There were cases (large circuits) when all paths in this subset were found to be untestable because of the length of the paths and the constraints imposed. Due to this, for all such circuits, we also included shorter paths, which serves the purpose of this experiment.

Tables 5.3, 5.4 compares the results for all cases. As seen in these tables, constraining the ATPG is definitely beneficial for testing path delay faults. The columns titled %FP list the total paths that were proved false or untestable during the ATPG. An increase in %FP indicates that the constraints imposed on ATPG made these additional paths untestable. Since these constraints were obtained from the circuit through higher unrolling, or static and inductive invariants, these additional untestable path delay faults are functionally untestable while structurally testable. Testing for such paths clearly would lead to non-trivial amount of over-testing.

Inductive invariants are clearly beneficial when it comes to effectively constraining an ATPG system. Consider the large circuits, such as s13207, s15850 from the ISCAS benchmark suite, as shown in Table 5.3. No significant additional false paths were identified through constraints from case 1, case 2 and case 3, but addition of inductive invariants increased the number of false paths. In s13207, the number of false paths identified increased from around 53% to 99% and in case of s15850, from approximately 84% to 93%. Similarly many circuits show tremendous improvement, but there are some circuits that do not benefit from such

constraints. This is due to the limitations imposed while computing the constraints/invariants.

The ITC benchmarks also show similar results. Circuits b03, b05, b07, b09 etc., all that are highlighted in Table 5.4 show increased false paths that were identified when the ATPG was constrained with inductive invariants. Larger circuits such as b15, shows a similar trend where cases 1, 2 and 3 showed no different in the %FP, but case 4 increased %FP from 79.10% to 87.31%.

The time required for testing all paths also varies with the different cases. As seen from the tables, increased constraints implies a larger CNF which affects the SAT solver performance leading to a higher ATPG computation time.

In summary, from the two ATPG applications, we see that invariants capture powerful relations that can aid such applications. Static invariants alone does show benefits, which can be significantly enhanced when they are combined with inductive invariants. The approaches presented in this thesis allow faster computation of such relations, due to which many more invariants can be deduced in any given time limit, thus increasing the impact on various applications that benefit from invariants.

Table 5.3: Testing path delay faults - ISCAS benchmarks

Ckt	#pdf	Case 1		Case 2		Case 3		Case 4	
				$k = 6$					
		%FP	T(s)	%FP	T(s)	%FP	T(s)	%FP	T(s)
s298	462	48.92	0.03	55.84	0.07	59.09	0.26	**59.09**	0.79
s344	710	24.65	0.05	41.41	0.14	42.96	0.17	**43.24**	0.27
s349	730	26.31	0.05	42.74	0.14	44.11	0.17	44.38	0.26
s382	800	49.50	0.05	49.5	0.15	49.5	0.34	**54.13**	0.42
s386	414	35.51	0.03	39.13	0.09	39.13	0.32	39.13	0.54
s400	896	53.68	0.06	53.68	0.17	53.68	0.43	**58.82**	0.52
s444	1070	59.25	0.08	59.25	0.23	59.25	0.63	**63.55**	0.75
s526	820	63.41	0.06	63.54	0.18	63.66	0.8	**65.24**	1.19
s641	3488	29.53	0.64	29.53	1.63	29.53	4.18	**60.95**	10.9
s713	43624	88.48	69.61	88.48	80.16	88.48	130.7	**94.62**	210.73
s820	984	25.71	0.16	25.71	0.49	25.71	5.01	25.71	5.66
s832	1012	27.47	0.18	27.57	0.52	27.57	4.85	27.57	5.56
s1196	6196	39.45	1.73	39.44	4.1	39.45	24.8	39.44	38.97
s1238	7118	48.03	2	48.03	4.53	48.03	28.94	48.03	44.47
s1423	89452	61.99	445.05	64.89	506.84	61.99	644.8	**77.05**	685.87
s1488	1924	17.57	0.76	17.57	2.87	17.57	11.98	17.57	15.87
s1494	1952	18.39	0.78	18.39	2.56	18.39	11.97	18.39	15.33
s4863	38038	98.06	104.46	98.06	168.70	98.06	452.38	98.06	497.96
s13207	22022	52.56	92.39	53.12	354.04	53.09	274.58	**99.98**	523.61
s15850	53064	84.62	317.82	84.97	818.43	84.97	548.77	**93.64**	1028.79
s38417	175592	52.44	2969.3	61.787	8819.87	54.39	4181.58	54.57	4964.06

Time includes the total time required to test all PDFs listed under #pdf.
%FP (False Paths): Total paths that were proved false/untestable

Table 5.4: Testing path delay faults - ITC benchmarks

Ckt	#pdf	Case 1		Case 2 $k = 6$		Case 3		Case 4	
		%FP	T(s)	%FP	T(s)	%FP	T(s)	%FP	T(s)
b01	182	13.19	0.01	13.19	0.02	13.19	0.02	13.19	0.03
b02	58	20.69	0.001	20.69	0.001	20.69	0.001	20.69	0.001
b03	1652	39.47	0.12	55.27	0.27	54.54	0.76	**84.87**	1.87
b04	11288	13.83	4.51	16.59	9.37	16.48	11.97	**25.56**	67.91
b05	146100	87.42	999.22	98.93	1048.84	98.39	1506.08	**99.76**	1780.44
b06	284	47.18	0.01	47.18	0.02	47.18	0.03	47.18	0.08
b07	10920	79.65	3.02	88.11	5.68	88.15	46.7	**90.92**	59.21
b08	4842	82.26	0.62	82.95	1.06	82.55	2.87	82.84	11.38
b09	1518	54.02	0.1	55.99	0.23	57.58	0.87	**67.59**	1.5
b10	1426	64.31	0.11	65.85	0.28	65.85	1.12	**67.53**	1.63
b11	89130	94.1	356.7	94.57	379.97	94.57	665.2	**95.10**	735.38
b12	26706	76.68	20.31	78.35	44.77	78.07	125.15	78.47	152.3
b13	1454	53.85	0.18	56.33	0.54	56.33	3.03	**68.50**	4.65
b14	20002	76.14	46.82	76.14	46.82	76.14	46.82	76.14	46.82
b15	20002	79.10	76.84	79.10	287.04	79.10	694.69	**87.31**	985.66
b20	20002	77.6	98.31	78.29	256.54	78.29	332.39	78.29	534.59
b21	20002	92.94	118.3	93.69	298.74	93.69	231.06	93.69	436.516
b22	20002	72.95	154.74	73.58	443.07	73.58	470.77	73.58	538.86

Time includes the total time required to test all PDFs listed under #pdf.
%FP (False Paths): Total paths that were proved false/untestable

Chapter 6

Conclusion

In this thesis, we focused on fast computation of invariants. Techniques for fast static learning and fast inductive reasoning were presented. These invariants were then applied to ATPG problems such as untestable fault identification and ATPG constraining for path delay fault testing, with promising results.

With respect to static learning, static filters, dynamic filters and their combination were discussed along with experimental results. All presented approaches showed significant speedup in comparison to the exhaustive (no-filter) approach. The filters were extended for multi-node learning. Fast learning can compute a substantially larger number of invariants in comparison to the no-filter approach, in a given fixed amount of time.

In the case of inductive reasoning, static filters were discussed to eliminate invariants that can easily be deduced from the CNF. To speed up the iterative fixed point inductive proof, dynamic filters were presented that can be applied to a diverse set of multi-node invariants. The dynamic filters provided a means to significantly reduce the number of SAT solver calls, even when the set contained variable sized potential invariants, providing tremendous

performance boost. Strategies to focus on specific types of powerful multi-node invariants were proposed. With experimental results, we showed that these filters enabled reasoning about large sets of multi-node invariants.

The static and inductive invariants computed from the proposed techniques were utilized for two ATPG problems: untestable fault identification and path delay fault ATPG. The multi-node invariants from fast learning were able to help identify many more untestable faults in comparison to the invariants from the exhaustive, non-filtered approach for many of the benchmarks. Adding inductive invariants to this set helped identify many more untestable faults. Similarly, we compared the number of false paths detected during path delay fault testing. Unrolling the circuit for six time-frames as well as constraining the ATPG using only static invariants helped identify a higher number of false paths in comparison to the unconstrained ATPG. When the constrains were upgraded to include both static and inductive invariants, the number of false paths identified substantially increased for many circuits, which points to the significant amount of overtesting that can be avoided.

Overall, we demonstrated the effect of invariants on two ATPG problems. The bottom line is that static and inductive invariants, when combined, hold great potential in aiding the shown applications as well as many more EDA areas. Techniques for faster invariant computation substantially enhances the effect of invariants on different applications that can utilize such relations.

Bibliography

[1] G. Moore, “Cramming more components onto integrated circuits,” *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan 1998.

[2] M. Schulz, E. Trischler, and T. Sarfert, “Socrates: a highly efficient automatic test pattern generation system,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 7, pp. 126–137, Jan 1988.

[3] A. El-Maleh, M. Kassab, and J. Rajski, “A fast sequential learning technique for real circuits with application to enhancing atpg performance,” in *Design Automation Conference, 1998. Proceedings*, pp. 625–631, June 1998.

[4] P. Tafertshofer, A. Ganz, and K. Antreich, “Igraine-an implication graph-based engine for fast implication, justification, and propagation,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, pp. 907–927, Aug 2000.

[5] S. Kajihara, K. Saluja, and S. Reddy, “Enhanced 3-valued logic/fault simulation for full scan circuits using implicit logic values,” in *Test Symposium, 2004. ETS 2004. Proceedings. Ninth IEEE European*, pp. 108–113, May 2004.

[6] M. Amyeen, W. Fuchs, I. Pomeranz, and V. Boppana, “Implication and evaluation techniques for proving fault equivalence,” in *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pp. 201–207, 1999.

[7] S. Prabhakar and M. Hsiao, “Using non-trivial logic implications for trace buffer-based silicon debug,” in *Asian Test Symposium, 2009. ATS ’09.*, pp. 131–136, Nov 2009.

[8] S. Prabhakar and M. Hsiao, “Multiplexed trace signal selection using non-trivial implication-based correlation,” in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, pp. 697–704, March 2010.

[9] D. Paul, M. Chatterjee, and D. Pradhan, “Verilat: verification using logic augmentation and transformations,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, pp. 1041–1051, Sep 2000.

[10] J. Marques-Silva and T. Glass, “Combinational equivalence checking using satisfiability and recursive learning,” in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pp. 145–149, March 1999.

[11] R. Arora and M. Hsiao, “Enhancing sat-based bounded model checking using sequential logic implications,” in *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 784–787, 2004.

[12] N. Goel, M. Hsiao, N. Ramakrishnan, and M. Zaki, “Mining complex boolean expressions for sequential equivalence checking,” in *Test Symposium (ATS), 2010 19th IEEE Asian*, pp. 442–447, Dec 2010.

[13] H. Ichihara and K. Kinoshita, “On acceleration of logic circuits optimization using implication relations,” in *Test Symposium, 1997. (ATS ’97) Proceedings., Sixth Asian*, pp. 222–227, Nov 1997.

[14] W. Kunz, D. Stoffel, and P. Menon, “Logic optimization and equivalence checking by implication analysis,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 16, pp. 266–281, Mar 1997.

[15] M. Iyer and M. Abramovici, “Fire: a fault-independent combinational redundancy identification algorithm,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 4, pp. 295–301, June 1996.

[16] M. Iyer, D. Long, and M. Abramovici, “Identifying sequential redundancies without search,” in *Design Automation Conference Proceedings 1996, 33rd*, pp. 457–462, Jun 1996.

[17] Q. Peng, M. Abramovici, and J. Savir, “Must: multiple-stem analysis for identifying sequentially untestable faults,” in *Test Conference, 2000. Proceedings. International*, pp. 839–846, 2000.

[18] M. Hsiao, “Maximizing impossibilities for untestable fault identification,” in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pp. 949–953, 2002.

[19] M. Syal and M. Hsiao, “Untestable fault identification using recurrence relations and impossible value assignments,” in *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 481–486, 2004.

[20] M. Schulz and E. Auth, “Improved deterministic test pattern generation with applications to redundancy identification,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 8, pp. 811–816, Jul 1989.

[21] W. Kunz and D. Pradhan, “Accelerated dynamic learning for test pattern generation in combinational circuits,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 12, pp. 684–694, May 1993.

[22] J. Rajski and H. Cox, "A method to calculate necessary assignments in algorithmic test pattern generation," in *Test Conference, 1990. Proceedings., International*, pp. 25–34, Sep 1990.

[23] S. Chakradhar, V. Agrawal, and S. Rothweiler, "A transitive closure algorithm for test generation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 12, pp. 1015–1028, Jul 1993.

[24] W. Kunz and D. Pradhan, "Recursive learning: a new implication technique for efficient solutions to cad problems-test, verification, and optimization," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 13, pp. 1143–1158, Sep 1994.

[25] J.-K. Zhao, J. Newquist, and J. Patel, "A graph traversal based framework for sequential logic implication with an application to c-cycle redundancy identification," in *VLSI Design, 2001. Fourteenth International Conference on*, pp. 163–169, 2001.

[26] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Design Automation Conference, 2001. Proceedings*, pp. 530–535, 2001.

[27] T. Larrabee, "Test pattern generation using boolean satisfiability," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 11, pp. 4–15, Jan 1992.

[28] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 15, pp. 1167–1176, Sep 1996.

[29] W. Hu, H. Nguyen, and M. Hsiao, "Sufficiency-based filtering of invariants for sequential equivalence checking," in *High Level Design Validation and Test Workshop (HLDVT), 2011 IEEE International*, pp. 1–8, Nov 2011.

[30] C.-A. Chen and S. Gupta, "A satisfiability-based test generator for path delay faults in combinational circuits," in *Design Automation Conference Proceedings 1996, 33rd*, pp. 209–214, Jun 1996.

[31] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and U. Glaeser, "An atpg-based framework for verifying sequential equivalence," in *Test Conference, 1996. Proceedings., International*, pp. 865–874, Oct 1996.

[32] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, C.-Y. Huang, and F. Brewer, "Aquila: an equivalence checking system for large sequential designs," *Computers, IEEE Transactions on*, vol. 49, pp. 443–464, May 2000.

[33] C. van Eijk, “Sequential equivalence checking based on structural similarities,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, pp. 814–819, Jul 2000.

[34] J. Jiang and W.-L. Hung, “Inductive equivalence checking under retiming and resynthesis,” in *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pp. 326–333, Nov 2007.

[35] F. Lu and K.-T. Cheng, “Sechecker: A sequential equivalence checking framework based on kth invariants,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, pp. 733–746, June 2009.

[36] L. Zhang, M. Prasad, and M. Hsiao, “Incremental deductive inductive reasoning for sat-based bounded model checking,” in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pp. 502–509, Nov 2004.

[37] H. Nguyen and M. Hsiao, “Sequential equivalence checking of hard instances with targeted inductive invariants and efficient filtering strategies,” in *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, pp. 1–8, Nov 2012.

[38] D. Krishnaswamy, M. Hsiao, V. Saxena, E. Rudnick, J. Patel, and P. Banerjee, “Parallel genetic algorithms for simulation-based sequential circuit test generation,” in *VLSI Design, 1997. Proceedings., Tenth International Conference on*, pp. 475–481, Jan 1997.

[39] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, “Dynamic state traversal for sequential circuit test generation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, pp. 548–565, July 2000.

[40] A. Krstic, J.-J. Liou, K.-T. Cheng, and L.-C. Wang, “On structural vs. functional testing for delay faults,” in *Quality Electronic Design, 2003. Proceedings. Fourth International Symposium on*, pp. 438–441, March 2003.

[41] X. Liu and M. Hsiao, “Constrained atpg for broadside transition testing,” in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pp. 175–182, Nov 2003.

[42] Y.-C. Lin, F. Lu, and K.-T. Cheng, “Pseudofunctional testing,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, pp. 1535–1546, Aug 2006.

[43] K. Chandrasekar and M. Hsiao, “Integration of learning techniques into incremental satisfiability for efficient path-delay fault test generation,” in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 1002–1007 Vol. 2, March 2005.

[44] M. Syal, K. Chandrasekar, V. Vimjam, M. Hsiao, Y.-S. Chang, and S. Chakravarty, “A study of implication based pseudo functional testing,” in *Test Conference, 2006. ITC ’06. IEEE International*, pp. 1–10, Oct 2006.