Static Learning for Problems in VLSI Test and Verification

Manan Syal

June 21, 2005

Bradley Department of Electrical and Computer Engineering,

Blacksburg, Virginia.

Keywords: Logic Implications, Untestable faults, ATPG, Equivalence checking

Static Learning for Problems in VLSI Test and Verification

Manan Syal

(Abstract)

Static learning in the form of logic implications captures Boolean relationships between various gates in a circuit. In the past, logic implications have been applied in several areas of electronic design automation (EDA) including: test-pattern-generation, logic and fault simulation, fault diagnosis, logic optimization, etc. While logic implications have assisted in solving several EDA problems, their usefulness has not been fully explored. We believe that logic implications have not been carefully analyzed in the past, and this lack of thorough investigation has limited their applicability in solving hard EDA problems. In this dissertation, we offer deeper insights into the Boolean relationships exhibited in a circuit, and present techniques to extract their full potential in solving two hard problems in test and verification: (1) Efficient identification of sequentially untestable stuck-at faults, and (2) Equivalence checking of sequential circuits. Additionally, for the dissertation, we define a new concept called multi-cycle path delay faults (M-pdf) for latch based designs with multiple clock domains, and propose an implications-based methodology for the identification of untestable M-pdfs for such designs.

One of the main bottlenecks in the efficiency of test-pattern-generation (TPG) is the presence of untestable faults in a design. State-of-the-art automatic test pattern generators (ATPG) spend a lot of effort (in both time and memory) targeting untestable faults before aborting on such faults, or, eventually identifying these faults as untestable (if given enough computational resources). In either case, TPG is considerably slowed down by the presence of untestable faults. Thus, efficient methods to identify untestable faults are desired. In this dissertation, we discuss a number of solutions that we have developed for the purpose of untestable fault identification. The techniques that we propose are fault-independent and explore properties associated with logic implications to derive conclusions about untestable faults. Experimental results for benchmark circuits show that our techniques achieve a significant increase in the number of untestable faults identified, at low memory and computational overhead.

The second related problem that we address in this proposal is that of determining the equivalence of sequential circuits. During the design phase, hardware goes through several stages of optimizations (for area, speed, power, etc). Determining the functional correctness of the design after each optimization step by means of exhaustive simulation can be prohibitively expensive. An alternative to prove functional correctness of the optimized design is to determine the design's functional equivalence w.r.t. some golden model which is known to be functionally correct. Efficient techniques to perform this process, known as equivalence checking, have been investigated in the research community. However, equivalence checking of sequential circuits still remains a challenging problem. In an attempt to solve this problem, we propose a Boolean SAT (satisfiability) based framework that utilizes logic implications for the purpose of sequential equivalence checking.

Finally, we define a new concept called multi-cycle path-delay faults (M-pdfs). Traditionally, path delay faults have been analyzed for flip-flop based designs over the boundary of a single clock cycle. However, path delay faults may span multiple clock cycles, and a technique is desired to model and analyze such path delay faults. This is especially essential for latch based designs with multiple clock domains, because the problem of identifying untestable faults is more complex in such design environments. In this dissertation, we propose a three-step methodology to identify untestable M-pdfs in latch-based designs with multiple clocks using logic implications.

Dedication

I dedicate this dissertation to my parents and my guru Shree K. D. Nagar - “I would not be who I am and where I am without their blessings and support. I owe everything to them.”

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Michael Hsiao for giving me the opportunity to work with him. His constant direction, support and motivation made this dissertation possible. I would also like to thank Dr. D. S. Ha, Dr. S. Shukla, Dr. R. M. Buehrer and Dr. T. M. Murali for graciously serving on my committee.

I am also grateful to Sreejit Chakravarty, Suriyaprakash Natarajan and Kiran Doreswamy of Intel corp., who mentored me during my internships there.

Last, but not the least by any means, I would like to thank my friends who have made my stay at graduate school enjoyable and the people in my research group who have made every aspect of research exciting and interesting for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Logic implications capture the effect of asserting logic values throughout a circuit. Over the past few decades, logic implications have been successfully applied in several areas of electronic design automation (EDA) such as: test-pattern-generation [1–3], logic and fault simulation [4], fault diagnosis [5], logic verification [6–8], logic optimization [9, 10], untestable fault identification [11–15], etc. Since a powerful implication engine can have a wide impact on EDA applications and tools, much effort has been invested in efficient computation of implications.

Schulz *et al.* were the first to improve the quality of implications by computing indirect implications in SOCRATES [1]. To further the application of these indirect implications, static learning was extended to dynamic learning [16, 17]. Cox *et al.* introduced the use of 16-valued algebra and reduction lists to determine node assignments in [18]. A transitive closure procedure on implication graph was proposed by Chakradhar *et al.* in [19]. A complete implication engine based on recursive learning was proposed by Kunz *et al.* [20] that can capture all pair-wise relations in a circuit. However, to keep simulation time within reasonable limits, the recursion depth is generally limited to low values. A graphical representation to store implications was proposed by Zhao *et al.* [21] and the concept of indirect implications based on transitivity of implications, along with extended backward implications were used to increase the number of implications learnt.

Learning non-trivial Boolean relationships via static implications can prove instrumental only when these relationships are analyzed and applied suitably. While much work has been focused on improving the quality of implications, little effort has been invested in investigating the properties of these Boolean relationships. As a result, while implications have found some success in assisting various EDA applications, their full-potential to solve hard problems in test and verification is still latent. We believe that by looking deeper into these Boolean relationships, interesting characteristics of the design can be discovered which can prove vital in solving hard, unsolved problems in test and verification.

This dissertation presents an attempt in this very direction. The unifying theme of the dissertation can be understood through the following statement, which presents the underlying contribution of this work: improving and analyzing logic implications to make powerful deductions, and applying these implications and deductions towards solving hard, unsolved problems in VLSI (very large scale integration) test and verification.

While the deductions made in this work can be applied towards several applications, we target two hard problems in test and verification: (a) identification of untestable stuck-at faults in sequential circuits and (b) equivalence checking of sequential circuits. In addition, we first present the formal definition of multi-cycle path delay faults (M-pdfs) and then provide a novel, scalable framework to identify untestable M-pdfs in latch based designs with multiple clocks. Since analyzing untestablility in such designs is particularly challenging and not well researched, our framework can have significant impact on how identification of untestable faults is viewed in such designs.

Untestable faults are faults for which there exists no test pattern that can both, excite the fault and propagate its effect to a primary output. A fault may be untestable either because (a) the conditions necessary to excite the fault cannot be met, or (b) the conditions necessary to propagate the fault effect to an observable point in the circuit cannot be achieved or (c) the conditions for fault excitation and fault propagation cannot be simultaneously satisfied. In combinational circuits untestable faults result due to redundancies in the circuit, while in sequential circuits untestable faults may also result because of unreachable states or impossible state transitions.

Test-pattern-generation (TPG) is the process of generating patterns that would be able to detect the presence of defects in a chip, and automatic-test-pattern-generators (ATPG) are tools that are responsible for performing this task. The presence of untestable faults in a design degrades the performance of a TPG tool; ATPG must exhaust the entire search space before identifying such faults as untestable. Thus, the performance of ATPG engines (and other fault-oriented tools such as fault-simulators) can be enhanced if knowledge of untestable faults is available *a priori*, in which case these tools can ignore targeting such faults. In addition, successful identification of untestable faults can aid other computer-aided applications, such as combinational equivalence checking [6]. There are additional indirect benefits of untestable fault identification because such faults can have other detrimental effects. The presence of untestable faults can potentially prevent the detection of other faults in the circuit [22]. Untestable faults in the form of redundancies increase the chip area; they may also increase the power consumption and the propagation delays through a circuit [22]. Also, untestable faults may result in unnecessary yield loss during partial or full-scan mode of testing, because even though the circuit remains fully operational in the presence of untestable faults, scan based testing may detect such faults and reject the chip. Thus, significant effort has been invested towards efficient identification of untestable faults.

The techniques that have been proposed in the past for untestable fault identification can be classified into fault-oriented methods based on ATPG [23–25], and fault-independent methods [11–15] generally based on conflict analysis. Generally, ATPG-based methods outperform fault-independent methods for smaller circuits; however, the computational complexity of branch-and-bound algorithms (ATPG) makes them impractical for large combinational and sequential circuits. Conflict-based analysis has thus been researched and improved over the years. Iyer *et al.* introduced FIRE [11] as a technique to identify untestable faults as faults that require a conflict on a single line as a necessary condition for their detection. Since it is impossible for a single line to have conflicting values at the same time, faults that require such an impossible assignment are untestable. FIRES [12] was introduced as an extension of FIRE to identify untestable faults in sequential circuits without explicit search. Because illegal states in sequential circuits result in untestable faults, a BDD based approach to identify illegal states was introduced in [26] as FILL. Next, FUNI [26] utilized this illegal state space information to identify untestable faults. MUST [13] introduced by Peng *et al.* was built over the framework of FIRES as a fault-oriented approach to identify untestable faults. However, the memory requirement for MUST can be exponential. Recently, Hsiao [14] presented a fault-independent technique to identify untestable faults using multiple-node impossible value combinations. However, this approach concentrated on identifying conflicts local to Boolean gates in a circuit.

In this dissertation, we provide efficient, implications-based fault-independent solutions for the purpose of untestable fault identification. The features that our engine incorporates for untestable fault identification are:

- Identifying global multiple-node conflicting assignments. Unlike previous techniques that concentrate on identifying local conflicts in the circuit [14], our approach efficiently extends the impossible combination analysis across multiple levels in the circuit to identify more conflicting value combinations, which helps us in identifying more untestable faults.

- Exploring characteristics associated with logic implications to derive conclusions about the controllability characteristics of nets in the design. Since a large set of untestable faults can be inferred from the knowledge of an uncontrollable net, efficient ways of identifying uncontrollable nets pay a rich dividend in terms of untestable faults.

- A new and efficient mechanism to improve Boolean learning through static implications. Our technique enables learning those relationships which were missed by previous techniques. The significance of our new learning criterion is clearly presented through the substantial leap achieved in the number of sequentially untestable faults identified.

Details about the techniques mentioned above are presented in Chapter 2.

The next problem we target in this dissertation is that of sequential equivalence checking. We propose a novel framework to verify equivalence of sequential circuits using Boolean Satisfiability (SAT). We tackle a problem that is harder than the traditional sequential hardware equivalence; specifically, we address the uninvestigated problem of verifying delay replaceability [27] of two sequential designs.

Logic synthesis has matured over the last few decades and now offers synthesis and optimization of sequential circuits with respect to several constraints such as area, power, speed, testability, etc. Current synthesis tools can handle sequential circuits with hundreds to thousands of memory elements (latches or flip-flops) and can perform sequential and combinational transformations (retiming, sequential/combinational redundancy removal, etc.). As a consequence of the maturity of synthesis tools, there has been an increasing use of sequential optimization techniques during the design cycle of sequential machines. The process of verifying that the input-output characteristics of the original sequential design are preserved after these transformation procedures is commonly referred to as sequential equivalence checking. While much success and progress has been seen with equivalence checking of combinational circuits, sequential equivalence checking remains extremely difficult.

In the literature, several notions or definitions of sequential equivalence have been provided. These definitions usually differ from one another in the assumption(s) each definition makes with respect to the operation of the sequential circuit and/or the environment in which the design may be embedded. For circuits with an external reset state, a notion of reset-equivalence exists; according to this notion, two sequential circuits are said to be equivalent if the input-output behavior of the two circuits is identical after the application of the external reset to the two circuits. This notion of equivalence assumes that the external environment is capable of generating the reset signal for both circuits under consideration. However, it was argued in [27] that many real designs do not have an external reset state. Also, it would be unrealistic to assume that the external environment is always capable of generating this reset state for the design under consideration. For circuits without an external reset state, several notions of sequential equivalence have been provided [27–30].

Sequential hardware equivalence (SHE) [28] refers to a notion according to which two circuits are said to be equivalent if their input-output behavior is identical after an initialization sequence has been applied; hence SHE is also sometimes referred to as post-synchronization equivalence [30]. However, SHE requires the preservation of only *one* initializing sequence; as a consequence, the notion of SHE can lead to erroneous circuit replacements if the design's environment cannot generate the specific synchronizing sequence under which the replacement is verified against the original design. Moreover, the notion of SHE does not work for designs which do not have an initializing sequence.

Cheng [29] provided a notion of design replacement in terms of sequential redundancy. A circuit in which sequential redundancy is removed according to this definition was argued to be a valid replacement for the original circuit. However, in [27] it was shown that this notion of sequential equivalence or design replacement may be inaccurate because logic that may be sequentially redundant in a stand-alone sequential circuit may not be redundant when considered in the context of the circuit's environment.

Another classification of sequential equivalence called 3-valued safe replaceability was introduced by Huang *et al.* [30]. This notion can be closely related to Cheng's [29] notion of equivalence using sequential redundancy; as a consequence, it can be shown that 3-valued safe replaceability can result in erroneous replacements when the composition of the transformed circuit with it's environment is taken into account. Moreover, like SHE, this definition is applicable only for designs that are initializable.

In contrast to the above classifications, the notions of safe replaceability and delay replaceability [27] do not make any assumptions either about the design-environment or about the steady state behavior of the design (such as initialization). Thus, proving safe replaceability or delay replaceability would provide more confidence in the functional correctness of the transformed or the optimized design (with respect to the original design). Although safe replaceability provides a more general form of design replacement compared to delay replaceability, due to reasons mentioned in this dissertation, it is more practical to target verification of delay replaceability of sequential circuits. In terms of complexity, however, verifying delay replaceability is considered as hard as verifying safe replaceability [27] of sequential circuits (conjectured as EXPSPACE complete). We also illustrate that delay replaceability is a harder verification problem compared to 3-valued safe replaceability (and hence also harder than SHE). Thus, verification of delay replaceability would automatically cover sequential equivalence according to 3-valued replaceability and SHE. While previous verification efforts have focused on easier notions of sequential equivalence (such as SHE), to the best of our knowledge, there has not been any significant contribution made towards the verification of either safe replaceability or delay replaceability of sequential circuits. Thus, the proposed framework presents a *unique and practical approach* to tackle a hard and uninvestigated verification problem: verifying delay replaceability of sequential circuits.

Conventional techniques for sequential equivalence checking are based either upon binary decision diagrams (BDD) [31–33] or upon automatic test pattern generators (ATPG) [30]. While BDDs suffer from memory explosion, sequential ATPGs suffer from exponential time complexity for large sequential circuits. Thus, a need is justified to identify alternative solutions for checking equivalence of sequential circuits. Recently, significant progress has been made in solving Boolean satisfiability (SAT) instances [34–36]. In terms of verification of sequential systems, SAT has shown success in bounded model checking [37–39]; however, modeling the problem of sequential equivalence checking with SAT is not well explored. In this dissertation, we present a novel SAT formulation that can be used for verifying sequential circuit equivalence (under the notion of delay replaceability). Three key features of our framework, which prove critical for sequential equivalence checking, are as follows: (a) a methodology to inductively prove equivalence (delay replaceability) of sequential circuits with no assumptions about any initial state; (b) a scheme to include sequential logic implications into the framework; and (c) a low-cost scheme to identify equivalent flip-flop pairs on the fly. We used our tool to successfully verify several sequential benchmark circuits. Low execution times make our framework practical and scalable.

Details about our SAT based framework are provided in Chapter 3.

Finally, in this dissertation, we propose a framework for identifying untestable multi-cycle path delay faults in latch based designs with multiple clocks. Traditionally, path delay faults have been analyzed in flip-flop based designs, with single clock domains. Also, traditionally, path delay faults have been analyzed over the boundary of a single clock cycle. However, many commercial designs have multiple clock domains, and incorporate latches as sequential elements. Also, increased performance requirement is contributing to an increase of such design characteristics. These factors along with the fact that several path delay faults can be gross enough to last for multiple clock cycles motivates the need for a framework for analyzing multi-cycle path delay faults for latch based designs with multiple clock domains.

Path delay faults model distributed defects that cause the delay associated with a particular path to exceed specified limits. Corresponding to each physical path in the circuit, there exist two logical paths: one associated with the rising transition (0 - 1), also known as the slow-to-rise path, and the other associated with the falling transition (1 - 0), also known as the slow-to-fall path. Generally speaking, a path-delay fault requires a pair of vectors (V_1, V_2) to detect its presence; an initialization vector (V_1) that (at-least) sets the initial value at the beginning of the path; and a propagation vector (V_2) that propagates the transition along the path to a primary output. Tests for path delay faults are classified into two broad categories: general robust test [44] (GRT) and general non robust test [44] (GNRT). GNRT is a less constrained test pattern generation scheme as compared to GRT. While GNRT requires the initialization vector V_1 to simply launch the transition at the beginning of the path (i.e., set the initial value corresponding to the logical path), GRT scheme may also require additional conditions to be satisfied by the initialization vector for the vector pair (V_1, V_2) to be classified as a GRT for the path-delay fault. Thus, a test that can classifies as a GRT for a path-delay fault would also serve as a GNRT for the same delay fault. Conversely, if a path delay fault is not testable by the GNRT scheme (also known as non-robustly untestable) it would also be untestable according to the GRT scheme (known as robustly untestable). A significant amount of research has been invested in identifying both robust and non-robust untestable faults. Most of the techniques that have been published rely on static learning using logic implications to identify untestable path delay faults [45–47]. In this dissertation, we target the identification of non-robustly untestable path delay faults using logic implications. However, unlike previous techniques, we target multi-cycle path delay faults as opposed to single-cycle path delay faults; also, we deal with latch based design with multiple clock domains, which makes our work novel.

Details of our technique to identify non-robustly untestable multi-cycle path delay faults are provided in chapter 4.

Since logic implications form the core element of our work, before discussing each problem targeted in this discussion, a preliminary discussion on logic implications is presented

1.1 Logic Implications

Logic implications identify the effect of asserting logic values throughout a circuit. In the past, logic implications have been categorized into direct, indirect and extended backward implications. Direct implications for a gate g are logic relationships learned without performing circuit simulation. Whereas, indirect and extended backward implications require circuit simulation and identify the logical effect of asserting a value on g with nodes in the circuit which may not be directly connected to g. While direct implications for a gate can be easily enumerated by traversing the immediate fan-in and fan-out of a gate, indirect and extended backward implications require the extensive use of the transitive property and the contrapositive property [21]. The following terminology will be used henceforth for describing implications:

1. $[N, v, t]$: Assign logic value v to gate N in time frame t. For combinational circuits, t is equal to 0, and is dropped from the expression, i.e. if t = 0, $[N, v, t]$ = $[N, v]$.

2. $[N, v, t] \rightarrow [M, w, t_1]$: Assigning logic value v to gate N in time frame t would imply or have the effect of assigning logic value w to gate M in time frame t_1.

3. $Impl[N, v, t]$: All implications resulting from the value assignment of logic value v to gate N in time frame t. For $t = 0$, $Impl[N, v, t]$ is represented as $Impl[N, v]$.

With this terminology, we will use the following example to explain the concepts for direct, indirect, and extended backward implications.

Example 1: Consider the circuit shown in Figure 1.1.

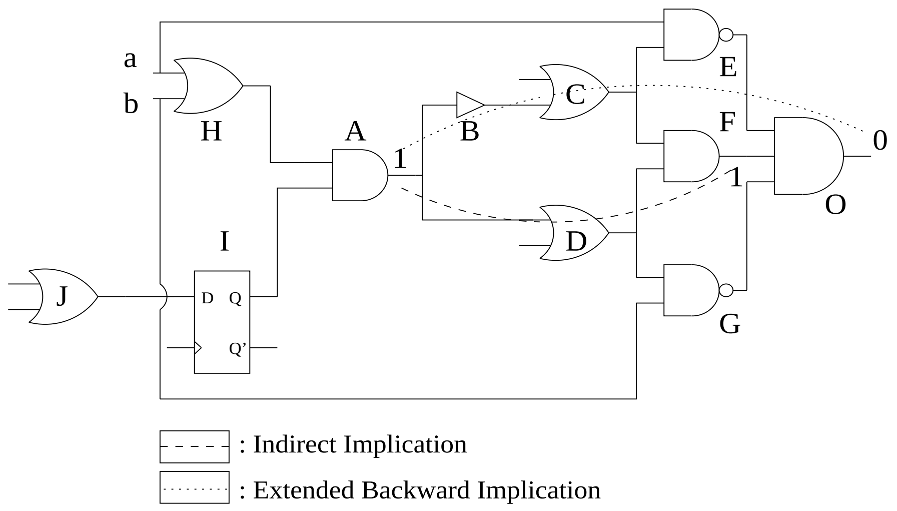


Figure 1.1: Segment of a Sequential Circuit

Let us compute the implications for gate A = 1.

1. *Direct Implications*: A logic value of 1 at the output of gate A would imply B = D = 1. Also $A = 1 \rightarrow H = 1$ and $I = 1$. Thus, the set $\{(A, 1, 0), (B, 1, 0), (D, 1, 0), (H, 1, 0), (I, 1, 0)\}$ represents the direct implications of $A = 1$. Similarly, direct implications associated with $B = 1$ are $\{(B, 1, 0), (C, 1, 0), (A, 1, 0)\}$ and so on. These implications are stored in the form of a graph with each node representing a gate (with value). A directed edge between nodes represents an implication, and a weight along with an edge represents the relative time-frame associated with the implication. Figure 1.2 (a) shows the graphical representation of the direct implications for $A = 1$. The complete set of implications resulting from setting A to 1 can be obtained by traversing the graph rooted at node $A = 1$ (transitive closure on $A = 1$).

   Thus, the complete set of direct implications for $A = 1$ is: $\{(A, 1, 0),\ (H, 1, 0),\ (B, 1, 0), (C, 1, 0), (D, 1, 0), (I, 1, 0), (J, 1, -1)\}$

2. *Indirect Implications* : Although neither $C = 1$ nor $D = 1$ implies a logic value

on gate F individually, together, they imply $F = 1$. Thus, indirectly, $A = 1$ would imply $F = 1$ (shown as a dashed line in Figure 1.1). This is called an indirect implication, and it can be computed by logic simulating the current set of implications of the target node. In this example, by inserting the implications of $A = 1$ into the circuit, followed by logic simulation, we will obtain $F = 1$ as a result. This new implication is added as an additional outgoing edge from A = 1 in the implication graph as shown in Figure 1.2 (b). Another non-trivial implication inferred from each indirect implication derives its roots from the contrapositive law. According to the contrapositive law, if $[N, v] \rightarrow [M, w, t_1]$, then $[M, \overline{w}] \rightarrow [N, \overline{v}, -t_1]$. Since $[A, 1] \rightarrow [F, 1, 0]$, by contrapositive law, $[F, 0] \rightarrow [A, 0, 0]$.

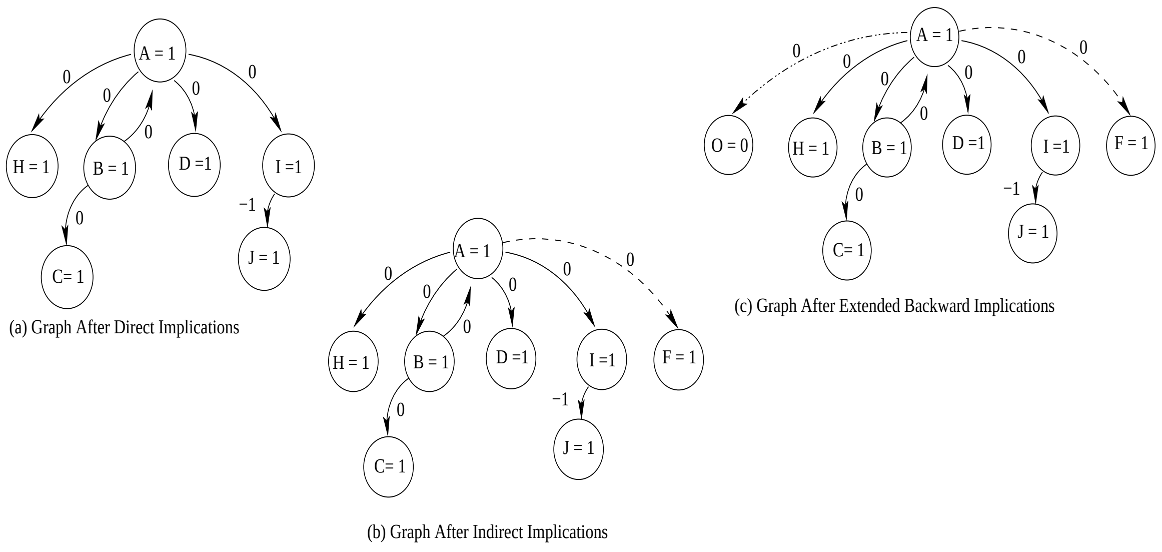

A = 1
H = 1
B = 1
D =1
I =1
C= 1
J = 1
F = 1
O = 0
0
−1
(a) Graph After Direct Implications
(b) Graph After Indirect Implications
(c) Graph After Extended Backward Implications

Figure 1.2: Implication Graph For Gate A = 1

3. *Extended Backward (EB) Implications*: Extended backward implications aim to learn additional relationships of the implying node with respect to any unjustified gates in its implication list. For the circuit shown in Figure 1.1, gate $H = 1$ is an unjustified gate in the implication list for $A = 1$, as none of its inputs is implied to a value of

logic 1. Thus, H is a candidate for the application of extended backward implications. To perform extended backward implications on H, a transitive closure is first performed for each of its unspecified inputs (i.e. a and b), obtaining $impl[a = 1]$ and $impl[b = 1]$, respectively. The implications of $A = 1$ are simulated *together* with each of H's unspecified inputs' implication sets in turn, creating a set (set_i) of newly found logic assignments for each input i. For our example, when the implications of $(a = 1)$ and $(A = 1)$ are simulated, the new assignments (set_1) found are $(E, 0, 0)$ and $(O, 0, 0)$. For the combined implication set of $(b = 1)$ and $(A = 1)$, new assignments found (set_2) are $(G, 0, 0)$ and $(O, 0, 0)$. All logic assignments (not currently in the implication set for $A = 1$) that are common among these sets, are the extended backward implications and added as new edges to the original node $A = 1$. In our example, since $(O, 0, 0)$ is common between set_1 and set_2, it is added as a new edge to the implication graph for $A = 1$ (depicted in Figure 1.2 (c)). $\square$

The rest of this dissertation is organized in the following manner: Chapter 2 provides details on our methodology for identification of untestable stuck-at faults. Chapter 3 discusses our SAT based framework used to verify equivalence of sequential circuits. In chapter 4 we provide our framework for identifying untestable multi-cycle path delay faults. Finally, chapter 5 concludes the dissertation.

Chapter 2

Identifying Untestable Stuck-at Faults

In this chapter, we present new and efficient techniques to identify untestable faults in both combinational and sequential circuits. Our techniques are fault-independent (per-fault analysis is not performed), and rely on logic implications and conflict based analysis to deduce untestable faults. The core element of our framework is the single-line conflict analysis proposed by Iyer *et al.*, which is first explained in the preliminaries subsection. This is followed by our new techniques and experimental results.

2.1 Preliminaries: Single Line Conflict Analysis

The underlying concept behind the single line conflict analysis is that faults which require a conflict on a line as a necessary condition for their detection are untestable. The analysis is fault independent as we begin with a possible conflict and identify faults that require the existence of this conflicting condition as a necessary condition for their detection. In this analysis, for each gate in the circuit, the following two sets are computed:

S_0: Set of faults untestable with $g = 0$. S_1: Set of faults untestable with $g = 1$. The faults in any set S_i require $g = \bar{i}$ to be testable.

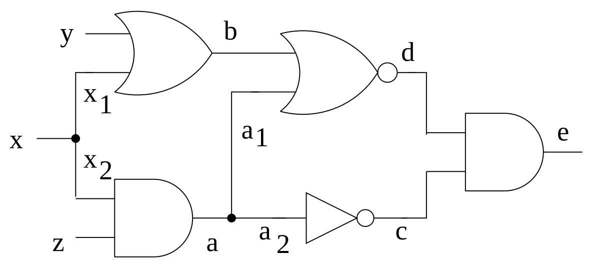


Figure 2.1: Illustration of single line conflict analysis

Any faults that lie in the intersection of sets S_0 and S_1 are identified as untestable because these faults require conflicting (impossible) values on g as a necessary condition for their detection.

Consider the following example with the provided terminology for better understanding:

Definition 1 *Fault g/v refers to gate g stuck-at logic v.*

Example 2: Consider the circuit shown in Figure 2.1. Consider the implications of x = 1. $Impl[x, 1, 0]$ = $\{(x, 1, 0), (x_1, 1, 0), (x_2, 1, 0), (b, 1, 0), (d, 0, 0), (e, 0, 0)\}$.

Faults unexcitable due to x = 1:

With x = 1, it would not be possible to set line b to 0, since $\{x = 1\} \rightarrow \{b = 1\}$. Thus, fault $b/1$ would be unexcitable with $x = 1$, and would require $x = 0$ as a necessary condition for its detection. Essentially, *if $[k, v, t] \in Impl[N, w, t_1]$, then fault k/v would be unexcitable in time frame t with $N = w$ in time frame t_1.*

As a result, faults $x/1, x_1/1, x_2/1, b/1, d/0, e/0$ would be unexcitable with $x = 1$.

Faults unobservable due to x = 1:

Since $\{x = 1\} \rightarrow \{d = 0\}$, any fault value appearing at line c cannot be propagated to the next level (fault effect is said to be blocked). Hence, faults $c/0$ and $c/1$ would require $x = 0$ as a necessary condition for their detection. Similarly, any faults appearing on lines y, a_1, a_2, etc. would also be blocked due to implications of $x = 1$. The complete set of faults that cannot be propagated (are unobservable) because of $x = 1$ is:

$\{y/0, y/1, a_1/0, a_1/1, c/0, c/1, a_2/0, a_2/1, a/0, a/1, z/0, z/1, x_2/0, x_2/1\}$.

Thus, S_1 = $\{x/1, x_1/1, x_2/1, b/1, d/0, e/0, y/0, y/1, a_1/0, a_1/1, c/0, c/1, a_2/0, a_2/1, a/0, a/1, z/0, z/1, x_2/0, x_2/1\}$.

Now, consider the implications of $x = 0$.

$Impl[x, 0, 0] = \{(x, 0, 0), (x_1, 0, 0), (x_2, 0, 0), (a, 0, 0), (a_1, 0, 0), (a_2, 0, 0), (c, 1, 0)\}$.

Similar to the analysis for $x = 1$, faults which are unexcitable and unobservable due to $x = 0$ are enumerated as:

S_0 = $\{x/0, x_1/0, x_2/0, a/0, a_1/0, a_2/0, c/1, z/0, z/1\}$

Thus, $S_0 \bigcap S_1$ = $\{x_2/0, a/0, a_1/0, a_2/0, c/1, z/0, z/1\}$ forms the set of untestable faults because these these faults require a conflicting assignment on line x as a necessary condition for their detection. $\square$

As observed from the example above, untestable faults are identified via the unexcitable and unobservable nets. For sequential circuits, we use the propagation of unobservability across multiple time frames and make sure that the fault we declare as sequentially untestable is truly untestable (i.e., the fault does not become testable after recombination with the copy of the same fault in a higher time frame) using the approach proposed in our earlier work [48].

In addition to the implication engine and the single line conflict analysis, the following definitions of three-valued-logic simulation and untestable faults [12] [49] are necessary for a better understanding of our untestable fault identification engine:

Definition 2 *Three-valued-logic simulation is a simulation framework wherein each gate is evaluated only with the logic values in the set $\{0, 1, X\}$ (where X denotes an unknown logic value). Also, in this simulation, X and its logical complement $\overline{X}$ are not differentiated.*

Definition 3 *A fault f is said to be testable if there exists an input sequence I such that for every pair of initial states S and S^f of the fault-free and the faulty circuit, respectively, the response $Z(I, S)$ of the fault-free circuit to I is different from the response $Z^f(I, S^f)$ of the faulty circuit (at some given time on some output).*

Definition 4 *A fault is said to be untestable if it is not testable.*

2.2 The First Step: Recurrence Relations and Multiple Line Conflict Analysis

The application of logic implications in identifying untestable faults is evident from Example 2. Logic implications are used to identify unexcitable and unobservable nets due to a given assignment, which results in identification of untestable faults. However, in this dissertation, we take application of logic implications towards identification of untestable faults to the next level. Specifically, we explore characteristics of logic implications to make useful deductions about unexcitable nets. These unexcitable nets play a vital role in identifying untestable faults. In this section, we present several new Lemmas, new concepts and a new Theorem that enable the use of logic implications in identifying untestable faults via unexcitable nets. We also present a technique to improve conflict analysis from only single line conflicts to multiple line conflicts: we quickly identify impossible value combinations on multiple nets in the circuit. These impossible value combinations are then used to identify untestable faults.

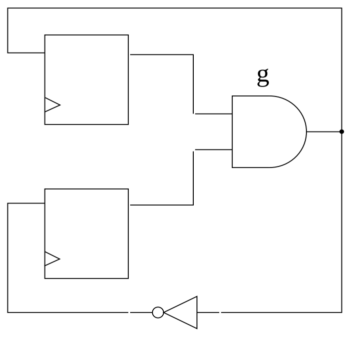
2.2.1 Identifying Recurrence Relations

A net N is said to be unexcitable to a particular logic value v if it is impossible to set N to logic v. Unexcitable nets generally cause a large set of faults to become untestable, because they not only imply that a particular fault on the net cannot be excited, but they also block the propagation paths for many other faults. Thus, identification of unexcitable nets can play a key role in untestable fault identification. We present a novel, low-cost mechanism to make deductions about unexcitable nets using logic implications. There has been related work in identifying unexcitable nets for untestable fault identification. Liang *et al.* [50] proposed a technique to identify unexcitable nets through symbolic simulation, and they showed that with their technique they could identify a substantial portion of the total faults as untestable. Although their results are encouraging, it was shown in [26] that the results in [50] could be over-optimistic i.e., the faults they identify as untestable could in fact be testable. Our method of identifying unexcitable nets is very different from the method used by Liang *et al.* We do not perform symbolic simulation to identify controllability characteristics of any net in the circuit. We examine certain properties of sequential implications to derive conclusions about the controllability characteristics of lines in the circuit. Before we describe our approach, the following definitions and lemmas are necessary:

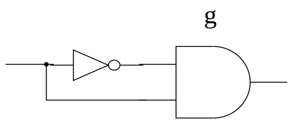
Definition 5 *We define a gate g to be n-cycle-constant to logic value v if it takes n-time frames to identify through implications that g cannot take a logic value of $\overline{v}$.*

Traditionally a gate g is defined to be a constant with value v if its opposite value implies conflicting assignments (i.e., $[g, \overline{v}, 0] \rightarrow [a, w, t]$ and $[g, \overline{v}, 0] \rightarrow [a, \overline{w}, t]$ (where t could be positive or negative)). However, the identification of a constant through implications depends upon the number of time frames the sequential circuit is analyzed for. For example, if the circuit in Figure 2.2 is analyzed for 1-time frame (combinational analysis), then gate g would not be identified as a constant to logic 0.

However, if the circuit is analyzed for 2-time frames, it would be observed that $[g, 1, 0] \rightarrow [g, 0, -1]$ and $[g, 1, 0] \rightarrow [g, 1, -1]$ indicating that gate g is a constant with value 0. Thus, in this case we would define g to be 2-cycle-constant. However, it would take just one time



(a) Two cycle constant



(b) One cycle constant

Figure 2.2: Illustration of n-cycle constant

frame for the circuit in Figure 2.2(b) to identify that gate g is a constant with value 0. Thus, in that case we identify g to be a 1-cycle-constant.

It is intuitive that an implication engine built over $n + m$ time frames may potentially identify more constants than the one built over m time frames (with $n \geq 1$).

Axiom 1: If a gate g is identified as an n-cycle-constant with value v, then $g = v$ is implied in every time frame of the sequential circuit.

Definition 6 *The value assignment $g = v$ is said to be unachievable if there exists no input sequence that can set gate g to logic v with the initial state of all flip-flops being unknown or X.*

Consider the circuit shown in Figure 2.3. It can be seen that the value assignment a = 1 and b = 1 are unachievable assuming that the starting state of the flip-flop F is X.

Definition 7 *Informally, an initialization sequence I is a sequence which brings a sequential circuit to a specific known state from an unknown initial state (where all flip-flops have an unknown logic value X).*

Lemma 1 *If the assignment $g = v$ is unachievable for gate g, then g would be constant to $\overline{v}$ if there exists an initialization sequence for the circuit. We call such a constant as a post-initialization constant.*

Proof: Assume that there exists a gate g for which logic value v is unachievable. Now, if there exists an initialization sequence for the circuit, then it would set g = $\overline{v}$ (g = v is unachievable). Then for every sequence applied after this initialization sequence, g would take the value $\overline{v}$ (since g = v is unachievable, and since the state machine can be initialized, g cannot take the value X (unknown) after initialization). Thus, after the state machine is initialized, g would be a constant $\overline{v}$. □

Definition 8 *We say that a recurrence relation exists for a gate g with value v if* $[g, v, t] \rightarrow [g, v, t - t_1]$. *For* $t_1 < 0$, *this relation is called a forward recurrence relation, while for* $t_1 > 0$, *the relationship is called a backward recurrence relation.*

Now we state the following Lemma that helps in the identification of unachievable nets using knowledge of implication relationships in a sequential circuit.

Lemma 2 *For a gate* g, *if the backward recurrence condition* $[g, v, t] \rightarrow [g, v, (t - t_1)]$ *exists (where* $t_1 \geq 1$ *), and* g *is not a constant with value* v, *then the value assignment* $g = v$ *is unachievable.*

Proof: In order to prove this lemma, consider Figure 2.4. Without loss of generality, let us assume that t_1 = 1, i.e., $[g, v, t] \rightarrow [g, v, t - 1]$ (the proof would hold true for any t and t_1). Figure 2.4 shows the iterative logic array (ILA) expansion of a sequential circuit where gate

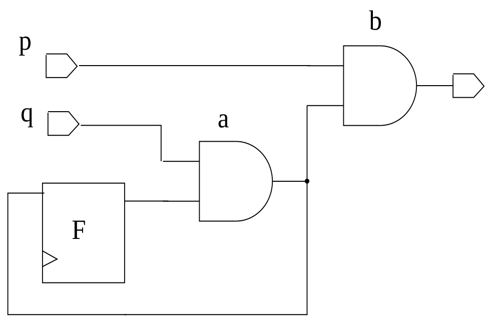


Figure 2.3: Illustration of unachievable assignments

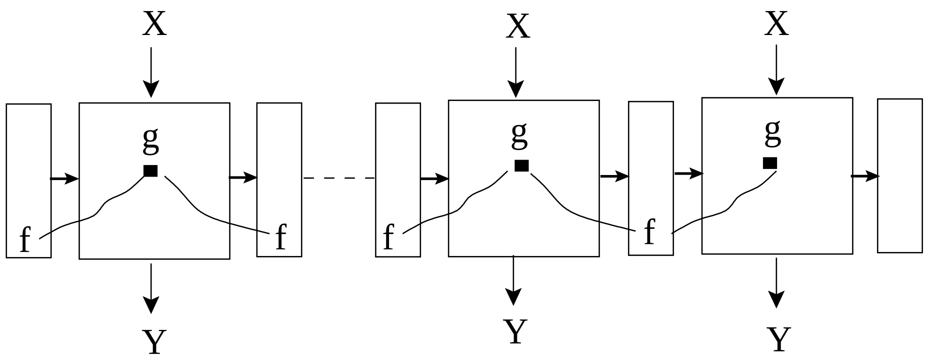


Figure 2.4: ILA expansion of a sequential circuit

g is shown as a colored box to maintain generality of the proof. Since $[g, v, t] \rightarrow [g, v, t-1]$, $[g, v, t]$ must also imply a logic value w on a flip-flop (say flip-flop f) in time frame t through which it implies $[g, v, t-1]$. That is,

$$[g, v, t] \rightarrow [f, w, t] \tag{2.1}$$

Since $[g, v, t] \rightarrow [g, v, (t-1)]$, it follows that $[g, v, (t-1)] \rightarrow [g, v, (t-2)]$, $[g, v, (t-2)] \rightarrow [g, v, (t-3)]$, ... , $[g, v, (t-n)] \rightarrow [g, v, (t-n-1)]$ (for any $n \geq 0$). Therefore, by using transitivity property of implications,

$$[g, v, t] \rightarrow [g, v, (t-n-1)] \tag{2.2}$$

Again, since $[g, v, t] \rightarrow [f, w, t]$ from equation 2.1, it follows that (by changing the reference from t to $t-n-1$)

$$[g, v, (t-n-1)] \rightarrow [f, w, (t-n-1)] \tag{2.3}$$

Thus, equations 2.2 and 2.3 further imply that $[g, v, t] \rightarrow [f, w, (t-n-1)]$. Since this relationship is valid for any value of $n \geq 0$, it follows that it would be impossible to assign gate g to logic value v starting from an all unknown because the assignment $g = v$ implies some flip flop f to always have a fully specified logic value (even at the starting state). However, it should be noted that this argument holds true only if $g = v$ is not a constant. If

$g = v$ is a constant, then according to Axiom 1, $[g, v, t]$ does not require a value assignment on any flip-flop to imply $[g, v, t - 1]$ (as a constant is implied in each frame). □

Lemma 3 *If an assignment of g = v is found to be unachievable by Lemma 2, then all faults that require this assignment as a necessary condition for their detection would be untestable.*

Proof: Refer to the ILA in Figure 2.4. Now, there would exist a subset of initial states S_i in the good machine (good machine is the circuit without any fault) which would contain the value assignment of $f = \overline{w}$. Since the assignment $g = v$ implies $f = w$ in every time frame (using the argument provided in Lemma 2), all states in S_i would imply $g = \overline{v}$ in every time frame, by contrapositive law [21]. Therefore, starting from any state in S_i, there cannot be applied a vector sequence that can set $g = v$ in any time frame and hence all faults that require $g = v$ as a necessary condition for their detection would not be detected at the output. So, by Definition 3 these faults cannot be classified as testable, and as a consequence all faults that require the unachievable assignment of $g = v$ would be untestable. It should also be noted that if the sequential circuit has a reset state and if the conditions defined in Lemma 2 are true, then $g = v$ can be marked as unachievable only if the reset state does not contain the assignment of $f = w$. □

Corollary 1 *Based on Lemma 2, given* $[g, v, t] \rightarrow [g, v, t - t_1]$*, then* $g = v$ *is unachievable only if* $g = \overline{v}$ *is possible.*

Proof: If $g = \overline{v}$ is possible, then $g = v$ is not a constant. Then according to Lemma 2, if $[g, v, t] \rightarrow [g, v, t - t_1]$, then $g = v$ is unachievable. However, if it cannot be guaranteed that $g = \overline{v}$ is possible (say by three-valued logic simulation), then it is inconclusive if $g = v$ is a constant or not. Therefore, it may be possible that even though $g = v$ is not identified as m-cycle-constant it may be identified as $(m + n)$-cycle-constant. In that case

$[g, v, t] \rightarrow [g, v, t - t_1]$ would be a legal implication (Axiom 1), and marking $g = v$ as unachievable would be erroneous. Thus, $g = v$ is unachievable according to Lemma 2 only if $g = \overline{v}$ is possible. □

Now, we present a theorem as an extension of Lemma 2 that can be used to identify untestable faults, by identifying unachievable nets and post-initialization constants.

Theorem 1 *If the backward recurrence relation $[g, v, t] \rightarrow [g, v, t - t_1]$ exists, and if an input sequence exists that can initialize gate g from unknown (X) to $\overline{v}$, then:*

1. *If t_1 = 1, then $g = v$ is unachievable, and $g = \overline{v}$ is a post-initialization constant, even if the sequential machine is only partially initializable;*

2. *If $t_1 > 1$, then g is only unachievable and $g = \overline{v}$ may not be marked as post-initialization constant.*

Proof: If there exists a sequence that can set gate g to logic value $\overline{v}$, then g is definitely not a constant with value v. Then according to Lemma 2, since $[g, v, t] \rightarrow [g, v, t - t_1]$, $g = v$ is unachievable. Now, if t_1 = 1, then $[g, v, t] \rightarrow [g, v, t - 1]$. By contrapositive law of implications [21], $[g, \overline{v}, t] \rightarrow [g, \overline{v}, t + 1]$. This means that if there exists a sequence that can set gate g to $\overline{v}$, in a time frame t then g would be $\overline{v}$ in time frame $t + 1$. Also, $[g, \overline{v}, t + 1] \rightarrow [g, \overline{v}, t + 2]$. Thus, by transitivity of implications, if gate g is set to logic value $\overline{v}$ in some time frame t, then g would always remain fixed at $\overline{v}$ from that time frame onwards. Thus, gate g can be marked as post-initialization constant. It should be noted that this characterization of gate g holds true even if the state machine is not completely initializable because once gate g attains the logic value $\overline{v}$, it would remain set to $\overline{v}$ in every time frame after t irrespective of whether the machine is totally initialized or not.

If $t_1 > 1$, then once gate g is set to $\overline{v}$, the value assignment would repeat after every t_1 time frames. This is true because since $[g, v, t] \rightarrow [g, v, t - t_1]$, by contrapositive law it follows that $[g, \overline{v}, t] \rightarrow [g, \overline{v}, t + t_1]$. However, if the machine is not completely initialized, then the value of gate g between time frames t (at which g was set to $\overline{v}$) and $t + t_1$ (at which the value of g is again implied to $\overline{v}$), the value of g could either be $\overline{v}$ or X. In that case, g may not be marked as a post-initialization constant to $\overline{v}$. $\square$

Implementation of the Theorem:

Before we discuss our implementation of Theorem 1, it is necessary to understand that from our perspective of identification of untestable faults, identification of $g = v$ being unachievable is sufficient for untestable fault identification. As long as $g = v$ is unachievable, all faults that require the assignment $g = v$ would be untestable. So, we do not need the complete statement of Theorem 1 for untestable fault identification, and Lemma 2 would be sufficient for identification of such faults. However, if untestable faults are classified into multiple categories as in [49], the statement in Theorem 1 can be useful to identify a larger subset of untestable faults in different fault categories.

If we consider the statement in Lemma 2, the conditions that are required to determine if an assignment $g = v$ is unachievable are:

1. The implication $[g, v, 0] \rightarrow [g, v, -t]$ and
2. That g = v is not a constant.

Once the implication engine is built, it is straight-forward to check if $[g, v, 0] \rightarrow [g, v, -t]$ by just performing a transitive closure on the implication graph for the node $g = v$. The way we ensure that $g = v$ is not a constant is through logic simulation using random vectors. During random simulation, one of the following two conditions can occur:

- Via random simulation, if a sequence can set $g = \overline{v}$, then we can definitely conclude that $g = v$ is not a constant. In that case, $g = v$ is marked as unachievable according

```
//Main program
1. Perform direct implications
2. For each gate assignment $\{g = v\}$ and $\{g = \overline{v}\}$:
      checkIfUnachievable($g$, val) //(val $in\{0, 1\}$)
3.    For every gate assignment $\{g = v\}$ and $\{g = \overline{v}\}$:
3.1   Perform indirect and extended backward implications
3.2   checkIfUnachievable($g$, val) //(val $in\{0, 1\}$)
4. For each gate $g$:
4.1   if($g = val$) unachievable (val $\in \{0, 1\}$):
        all faults that require ($g = val$) as a necessary condition for
        their detection are untestable.
        else: perform single line conflict analysis for $g$
//unachievable nets
checkIfUnachievable($g$, val)
1. If$[g, val, 0] \rightarrow [g, val, -t]$
1.1  Perform logic simulation using $k$ random vectors
      (In our framework, $k$ = 10,000)
1.2  if($g = \overline{val}$) during simulation:
      mark ($g = val$) as unachievable
```

Figure 2.5: Algorithm to identify untestable faults using unachievable nets

to Lemma 2, and all faults that require $g = v$ as a necessary condition for their detection are marked as untestable.

- If however, at the end of random simulation, gate g is still uninitialized (i.e. $g = X$), then it cannot be guaranteed that $g = \overline{v}$ is possible, and according to Corollary 1, $g = v$ cannot be marked as unachievable.

Using the discussion provided above, the overall algorithm to identify untestable faults with the addition of identification of unachievable nets can be summarized as shown in

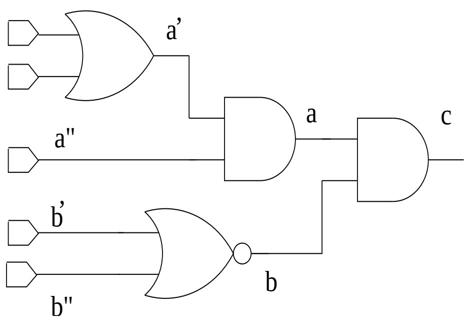


Figure 2.6: Circuit to illustrate multi-node impossible combination

Figure 2.5

It can be observed that since the success of the technique based on recurrence relations depends upon the identification of implications of the sort $[g, v, 0] \rightarrow [g, v, -t]$, it would not offer identification of additional untestable faults for combinational circuits. Moreover, not all sequential circuits exhibit recurrence relations of the type $[g, v, 0] \rightarrow [g, v, -t]$. In this regard, additional techniques can be used to help identification of additional untestable faults for both combinational and sequential circuits. In the next section, we describe a new approach that aims to identify untestable faults by quick identification of impossible or conflicting value combinations inside the circuit.

2.2.2 Identifi cation of Multiple-line Conflcts

Single line conflict analysis efficiently identifies untestable faults which require an impossible assignment on a single line as a necessary condition for their detection. However, there are faults which may require an impossible combination of values on more than one net in the circuit. Though these faults are also untestable, single line conflict analysis would be incapable of identifying these faults. To address this shortcoming of the single line conflict analysis, we propose a technique to quickly identify impossible combinations on multiple nets, and then use these impossible combinations to identify untestable faults.

Our approach is motivated from the work presented in [14] where conflicting value assignments are identified local to each Boolean gate. To understand the approach employed in [14], consider the circuit shown in Figure 2.6.

The technique presented in [14] would identify the value combination of $\{a = 1, b = 1, c = 0\}$ as impossible to achieve, and then untestable faults would be identified by creating the following sets:

S_0: Set of faults untestable with a = 0. S_1: Set of faults untestable with b = 0. S_2: Set of faults untestable with c = 1.

The faults in S_0, S_1, S_3 require a = 1, b = 1 and c = 0 respectively, as a necessary condition for their detection. Then, the set of untestable faults would be the intersection of S_1, S_2 and S_3. It is important to note that if ($a = v, b = w$) forms an impossible combination pair, then untestable fault identification requires the enumeration of faults untestable due to $a = \overline{v}$ and $b = \overline{w}$.

Now, it is interesting to note from Figure 2.6 that the value combination $\{(a' = 1, a" = 1), (b' = 0, b" = 0), c = 0\}$ also forms a conflict or an impossible assignment. Also, since $Impl[a', 0, 0] \supset Impl[a, 0, 0]$ and $Impl[a", 0, 0] \supset impl[a, 0, 0]$, the set of faults untestable due to $a' = 0$ and $a" = 0$ could potentially be greater than that due to $a = 0$. Similarly, the set of faults that can be identified as untestable due to $b' = 1$ and $b" = 1$ could be greater than that untestable due to $b = 0$. As a consequence, the set of untestable faults identified using this new conflicting combination could be greater than that identified with the original, i.e. $\{a = 1, b = 1, c = 0\}$.

It is evident that the number of sets to be intersected for the conflict $\{(a' = 1, a" = 1), (b' = 0, b" = 0), c = 0\}$ would be greater than that for $\{a = 1, b = 1, c = 0\}$. However, since set-intersection is performed on the fly, the conflicting combination is discarded as soon as the intersection becomes null and a larger set of conflicting values does not hurt as much in terms of computational complexity. A larger set of elements in an impossible combination value set would hurt the most in terms of execution time only if the set-intersection is non-empty until the last intersection is performed, and becomes empty only after the last intersection operation. However, we observed that this does not happen often, and that the computational overhead due to a bigger set of impossible value combinations remains acceptable. These observations form the motivation for our approach.

Before we illustrate our algorithm, we need to define the following terms:

Definition 9 *Non-Terminating necessary Condition set (NTC): NTC for an assignment* $g = v$ *is defined as the set of value assignments* $\{a_i = w_i \mid w_i \in \{0, 1\}\}$ *that are necessary to achieve* $g = v$*. However, there may exist other assignments that are necessary to achieve some or all conditions in NTC.*

For example, in Figure 2.6, $a = 1$ and $b = 1$ are necessary for $c = 1$. However, there exist assignments ($a' = 1, a" = 1, b' = 1 and b" = 1$) that are necessary to achieve $a = 1$ and $b = 1$. Thus, $a = 1$, $b = 1$ forms the NTC for $c = 1$.

Definition 10 *Terminating Necessary Condition set (TNC): TNC for an assignment* $g = v$ *is the set of value assignments* $\{a_i = w_i \mid w_i \in \{0, 1\}\}$ *necessary to achieve* $g = v$ *such that there exist no additional assignments that are necessary to achieve any conditions in this set.*

For example, in Figure 2.6, $a' = 1$, $a" = 1$, $b' = 0$ and $b" = 0$ form the TNC for $c = 1$.

Thus, according to Definitions 9 and 10, the set of conflicting conditions obtained in [14] would be the NTC for any assignment $g = v$. These conflicting conditions would form the TNC only if $|$NTC$|$ = $|$TNC$|$ for any $g = v$.

Our Approach: In our approach, we identify the TNC for any assignment $g = v$ (rather than the NTC). Then the set TNC, $g = \overline{v}$ forms a conflicting assignment. As the size of the TNC is greater than that of NTC, our approach may take more execution time than that taken by the approach in [14], but we guarantee with the following Lemma that we

```
1. Construct implication graph and identify additional
unachievable nets using Lemma 2.
/*identification of impossible combinations */
2. For each gate assignment $g = val$ //(val $in\{0,1\}$)
2.1     Identify the TNC for $g = val$
2.2     Impossible Combination (IC) set = TNC, $g = \overline{val}$
2.3     $i = 0$, $S_{untest} = \phi$.
2.4     For each assignment $a = w$ in IC:
2.4.1       $S_i$ = fault untestable with $a = \overline{w}$.
2.4.2       if($i = 0$), $S_{untest} = S_{untest} \cup S_i$;
            else $S_{untest} = S_{untest} \bigcap S_i$
2.4.3       if($S_{untest} = \phi$), break; else $i++$;
```

Figure 2.7: Algorithm to identify untestable faults using multiple line conflicts

would always identify at least as many (and potentially more) untestable faults as identified in [14].

Definition 11 *Related elements: We say that gates a and b are related elements if there exists at least one path from a to b.*

Lemma 4 *If two related elements a and b exist such that the assignment $a = v$ is a part of TNC for a gate $g = l$ and $b = w$ is a part of NTC for the same gate $g = l$, then $impl[a, \overline{v}, 0] \supseteq impl[b, \overline{w}, 0]$.*

Proof: Since $b = w$ is not a terminating necessary condition for $g = l$, there must exist some necessary conditions to achieve $b = w$. Now, since $a = v$ is a terminating condition for $g = l$ and since a and b are related, then $a = v$ must be a part of the conditions necessary to set $b = w$. This means that in order to set $b = w$, gate a must be set to v, or in other words, $[b, w, 0] \rightarrow [a, v, 0]$. By contrapositive law, $[a, \overline{v}, 0] \rightarrow [b, \overline{w}, 0]$. Thus, $impl[a, \overline{v}, 0] \supseteq impl[b, \overline{w}, 0]$. □

Thus, according to Lemma 4, the implications of the complement of all elements in a TNC are a superset of the complemented related elements in a NTC for any given assignment. Therefore, we would always identify at least as many untestable faults using TNCs as can be identified using NTC as used in [14]. The complete algorithm to identify untestable faults using multiple line conflict analysis is shown in Figure 2.7.

2.2.3 Experimental Results: Theorem 1 and Multiple Line Conflcts

The proposed techniques (Theorem 1 and multiple line conflict analysis) were implemented in C++ and experiments were conducted on ISCAS '85 and ISCAS '89 circuits on a 3.2 GHz, Pentium-4 workstation with 1 GB RAM, with Linux as the operating system.

The experimental results (in terms of untestable faults identified and execution times) obtained using our techniques are shown in Table 2.1. Column 1 in Table 2.1 lists the circuits for which our proposed techniques were applied. Columns 2 and 3 report the number of untestable faults identified (Unt) with the traditional single-line-conflict (S-L-C) analysis along with the time taken for such an analysis. For combinational circuits, the implication engine was built over one time frame, while for the sequential circuits reported here, implications were built over three time frames. Column 4 illustrates the number of untestable faults identified with the single-line-conflict analysis which utilizes the concept of identification of unachievable nets using backward recurrence analysis (as proposed in Theorem 1). As described in the implementation of Theorem 1, random simulation is used to identify unachievable nets for which backward recurrence relations exist. For the results shown in Table 2.1, simulation using 10,000 random vectors was performed for this purpose. Column 5 reports the time taken for untestable fault identification using Theorem 1. It can be observed that for large subset of sequential circuits, Theorem 1 facilitates identification of a significant number of additional untestable faults at practically no or low overhead in terms of execution time.

Table 2.1: Untestable Faults Identified Using Theorem1 and Multiple Line Conflicts

Ckt	**S-L-C**		**S-L-C + Thm-1**		**M-L-C + Thm-1**	
	Unt	Time	Unt	Time	Unt	Time
c432	0	0.01	0	0.01	**2**	0.02
c1908	4	0.42	4	0.43	**9**	1.2
c2670	39	0.33	39	0.34	**93**	1.15
c3540	105	1.7	105	1.8	**137**	5.8
c5315	20	1.01	20	1.03	**58**	2.39
c7552	42	4.5	42	4.6	**66**	6.76
s298	2	0.03	2	0.06	6	0.1
s386	60	0.16	60	0.26	63	0.65
s400	8	0.11	8	0.15	10	0.22
s641	0	0.14	**59**	0.18	**59**	0.20
s713	32	0.18	**101**	0.20	**101**	0.21
s832	0	0.75	0	1.0	**4**	1.4
s1238	9	0.95	9	1.8	**25**	4.5
s1423	9	0.22	9	0.28	**14**	0.96
s5378	879	14.8	879	15.2	**882**	15.6
s9234.1	195	41.6	**202**	44.6	**371**	99.1
s13207.1	397	156.2	397	194.6	**453**	232.1
s15850	445	244.6	**756**	254.8	**835**	394.3
s15850.1	317	104.2	**887**	161.3	**951**	192.6
s38417	356	335.5	**454**	679.1	**511**	787.2
s38584	1691	725.6	**2069**	773.5	**2283**	2187.4

Time measured in seconds

For example, for s641, s731 and s15850.1, the application of the theorem helps single-line-conflict analysis to identify additional 59, 63 and 570 untestable faults respectively, without significant overhead. As previously indicated, the proposed Theorem does not assist in the identification of additional redundancies in combinational circuits or in sequential circuits that do not exhibit backward recurrence relations. Thus, we applied Multiple-line-conflict (M-L-C) analysis on top of the tool that implements Theorem 1, to identify more untestable faults. It can be observed from the results reported in column 6 that this technique not only works well for combinational circuit but it also helps in identification of additional untestable fault even for sequential circuits that exhibit recurrence relations. For example, for c3540 an additional 32 untestable faults were identified (thereby identifying all the 137 untestable faults) in less than 13 seconds. Similarly, for c2670, an additional 54 untestable faults were identified using this technique taking the total number to 93. Also, for circuits such as s38584 that benefited from Theorem 1, this technique facilitated identification of substantial number of additional untestable faults.

Table 2.2 compares our tool with some of the fault-independent techniques proposed earlier. Column 2 in Table 2.2 shows the best results (in terms of the number of untestable faults identified) among Hsiao [14] and MUST [13]. Column 4 shows the number of untestable faults identified in FILL and FUNI [26]. Column 6 shows the number of untestable faults identified by our tool. For most of the circuits, our tool outperforms the earlier fault-independent techniques that targeted untestable fault identification. It should be noted that since MUST [13] is a fault oriented technique (multiple-stem analysis in their technique is performed on a per-fault basis), it performs better than our fault-independent approach for a few circuits (such as s1238 and s13207).

Table 2.2: Comparison with previous fault-independent techniques

Ckt	**Best([14], [13])**		**FILL+FUNI [26]**		**Our Tool**	
	Unt	Time	Unt	Time	Unt	Time
c1908	8	8.7	-	-	**9**	1.2
c2670	**97**	6.32	-	-	**93**	1.15
c3540	131	11.0	-	-	**137**	5.8
c5315	**59**	16.5	-	-	**58**	2.39
c7552	64	18.7	-	-	**66**	6.76
s386	**63**	0.52	36	0.4	63	0.65
s400	10	0.7	**16**	2.0	10	0.22
s641	-	-	55	1.7	**59**	0.20
s713	38	0.38	91	1.7	**101**	0.21
s1238	**58**	11.42	6	4.1	**25**	4.5
s1423	**14**	1.3	5	12.3	**14**	0.96
s5378	**884**	120.0	414	43.5	**882**	15.6
s9234	-	-	257	108.8	**371**	99.1
s9234.1	360	114.4	-	-	**371**	99.1
s13207	**1125**	238	654	150.8	**453**	232.1
s13207.1	451	1795.2	-	-	**453**	232.1
s15850	-	-	816	114.0	**835**	394.3
s15850.1	368	710.2	-	-	**951**	192.6
s38417	466	6456.2	381	373.4	**511**	787.2
s38584	1923	6553.5	1582	405.4	**2283**	2187.4

Time measured in seconds

2.3 What Next?

In the preceding sections, we presented new techniques to identify untestable faults through clever observations about sequential implications. Now we present techniques which can further improve the capability of our tool to identify untestable faults via logic implications.

First we present a novel technique to enhance Boolean learning through a new class of logic implications called *extended forward implications*. Using a new concept called *implication-frontier*, extended forward implications are able to identify those non-trivial relationships which previous techniques fail to identify. For single-line and multiple-line conflict analysis, unexcitable and unobservable nets due to a particular logic assignment $g = v$ are enumerated using implications of $g = v$. Therefore, increasing the power of implications would enable identification of more unexcitable/unobservable nets due to a given assignment, translating to an increased number of untestable faults identified. Also, the effectiveness of Theorem 1 described earlier is based on the strength of the implication engine. Thus, increasing static learning through implications can potentially pay rich dividends in terms of untestable fault identification. Moreover, since logic implications find applications in many other areas such as pattern-generation, model-checking, logic optimization, etc., the additional knowledge provided by the new extended forward implications can play a key role in many facets of test and verification.

Next, we present a new concept called *dual recurrence relations* in sequential circuits, and propose another Theorem based on this concept to quickly identify sequentially untestable faults. This theorem efficiently looks for special characteristics exhibited by sequential implications to deduce untestable faults.

Details of both techniques are discussed in the following subsections.

2.3.1 Enhancing Implications

As mentioned previously, logic implications form the core element several tools geared towards the identification of untestable faults. Thus, efficiently increasing the strength of the implication engine would potentially have a wide impact on the performance of all such tools.

In this section, we propose a new technique to improve Boolean learning through logic implications. This results in a new class of implications, termed as extended forward implications. Before explaining extended forward implications, we define a new concept called implication frontier (I-Frontier), and provide a motivating example.

Definition 12 *Implication Frontier (I-Frontier): The I-Frontier of a logic assignment $g = v$ is the set of gates that are currently unspecified but each gate in the set has one or more inputs implied by $g = v$. If g_{ik} represents the k^{th} input of an n-input gate g_i, then I-Frontier can be represented as:*

I-Frontier$[g = v]$: $g_i \mid g_i \notin Impl[g, v, 0], g_{ik} \in Impl[g, v, 0], 1 \leq k \leq n$

The motivation behind extended-forward implications can be understood through the following example:

Example 3: $A = 1$ in Figure 2.8 does not imply any assignments (by direct, indirect and extended backward implications). Thus, $Impl[A, 1, 0] = \{(A, 1, 0)\}$ (reflexive property). The I-Frontier of $A = 1$ consists of two gates: gate C and gate G. Each of these gates has one specified input (gate A), and one unspecified input (gate B). Let us focus on gate C. If we set gate B to logic 0 (the unspecified input of gate C), and simulate the circuit with $Impl[A, 1, 0] \cup Impl[B, 0, 0]$, a set ($R_0$) of new logic assignments can be learned: $(C = 0), (G = 1), (H = 0)$.

Next, if we set $B = 1$, and again simulate the circuit, another set (R_1) of new assignments can be learned: $(C = 1), (D = 1), (E = 1), (F = 1), (G = 0), (H = 0)$. Now,

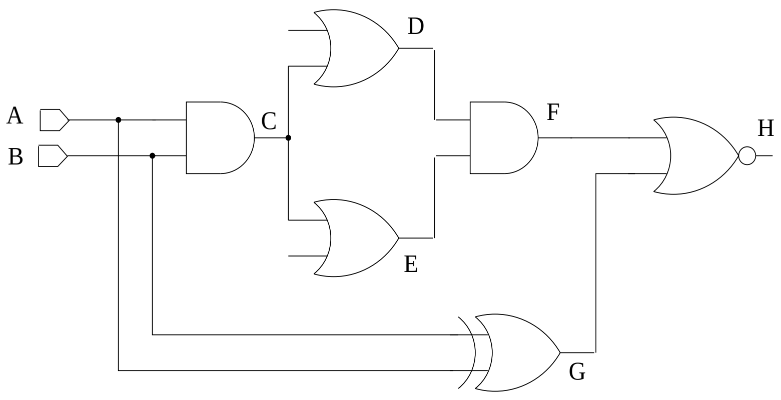


Figure 2.8: Illustration of n-cycle constant

$Impl[A,1,0] \cup Impl[B,0,0]$ implies R_0 while $Impl[A,1,0] \cup Impl[B,1,0]$ implies R_1. Thus, $R_0 \bigcap R_1 \in (Impl[A,1,0] \cup Impl[B,0,0]) \bigcap (Impl[A,1,0] \cup Impl[B,1,0])$.

Or, $R_0 \bigcap R_1 \in Impl[A,1,0] \cup (Impl[B,0,0] \bigcap Impl[B,1,0])$;

Since $Impl[B,0,0] \bigcap Impl[B,1,0] = \phi$, $R_0 \bigcap R_1 \in Impl[A,1,0]$. Finally, because $R_0 \bigcap R_1$ = ($H = 0$), we can learn that $[A,1,0] \rightarrow [H,0,0]$.

By setting the unspecified input of C (a gate in the I-Frontier$[A = 1]$) to both logic values, we were able to identify a new logic implication which was missed by all of direct, indirect and extended backward implications. □

The main idea behind extended forward implications is to learn new relations using the I-Frontier for an assignment $g = v$. Now we formally define extended forward implications (EF for short):

Definition 13 *Extended Forward (EF) Implications: If $g_i \in$ I-Frontier$[g = v]$, then:*

a) If $g_i k$ is the only input of g_i that is currently unspecified,

- R_0 = *logic-simulate(*$Impl[g,v,0] \cup Impl[g_{ik},0,0]$*).*
- R_1 = *logic-simulate(*$Impl[g,v,0] \cup Impl[g_{ik},1,0]$*).*

(b) If g_i has more than one unspecified input,

- R_0 = *logic-simulate(*$Impl[g,v,0] \cup Impl[g_i,0,0]$*)*

- R_1 = *logic-simulate(*$Impl[g,v,0] \cup Impl[g_i,1,0]$*)*

EF$\{g = v\}$ = $R_0 \bigcap R_1$*;* $Impl[g,v,0] \cup = EF\{g = v\}$.

The motivation behind extended-forward (EF) implications is to push the envelope of implications for $g = v$ beyond the I-Frontier. For case (a) above, the attempt to go beyond the I-Frontier is performed by trying both logic values for the unspecified input of g_i $g_{ik} = 0 and g_{ik} = 1$ and taking an intersection of the set of new logic assignments for each logic value. For case (b) more than one inputs of g_i are unspecified: it would be too expensive (computationally) to try the complete value combinations for all unspecified inputs. Instead, both logic values for the gate output $g_i = 0$ and $g_i = 1$ are simulated. Since the underlying concept behind extended forward implication tries to extend implications beyond the point forward implications reach (bounded by the I-Frontier), hence the name.

Now we provide important characteristics of EF implications and I-Frontier through the following Lemmas. As explained later, these characteristics are used in making EF implications efficient.

Lemma 5 *If* $Impl[g_1,v,0] \supseteq Impl[g_2,w,0]$*, I-Frontier*$\{g_1 = v\} \supseteq$ *I-Frontier*$\{g_2 = w\}$.

Proof: By contradiction, assume that $Impl[g_1,v,0] \supseteq Impl[g_2,w,0]$ and I-Frontier$\{g_1 = v\} \subset$ I-Frontier $\{g_2 = w\}$. Thus, there exists at-least one gate g_i, such that $g_i \in$ I-Frontier$\{g_2 = w\}$ and $g_i \notin$ I-Frontier$\{g_1 = v\}$. By definition of I-Frontier, there must exist at-least one input g_{ik} of g_i such that $g_{ik} \in Impl[g_2,w,0]$ and $g_{ik} \notin Impl[g_1,v,0]$. However, this would violate our initial assumption that $Impl[g_1,v,0] \supseteq Impl[g_2,w,0]$; which proves the Lemma. $\square$

Lemma 6 *If* $Impl[g_1,v,0] \supseteq Impl[g_2,w,0]$*, EF*$\{g_1 = v\} \supseteq$ *EF*$\{g_2 = w\}$.

Proof: Given $Impl[g_1,v,0] \supseteq Impl[g_2,w,0]$. Consider Definition 13: to enumerate EF$\{g_1 = v\}$ and EF$\{g_2 = w\}$, I-Frontier$\{g_1 = v\}$ and I-Frontier$\{g_2 = w\}$ are used.

For each gate g_i in these I-Frontiers, R_0 and R_1 are created and intersected. Thus, the number of EF implications learned is directly proportional to the size of the I-Frontier, and the size of R_0 and R_1 for each gate in the I-Frontier. From Lemma 5, since $Impl[g_1, v, 0] \supseteq Impl[g_2, w, 0]$, I-Frontier$\{g_1 = v\} \supseteq$ I-Frontier$\{g_2 = w\}$. Also, since R_0 and R_1 are proportional to $Impl[g_1, v, 0]$ and $Impl[g_2, w, 0]$, it follows that R_0 and R_1 with $\{g_1 = v\}$ will be a superset of the corresponding sets for $\{g_2 = w\}$. Thus, EF$\{g_1 = v\}$ $\supseteq$ŁEF$\{g_2 = w\}$.□

While learning additional Boolean relationships through EF implications can be useful, this learning should not come at the cost of high computational overhead. Observe from the equations used in case (a) and case (b) in Definition 13 that the computation of EF$\{g = v\}$ involves two passes of logic simulation corresponding to each gate in the I-Frontier of $\{g = v\}$. Although EF implications can significantly increase the number of implications learned, the cost associated with logic simulation can potentially become prohibitive as circuit size grows. The time spent in logic simulation is worthwhile if all the gates in the I-Frontier result in new EF implications. However, we observed that a large subset of gates in the I-Frontier do not result in identification of new implications, but still account logic simulation time. Therefore, techniques that can quickly identify and prune out such 'useless' gates from the I-Frontier is desirable. In the following discussion, we describe two efficient techniques for this purpose.

2.3.2 EF Pruning Techniques

A. Pruning Based on I-Frontier: First, we present an important characteristic of I-Frontiers through Lemma 7. This characteristic can prove vital not only in improving the efficiency of learning EF implications, but also accelerate any other Boolean learning technique based on simulation (e.g. extended backward learning).

Lemma 7 *If [I-Frontier$\{g_1 = v\}$ $\bigcap$ I-Frontier$\{g_2 = w\}$] = ϕ, simulating $\{Impl[g_1, v, 0] \cup Impl[g_2, w, 0]\}$ would not result in the identification of any new value assignments.*

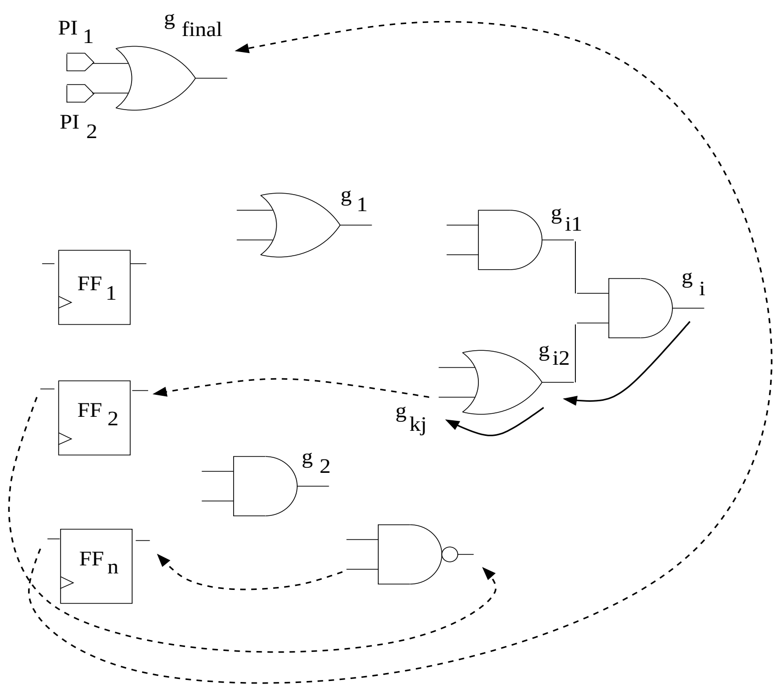


Figure 2.9: Illustration for Lemma 7

Proof: We prove this Lemma by contradiction. Without loss of generality, consider a sequential circuit with only two-input gates and flip-flops. Refer to Figure 2.9 for the following discussion.

Statement of contradiction: By contradiction assume [I-Frontier$\{g_1 = v\}$ $\bigcap$I-Frontier$\{g_2 = w\}$] = ϕ, but a set of new logic assignments, S, is obtained on simulating $Impl[g_1, v, 0] \cup Impl[g_2, w, 0]$.

Let, $g_i \in S$ (i.e. $\{g_i = v_i\} \in S$ is learned through simulating $\{Impl[g_1, v, 0] \cup Impl[g_2, w, 0]\}$). Note that if $\{g_i = v_i\} \in S$, $\{g_i = v_i\} \notin \{Impl[g_1, v, 0] \cup Impl[g_2, w, 0]\}$, and vice versa.

Step 1: Since $g_i \in S$, either $\{g_i = 1\} \in S$ or $\{g_i = 0\} \in S$. For ease of understanding, let g_i be AND gate as shown in Figure 2.9, and have two gates g_{i1} and g_{i2} as its inputs.

- Case A: Assume $\{g_i = 1\} \in S$. Now, $\{g_i = 1\}$ can be learned through the simulation of $\{Impl[g_1, v, 0] \cup Impl[g_2, w, 0]\}$, if:

  1. $g_{i1} = 1$ and g_{i2} = 1 are specified prior to logic simulation; OR

2. $g_{i1} = 1$ and g_{i2} = 1 are learned during logic simulation.

Consider part 1. Both $g_{i1} = 1$ and $g_{i2} = 1$ can be specified prior to logic simulation only if one of these conditions is true:

i. $\{(g_{i1} = 1), (g_{i2} = 1)\} \in Impl[g_1, v, 0]$

ii. $\{(g_{i1} = 1), (g_{i2} = 1)\} \in Impl[g_2, w, 0]$

iii. $(g_{i1} = 1) \in Impl[g_1, v, 0]$ and $(g_{i2} = 1) \in Impl[g_2, w, 0]$

iv. $(g_{i1} = 1) \in Impl[g_2, w, 0]$ and $(g_{i2} = 1) \in Impl[g_1, v, 0]$

If either condition (i) or (ii) were true, then $(g_i = 1) \in \{Impl[g_1, v, 0] \cup Impl[g_2, w, 0]\}$ (by indirect implications). Since $\{g_i = 1\} \in S$, conditions (i) and (ii) are not possible. If either condition (iii) or (iv) were true, then by Definition 12, $g_i \in$ [I-Frontier$\{g_1 = v\} \bigcap$ I-Frontier$\{g_2 = w\}$]. This conflicts the assumption that [I-Frontier$\{g_1 = v\} \bigcap$ I-Frontier$\{g_2 = w\}$] = ϕ.

Thus, part 1 cannot be true if $\{g_i = 1\} \in S$. As a result, both $g_{i1} = 1$ and $g_{i2} = 1$ must also be learnt during logic simulation.

- Case B: Assume that $\{g_i = 0\} \in S$. $\{g_i = 0\}$ can be learnt through the simulation of $\{Impl[g_1, v, 0] \cup Impl[g_2, w, 0]\}$, if:

1. $g_{i1} = 0$ or $g_{i2} = 0$ is specified prior to logic simulation; OR

2. $g_{i1} = 0$ or $g_{i2} = 0$ is learnt during logic simulation.

Again, consider Part-1. If either $g_{i1} = 0$ or $g_{i2} = 0$ is specified prior to logic simulation, then $\{g_i = 0\}$ will also be specified prior to simulation. But, by assumption, $g_i = 0 \in S$ (learnt after simulation). As a result, part-1 cannot be true. Thus, either $\{g_{i1} = 0\}$ or $\{g_{i2} = 0\}$ must be learnt during logic simulation.

From Case A and Case B, if $g_i \in S$, at least one gate at the input of g_i must also be learnt through simulation. That is, either $g_{i1} \in S$ and/or $g_{i2} \in S$. Assume $g_{i2} \in S$.

Step 2: The arguments used for g_i can also be applied for g_{i2}. Given [I-Frontier$\{g_1 = v\} \bigcup$ I-Frontier$\{g_2 = w\}$] = ϕ, $g_{i2} \in S$ is possible only if there exists some gate g_{kj} at the input of gate g_{i2} such that $g_{kj} \in S$. Since, $g_i \in S$ is possible only if $g_{i2} \in S$, by transitivity, it follows that $g_i \in S$ is possible only if $g_{kj} \in S$.

Step 3: Applying this argument inductively, we would reach gate g_{final} in the input-cone of g_i such that both inputs of g_{final} are primary inputs (say PI_1 and PI_2). Note that, as shown in Figure 2.9, g_{final} may be reached after passing through several gates, which may include flip-flops. From induction, $g_i \in S$ is possible only if $g_{final} \in S$. From Step 1, if $g_{final} \in S$ either $PI_1 \in S or PI_2 \in S$. However, since both PI_1 and PI_2 are primary inputs, new assignments on PI_1 or PI_2 cannot be learnt through simulation. That is, $PI_1 \notin S$ and $PI_2 \notin S$. As a result, $g_{final} \notin S$. Since $g_i \in S$ is possible only if $g_{final} \in S$, $g_i \in S$. Thus, if [I-Frontier$\{g_1 = v\} \bigcap$ I-Frontier$\{g_2 = w\}$] = ϕ, no new assignments can be identified by simulating $\{Impl[g_1, v, 0] \cup Impl[g_2, w, 0]\}$. □

Let us understand how Lemma 7 can be used to compute EF implications more efficiently. From the equations in case (a) and case (b) in Definition 13, the implications of $g = v$ are combined with the implications of gate g_{ik} or g_i, followed by simulation.

Proposition 1 *Using Lemma 7, new EF implications would not be learnt if:*

i. *For equations in case (a) for Definition 13:*

   [I-Frontier$\{g = v\} bigcap$ I-Frontier$\{g_{ik} = 0\}$] = ϕ or if [I-Frontier$\{g = v\} \bigcap$ I-Frontier$\{g_{ik} = 1\}$] = ϕ;

ii. *For equations in case (b) for Definition 13:*

   [I-Frontier$\{g = v\} \bigcap$ I-Frontier$\{g_i = 0\}$] = ϕ or if [I-Frontier$\{g = v\} \bigcap$ I-Frontier$\{g_i = 1\}$] = ϕ.

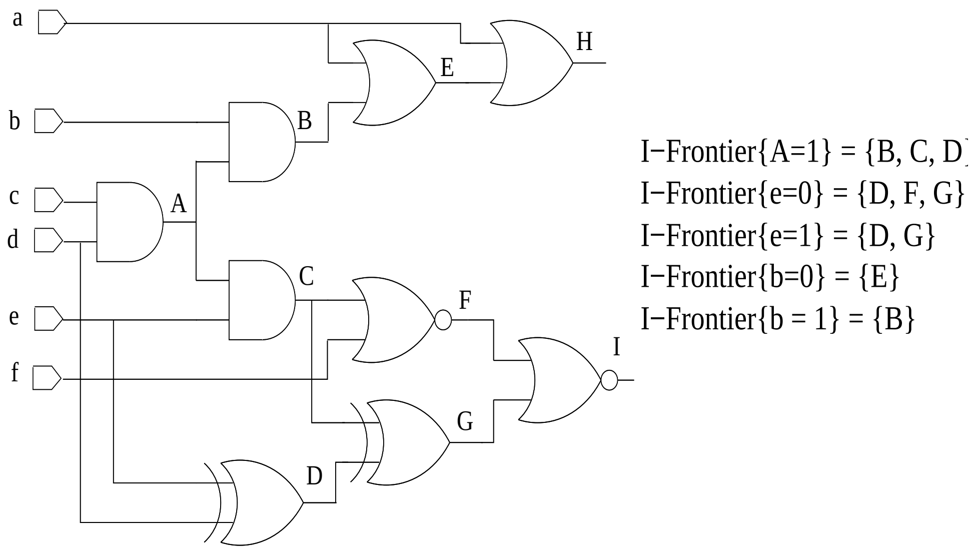


Figure 2.10: Illustration of pruning based on I-Frontier

In either case, g_i can be dropped from the I-Frontier of $\{g = v\}$ without loss of new EF implications.

Consider the impact of Lemma 7: by performing a simple check (intersection) on the I-Frontiers, Lemma 7 enables the identification those gates which do not contribute to any new learning via EF implications. By removing such gates from the I-Frontier, saving in simulation cost is achieved. We observed that this pruning technique reduced the execution time of learning implications by as much as 30% - 40% for benchmark circuits.

Consider the following example for an illustration of Proposition 1.

Example 4: Figure 2.10 shows the combinational portion of a circuit and the I-Frontiers of some nets of interest.

By direct, indirect and extended backward implications, $Impl[A, 1, 0] = \{(A = 1), (c = 1), (d = 1)\}$. Assume that EF$\{A = 1\}$ need to be enumerated. Candidates in the I-Frontier for $\{A = 1\}$ are $\{B, C, D\}$. First consider candidate B. Since [I-Frontier$\{A = 1\} \bigcap$ I-Frontier$\{b = 0\}$] = ϕ, Proposition 1 allows us to drop gate B from I-Frontier$\{A = 1\}$. It can be verified that simulating $Impl[A, 1, 0] \cup Impl[b, 0, 0]$ would not result in any new logic assignments.

Now, consider candidates C and D. The unspecified input of gates C and D is gate e. Since [I-Frontier$\{A = 1\} \bigcap$ I-Frontier$\{e = 1\}] \neq phi$ and [I-Frontier$\{A = 1\} \bigcap$ I-Frontier$\{e = 0\}] \neq phi$, EF implication process is performed for both candidates C and D. EF for candidates C and D results in identification of new implications ($\{G = 1\}, \{I = 0\}$). □

B. Pruning Based on Value: In this heuristic, we use the characteristics of EF implications described earlier. Specifically, using Lemma 6, we propose the following Lemma to further prune out those gates which do not contribute to identification of EF implications.

Lemma 8 *Assume that the value of gate g_i is controlled to logic w when one or more inputs $g_{ik} = c$. If:*

(i). EF has been performed for all inputs $g_{ik} = c$, and

(ii). extended backward implications are performed for $\{g_i = w\}$;

Then: new EF implications would not be obtained for $\{g_i = w\}$.

Proof: Without loss of generality, assume g_i to be AND gate. Let g_{ik} be the gate at k^{th} input of gate g_i. Thus, $\forall k \{g_i k = 0\} \rightarrow \{g_i = 0\}$ (g_i is controlled to logic 0, when one or more inputs $g_{ik} = 0$). As a result, $\forall k, Impl[g_{ik}, 0, 0] \supseteq Impl[g_i, 0, 0]$. From Lemma 6, $\forall k$, EF$\{g_{ik} = 0\} \supseteq$ EF$\{g_i = 0\}$.

Let us assume that gate g_i has n gates $g_{i1} - g_{in}$ as inputs. Thus,

$$EF\{g_{i1} = 0\} \supseteq EF\{g_i = 0\} \tag{2.4}$$

Intersecting $EF\{g_{i2} = 0\}$ on both sides of equation 2.4,

$$[EF\{g_{i1} = 0\} \bigcap EF\{g_{i2} = 0\}] \supseteq [EF\{g_i = 0\} \bigcap EF\{g_{i2} = 0\}] \tag{2.5}$$

Since $EF\{g_{i2} = 0\} \supseteq EF\{g_i = 0\}$ (note right side of equation 2.5),

$$[EF\{g_{i1} = 0\} \bigcap EF\{g_{i2} = 0\}] \supseteq [EF\{g_i = 0\}] \tag{2.6}$$

Performing this intersection operation for all inputs,

$$[EF\{g_{i1} = 0\} \bigcap EF\{g_{i2} = 0\}....EF\{g_{in} = 0\}] \supseteq EF\{g_i = 0\} \tag{2.7}$$

Assume that each term on the left-hand-side of equation 2.7 has been individually evaluated (i.e. $EF\{g_{ik} = 0\}$ has been evaluated for all inputs). By definition of extended backward implications, left side of equation 2.7 can be identified by performing extended backward implications on $\{g_i = 0\}$. Thus, equation 2.7 implies that if $EF\{g_{ik} = 0\}$ has been performed for all inputs of g_i, performing extended-backward implications on $\{g_i = 0\}$ would be sufficient; new EF implications would not be identified for $\{g = 0\}$. Similar reasoning applies to other gates for which $g_{ik} = c$ controls the output of g_i to logic w (e.g. if g_i is OR gate, any input $g_{ik} = 1$ controls output of g_i to logic 1). □

Consider the impact of Lemma 8: For any gate g_i with n inputs $g_{i1} - g_{in}$, once both conditions in Lemma 8 are satisfied, $EF\{g_i = w\}$ can be completely avoided. By not performing $EF\{g_i = w\}$, logic simulation is avoided for all gates in the I-Frontier for $\{g_i = w\}$.

For better understanding, consider the following example:

Example 5: Let us identify $EF\{A = 0\}$ in Figure 2.11 using two procedures: in procedure 1 we explicitly perform $EF\{A = 0\}$ using Definition 13 and in procedure 2, we use Lemma 8.

Procedure 1: Since B is the only gate in the I-Frontier of $\{A = 0\}$, $EF\{A = 0\}$ would be enumerated as: R_0 = logic-simulate($Impl[A, 0, 0] \cup Impl[c, 0, 0]$);

Or, R_0 = $\{(B = 0), (C = 1), (D = 0)\}$

R_1 = logic-simulate($Impl[A, 0, 0] \cup Impl[c, 1, 0]$)

Or, R_1 = $\{(B = 1), (D = 0)\}$

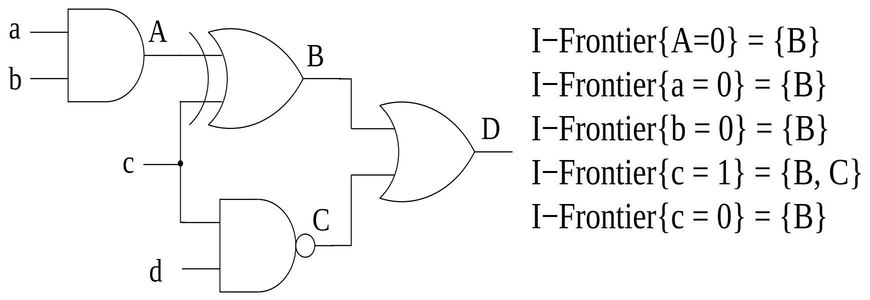


Figure 2.11: Illustration of pruning based on I-Frontier

Thus, $EF\{A = 0\}$ = $R_0 \bigcup R_1$ = $\{(D = 0)\}$.

Procedure 2: It can be verified that $EF\{a = 0\} = EF\{b = 0\} = \{(D = 0)\}$. Now, by performing extended backward implications on $\{A = 0\}$, we can directly learn the implication $[A, 0, 0] \rightarrow [D, 0, 0]$.

Unlike Procedure 1, Procedure 2 learns $EF\{A = 0\}$ without explicitly performing logic simulation. Once $EF\{a = 0\}$ and $EF\{b = 0\}$ are performed, by Lemma 8, only extended backward implications on $\{A = 0\}$ are needed to identify $EF\{A = 0\}$ (eliminating the need for additional logic simulations). □

This pruning technique based on value is more powerful than the pruning based on I-Frontier and helped achieve an additional 10% -15% reduction in execution time on top of the latter. These two pruning techniques together make EF implications an efficient way of improving the l earning capabilities of our implication engine. Figure 2.12 is complete algorithm for enumerating implications:

```
/* Implications Engine */
1. For all gate assignments in levelized order ($\{g = w\}$, $w \in \{0, 1\}$)
2.      if $w$ is the non-controlling value of $g$ (pruning based on value)
2.1         Identify I-Frontier$\{g = w\}$
2.2         Prune I-Frontier based on Proposition 1
2.3         Perform $EF\{g = w\}$ using Defi nition 13
```

Figure 2.12: Algorithm to enumerate extended forward implications

2.3.3 New Theorem using Dual Recurrence Relations

Learning non-trivial Boolean relationships via EF implications can prove instrumental only when these relationships are analyzed and applied suitably. In this section, we propose a new Theorem that efficiently utilizes the extra knowledge provided by EF implications towards identifying untestable faults. This Theorem is based on a new concept called dual recurrence relations. Dual recurrence relations explore certain unique characteristics of sequential implications to help derive conclusions about untestable faults.

Earlier, we proposed the following Lemma based on the knowledge of recurrence relations was used to identify unachievable assignments:

For a gate g, if

a. A backward recurrence relation exists for $g = v$, and

b. $g = v$ is not a constant assignment;

Then the value assignment g = v is unachievable.

If condition (a) is true for a gate g (i.e. a backward recurrence relation exists for $\{g = v\}$), we perform logic simulation using random vectors (starting from unknown initial state) to determine if condition (b) also holds true for g. Condition (b) is declared true only if $g = \overline{v}$ is achieved during simulation, which results in the assignment $g = v$ being declared as unachievable according to the Lemma.

However, it is possible that during simulation gate g always remains un-initialized, i.e. g always remains X. In such a scenario, our simulation based technique cannot make any decisions regarding the controllability of gate g.

To address this shortcoming, we propose a new and more powerful Theorem, which would enable meaningful decisions to be made about the controllability of a gate without performing random simulation.

We define dual-recurrence relations and discuss Lemma 9 before stating the Theorem.

Definition 14 *A dual recurrence relation exists for a gate g if a backward recurrence relations exist for $\{g = v\}$ and $\{g = \overline{v}\}$ (i.e. $[g, v, t] \rightarrow [g, v, t - t_1]$ and $[g, \overline{v}, t] \rightarrow [g, \overline{v}, t - t_1]$ for $t_1 > 0$).*

Definition 15 *If $\{g = v\}$ is a sequentially constant assignment, $\{g = v\}$ is true in every time frame of the sequential circuit after circuit synchronization. If the sequential machine powers up into an illegal starting state that sets gate g to $\overline{v}$, the machine eventually goes into a state that sets $g = v$ after which g retains value v in all time frames.*

Lemma 9 *If a dual recurrence relation exists for a gate g, then neither $\{g = v\}$ nor $\{g = \overline{v}\}$ is a constant assignment.*

Proof: If a dual recurrence relation exists for gate g, then

$$[g, v, t] \rightarrow [g, v, t - t_1] \tag{2.8}$$

$$[g, \overline{v}, t] \rightarrow [g, \overline{v}, t - t_1] \tag{2.9}$$

Applying contrapositive law to equation 2.9, we obtain

$$[g, v, -t] \rightarrow [g, v, t_1 - t] \tag{2.10}$$

Adding (2*t) to the time portion of equation 2.10,

$$[g, v, t] \rightarrow [g, v, t + t_1] \tag{2.11}$$

Equations 2.8 and 2.11 indicate that if the sequential machine enters a state that sets $\{g = v\}$, the assignment $\{g = v\}$ would appear indefinitely at an interval of every t_1 time frames. Thus, according to Definition 15, it can be concluded that the assignment $\{g = \overline{v}\}$ cannot be constant. Using a similar argument, since the dual recurrence relation exists on gate g, $\{g = v\}$ cannot be constant. $\square$

Now, we present our theorem which enables identification of unachievable nets based on dual recurrence relations.

Theorem 2 *If a dual recurrence relation exists for a gate g, then both logic assignments $\{g = v\}$ and $\{g = \overline{v}\}$ are unachievable.*

Proof: Since a dual relationship exists for gate g, $[g, v, t] \rightarrow [g, v, t - t_1]$ (with $t_1 > 0$). Also, since $\{g = v\}$ is not constant (Lemma 9), then according to the Lemma 2, $\{g = v\}$ is unachievable. Using a similar argument, $\{g = \overline{v}\}$ is unachievable. $\square$

It should be noted that if a dual recurrence relation exists for gate g, random logic simulation employed in our previous approach (Lemma 2) would return with $\{g = X\}$. In such a scenario, we would not be able to make any decision about the controllability characteristics of g. However, Theorem 2 enables us to quickly learn that both logic assignments on gate g ($g = v$ and $g = \overline{v}$) are unachievable. As shown in the results, this additional knowledge results in a significant increase in the number of untestable faults identified.

Finally, we provide the following Lemma that applies Theorem 2 towards identification of untestable faults.

Lemma 10 *If a dual recurrence relationship exists for gate g, all faults that require $\{g = v\}$ or $\{g = \overline{v}\}$ as a necessary condition for their detection would be untestable.*

Proof: Since a dual recurrence relation exists for gate g, assignment $g = v$ is unachievable. Thus, by the hypothesis on unachievable nets presented in Lemma 3, all faults that require assignment $\{g = v\}$ as a necessary condition for their detection are untestable. Similar argument can be used for $\{g = \overline{v}\}$. $\square$

Implementation of Theorem 2

Implementation of Theorem 2 does not add any memory overhead or significant overhead in terms of execution time (unlike Lemma 2 which required random logic simulation).

The algorithm for implementation of Theorem 2 (on top of the Lemma 2) is shown in Figure 2.13.

```
 1. For all gate assignments ($\{g = w\}, w \in \{0, 1\}$)
2.      if $[g, v, 0] \rightarrow [g, v, -t_1]$ and $[g, \overline{v}, 0] \rightarrow [g, \overline{v}, -t_1]$
            mark $\{g = v\}$ and $\{g = \overline{v}\}$ as unachievable.
3.      else if $[g, v, 0] \rightarrow [g, v, -t_1]$
                i. Perform random simulation using 10,000 vectors
                ii. If $\{g = \overline{v}\}$ achieved, mark $\{g = v\}$ as unachievable (Lemma 2)
```

Figure 2.13: Algorithm to implement Theorem 2

Note that we implement new Theorem on top of the Lemma 2 to estimate the increase in the number of untestable faults identified by Theorem 2 on top of Lemma 2. As we show in our results, the contribution of the new Theorem is significant.

2.3.4 Experimental Results: EF implications and Dual-Recurrence Relations

The proposed techniques were implemented in C++ and experiments were conducted on ISCAS '85 and ISCAS '89 circuits on a 3.2 GHz, Pentium-4 workstation with 1 GB RAM, with Linux as the operating system. Table 2.3 illustrates the experimental results (# of untestable faults identified and execution times) obtained using our techniques. For each circuit listed in Column 1, Columns 2 and 3 respectively report the # of untestable faults identified (UNT) and the time taken by our previous techniques (Lemma 2 and multiple line conflict analysis). Column 4 shows the # of untestable faults identified for each of the circuits with only extended forward implications (no Theorem 2). It can be seen from Column 4 that for most of the circuits, the # of untestable faults identified increased when extended forward implications were incorporated into the framework.

Table 2.3: Untestable Faults Identified Using Theorem2 and EF implications

Ckt	**Lemma 2**		**EF**	**Thm 2**	**EF + Thm 2**	
	Unt	Time	Unt	Unt	Unt	Time
c1908*	9	1.2	9	9	9	1.2
c2670	93	1.15	101	93	**101**	1.15
c3540*	137	5.8	137	137	137	7.5
c5315*	58	2.39	59	58	**59**	2.9
c7552	66	6.76	67	66	**67**	15.1
s386	63	0.65	65	63	**65**	0.7
s400	10	0.22	10	10	10	0.25
s641	59	0.20	59	59	59	0.24
s713	101	0.21	101	101	101	0.29
s1238	25	4.5	28	25	**28**	5.12
s1423	14	0.96	14	14	14	1.07
s5378	882	15.6	884	882	**884**	20.25
s9234	434	142.0	438	3490	**3602**	211.9
s9234.1	371	99.1	382	371	**389**	134.3
s13207	897	127.2	937	2833	**3812**	242.1
s13207.1	453	232.1	457	885	**889**	425.9
s15850	835	394.3	838	4411	**4636**	454.2
s15850.1	951	192.6	954	1043	**1045**	257.1
s38417	511	787.2	511	511	511	1014.1
s38584	228	2187.4	2308	5405	**5616**	2758.5

Time measured in seconds; *: All redundant faults identifi ed

Column 5 shows the # of untestable faults identified only through the use of Theorem 2. It is easy to observe that Theorem 2 cannot aid in the identification of additional untestable faults for combinational circuits because the theorem uses dual recurrence relations to identify untestable faults. For the larger sequential circuits (except for s38417), Theorem 2 significantly increases the # of identified untestable faults. For example, for s9234, s13207, s38584, Theorem 2 increased the # of identified untestable faults by multiples of thousands. Finally, columns 6 and 7 report the # of untestable faults identified and the time taken for analysis when both extended-forward implications and Theorem 2 are applied in conjunction. Key observations made from Table 2.3 are:

- Through our new techniques, we could identify a large subset of additional faults as untestable, with little overhead in terms of execution time. As an example, for circuit s13207 we could identify an additional 2915 untestable faults (over Lemma 2) with an overhead of less than 120 seconds. Even for large circuits such as s38584, we could additionally identify more than 3000 additional untestable faults with an additional overhead of about 600 seconds. An overhead of 600 seconds is insignificant considering that deterministic sequential ATPGs would potentially spend hours targeting untestable faults for such designs.

- By enhancing Boolean learning through EF implications, we were able to identify several critical recurrence relationships which were missed earlier. Knowledge of these additional recurrence relations in turn increases the number of untestable faults considerably. As an example, for s13207, without EF implications, we could identify 2833 faults as untestable using Theorem 2 (Column 5). When EF implications were also used in conjunction with Theorem 2, new recurrence relationships were uncovered, which increased the number of untestable faults by almost 1000 (Column 6) via Theorem 2. Similar results can be observed for s9234, s15850, etc. These results illustrate the significance of EF implications in learning critical and useful sequential relationships which cannot otherwise be learnt.

Finally, even though there exist circuits for which the # of identified untestable faults did not increase, the total # of implications identified for each circuit increased by an average of 15-20% by using EF implications.

Table 2.4 compares our results with some of the previously published work. Columns 2 and 3 report the # of untestable fault identified by MUST [13] (a combination of fault-independent and fault-oriented approaches) and the time taken for analysis, while columns 4 and 5 report results for SFT or single-fault-theorem [24] (based on ATPG). It can be seen that for some small sequential circuits (such as s1238 and s386), ATPG-based techniques can outperform our approach. This can be attributed to: (a) complete branch-and-bound nature of ATPG which may suit some small designs; (b) implications learnt in our engine, like most other fault-independent techniques, do not represent the complete set of logic relations (learning the complete set of implications is computationally very expensive). However, for larger sequential circuits, our technique outperforms both SFT and MUST by large margins in terms of untestable faults.

Table 2.4: Comparison with other engines

Ckt	**MUST [13]**		**SFT [24]**		**EF + Thm 2**	
	Unt	Time	Unt	Time	Unt	Time
c1908	8	8.7	-	-	**9**	1.2
c2670	97	6.32	-	-	**101**	1.15
c3540	127	37.4	-	-	**137**	7.5
c5315	**59**	16.5	-	-	**59**	2.9
c7552	62	24.2	-	-	**67**	15.1
s386	-	-	**70**	51.5	65	0.7
s400	9	5.36	**10**	1.7	**10**	0.25
s641	-	-	0	8.0	**59**	0.24
s713	38	2.9	38	8.9	**101**	0.29
s1238	58	11.42	**69**	14.8	28	5.12
s1423	**14**	6.96	**14**	25.4	**14**	1.07
s5378	622	232	470	862.0	**884**	20.25
s9234	-	-	524	4328.7	**3602**	211.9
s13207	1125	238	961	3237.2	**3812**	242.1
s15850	-	-	448	4197.5	**4636**	454.2
s38417	189	972.9	391	6995.5	**511**	1014.1
s38584	-	-	2142	9076.3	**5616**	2758.5

Time measured in seconds

To illustrate the impact of our contribution, we show the performance of deterministic ATPG for the faults identified as untestable by our tool. Any tool designed for untestable fault identification can enhance the performance of ATPG if:

a. Such a tool can identify those untestable faults which ATPG fails to identify as untestable. In this case, performance gain to ATPG will be in terms of increased effective fault coverage or EFC (EFC = (#faults detected by ATPG)/ (#Total Faults #untestable faults));

b. Such a tool can identify untestable faults faster than ATPG. In this case, performance gain to ATPG will be in terms of improved speed (by ignoring already known untestable faults).

We show the importance of our tool for ATPG with respect to both these criteria. We use an in-house ATPG (based on PODEM [51], using SCOAP [52] testability measures) to target only those faults that are identified as untestable by our tool. For a fair estimation of the performance of ATPG on untestable faults, we perform ATPG in an incremental manner: each sequential circuit is first unrolled into one time frame, and ATPG targets all faults identified as untestable by our tool. Faults identified as untestable by ATPG within one time frame are dropped for further consideration. Next, the circuit is unrolled into two time frames, and only those faults which were not identified as untestable by ATPG in one time frame are now targeted. Again, faults identified as untestable in two time-frames are dropped when the circuit is unrolled into three time frames. Results in the context of ATPGs performance on untestable faults are shown in Table 2.5. Column 2 in Table 2.5 shows the number of untestable faults fed to ATPG for analysis. Note that ATPG is made to target only those faults which we have already identified as untestable using our tool. Columns 3 and 4 show the number of untestable faults identified by ATPG in one time frame, and the corresponding time taken. Next, Columns 5 and 6 show similar results when ATPG was performed on the two time frame unrolled circuit, and Columns 7 and 8 show results for three time frames. Finally, Columns 9 and 10 show the total number of faults identified as untestable by ATPG over three-time frames and the total time taken by ATPG to analyze untestable faults. Key observations made from Table 2.5 are:

- For small circuits, such as s386 and s400, ATPG is also able to identify all faults as untestable (shown in bold in Column 9). Not only that, ATPG is also faster than our fault-independent tool. This is a result of the small-search space that ATPG has to explore for such circuits (these circuits have few inputs and few flip-flops). Thus, the branch and bound nature of ATPG works well for such small circuits.

- For medium sized circuits such as s641-s5378, ATPG is not able to identify all faults as untestable. Our tool can identify several untestable faults which ATPG either aborts or falsely detects in the unrolled circuit (untestable faults get detected because flip-flops are converted into primary inputs in unrolled circuits). Moreover, ATPG may spend significantly more time than taken by our tool to analyze these faults (e.g. s5378).

- or large circuits such as s9234-s38584, ATPG identifies a very small fraction of faults as untestable. The number of untestable faults identified by our tool are more than that identified as untestable by ATPG by several factors (e.g. s9234, s15850 etc.). Moreover, the amount of time spent by ATPG on untestable faults is significantly more compared to the time spent by our tool on these faults (e.g. s38584, s9234 etc).

- As the number of time frames is increased linearly, the search space for ATPG grows exponentially. Thus, ATPG takes more time to analyze untestable faults. Also, with an increase in time frames, ATPG identifies fewer untestable faults (ATPG aborts on most faults when the circuit is unrolled for 3 or more time frames).

Table 2.5: Performance of ATPG for untestable faults

Ckt	**#Faults**	**ATPG* (1-TF)**		**ATPG* (2-TF)**		**ATPG* (3-TF)**		**ATPG (Total)**	
		Unt	Time	Unt	Time	Unt	Time	Unt	Time
s386	65	0	0.01s	65	0.08s	-	-	**65**	0.09s
s400	10	6	0.01s	4	0.01s	-	-	**10**	0.02s
s641	**59**	0	0.02s	0	0.42s	0	1.6s	0	2.04s
s713	**101**	38	0.16s	0	0.45s	0	10.2s	38	10.81s
s1238	**28**	27	0.04s	0	0.01s	0	0.01s	27	0.06s
s5378	**884**	35	0.53s	286	3m45s	100	9m54s	421	13m39s
s9234	**3602**	396	3m25s	52	20m53s	3	50m13s	451	71m31s
s9234.1	**389**	311	2m38s	14	4m42s	0	7m18s	325	14m32s
s13207	**3812**	145	16.1s	776	1m1s	36	11m15s	957	12m32s
s13207.1	**889**	115	8.6s	241	14.4s	9	15m2s	365	15m25s
s15850	**4636**	375	1m5s	49	9m32s	4	72m2s	428	82m39s
s15850.1	**1045**	352	46s	37	1m42s	4	4m22s	393	6m50s
s38417	**511**	123	2m18s	43	7m20s	115	29m48s	281	39m26s
s38584	**5616**	1444	39s	300	61m31s	37	85m10s	1781	147m20s

Time measured in minutes and seconds (m: minutes; s: seconds)

*Number of backtracks used in ATPG: 100,000

To illustrate the impact of our contribution, we show the performance of deterministic ATPG for the faults identified as untestable by our tool.

Experimental results show that our tool identifies those untestable faults which ATPG fails to identify as untestable; also for large circuits, ATPG spends exponential amount of time targeting the faults identified as untestable by our tool. Thus, our tool can benefit ATPG in both increasing the effective fault coverage and in increasing the efficiency of ATPG through a priori knowledge of untestable faults.

2.4 Summary

To summarize, we presented several inexpensive and novel techniques to identify untestable faults using logic implications. We explored certain characteristics exhibited by sequential implications, namely recurrence relations to infer about untestable faults via unachievable nets. We also extended single line conflict analysis to multiple line conflict analysis by quickly identifying impossible value combinations. Finally, since Boolean learning through logic implications form the core element of untestable fault identification, and also for other EDA tools, we enhanced logic implications through a new class of implications, which we refer as extended forward implications. Experimental results showed that using our techniques we could identify several untestable untestable faults which previous techniques failed to identify. Also, the impact of our contributions was shown through the gain provided to ATPG by the untestable faults identified by our tool.

Chapter 3

Verifying Equivalence of Sequential Circuits

In this chapter we present a novel framework to verify equivalence of sequential circuits using Boolean Satisfiability (SAT). We tackle the uninvestigated problem of verifying delay replaceability [27] of two sequential designs. This notion of sequential equivalence does not make any assumptions either about the design-environment or about the design's steady state behavior. Thus, verifying delay replaceability is considered as hard as verifying safe replaceability [27] of sequential circuits (conjectured as EXPSPACE complete). Our SAT-based framework has the following salient features: (a) a methodology to inductively prove equivalence (delay replaceability) of sequential circuits with no assumptions about any initial state; (b) a scheme to include sequential logic implications into the framework; and (c) a low-cost scheme to identify equivalent flip-flop pairs on the fly. Each feature of the framework is discussed in detail, and experimental results for our framework on benchmark circuits are provided.

3.1 Preliminaries

While significant success has been achieved in the area of combinational equivalence checking, the problem of verifying the equivalence of sequential circuits remains challenging. Before describing our framework, we first discuss necessary definitions and present an overview of related work on sequential equivalence checking.

3.1.1 Definitions

Any synchronous sequential design, M, can be represented through its corresponding deterministic finite state machine (FSM). Formally, an FSM can be represented as a quintuple (Q, I, O, λ, δ), where Q is the set of states that the FSM can traverse, I is the set of inputs, O is the set of outputs, λ is the output function, and δ is the next state function. Also, $\lambda(Q, I) : Q \times I \to O$, and $\delta(Q, I) : Q \times I \to Q$. In the following discussion, the quintuple for design M_i is represented as (Q_{M_i}, I_{M_i}, O_{M_i}, λ_{M_i}, δ_{M_i}), unless otherwise specified. Also, we use M_0 to represent the original design (also called specification), and M_1 to represent the design which is a replacement for M_0. M_1 is also referred as the implementation, transformed design or the replacement design.

Definition 16 *Miter: A miter is a design configuration in which two circuits (implementation and specification) are connected such that their corresponding inputs are tied together, and their corresponding outputs feed XOR gates.*

Such a configuration for single-output sequential circuits is shown in Figure 3.1. Note that the two designs in the miter may have different numbers of flip-flops.

Definition 17 *Product Machine: An FSM that represents the behavior of a miter is defined a product machine. More specifically, a product machine represents the partial cross-product of the FSM of the two sequential circuits present in the miter (partial cross-product*

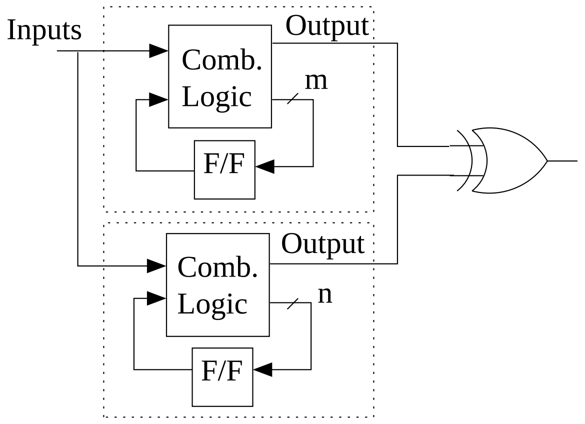


Figure 3.1: Miter for two, single-output sequential circuits (M_0 and M_1)

because some edges in the state-transition graph corresponding to logically opposite values on the same input are invalid for the FSM of the miter).

In any given state, the product machine can only have a binary output; logic 1 if the output response of the two sequential circuits in the miter differs or logic 0 if the output response of the two circuits is identical. As an aside, if the FSM of the two sequential circuits in the miter have n and m states respectively, the product machine would have $n \times m$ states.

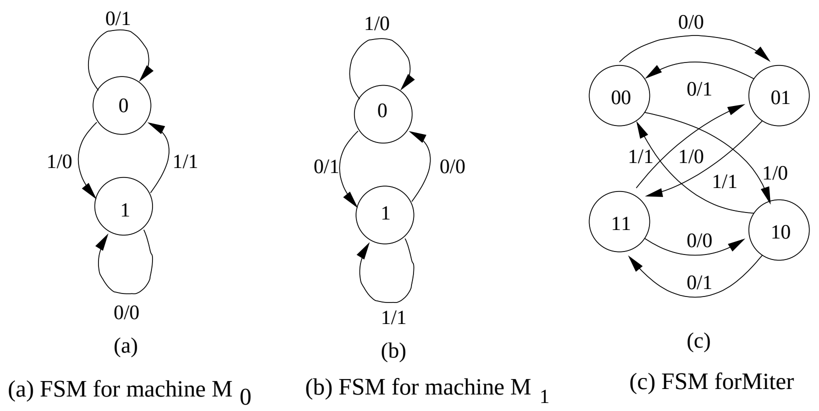


Figure 3.2: Product machine (FSM for the miter) for two sequential circuits (M_0 and M_1)

An example product machine is shown in Figure 3.2. Figure 3.2 (a) and (b) show the FSM for two sequential circuits M_0 and M_1 (each having one memory element, one input and one output). Figure 3.2 (c) shows the FSM corresponding to the miter. As can be observed, the FSM for the miter is a partial cross product on the FSMs for M_1 and M_2 (the edge corresponding to logic 0 (logic 1) at the input in M_1 (M_2) and logic 1 (logic 0) in M_2 (M_1) is absent).

Definition 18 *Strongly Connected Component (SCC): A set of states S for an FSM forms a strongly connected component if it is a maximal set of states such that for any pair of states s_0 and $s_1 \in S$, there exist input sequences π_0 and π_1 such that $\delta(s_0, \pi_0) = s_1$ and $\delta(s_1, \pi_1) = s_0$.*

In other words, a set of states S form an SCC if in the state-transition graph of S there exists a path between every pair of states in S.

Definition 19 *Terminal Strongly Connected Component (TSCC): An SCC, S, such that for each state $s \in S$, for any input sequence π, $\delta(s, \pi) \in S$.*

In other words, an SCC is classified as a TSCC if in the state-transition graph, there exists no outgoing edge from the SCC.

Definition 20 *Equivalent State Pair: States $s_0 \in M_0$ (=(Q_{M_0}, I, O, λ_{M_0}, δ_{M_0})) and $s_1 \in M_1$ (=(Q_{M_1}, I, O, λ_{M_1}, δ_{M_1})) are said to be equivalent (denoted as $s_0 \sim s_1$) if for any sequence of inputs $\pi \in I^*$, $\lambda_{M_0}(s_0, \pi) = \lambda_{M_1}(s_1, \pi)$.*

Definition 21 *Initializing sequence: $\pi \in I^*$ is the initializing sequence for design M_0 if for any pair of states a_0, $b_0 \in Q_{M_0}$, $\delta_{M_0}(a_0, \pi) \sim \delta_{M_0}(b_0, \pi)$.*

Definition 22 *3-valued logic: In 3-valued logic, any gate in a circuit can have a logic value $v \in \{0, 1, u\}$. Here u represents an unknown logic value, and can either be logic 0 or logic 1 depending upon the power-up state of the circuit.*

Definition 23 *Unknown State (x): In 3-valued logic, x represents the unknown state in which all state elements have logic value u.*

Definition 24 *Degree of Determinism: Design M_1 is said to be more deterministic than design M_0 if $\forall \pi \in I^*$, $\{\lambda_{M_0}(x, \pi), \lambda_{M_1}(x, \pi)\} \notin \{(1, u), (0, u)\}$*

Definition 25 *n-cycle Delayed Design [27]: Given design M, the n-cycle delayed design (M^n) is the restriction of M to the set of states $s \mid \exists \pi \in I^n, s' \in Q_M : \delta_M(s', \pi) = s$. In other words, state $s \in M^n$ if and only if there exists a power up state s' in M and an input sequence of length n that drives s' to s.*

3.1.2 Various Notions of Sequential Equivalence

While the following notions of sequential equivalence have been mentioned in the Introduction chapter, we formally describe them in this subsection, elaborating on their applicability while also justifying the need to target delay replaceability.

Definition 26 *Sequential Hardware Equivalence (SHE) [28]: Designs M_0 and M_1 are sequentially hardware equivalent (denoted as $M_0 \approx M_1$) if and only if there exists an input sequence $\pi \in I^*$ such that for any state pair (s_0, s_1) $\in Q_{M_0} \times Q_{M_1}$, we have $\delta_{M_0}(s_0, \pi) \sim \delta_{M_1}(s_1, \pi)$.*

An input sequence that satisfies the definition of SHE is also an initializing sequence for each of the two designs. This also implies that two designs would be equivalent according to the definition of SHE if their respective TSCC's are equivalent (each state in the TSCC of one design would be equivalent to at least one state in the TSCC of the other design). Although the notion of SHE preserves the core behavior (defined by the TSCC) of a design (M_0) with its replacement (M_1), SHE may lead to erroneous replacements because of the following reasons:

- SHE requires the preservation of only one initializing sequence. Assume that design M_0 can be initialized with either sequence I_1 or I_2, while design M_1 can be initialized only with sequence I_1. According to SHE, design M_1 can be verified as a valid replacement for M_0 under I_1. However, if the design-environment can only generate the initializing sequence I_2 and not I_1, the input-output behavior of M_1 will not resemble that for M_0. An example of such an erroneous replacement under SHE is provided in [27].

- SHE applies only to designs that have an initializing sequence. For designs that do not have an initializing sequence, the notion of SHE would not consider a design as a valid replacement even for itself. An example of a design that does not have an initializing sequence would be a sequential circuit that has more than one steady state behavior (more than one TSCC); the circuit may be driven into any one based on the user's requirement.

Due to the reasons mentioned above, it may be necessary to verify sequential equivalence using a stronger notion of sequential equivalence such as 3-valued safe replaceability, safe replaceability or delay replaceability.

Definition 27 *3-valued safe replaceability [30]: M_1 is a 3-valued safe replacement for design M_0 if and only if for all input sequences T, $(\lambda_{M_0}(x, T), \lambda_{M_1}(x, T)) \in \{\ (0, 0),\ (1, 1), (u, u), (u, 0), (u, 1)\}$ under the 3-valued logic.*

In other words, starting from unknown initial state (x), under any given sequence π: (a) if design M_0 has a deterministic output, 3-valued safe replaceability requires the replacement design M_1 to have the same deterministic output as M_0 (b) if design M_0 has an unknown output, u, 3-valued safe replaceability allows the replacement design M_1 to either have a deterministic output (logic 0 or 1) or an unknown output (u).

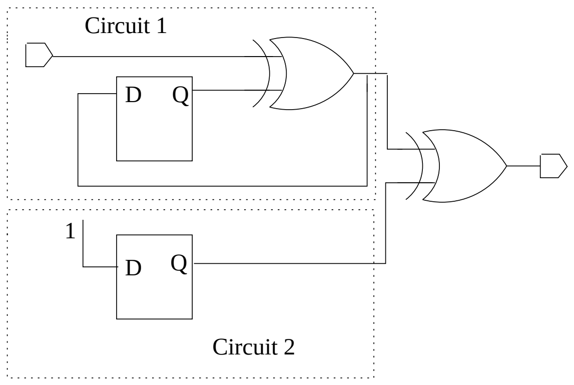


Figure 3.3: Miter to illustrate erroneous replacement with 3-valued safe replaceability

Unlike SHE, the notion of 3-valued safe replaceability preserves all initializing sequences [30]. However, like SHE, 3-valued safe replaceability can be applied only for designs that have an initializing sequence. For designs that are not initializable, 3-valued safe replaceability may lead to erroneous replacements. An example of such a erroneous replacement under 3-valued safe replaceability is shown in Figure 3.3. Consider the miter circuit shown in Figure 3.3. Circuit 1 is uninitializable using 3-valued logic: the output of the flip-flop and the XOR-gate in circuit 1 would always remain at logic u if the initial state is assumed to be u. It can be verified that circuit 2 is a 3-valued safe replacement for circuit 1; using 3-valued logic, the output of (circuit 1, circuit 2) is always $(u, 1)$. However, irrespective of the power-up state of the circuit 1 (logic 0 or 1), it is possible to generate an output (0, 1) for (circuit 1, circuit 2). This indicates that circuit 2 is an erroneous replacement for circuit 1.

Moreover, as explained later, delay replaceability is a stronger notion of sequential equivalence compared to 3-valued safe replaceability. Therefore, verification of delay replaceability would imply verification of 3-valued safe replaceability.

Definition 28 *Safe Replaceability [27]: Design M_1 is a safe replacement for design M_0 (denoted by $M_1 \preceq M_0$) if given any state $s_1 \in Q_{M_1}$ and any finite input sequence $\pi \in I^*$,* *there exists some state $s_0 \in Q_{M_0}$, such that $s_0 \sim s_1$.*

For M_1 to be a valid safe replacement for M_0, safe replaceability requires each state in M_1 to have at least one equivalent state in M_0. In other words, for each finite input sequence I and for each initial state in the transformed circuit, the above definition requires that there must exist a state in the original circuit that has the same output response for that particular input sequence. As a consequence, checking safe replaceability is more difficult than checking replaceability according to other definitions such as SHE: to verify safe replaceability, every state of the transformed circuit needs to be examined. If the BDD for the miter can be constructed, this may be feasible; however, for most circuits, BDDs cannot be build due to memory constraints. Thus, verifying safe-replaceability, conjectured to be EXPSACE complete [27], remains a challenging problem.

Definition 29 *Delay Replaceability [27]: A design M_1 is an n-delay replacement for design M_0 if ${M_1}^n \preceq M_0$. In other words, for any state $s_1 \in Q_{{M_1}^n}$ and any finite input sequence π, there exists some state $s_0 \in Q_{M_0}$ such that $s_0 \sim s_1$.*

Delay replaceability provides a relaxed form of safe replaceability because it checks for safe replaceability between the specification (M_0) and the n-cycle delayed version of the implementation (${M_1}^n$). Like safe replaceability, delay replaceability does not make any assumptions about the steady state behavior of the design or about the design-environment. The only condition imposed by delay replaceability is with respect to ignoring the design's response for the first n cycles after power up. This condition is realistic because, in practice, most designs are allowed to run for a few cycles before the design is used by its environment. Though safe replaceability is a stronger form of design replacement, the following reasons make delay replaceability more practical:

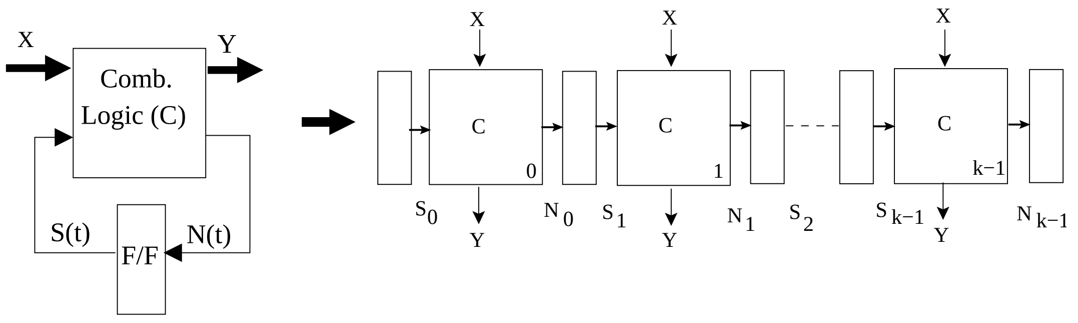
- Ephemeral states [27] are states that have no incoming edges in the state transition graph of a design. Some ephemeral states in the transformed design (M_1) may not be equivalent to any state in the original design (M_0). In such a scenario, the notion of safe replaceability would not qualify M_1 as a valid replacement for M_0. Rejecting a replacement because of ephemeral states may be too conservative as these states cannot be a part of the core behavior of the design. By ignoring the design's response for few cycles after power-up, delay replaceability eliminates ephemeral states from consideration. Thus, while preserving all states that constitute the steady state behavior of design (TSCC), by removing ephemeral states, delay replaceability provides more flexible and realistic replacement criteria.

- The notion of delay replaceability can allow better design optimization compared to safe replaceability.

Now we provide the following Lemma to show that the notion of delay replaceability is stronger than the notion of 3-valued safe replaceability if the replacement design, M_1, is more deterministic than the original design, M_0.

Lemma 11 *If M_1 has more deterministic behavior compared to M_0, and M_1 is verified to be a valid n-delay replacement for M_0, then M_1 is also a valid 3-valued safe replacement for M_0.*

Proof: Assume that design M_1 is an n-delay replacement for design M_0. Thus, given any sequence T, for each state $s_{M_1{}^n} \in Q_{M_1{}^n}$, there exists $s_{M_0} \in Q_{M_0}$ such that $(\lambda_{M_1}(s_{M_1{}^n}, T), \lambda_{M_0}(s_{M_0}, T)) \in \{(0,0),(1,1)\}$. Here ${M_1}^n$ represents n cycle delayed version of M_1 and $Q_{M_1{}^n}$ represents the set of states for ${M_1}^n$. If each state element present in s_{M_0} is changed from deterministic value to logic u, $\lambda_{M_0}(x, T)$ would either be u or retain the same deterministic value as $\lambda_{M_0}(s_{M_0}, T)$. Similarly, if each state element present in $s_{M_1{}^n}$ is changed from deterministic value to logic u, $\lambda_{M_1}(x, T)$ would either be u or retain the same deterministic value as $\lambda_{M_1{}^n}(s_{M_1{}^n}, T)$. Also, since M_1 has more deterministic behavior as compared to M_0, if $\lambda_{M_0}(x, T)$ is deterministic, so must be $\lambda_{M_1{}^n}(x, T)$. Thus, if M_1 is a n delay



(a) Sequential Circuit Model (b) k−Frame ILA

Figure 3.4: Model of a sequential circuit and corresponding k-frame ILA

replacement for M_0, then, $(\lambda_{M_0}(x,T), \lambda_{M_1{}^n}(x,T)) \in \{(u,0),(u,1),(u,u),(1,1),(0,0)\}$. By definition 27, this relationship implies that M_1 is also a valid 3-valued replacement for M_0. □

3.2 Verifying Delay Replaceability

This section presents our SAT-based framework for verifying delay replaceability. First, we define an iterative logic array (ILA) as follows:

Definition 30 *ILA: An iterative logic array of length k consists of k copies of the combinational portion of a sequential circuit connected in such a manner that the next state variables generated by i^{th} copy (N_i) feed the present state variables of $(i+1)^{th}$ copy (S_{i+1}).*

Each copy in the ILA represents the operation of the sequential circuit for one clock-cycle or time-frame; each copy in the ILA is hence also referred to as one time-frame. Figure 3.4(a) and (b) show the generic model of a sequential circuit and the corresponding ILA of size k. In this figure, S_i and N_i represent the present and next state variables for time-frame i and X (Y) represent the set of inputs (outputs) for the sequential circuit.

Such a k-frame ILA forms our model of computation: we use an ILA of the miter circuit to verify delay replaceability. Our model of computation, the k-frame ILA of a miter is shown in Figure 3.5. For simplicity, Figure 3.5 shows the miter for two single-output circuits M_0 and M_1. If the circuits connected together in a miter have more than one output, then corresponding output pairs feed XOR gates, and all XOR gates feed an OR-gate. In Figure 3.5, X represents the inputs for the k-frame ILA; X_i represents the input to frame i; Y_i represents the output of the miter in time-frame i; $S_i^{M_j}$ represents the present state variables for time-frame i to design M_j, and $N_{k-1}^{M_j}$ represents design $M_j's$ next-state variables for the last time-frame. Finally, in the ILA (Figure 3.5), *the present state variables for the first time-frame, i.e., $S_0^{M_0}$ and $S_0^{M_1}$, are fully controllable to any logic value*. In other words, the initial state is unrestricted.

Remark 1:

For the ILA shown in Figure 3.5, because the initial states for both designs are unconstrained, in such a setup, the reachable state-space shrinks monotonically from time-frame 0 to time-frame $k-1$. In other words, $S_j \supseteq S_{j+1}; 0 \leq j \leq k-2$.

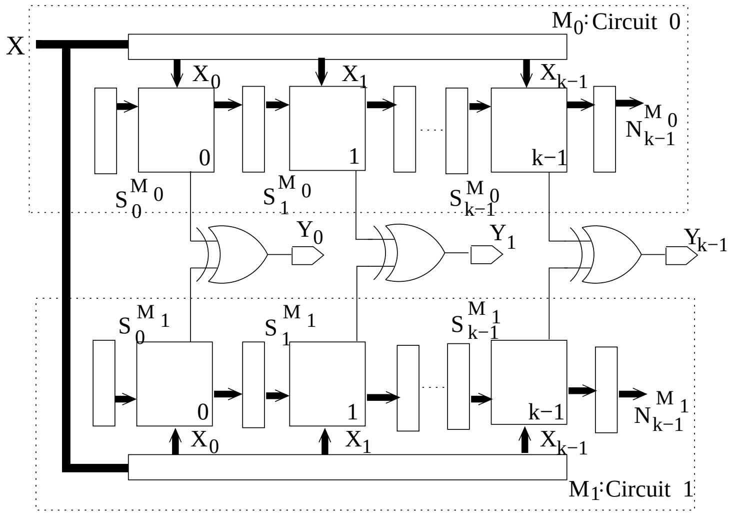


Figure 3.5: Computation model: k-frame ILA of the miter

We now present a Theorem that enables using a bounded model of a sequential circuit (the k-frame ILA of the miter) to verify delay replaceability.

Theorem 3 *If the output of the miter in the last time-frame of a k-frame ILA (Y_{k-1} in Figure 3.5) is unsatisfiable to logic 1, then the transformed circuit (M_1) is a valid $k-1$ delay replacement for the original circuit (M_0).*

Proof: Since Y_{k-1} = 1 is not satisfiable, for all state pairs in $(S_{k-1}^{M_0}, S_{k-1}^{M_1})$, Y_{k-1} = 0 must be true. Given that $S_j \supseteq S_{j+1} \forall j \geq 0$, if the size of the ILA is increased beyond k, the states reachable for any time-frame $t \geq k-1$ would be bounded by $(S_{k-1}^{M_0}, S_{k-1}^{M_1})$. As a consequence, even if the size of the ILA is increased beyond k, it would not be possible to achieve Y_j = 1 for $j \geq k-1$. Thus, if Y_{k-1} = 1 is unsatisfiable in the k frame ILA (Figure 3.5), no input sequence can differentiate the outputs of M_0 and M_1 *after* $k-1$ time-frames.

Next, $S_{k-1}^{M_1}$ represents the set of states reachable from any power-up state in M_1 after $k-1$ time-frames. Thus, by Definition 25, $S_{k-1}^{M_1}$ and the state and output transition functions defined by time-frame $k-1$ represent the $(k-1)$-cycle delayed design M_1^{k-1}. Since $S_{k-1}^{M_0} \subseteq S_0^{M_0}$, for every state in $S_{k-1}^{M_1}$, there exists at least one power up state in $S_0^{M_0}$ such that for all test sequences, $Y_j = 0 \; \forall j \geq k-1$. Thus, for all states $s_1 \in Q_{M_1^{k-1}}, \exists s_0 \in Q_{M_0}$ such that for all test sequences $s_1 \sim s_0$. By Definition 29, M_1 is a valid $k-1$ delay replacement for M_0. $\square$

It should be noted that the converse of Theorem 3 is not true; that is, if Y_{k-1} = 1 is satisfiable in the k-frame ILA of the miter, it *cannot* be concluded that the transformed circuit is not a $k-1$ delay replacement for the original circuit. If Y_{k-1} = 1 is satisfiable, it only indicates that there exists a pair of states $(s_0, s_1) \in (S_{k-1}^{M_0}, S_{k-1}^{M_1})$ such that Y_{k-1} can be set to logic 1. It may still be possible that s_1 is equivalent to some other state $s_j \in S_{k-1}^{M_0}$.

We implement Theorem 3 by first representing the ILA for the miter circuit in the form of a propositional formula, and then using a satisfiability solver (zChaff [35], [40]) to check if $Y_{k-1} = 1$ is satisfiable. If $Y_{k-1} = 1$ is unsatisfiable, then by Theorem 3, the transformed design (M_1) is a valid $k-1$ delay replacement for the original design (M_0). If however, Y_{k-1} = 1 is satisfiable, we do not make any conclusions on the validity of M_1 as a $k-1$ delay replacement for M_0.

However, using the simple framework shown in Figure 3.5, we observed that Theorem 3 is usually insufficient to prove delay replaceability. In other words, the output of the miter in the last time-frame is almost always satisfiable to logic 1. The main reasons for this include the following:

- **No Correlation**: The simple framework of Figure 3.5 does not exploit any correlation between the behavior of M_0 and M_1. The two designs are connected only at their primary outputs; satisfaction of Y_{k-1} = 1 would result in independent assignments made in M_0 and M_1. Even if some of these assignments are not logically possible (due to the correlation), the framework of Figure 3.5 will not identify these impossible assignments.

- **Unrestricted Initial State**: In the framework of Figure 3.5, the initial state for the miter is unrestricted. This results in a large state space, from which Y_{k-1} = 1 is usually achieved. If the initial state space of the miter can be reduced, then the solution space resulting in Y_{k-1} = 1 can be avoided.

- **SAT Limitation**: The SAT solver used (zChaff) cannot identify all the sequential correlations between the signals across time-frames. As an example, assume that a sequential relation exists between two signals a and b (in the same circuit) in successive time-frames. The SAT solver views the same net in different time-frames as unique variables (signal a would be interpreted as k unique variables $a_0, a_1, ..., a_{k-1}$ corresponding to k time-frames). As a result, even if the SAT engine identifies a correlation between two variables a_{k-1} and b_{k-2}, it may not be able to generalize that relationship for a_{k-2} and b_{k-3}, or for the variables in other time-frames.

In the subsequent subsections, we illustrate how our framework is enhanced to overcome each of the limitations described above.

3.2.1 Identifying Equivalent Flip-Flops

It is well-known that establishing correspondence or equivalence between some flip-flops of the two sequential circuits in the miter can simplify the problem of sequential equivalence checking [41–43]. Also, by establishing correspondence between some flip-flops of the two designs, the problem of *'no correlation'* and *'unrestricted initial state'* discussed earlier are reduced (correlation is established via flip-flops and logically opposite values on equivalent flip-flops are illegal).

To establish equivalence between flip-flops of the two designs in a miter, an inductive technique was proposed in [30]. In the context of our framework, the following procedure (termed **identifyEquivFF**) illustrates this technique:

1. Identify a set, P, of permissible flip-flop pairs $\{f^{M_0}, f^{M_1}\}$ via random simulation. Flip-flops $\{f^{M_0} \in M_0, f^{M_1} \in M_1\}$ form a permissible pair if f^{M_0} and f^{M_1} have the same signature with random vectors.

2. *Assumption Step*: Assume each pair $\{f^{M_0}, f^{M_1}\} \in P$ to be equivalent. That is, constraint the present-state variables $(S_0 - S_{k-1})$ for each pair to the same logic value.

3. *Verification Step:* Pick flip-flop pair (f^{M_0}, f^{M_1}) from P. Let $(f^{M_0}_{N_{k-1}}, f^{M_1}_{N_{k-1}})$ represent the final next-state variables for (f^{M_0}, f^{M_1}) in the k-frame ILA of the miter. Connect these final next state variables to an XOR gate (say X_0). If $X_0 = 1$ cannot be satisfied, then pair (f^{M_0}, f^{M_1}) is truly equivalent; pick next flip-flop pair and repeat Step-3. If $X_0 = 1$ can be satisfied, then the pair (f^{M_0}, f^{M_1}) is inequivalent in the finite sized ILA; remove this pair from P and repeat iteration defined by step-2 and step-3.

4. Iterations are performed till either all pairs in P are proven to be equivalent in a single iteration or till P is empty.

Remark 2: Note that the inductive mechanism mentioned above [30] allows any flip-flop in M_0 to be permissible (and also equivalent) to more than one flip-flop in M_1 (allowing a one-to-many permissibility). Also, multiple flip-flops in M_0 are allowed to be permissible in [30] to any one flip-flop in M_1 (also allowing many-to-one permissibility). However, while verifying delay replaceability, one-to-many permissibility is not allowed: (i) if any one flip-flop in M_0 is paired with more than one flip-flop in M_1, multiple flip-flops in M_1 will be constrained to the same logic value; (ii) As a result, due to the one-to-many permissibility, some power-up states in M_1 would be impossible; in other words, $S_0^{M_1}$ would not represent the set of all power-up states in M_1 (iii) Consequently, $S_{k-1}^{M_1}$ would not represent M_1^{k-1}, and M_1 can no longer be classified as a $k-1$ delay replacement for M_0. Thus, for verifying delay replaceability, multiple flip-flops in M_0 can be paired as permissible with any one flip-flop in M_1, but not vice-versa.

In our experiments, we observed that step-1 in the inductive methodology described above [30] results in a very large set of permissible flip-flop pairs. The impact of having a large set of permissible flip-flop pairs can be understood in the following manner: Assume P initially contains m permissible pairs. If a permissible flip-flop pair is found to be inequivalent in the ILA (step-3), this pair must be removed from P and the iteration defined by steps 2 and 3 must be repeated. If n pairs from the total m pairs are found to be inequivalent, then n iterations must be performed. If m is large, the cost of each iteration can be prohibitive. Moreover, if m is large, the probability of large n also increases; this would increase the cost of identifying equivalent flip-flops, thereby increasing the overall verification time.

For several large sequential circuits, the initial size of P was found to be almost an order of magnitude larger than the number of flip-flops in each design. Moreover, a signif-

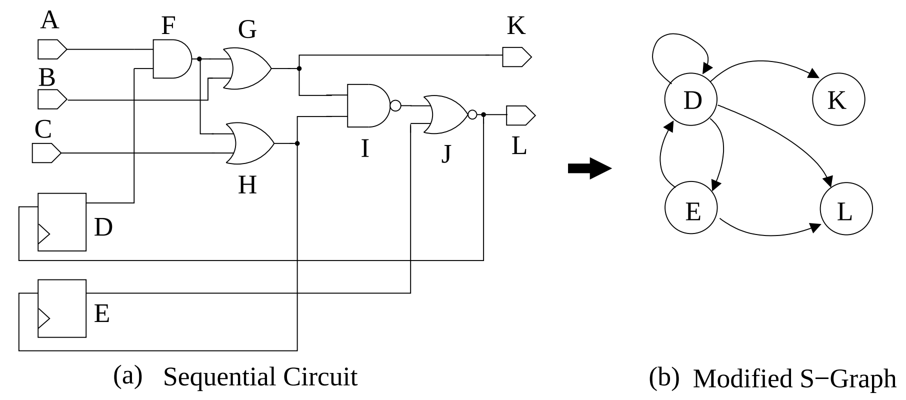


Figure 3.6: Illustration of modified s-graph

icant portion of the permissible pairs in P were found to be inequivalent. This increased the verification computation cost significantly. In order to overcome this bottleneck, we propose the following pruning. We first create a *modified s-graph* for each design in the miter. We define a *modified s-graph* as a graph $G(V, E)$ such that $v \in V$ if v is a either flip-flop or primary output in the circuit, and edge $e \in E$ exists between nodes v_i and v_j if a structural path exists between the gates corresponding to v_i and v_j in the circuit. An example for modified s-graph is shown in Figure 3.6. In the modified s-graph, nodes for primary outputs have only incoming edges, while nodes for flip-flops have incoming and outgoing edges. Also, if node v connects to another node v_i through n intermediate nodes, then, in the corresponding circuit, v can potentially affect v_i after n time-frames (intermediate nodes are flip-flops). The modified s-graph is used to prune permissible pairs using the following procedure (termed **pruneList**):

1. Pick flip-flop pair (f^{M_0}, f^{M_1}) from the set of permissible pairs (P).

2. Traverse the modified s-graph for f^{M_0} (f^{M_1}) and record the set output nodes that f^{M_0} (f^{M_1}) can reach in various time-frames as O_0 (O_1).

3. Perform a check to determine if $O_1 \subseteq O_0$. That is, check that each output node reached by f^{M_1} in each time-frame is also reached by f^{M_0} in the corresponding

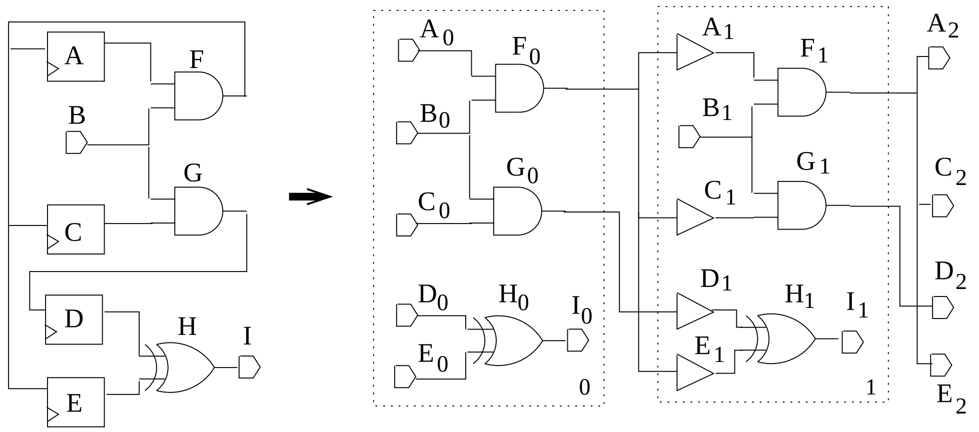
time-frame. If $O_1 \nsubseteq O_0$ remove the pair (f^{M_0}, f^{M_1}) from P (in some time-frame t, flip-flop f^{M_1} can potentially affect the logic value at some output which cannot be affected by f^{M_0} in the same time-frame).

The motivation behind pruning pairs that do not satisfy the subset operation is the following: If flip-flop f_{opt} in M_1 can affect the logic value at an output, then any equivalent flip-flop f_{equiv} in M_0 must also affect the same output. The converse, however, may not be true: f_{equiv} in M_0 may have a structural path to outputs which cannot be reached by f_{opt} in M_1 due to redundancy removal during optimization.

We observed that this pruning technique resulted in significant reduction in the number of permissible pairs for large sequential circuits, without affecting the verification result (verification result remained the same before/after pruning). This translated into a near exponential reduction in the verification time.

3.2.2 Using Logic Implications

As mentioned earlier, the SAT engine treats the same net in the sequential circuit as k different variables in a k-frame ILA. So, a sequential relation that exists between nets in the sequential circuit may not be identified by the SAT engine in the ILA. Also, sequential relationships may help to capture valuable information regarding illegal states that would otherwise be missed in the ILA. An example of such a sequential relationship is shown in Figure 3.7. In the sequential circuit (Figure 3.7(a)), two sequential conditions exist: (i) $F = 1$ implies $G = 1$ (in the same time-frame), and (ii) sequentially, $H = 1$ cannot be achieved. Now consider the 2-frame ILA for the sequential circuit (Figure 3.7(b)). It can be observed that the first relationship (on nets F and G) can only be captured in time-frame 1 (this relationship is completely missed in time-frame 0). Also, the second condition (sequentially, $H = 1$ is impossible) cannot be captured in either time-frame 0 or time-frame 1. The sequential conditions also efficiently capture



(a) Sequential Circuit (b) Corresponding 2−frame ILA

Figure 3.7: Illustration of sequential implications

information about illegal states present in the sequential circuit, namely: $(A, C, D, E) = \{(-, -, 0, 1), (-, -, 1, 0), (1, 0, -, -), (0, 1, -, -)\}$, where '$-$' represents 'don't care'. In our running example, this information is completely lost in time-frame 0, and only partially identified in time-frame 1 of the ILA.

In our framework, we use sequential logic implications to identify such sequential constraints. We use the graph-based implication engine proposed in [21] since this representation compactly stores sequential relationships in a design. We identify single node implications of the form $a = v \rightarrow b = w$ ($a = v$ implies $b = w$; $(v, w) \in 0, 1$) and convert them into CNF (conjunctive normal form) clauses. These clauses are then added into the propositional formula of the miter ILA. For example, for Figure 3.7(a), the relationship $F = 1 \rightarrow G = 1$ is converted into these clauses for the ILA: $(\neg F_0 \;\; G_0)$, $(\neg F_1 \;\; G_1)$. Also, conditions about impossible value assignments ($H = 1$ in Figure 3.7(a)) are represented as single variable clauses. So, the impossible assignment in Figure 3.7 is converted into two clauses: $(\neg H_0)$, $(\neg H_1)$. By introducing these additional clauses into the propositional formula, 'SAT limitation' discussed earlier is addressed. Moreover, since these clauses prevent some illegal states from being explored, the problem of 'unrestricted initial state' is indirectly handled via sequential implications.

Now, we show that adding the sequential relationships into the original design M_0 of the ILA of the miter would not result in erroneous delay replacements under Theorem 3 (note that sequential relations are not added into M_1).

Lemma 12 Given*: Set of sequential relationships for design M_0 as R_{M_0}.*
Statement*:* **If** *(i) R_{M_0} is added for each time frame of the ILA;* **AND** *(ii) Theorem 3 is true, i.e., $Y_{k-1} = 1$ (Figure 3.5) is unsatisfiable;* **then** *the transformed circuit (M_1) is a valid $k-1$ delay replacement for the original circuit (M_0).*

Proof:

- Let the illegal states captured by logic implications for M_0 be IS_{M_0}. Since logic implications are added to each time-frame in the ILA for M_0, IS_{M_0} cannot be reached in any time-frame for M_0.
- Let $S_i^{M_j}$ be the set of present state variables reachable for design M_j at time-frame i without any sequential constraints.

Step-1:
Since Y_{k-1} = 1 is not satisfiable, for all states in $((S_{k-1}^{M_0} - IS_{M_0}), (S_{k-1}^{M_1}))$, Y_{k-1} = 0 must be true. Now, (i) $S_j \supseteq S_{j+1}; j \geq 0$ and (ii) IS_{M_0} cannot be reached in any time frame for M_0. Conditions (i) and (ii) indicate that even if the size of the ILA is increased beyond k to $k + m$ $(m > 0)$, the reachable present state variables for any time-frame $t \geq k$ would be a subset of $((S_{k-1}^{M_0} - IS_{M_0}), (S_{k-1}^{M_1}))$. As a result, if Y_{k-1} = 1 is not satisfiable, Y_j = 1 cannot be satisfied for $j \geq k-1$.
Step-2:
$S_{k-1}^{M_1}$ represents the set of states reached from all power-up states in M_1 after $k-1$ time-frames. Thus, by Definition 25, $S_{k-1}^{M_1}$ and the state and output transition relationship defined by time-frame $k-1$ represent M_1^{k-1}. In addition, $(S_{k-1}^{M_0} - IS_{M_0}) \subseteq S_0^{M_0}$. Thus, for every

state $s_1 \in S_{k-1}^{M_1}$, there exists at least one power up state $s_0 \in S_0^{M_0}$ such that for all test sequences, $Y_j = 0;\ \forall j \geq k-1$. Therefore, by Definition 29, M_1 is a valid $k-1$ delay replacement for M_0. $\square$

Remark 3:

Note that sequential relationships are injected **only** for M_0. If sequential relationships for M_1 are also injected, then: (i) Constraints imposed by sequential relationships cause $S_0^{M_1}$ to represent only a subset of all possible power-up states for M_1; (ii) As a result, $S_{k-1}^{M_1}$ represents a subset of states reached from all power-up states in M_1; hence, $S_{k-1}^{M_1}$ would not represent M_1^{k-1}, and M_1 would not classify as $(k-1)$ delay replacement for M_0.

Finally, the overall verification algorithm can be understood through the following steps:

1. Identify set (P) of permissible flip-flop pairs between M_0 and M_1.
2. Prune P using modified s-graphs for M_0 and M_1 (pruneList: Section 3.2.1).
3. Identify set of sequential relations for M_0 as R_{M_0}.
4. Create the miter for M_0 and M_1. Expand the miter into k-frame ILA; convert the ILA into a propositional formula (ϕ).
5. Convert R_{M_0} into clauses. Add clauses for M_0 (all time-frames) into ϕ.
6. Check equivalence of flip-flops in P (identifyEquivFF: Section 3.2.1).
7. If $Y_{k-1} = 1$ (miter-output in last time frame) is not satisfiable, declare M_1 as valid $(k-1)$-delay replacement for M_0 (Lemma 12). Else, draw no conclusion.

3.3 Experimental Results

The algorithm described in the previous section was implemented in C++. Experiments were performed to verify delay replaceability of ISCAS '89 benchmark circuits with their optimized versions generated using SIS. All experiments were conducted on Pentium-4 3.2 GHz processor with 512MB RAM, running Linux.

Table 3.1 reports the results using our SAT-based formulation. Column 1 shows the instance for which equivalence was verified between the original circuit ckt with its *sequentially* optimized version, ckt_{opt}. The miter composed of ckt and ckt_{opt} was unrolled into a 3-time-frame ILA. By our Theorem 3, using this ILA, we would verify if ckt_{opt} is a valid 2-delay replacement for ckt. Columns 2 and 3 report the number of gates and flip-flops in the original and the optimized circuits, respectively. Column 4 (T-1) reports the result of verification for **only** Theorem 3 with **neither** sequential implications **nor** identification of equivalent FF (flip-flops). For Columns 4 - 7, "Y" indicates verification success, and "NC" indicates no conclusion. As described earlier, it can be observed that Theorem 3 in itself is insufficient for proving delay replaceability of sequential circuits.

Table 3.1: Results: Proving delay replaceability

Circuit	**#gates orig/opt**	**#FF orig/opt**	**T-1**	**Only Impl**	**Only EqFF**	**ALL**	**Time Impl**	**#Eq FF**	**Time Eq FF**	**Time Total**
s298	143/122	14/14	NC	NC	Y	Y	0.18	14	0.95	1.05
s344	196/157	15/15	NC	NC	Y	Y	0.20	15	1.2	1.48
s349	197/157	15/15	NC	NC	Y	Y	0.23	15	1.07	1.45
s382	189/169	21/21	NC	NC	Y	Y	0.21	21	2.14	2.54
s386	180/126	6/6	NC	NC	Y	Y	0.48	6	0.65	1.24
s400	195/170	21/21	NC	NC	Y	Y	0.22	21	2.2	2.66
s444	212/168	21/21	NC	NC	Y	Y	0.24	21	2.4	2.83
s510	244/227	6/6	NC	NC	Y	Y	0.4	6	1.2	1.73
s526	224/180	21/21	NC	NC	Y	Y	0.3	21	2.6	3.05
s713	471/223	19/17	NC	NC	Y	Y	0.43	17	2.1	2.71
s820	332/282	5/5	NC	NC	Y	Y	1.4	5	1.35	2.93
s832	330/275	5/5	NC	NC	Y	Y	0.8	5	1.94	2.85
s1238	555/502	18/18	NC	NC	Y	Y	2.7	18	2.9	5.85
s1423	754/686	74/74	NC	NC	Y	Y	0.58	74	11.6	13.7
s5378	3043/1430	179/163	NC	NC	NC	NC	13.4	111	522.6	540.2
s9234	5867/1085	228/138	NC	NC	NC	Y	45.3	83	52.4	103.2
s13207	8773/2521	669/453	NC	NC	NC	Y	77.8	185	362.6	465.6
s15850	10471/3847	597/540	NC	NC	NC	Y	205.2	340	532.2	782.8
s35932	18149/10795	1728/1728	NC	NC	Y	Y	283.4	1728	1285.5	1611.7

Time measured in seconds **NC: no conclusion**

Column 5 (Only Impl) reports the results of verification when Theorem 3 was used in conjunction with sequential implications, but **without** identification of equivalent FF. Column 6 (Only Eq. FF) reports verification results when Theorem 3 was used in conjunction with identification of equivalent FF but **without** sequential implications. While sequential implications and identification of equivalent FF individually enhance the setup, each addresses separate limitations of the simple SAT framework. For instance, sequential implications specifically address “SAT Limitation”, while identification of equivalent FF addresses the problem of “No Correlation”. Both techniques partly address the problem of “Unrestricted Initial State”. Thus, none of these techniques are individually sufficient in assisting Theorem 3 to prove equivalence for all sequential circuits. For smaller circuits (e.g. s1238, s1423, etc.), where the number of FF were equal between the original and optimized designs, identification of equivalent FF in conjunction with Theorem 3 was sufficient to prove equivalence (Column 6). Since all FF for such circuits were paired as equivalent, the problem of sequential equivalence checking reduced to combinational equivalence checking. However, for designs where the number of FF in the optimized and original design are significantly different and do not bear equivalence (e.g. s9234, s13207, etc.), identification of equivalent FF pairs together with Theorem 3 were insufficient.

Next, Column 7 (ALL) reports the verification result when Theorem 3 was applied in conjunction with both of our proposed enhancements (equivalent FF and sequential implications). Results in Column 7 indicate that sequential implications can aid in proving equivalence for sequential circuits where only identification of equivalent FF was not sufficient (e.g. s9234, s15850). Column 8 reports the time taken to identify sequential implications; Columns 9 and 10 show the number of equivalent FF pairs identified and the corresponding time spent and Column 11 reports the total time taken to prove equivalence. Note that we could verify circuits (such as s9234) for which only a fraction of FF pairs in the two designs were found to be equivalent. This indicates that our framework can be successfully applied for proving equivalence even for designs where the state-encoding is significantly changed during optimization. For all but one instances, we could successfully verify the optimized design as a valid 2-delay replacement for the original design without significant computational effort. Moreover, the time complexity of our framework grows only linearly with the circuit size. The only sequential circuit for which no conclusion could be drawn in the 3-frame ILA was s5378. However, for s5378, even when the ILA size was increased to as much as 50 time-frames, the "NC" result did not change. One potential reason for this could be the high sequential depth of s5378. Even when the ILA size is increased, not all illegal/ephemeral states are eliminated due to high sequential depth. Such states may result in the miter output getting satisfied to logic 1.

Finally, our framework is the first attempt on verifying delay replaceability of sequential circuits. Earlier approaches target easier notions of sequential equivalence; for instance, AQUILA [30] targets 3-valued safe replaceability, while BDD based techniques such as [33] target sequential equivalence after initialization (SHE). These definitions of sequential equivalence are automatically covered by delay replaceability.

3.4 Summary

To summarize, we have presented a new and effective framework to target a hard, uninvestigated sequential equivalence problem: verifying delay replaceability of sequential designs. We presented a new theorem to inductively prove equivalence of two sequential circuits in a finite ILA, via learning of sequential implications and pruning of permissible flip-flop pairs in a SAT-based framework. The limitations of the framework were discussed and techniques to overcome these limitations were incorporated into the framework. Specifically, we efficiently identified equivalent flip-flops between the two sequential circuits and used sequential implications to aid the process of verifying delay replaceability. Experimental results for benchmark circuits show the effectiveness and scalability of our methodology to prove sequential equivalence under the notion of delay replaceability.

Chapter 4

Untestable Multi-Cycle Path Delay Faults in Latch-Based Designs with Multiple Clocks

With shrinking feature size and higher operating frequencies, there has been a growing concern to target defects that increase the signal delays in a circuit. Such delay defects have been studied using the following three popular fault models: the transition fault model [53], the gate delay fault model [54] and the path delay fault model [55]. A transition fault models accumulated defects that cause the transition at a gate output to be delayed independent of the path through which it propagates in the circuit. A gate delay fault models defects that increase the input-to-output delay of single logic gate while other gates retain some nominal values of delay [56]. A path delay fault (pdf) models distributed defects that cause the signal along a particular path to get delayed. Transition and gate delay fault models assume isolated gross failures at distinct gates, while the path delay fault model handles distributed failures along a particular path. Since the path delay fault model is more generic in modeling actual defects, it has been the focus of significant research.

However, one major problem with path delay fault testing is the existence of an extremely large number (several millions) of paths even in moderately sized circuits. This makes test-generation for path-delay faults (pdfs) a difficult problem. It has been indicated in the literature that a very large portion of the paths that exist in a circuit are usually untestable. In order to reduce test generation effort for pdfs and to significantly reduce the effort in the diagnosis of speed failures during the silicon debug process, identification of untestable pdfs is needed. Thus, significant effort has been invested in the last decade towards efficient identification of untestable pdfs [45, 47, 57–60].

Though techniques proposed in the literature have achieved success in the identification of untestable pdfs, these techniques focus only on delay faults that can be captured within a single clock cycle. Moreover, these techniques have dealt with flip-flop based designs that are clocked with a single synchronous clock. However, the demand for higher operating frequencies and pipelined architectures has resulted in a growth of latch-based designs which are clocked with multiple clocks. These design methodologies coupled with time-sharing across latches have resulted in signals which propagate for multiple cycles through paths with multiple latches. Testing of such multi-cycle paths for delay defects, though critical (to ensure reliability), requires significant computational resources. The process of fault grading using functional tests for the large set of multi-cycle paths generated by a static-timing analysis (STA) tool is computationally intensive. Moreover, this process suffers from severe performance loss due to the time wasted in simulating paths that are inherently untestable. The performance of an ATPG approach to generate patterns for such paths is also significantly degraded by untestable paths. To reduce the delay test selection/generation effort for multi-cycle paths and also to reduce silicon debug effort for such paths, a framework that efficiently identifies untestable multi-cycle paths is needed.

Two main contributions which we present in this dissertation are:

- First, we formally present the conditions to non-robustly test a multi-cycle path. Conditions to test a multi-cycle path have been informally introduced [61] with assumptions on signal propagation in different cycles along a multi-cycle path. In the dissertation, we present non-robust conditions for a multi-cycle path in a comprehensive manner without these assumptions.

- We present a novel and unique methodology for identifying non-robustly untestable multi-cycle pdfs (M-pdfs) for latch-based designs with multiple clocks. To the best of our knowledge, identification of untestable M-pdfs has not been addressed in the literature. Techniques previously published target the identification of only single-cycle pdfs [45, 47, 57–60] (combinationally or sequentially untestable pdfs). Also, identification of untestable path-delay faults in latch based designs with multiple clocks has also not been investigated. In these contexts, our framework is the first-of-its-kind.

Key features of our framework are:

- Fast, incremental analysis: We present a three-step methodology that incrementally identifies and prunes untestable M-pdfs. We statically partition each path into segments at latch boundaries and use a novel, graphical representation of the M-pdf to analyze the delay fault.

- Low computational complexity and scalability.

- Per-path analysis: Unlike earlier non-enumerative techniques, we independently analyze each multi-cycle path provided in a fault list. Usually, STA tools generate a finite path-list (corresponding to some percentage of worst-delay-paths in the circuit). Although performing per-path analysis on all path faults in a circuit would be prohibitive, performing per-path analysis on this subset is feasible.

Experimental results show that our tool can be used to efficiently identify non-robustly untestable M-pdfs. In this chapter, first we discuss relevant definitions and characteristics of the problem; next, we describe the three-step methodology used in our framework to identify untestable M-pdfs, which is followed by experimental results for industrial designs.

4.1 Preliminaries

This section presents relevant definitions, issues about latch based designs with multiple clocks, the static segmentation of a multi-cycle path and a discussion on logic implications.

4.1.1 Relevant Definitions

Definition 31 *Multi-cycle Path Delay Fault (M-pdf):*
An M-pdf models distributed delay defects that cause a transition to be delayed along a path over multiple clock cycles. In our framework, M-pdf has two attributes associated with it:

i. A physical path, i.e., an interconnection of gates and lines ($g_0, l_0, g_1, l_1, ..., l_{n-1}, g_n$), where g_0 is the gate at which the path originates, $g_1 - g_{n-1}$ are intermediate gates along the path, $l_0 - l_{n-1}$ are lines through which the transition propagates and g_n is the termination point of the path; g_0 can either be a latch-output or primary input (PI), and g_n can be a latch-output or primary output (PO). Any intermediate gate, g_i, along the M-pdf can be a combinational (AND, OR, etc.) or sequential element (latch).

ii. A transition type (slow-to-rise or slow-to-fall).

Note that the definition of an M-pdf differs from conventional single-cycle pdf not in the structure of the pdf, but in the behavior of the pdf. While both single and multi-cycle pdfs may have similar physical structure, unlike single-cycle pdf, for an M-pdf, the transition gets delayed only over multiple clock cycles (i.e. an M-pdf cannot have a single-cycle test).

Definition 32 *On-path and Off-path lines:*

Lines $l_0 - l_{n-1}$ in a path ($g_0, l_0, g_1, l_1, ..., l_{n-1}, g_n$) are called the on-path inputs of the M-pdf; other lines in the circuit that feed $g_1 g_n$ are off-path inputs of the M-pdf.

Definition 33 *Controlling and Non-Controlling Values:*

If logic value v at any one input of gate g can determine the logic value at the output of the gate, then v is called the controlling value for g; else, v is the non-controlling value for g.

Definition 34 *Propagating Values:*

Some non-primitive gates (such as multiplexers) do not have controlling or non-controlling logic values. For such gates, we define logic value v as the propagating value if logic v at one of the inputs enables propagation of the value at the on-path input to the gate's output. For primitive gates (AND, OR, NOR, NAND), non-controlling values serve as propagating values.

Definition 35 *Non-robust Test for an M-pdf:*

A test sequence $< T_1, T_2, T_3 >$ (T_1, T_2 and T_3 are sets of vectors) is a non-robust test for an M-pdf ($g_0, l_0, g_1, l_1, ..., l_{n-1}, g_n$) if:

i. Test sequence T_1 can initialize the output of gate g_0 to the initial value associated with the transition (logic 0 for slow-to-rise, and logic 1 for slow-to-fall) associated with the path; and

ii. Test sequence T_2 launches the transition at the output of g_0 (sets it to logic 1 for slow-to-rise or logic 0 for slow-to-fall fault) and propagates it non-robustly through the entire path to the output of g_n. Non-robust propagation entails setting all off-path inputs to their propagating values under T_2; and

iii. If g_n is a latch, T_3 propagates the fault-effect latched at the output of g_n to a primary output. If g_n is a PO, T_3 is empty.

Remark 4:

a. In the conventional definition of non-robust tests ([45,47,62]), that applies to single-cycle path delay faults, two vectors are necessary to non-robustly detect a path delay fault. For an M-pdf, a non-robust test consists of at-least two vectors.

b. If an M-pdf terminates at a PO, a test sequence that satisfies conditions (i) and (ii) from Definition 35 is a non-robust test for the M-pdf (the fault effect reached PO).

c. Individually, each condition in Definition 35 is a necessary condition to be satisfied by a non-robust test for an M-pdf.

Definition 36 *Non-robustly untestable M-pdf:*
An M-pdf is non-robustly untestable if no test sequence exists that can satisfy all conditions in Definition 35.

Remark 5:

a. In this dissertation we provide a framework that identifies non-robustly untestable M-pdfs by checking violation of conditions (i) and (ii) in Definition 35. If no test sequence exists that satisfies conditions (i) and/or (ii) from Definition 35, the M-pdf is classified as untestable. The framework can be extended to additionally determine violation of condition (iii) to potentially identify more untestable M-pdfs.

b. The main focus of this dissertation is to provide a framework for identifying non-robustly untestable M-pdfs. Conditions necessary for robust tests for M-pdfs can be derived from those defined for non-robust tests for M-pdfs.

4.1.2 Multiple Clock Domains and our Computation Model

In this section, we present issues related to designs with multiple clocks are presented along with the computation model that is used for identifying untestable M-pdfs. A similar com-

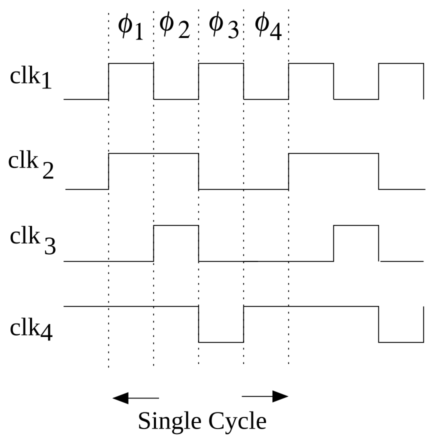
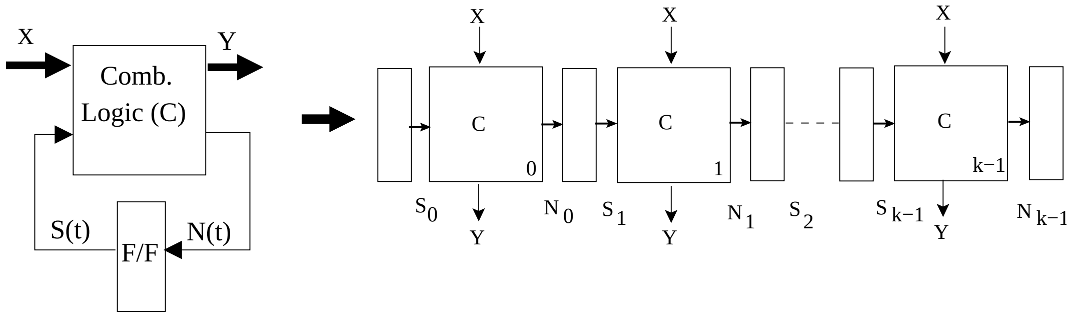


Figure 4.1: Clock waveforms in a multi-clock design

putation model was used in one of our previous work [63] to identify untestable transition faults. First, consider a design with four different clocks as shown in Figure 4.1. These four clocks divide the design into four unique phases, $\phi_1 \;\; \phi_4$. A single-cycle in this waveform represents the time period after which the clocking behavior of the circuit repeats. While test-generation looks for one phase in which it can detect a given fault, untestable fault identification must exhaust all clock-phases to conclude that a given fault is untestable. This increases the complexity of identifying untestable faults in designs with multiple clocks.

The model of computation used in our framework represents the sequential circuit as an iterative logic array (ILA). An ILA represents a sequential circuit as an interconnection of several copies of the combinational logic. Such an ILA was presented earlier in Chapter 3 and is shown again in Figure 4.2.

The ILA shown in Figure 4.2 corresponds to a design that contains a single clock and also incorporates flip-flops as sequential elements. For such designs, the circuit behavior does not change within the boundary of a single-clock cycle. Thus, each combinational copy in the ILA represents one cycle of the system clock or one time-frame. However, for latch based designs with multiple clocks, the behavior of the circuit may change in different phases of the same cycle. Specifically, if each clock cycle has m-unique clock phases, then



(a) Sequential Circuit Model (b) k−Frame ILA

Figure 4.2: ILA for flip-flop based designs with single clock

the same physical circuit can have m-different functional behaviors in one cycle.

To model this characteristic in the ILA, for latch based designs with multiple clocks, each combinational copy of the ILA must correspond to one of the m-unique clock-phases, and m-combinational blocks of the design must represent one cycle. The ILA for a latch based design with multiple clocks, having m-unique clock phases in a single cycle is shown in Figure 4.3. Such an ILA forms the computation model for our framework.

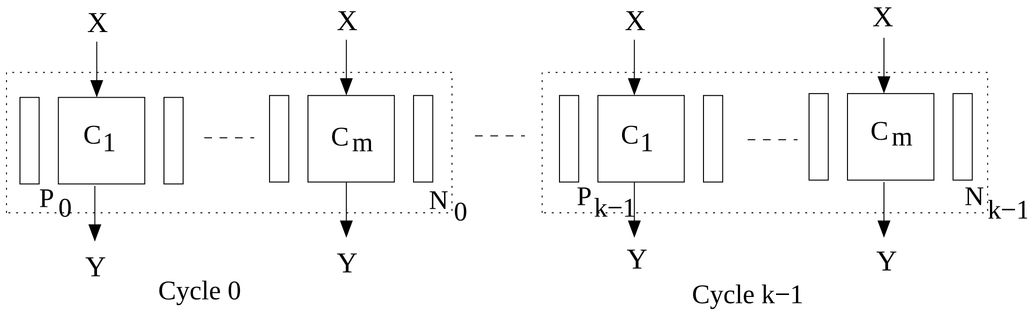


Figure 4.3: ILA for latch based designs with multiple clocks

4.1.3 Static Path Segmentation

As indicated earlier, a physical path (M-pdf) in our analysis can originate at a primary input or a latch output and terminate either at a latch output or primary output. This path can also include several intermediate latches. These intermediate latches can cause a transition to propagate across different clock phases; as the physical path crosses a latch boundary, the transition associated with the M-pdf can cross phase-boundaries. Thus, analyzing an M-pdf entails simultaneous analysis of different portions of the M-pdf that belong to different phases of various clocks. This complicates analyzing untestability of an M-pdf for designs with multiple clock domains. For ease of analysis, we statically split the path into segments, dividing the path at latch boundaries. This enables handling M-pdfs through a divide-and-conquer approach, as would be explained later. To understand static segmentation, consider the M-pdf shown in Figure 4.4. The path shown in Figure 4.4 originates at a primary input, passes through four latches, $L_1 - L_4$ and contains four combinational portions (cp) of the logic. We segment the path at latch inputs, so, the first segment (S_1) contains only the combinational portion cp_1 that extends from the primary input at which the M-pdf originates till the input of latch L_1. The second segment (S_2) extends from the latch L_1 to the input of latch L_2 and so on. The final segment (S_5) contains only latch L_4.

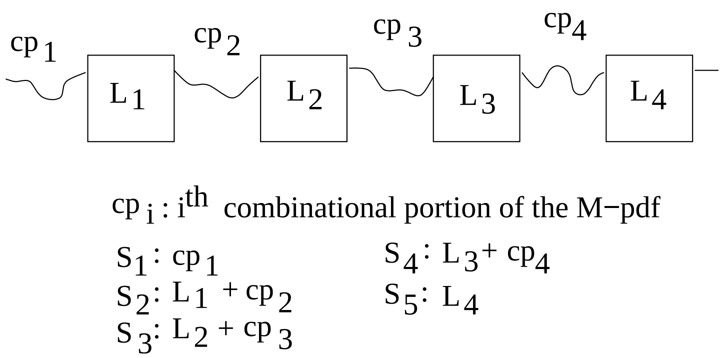


Figure 4.4: Illustration of path segmentation

Before we proceed, the following definition for satisfaction of conditions along a segment of an M-pdf is provided:

Definition 37 *Segment Sensitization:*
A segment is sensitized in a given clock phase ϕ_i if all off-path inputs to the segment are set to their propagating values in ϕ_i.

Note that a non-robust test for an M-pdf can sensitize two or more successive static segments in a single clock phase, thus forming a dynamic segment. For example, for the M-pdf in Figure 4.4, a test may sensitize static segments S_1 and S_2 simultaneously in a phase i. Then S_1 and S_2 form a dynamic segment. Though we segment the path statically, our 3-step methodology (section three) implicitly handles all possible dynamic segments that can exist due to the clocking relationship between various static segments.

4.1.4 Logic Implications

As indicated earlier, logic implications have successfully been used in the past in various facets of electronic design automation. Most of the earlier work on untestable pdfs [45, 47, 57, 58] has relied on logic implications and experimental results indicate that a large subset of single-cycle path delay faults can quickly be identified as untestable using implications. This motivates the use of logic implications to also identify untestable M-pdfs in our framework. Since we compute implications using the computation model described in the previous section, we identify sequential relationships which are missed in conventional approaches that usually assume a full-scan circuit and rely only on combinational logic relationships to identify untestable path delay faults. An example that illustrates the use of implications to identify untestable path-delay faults is given below. For simplicity, we consider a combinational design.

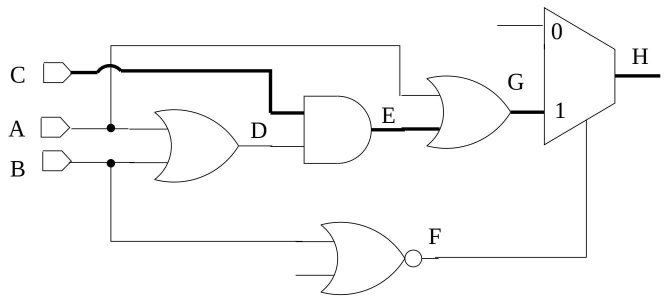


Figure 4.5: Illustration of implications for untestable pdfs

Example: 5 Consider a slow-to-rise transition on the path $\{C - E - G - H\}$, shown in bold in Figure 4.5. The conditions necessary to non-robustly propagate the transition to the output of gate H (multiplexer) are: $\{(D = 1), (B = 0), (F = 1)\}$. By implying these necessary assignments, a conflict would be identified ($\{D = 1\}$ and $\{B = 0\}$ imply $\{A = 1\}$. $\{A = 1\}$ implies $\{F = 0\}$ which conflicts with the set of necessary conditions for non-robust propagation). Thus, using implications, the path-fault would be declared as untestable. □

4.2 Three-Step Methodology to Identify Untestable M-pdfs

The static segmentation of M-pdfs introduced in the earlier section enables a divide-and-conquer approach to identify non-robustly untestable M-pdfs. In this section, we describe a three-step methodology for that purpose. The first step is called *independent sensitization*, where each segment is analyzed in each clock phase, independent of other segments. The second step is called *flow-analysis*, where the clocking scheme in the circuit is analyzed to determine whether the transition can flow from one segment to another through latches to reach the termination point in the M-pdf. Finally, the third step is called *simultaneous sensitization*. Flow-analysis (step-2) provides a clocking arrangement (if it exists) which allows the transition to flow through the path. For each such clocking arrangement identified by step-2, step-3 determines if all segments can be sensitized simultaneously in that clocking arrangement. Before describing each step, the following terms need to be defined:

Definition 38 *Successful Launch:*
A slow-to-rise (slow-to-fall) transition is said to be a successful launch in a clock phase ϕ_i if:

a. *The gate at which the M-pdf originates can be set to logic 0 (logic 1) in ϕ_{i-1} and to logic 1 (logic 0) in ϕ_i*

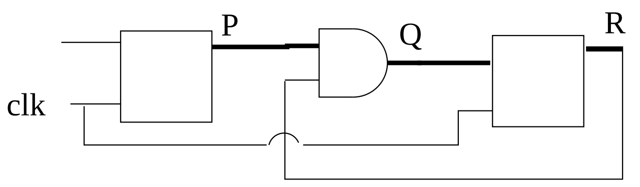
b. The first segment in the path can be sensitized in ϕ_i.

Definition 39 *Latch sensitivity, transparency and hold-state:*
Latch L_1 is sensitive to logic v ($v \in \{0,1\}$) if the input to L_1 gets transferred to its output when clock to L_1 is set to v. L_1 is said to be transparent while the clock to L_1 is set to v. When L_1 is not transparent, L_1 is in the hold-state: it holds the value it last acquired when its clock was set to logic v.

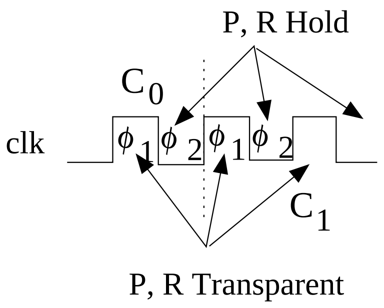
Step-1: Independent Sensitization

As the name suggests, in this step, each segment is treated independently of other segments. The sensitization conditions for each segment S_i are identified and implied in each unique clock phase to determine the clock phases in which S_i can be sensitized. If there exists a segment in the M-pdf which cannot be sensitized in any given clock phase, the M-pdf is declared non-robustly untestable. An example of such an unsensitizable segment is $\{C - E - G - H\}$ in Figure 4.5. Unlike other segments in the M-pdf, for the first segment, conditions necessary for a successful launch are examined. If the conditions for a successful launch cannot be satisfied for the first segment in any clock phase, the M-pdf would be declared as non-robustly untestable. The main purpose of step-1 is to prune those untestable M-pdfs for which either a successful launch is impossible or which have at-least one unsensitizable segment. Consider the following example for better understanding.

Example 6: Consider a slow-to-rise transition on the path $\{P - Q - R\}$, shown in bold in Figure 4.6 (a). For simplicity, a design with single clock is considered. By the segmentation procedure described earlier, the path would be divided into two segments: segment 1 (latch P, gate Q and data input of latch R) and segment 2 (output of latch R). Figure 4.6(b) shows the clocking waveform. Assuming that the two latches are sensitive to level-1 1 of the clock, both latches would be transparent during phase ϕ_1 and during phase ϕ_2, both latches would hold their previous values. The transition on segment 1 can be launched either in phase ϕ_1 or ϕ_2 (of any clock cycle). Assume launch in cycle C_1. Consider ϕ_2 first.



(a) Circuit to illustrate independent sensitization



(b) Clock and latch states

Figure 4.6: Illustration for Step-1

a. Launch in ϕ_2, C_1: For the transition to be launched in ϕ_2, the following conditions are required: $\{P = 0\}$ in ϕ_1 and $\{P = 1\}$ in ϕ_2. Latch P is in hold-state in phase ϕ_2: it must hold the value acquired in ϕ_1. Since $\{P = 0\}$ is required in ϕ_1, $\{P = 0\}$ will be implied in ϕ_2 (latch P holds value from ϕ_1). This conflict makes launching the transition in ϕ_2 impossible.

b. Launch in ϕ_1, C_1: For the transition to be launched in ϕ_1, the following conditions are required: $\{P = 1\}$ in ϕ_1 (in cycle C_1) and $\{P = 0\}$ in ϕ_2 (in previous cycle, C_0). $\{P = 0\}$ in ϕ_2 (C_0) implies $\{Q = 0\}$ in ϕ_2 (C_0). Since clk = 0 disables latch R in ϕ_2 (C_0), logic 0 at the input of latch R does not imply any value at the output of the latch. In ϕ_1 (C_1), clk goes high, implying $R = 0$. In ϕ_1 (C_1), the transition is launched by setting $\{P = 1\}$. However, $R = 0$ (implied when clk goes high) blocks the transition at the input of gate Q after it is launched in ϕ_1.

Case (a) and (b) indicate that the conditions for a successful launch of the transition cannot be met in any clock phase. Thus, the path-fault would be declared as untestable. □

During the process of independent-sensitization, a record is kept of the clock phases in which each segment can get sensitized (shown as sensitization information in Figure 4.7).

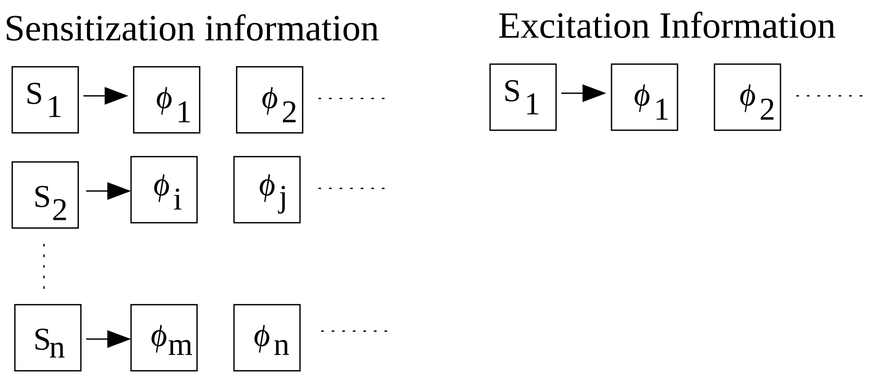


Figure 4.7: Data structures at the end of step-1

Also, the clock phases in which a transition can be successfully launched (according to Definition 38) are recorded (shown as excitation information in Figure 4.7). This information is used like a look-up table for the second step, i.e. flow analysis. The two data structures available at the end of step-1 are shown in Figure 4.7. If the sensitization list for any segment is empty or if the excitation list for the first segment is empty, the delay fault is concluded as non-robustly untestable, and is not considered for further analysis. It should be noted that the first segment appears under both sensitization information and excitation information in Figure 4.7. Maintaining sensitization information for the first segment may seem redundant (because the excitation condition contains information about transition launch and simultaneous sensitization of the first segment); the need for this information would become clear in step-2.

Step-2: Flow Analysis Before discussing the methodology employed during this step, we first define the following terms.

Definition 40 *Upstream and Downstream Latch:*

The latch at the beginning (end) of a segment is called the upstream (downstream) latch for that segment.

This step of the analysis focuses on determining if, once successfully launched, a transition can be propagated from one segment to another through intermediate latches or not. In this step, we model the M-pdf as a weighted tree called the flow-tree. In this flow-tree, each node represents one segment in the path, an edge between two nodes represents a flow of transition between the two segments and a weight associated with each edge represents the clock cycle and the clock-phase in which the transition flows from one segment to another. This tree is built incrementally; initially there is only one node in the tree corresponding to the first segment; when it is determined that the transition can flow from the first segment to the second segment in a particular cycle and phase $< C_i, \phi_i >$, a second node is added to the tree representing the second segment. A directed edge (from the first segment to the second segment) connects the two nodes, and weight $< C_i, \phi_i >$ is associated with the edge representing the cycle and phase in which the transition moved from segment 1 to segment 2. This continues till a complete path in this tree is identified that connects the root of the tree (the first segment) with the leaf node (the last segment). When such a path is identified, a clocking relationship between various segments in the path is identified which potentially detects the delay fault along the path. If no such path can be identified that connects the root to the leaf, the M-pdf is declared as untestable: although each segment can be sensitized individually, and a transition can be launched (from step-1), but the transition cannot propagate from the first segment to the last. Figure 4.8 shows an example of a flow-tree.

Assume that the flow-tree shown in Figure 4.8 corresponds to an M-pdf with 4 segments (S_1S_4) in a design with a two-phase clock (two phases are ϕ_1 and ϕ_2). The first node is created for segment S_1 corresponding to the clock cycle and phase in which a transition can be launched. In Figure 4.8, this corresponds to clock cycle C_1, phase ϕ_1. Then, some flow-conditions are examined to determine if the transition that was launched in ϕ_1 can get propagated from S_1 to S_2. The flow conditions that are examined at this stage would be explained shortly. Assume that the flow-conditions evaluate such that the transition flows through to S_2 in phase ϕ_1 (*flow conditions resulted in success*). So, a new node for segment S_2 is added to the flow-tree and an edge with weight $< C_1, \phi_1 >$ is established between

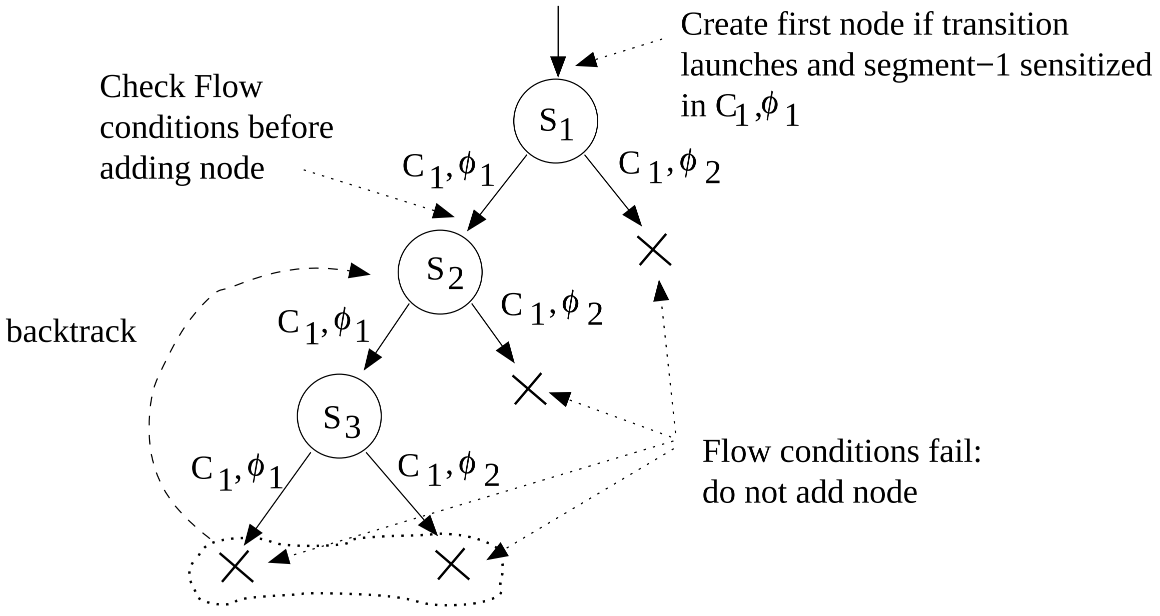
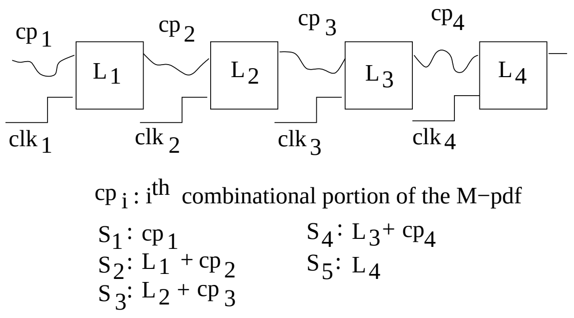
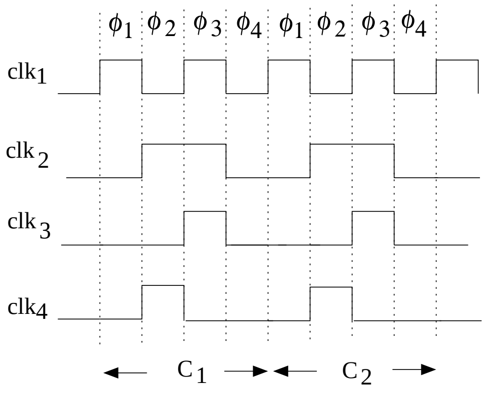

Create first node if transition launches and segment−1 sensitized in C1,φ1
Check Flow conditions before adding node
S1
C1,φ1
C1,φ2
S2
backtrack
C1,φ1
C1,φ2
S3
C1,φ1
C1,φ2
Flow conditions fail: do not add node

Figure 4.8: An example flow-tree

S_1 and S_2. Next, flow-conditions are again evaluated, on this occasion, to determine if the transition present on segment S_2 in phase ϕ_1 can be propagated to S_3 in ϕ_1. Assuming that the flow conditions result in a success, a node corresponding to S_3 is added to the tree, and an edge is established between S_2 and S_3. Similar flow-conditions are examined for S_3. Assume that the flow conditions at S_3 evaluate such that the transition present along segment S_3 cannot propagate to S_4 in either phase ϕ_1 or ϕ_2 (flow conditions result in failure for all unique clock phases). In such a situation, the tree is not built beyond S_3. When the flow-conditions result in failure for all possible phases, a 'backtrack' is performed. The 'backtrack' results in the re-examination of flow-conditions at S_2 (the node preceding S_3) for the next unexplored phase, i.e. ϕ_2. Assuming that the flow conditions at S_2 for phase ϕ_2 also result in a failure, backtrack would be performed to reach S_1. This process goes on till either a path in the flow-tree from the root to the leaf node is established or till all possible paths in the tree result in a failure in flow conditions. The need for maintaining sensitization information for the first segment in the path in addition to the excitation information should be apparent now: if only excitation information is maintained for S_1 in Figure 4.8, then once



(a) Five segment M−pdf



(b) Clocking scheme in the design

Figure 4.9: Illustration for flow-conditions

the root node is created, only the left subtree can be explored and built. If no information about the sensitization of S_1 is maintained then the edge corresponding to phase ϕ_2 between S_1 and S_2 cannot be created.

The following example explains flow-conditions:

Example 7: Consider the 5-segment M-pdf shown in Figure 4.9(a). Assume:

- The design has four different clocks, creating four unique clock phases ($\phi_1 \;\; \phi_4$) as shown in Figure 4.9(b).
- Independent sensitization (step-1) indicates that successful launch of the transition (slow-to-rise/slow-to-fall) can occur only in phase ϕ_1 and all segments can be individually sensitized in all clock phases.
- All latches in the design are sensitive to logic 1 of their respective clocks.

Refer to Figure 4.10 for the following discussion related to the flow-tree. Also, in the following discussion, the edge-weight in the flow-tree would be represented only by the clock phase; implicitly it represents clock cycle C_1 and the clock phase associated with it. Only when the transition crosses a cycle boundary, the clock cycle would be explicitly referred along with the clock phase.

Step-1: node S_1: Since successful launch is possible only in ϕ_1 (by assumption), a node corresponding to segment S_1 is created in the flow tree, and the edge weight for the edge coming into the root is set to ϕ_1.

Step-2: node S_2: Next, flow conditions must be examined to determine if the transition can propagate to S_2 in ϕ_1. The only flow-condition necessary to examine would be if the downstream latch, L_1, is transparent in ϕ_1. Since clk_1 = 1 in ϕ_1, the transition can flow through to S_2, resulting in the creation of a node for S_2 in the flow-tree (refer Figure 4.10).

Step-3: node S_3: Now, flow conditions must be examined to determine if the transition can propagate along S_2 in ϕ_1 and through the downstream latch L_2 to segment S_3. Thus, flow-conditions to be examined at node S_2 would be: (a) is S_2 sensitized in phase ϕ_1; (b) is the downstream latch L_2 transparent in ϕ_1. Condition (a) can be evaluated by performing a look-up on the sensitization information provided by step-1. In this case, condition (a) evaluates to true (by assumption, each segment can be sensitized in every phase). However, condition (b) evaluates to false (clock to latch L_2 (clk_2) is at logic 0 during ϕ_1; L_2 is not transparent). Since all the flow conditions did not evaluate to true, no node is added along this edge.

Now, the next unexplored clock phase i.e. ϕ_2 is explored at node S_2, and the following flow-conditions are now evaluated: (a) is the upstream latch, i.e. L_1 in the hold-state in ϕ_2; (b) is S_2 sensitized in ϕ_2; (c) is downstream latch L_2 transparent in ϕ_2. Condition (a) was not examined earlier, and is a new condition. Let us first understand the need to evaluate condition (a). If upstream latch L_1 is in the hold-state (i.e. the latch is not transparent), it holds its value from previous clock phase, indicating that the final value associated with the transition is still available at segment S_2. However, if latch L_1 is transparent, then we must ascertain if the previous segment (S_1) is sensitizable in ϕ_2. If S_1 is no longer sensitizable in ϕ_2, then the transition would be killed. It was not necessary to evaluate condition (a) earlier (during phase ϕ_1) because it was already known that the transition propagated through S_1

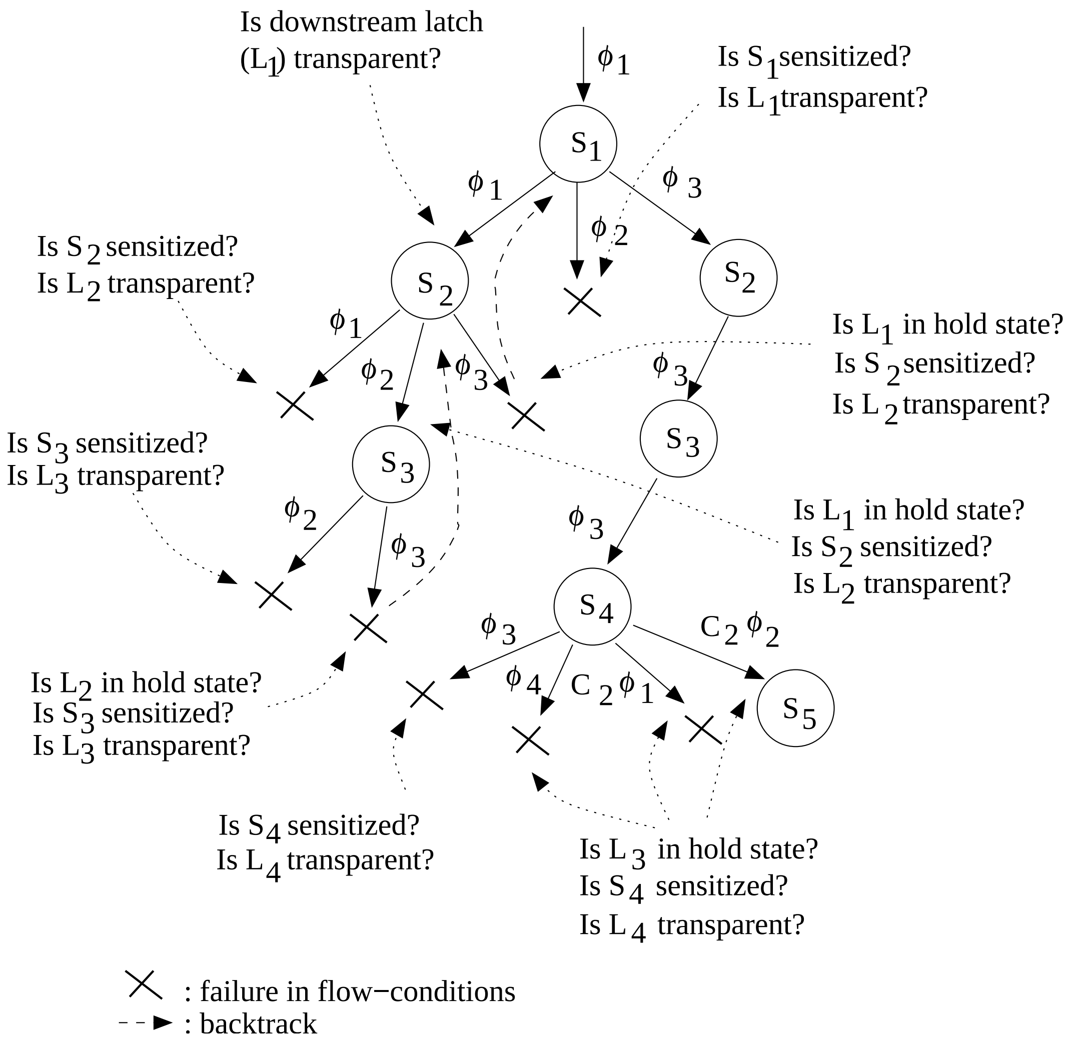


Figure 4.10: Flow-tree for the M-pdf shown in Figure 10

to S_2 in ϕ_1. Now let us determine if condition (a) evaluates to true or false in phase ϕ_2. Since the clock to latch L_1 goes low during ϕ_2, condition (a) evaluates to true (i.e. L_1 holds the value latched during the previous phase). Condition (b) is evaluated through a lookup on the sensitization information provided by step-1. Since S_2 is sensitizable in ϕ_2 (by assumption, each segment can be sensitized in every phase), condition (b) evaluates to true. Finally, condition (c) evaluates to true because clock to L_2 goes high in ϕ_2. Since all conditions evaluate to true, a node for segment S_3 is added to the flow-tree as shown in Figure 4.10.

Step-4: node S_4: Flow conditions must be examined to determine if the transition available at S_3 can reach S_4 in ϕ_2. At ϕ_2 these flow-conditions would be: (a) is S_3 sensitized in ϕ_2; (b) is downstream latch L_3 transparent in ϕ_2. Condition (a) would evaluate to true (by assumption). Since condition (b) evaluates to false (clock to L_3 is low), the next unexplored clock-phase (ϕ_3) would be examined. Flow conditions for ϕ_3 would be: (a) is the upstream latch, i.e. L_2 in the hold-state in ϕ_3; (b) is S_3 sensitized in ϕ_3; (c) is downstream latch L_3 transparent in ϕ_3. Since condition (a) evaluates to false (clock to L_2 is high in ϕ_3), a backtrack is performed to the previous segment, S_2.

The flow-conditions at S_2 for the next-unexplored clock-phase (ϕ_3) must be examined. These conditions at S_2 are: (a) is the upstream latch, L_1 in the hold-state in ϕ_3; (b) is S_3 sensitized in ϕ_3; (c) is downstream latch L_2 transparent. Condition (a) evaluates to false because L_1 is transparent in ϕ_3. So, a backtrack is performed, and the flow-conditions are examined for S_1 for the next-unexplored phase. The next unexplored phase for S_1 is ϕ_2. Flow conditions for S_1 are: (a) is S_1 sensitized in ϕ_2; (b) is downstream latch L_1 transparent. Condition (a) evaluates to true, but condition (b) evaluates to false (clock to L_1 is low in ϕ_2). Both these flow conditions evaluate to true for the next phase, ϕ_3; so a new node for segment S_2 is created in the tree, corresponding to the flow of transition from S_1 to S_2 in ϕ_3, as shown in Figure 4.10. Similar flow-conditions examined next for S_2 and S_3 also evaluate to true in phase ϕ_3, so, new nodes for S_3 and S_4 are created with edge weights ϕ_3, as shown in the right half of Figure 4.10.

Step-5: node S_5: Flow conditions must be examined at node S_4 to determine if the transition can reach the final segment, S_5. Using similar flow conditions as used for previous segments, a node for S_5 is created in the flow-tree as shown in Figure 4.10.

When a node for S_5 is created in the flow-tree, a complete path from the root node to the leaf node is created, which represents a potential multi-cycle clocking scheme using which the path can become testable. We use the word potential because the simultaneous sensitization of all segments corresponding to this clocking scheme is yet to be determined. Currently, all segments are known to be sensitized only independently. □

Having explained the flow conditions through the example above, we now state the flow-conditions formally, which need to be applied for various portions of the flow-tree.

Flow Conditions:

I. Flow from segment 1 to segment 2:

A. *If transition has been launched in ϕ*:

Flow from the 1^{st} segment to the 2^{nd} segment is possible in phase ϕ if (e.g. 1^{st} edge from node S_1 in Figure 4.10):

1. Downstream latch is transparent in ϕ.

If condition (1) fails, next phase is considered for sensitization of 1^{st} segment. Sensitization for the 1^{st} segment is not checked for phase ϕ because a successful launch in ϕ (Definition 38) includes sensitization conditions implicitly. If condition (1) passes, node for 2^{nd} segment is created.

B. *If transition has been launched prior to phase ϕ*:

Flow from the 1^{st} segment to the 2^{nd} segment is possible in phase ϕ if (e.g. 2^{nd}/3^{rd} edge from S_1 in Figure 4.10):

1. 1^{st} segment can be sensitized in ϕ
2. Downstream latch is transparent in ϕ

If either condition fails, consider sensitization in the next phase; otherwise node for 2^{nd} segment is created.

II. Flow from any other segment-i to segment (i+1)

In either case: transition launched in ϕ or prior to ϕ, to determine the flow of transition for any i^{th} segment to segment-(i+1) in phase ϕ ($i \neq 1$):

- If this is the 1^{st} out-going edge from node for segment-i, check if:
    1. segment-i is sensitized in ϕ and
    2. downstream latch is transparent in phase ϕ.

If either condition fails, next phase is considered for sensitization. If both conditions pass, node for segment $(i+1)$ is created. An example is the 1^{st} edge from S_2 in Figure 4.10 (for S_2 in the left part of the tree conditions fail in ϕ_1; for S_2 in the right part, conditions pass in ϕ_3).

- If this is not the 1^{st} out-going edge from the node for segment-i, check if:
  1. upstream latch is in the hold-state in ϕ;
  2. segment-i is sensitized in $\phi$and
  3. downstream latch is transparent in ϕ.

  If condition (1) fails, backtrack to previous segment. If condition (2) or (3) fails, next phase is considered for sensitization of segment-i. If all conditions pass, node for segment $(i+1)$ is created. An example of conditions passing is the 2^{nd} edge from node S_2 in the left part of the tree in Figure 4.10. An example of conditions failing is 2^{nd} edge from node S_3 in left part of the tree in Figure 4.10.

Step-3: Simultaneous Sensitization

This step takes as input the information provided by step-2 and determines if simultaneous sensitization of all segments is possible in the clock-phases as indicated by the flow-tree. As an example, the flow-tree in Figure 4.10 indicates that after launching the transition in phase ϕ_1 during cycle C_1, a possible flow of the transition along the path can occur when:

Segment S_1 is sensitized for ϕ_1 ϕ_3 during cycle C_1; segments S_2 and S_3 are sensitized in ϕ_3 during C_1; S_4 is sensitized during ϕ_3 and ϕ_4 during C_1 and during ϕ_1 and ϕ_2 during cycle C_2.

The goal of step-3 is to determine if the simultaneous sensitizations of different segments in various clock phases during different clock cycles results in a conflict. Sequential logic implications are used to imply the conditions necessary for simultaneously satisfying all conditions imposed by the flow-tree. If a conflict occurs during logic implications, the flow provided by step-2 is discarded, and other potential flows must be explored in the flow-tree. A conflict during step-3 causes the execution of step-2 to be resumed at the instance from where step-3 was invoked. Conflict during step-3 is treated as a failure in flow-conditions, resulting in the exploration of other paths in the flow-tree that connect the root (first segment) to the leaf (last segment).

For example, if the flow-indicated by Figure 4.10 results in a conflict during simultaneous sensitization, then step-2 would resume execution at node S_4 (right part of the tree). Phase ϕ_2 in cycle C_2 resulted in a failure (conflict during step-3); as a result flow-conditions must be checked for the next unexplored phase at node S_4. However, since all unique phases have already been explored at node S_4 (phase ϕ_3 and ϕ_4 in cycle C_1 and phase ϕ_1 and ϕ_2 in cycle C_2), backtrack must be performed to node S_3 and flow-conditions must be checked for the next unexplored phase at node S_3. This process would continue till a new path is identified from the root to the leaf in the flow-tree or till all possibilities of propagating the transition from the root (S_1) to the leaf (S_5) have been exhausted.

As explained, our three-step methodology incrementally identifies untestable M-pdfs from identifying combinationally untestable segments (step-1) to identifying impossible propagation of signal transition across multiple segments (step-2) to impossible conditions for simultaneous sensitization of all segments in the M-pdf (step-3). Each step plays a key role in our framework, and experimental results in the next section demonstrate that our 3-step methodology described above helps in the efficient identification of non-robustly untestable M-pdfs.

4.3 Experimental Results

In this section we provide results of applying our framework towards identifying untestable M-pdfs in industrial designs. Our 3-step methodology was implemented in C++ and experiments were conducted on an Intel Xeon 2.8 GHz processor, running Linux as the operating system. Experimental results are shown in Table 4.1. Column 1 and Column 2 respectively show various circuits and the number of gates in each circuit. All circuits incorporate latches as sequential elements and have more than one design clock. Column 3 shows the number of M-pdfs we used for our experiments. These M-pdfs were randomly generated for the purpose of experiments, and for each circuit, the number of M-pdfs were limited to 400K paths. Each M-pdf has at most 5 static segments. Column 4, 5 and 6 show the number of M-pdfs that were identified as non-robustly untestable. Column 4 shows the number of M-pdfs identified as untestable only with step-1 (independent sensitization). Column 5 shows the number of untestable M-pdfs identified when step-2 (flow-analysis) was also performed in addition to step-1. Finally, Column 6 shows the number of untestable M-pdfs when step-3 (simultaneous sensitization) was performed in addition to step-1 and step-2. The following observations can be made from the data in Columns 4 through 6:

- A large proportion of untestable M-pdfs is identified during independent sensitization. This indicates that these paths contain segments which are non-robustly untestable combinationally (each segment is analyzed independently of other segments, and no phase relationship between segments is accounted for in step-1).
- When step-2 was performed after step-1, for some circuits (such as Ckt_1, Ckt_2, Ckt_3), the number of untestable faults identified went up by as much as 35-50%. This indicates that for these circuits, a large subset of the M-pdfs remaining after step-1 contain segments which have phase relationships that do not allow signal transitions to propagate from the 1^{st} segment to the last. However, for some circuits (such as Ckt_4, Ckt_5, Ckt_6), step-2 did not help identify any additional untestable M-pdfs over step-1. In these circuits for the M-pdfs remaining after step-1, flow analysis could identify at-least one clocking scheme for which the transition could potentially propagate from the 1^{st} segment to the last.

Table 4.1: Results: Experiments performed on industrial designs

Circuit	**#gates**	**#M-pdfs**	**#UNT M-pdfs**			**Time (seconds)**		
			Step-1	**Step-2**	**Step-3**	**Step-1**	**Step-2**	**Step-3**
Ckt_1	11K	16K	8652	13055	13055	102.1	113.1	117.2
Ckt_2	16K	84K	44355	68095	68104	1756.6	1803.6	2074.1
Ckt_3	25 K	29K	18723	25384	25384	174.2	183.5	186.4
Ckt_4	6K	55K	4992	4992	9798	327.1	340.0	477.6
Ckt_5	23K	400K	158591	158591	158591	3992.6	4015.5	4796.8
Ckt_6	177K	400K	44064	44064	44411	4221.2	4248.0	6430.0
Ckt_7	324K	400K	226664	226687	226687	1254.7	1259.9	1421.6
Ckt_8	474K	400K	315079	320377	320377	820.5	941.3	1062.6
Ckt_9	602K	400K	297622	298109	298110	1032.7	1036.8	1174.1

- When step-3 was performed after step-1 and step-2, for Ckt_4, the number of untestable paths increases significantly (almost doubles). This demonstrates the value of step-3 in identifying a significant percentage of untestable faults in certain types of circuits. Also, for Ckt_2 and Ckt_6, there is a small increase in untestable paths. For M-pdfs identified as untestable in step-3, all clocking scenarios generated during flow-analysis (step-2) resulted in conflicts when all segments were sensitized simultaneously.

Columns 7, 8 and 9 show the run times of our tool for performing (a) step-1, (b) steps 1 and 2, and (b) steps 1, 2 and 3, respectively. The relatively low run-times indicate that our framework scales well even for very large industrial designs. Note that if conventional techniques such as [45, 47, 57–60] which are proposed for identifying untestable single-cycle pdfs were used for multi-cycle paths, only single-cycle untestable faults (similar to data reported in Column 4 of Table 4.1) would be identified. In our work, we contribute by providing a framework that would also efficiently take into account the clocking relationships between segments of a multi-cycle path to additionally identify untestable multi-cycle paths which would not be captured by previous techniques. Also, note that while we spend at most a few hours in identifying untestable M-pdfs, the knowledge provided by our tool can potentially save several man days wasted during fault-grading and silicon debug while targeting these untestable M-pdfs.

4.4 Summary

In this chapter, we first defined conditions for a non-robust path delay test for a multi-cycle path delay fault (M-pdf). We then proposed a novel 3-step framework to identify non-robustly untestable M-pdfs with the aid of sequential implications. Compared to previous techniques that primarily address untestability in single-cycle path delay faults, we proposed a first-of-its-kind framework geared towards identifying untestability in M-pdfs in practical latch-based industrial circuits with multiple clock domains. Experimental results demonstrate the value of each of the 3 steps in our approach. The results also show that our framework scales well with large industrial designs and enables the identification of those path-delay faults which conventional techniques do not address.

Chapter 5

Conclusion

Several new concepts were presented in this dissertation which efficiently apply logic implications to various applications in VLSI test and verification. We presented novel insights into logic implications, and used characteristics associated with sequential implications to identify significantly more untestable faults in benchmark circuits than could be identified by earlier techniques. Specifically, we identified the existence of single and dual recurrence relations in sequential circuits and used this knowledge to infer unachievable logic assignments in these circuits. The information about unachievable nets paid rich dividends in terms of untestable fault identification. We also extended single line conflict analysis to multiple line conflict analysis to quickly identify impossible value assignments in logic circuits. Since the identified value assignments are impossible to achieve, all faults which require these assignments as their necessary condition for detection were identified as untestable. The knowledge of unachievable nets coupled with multiple line conflict analysis resulted in a significant leap in the number of identified untestable faults.

Since logic implications find several applications in various facets of VLSI test and verification, we presented an efficient mechanism to improve Boolean learning through logic implications. We proposed a new kind of Boolean learning by means of extended forward implications. The new learning criterion used the novel concept of implication frontier to identify those non-trivial relationships which previous techniques failed to identify. While several applications can benefit from logic implications, in this dissertation we showed the impact of improved learning through the additional untestable faults extended forward implications helped to identify.

Next, we applied logic implications towards equivalence checking of sequential circuits. We targeted an uninvestigated verification problem, which is significantly harder than conventional sequential hardware equivalence. Specifically, we addressed the problem of verifying the delay replaceability of sequential circuits. We proposed a unique Boolean satisfiability based framework that utilized sequential implications to prove the delay replaceability of sequential circuits using an inductive methodology.

Finally, we presented the formal definition of multi-cycle path delay faults (M-pdfs) along with the definition of a non-robust test for such faults. Testing of such path-delay faults is critical to ensure the reliability of high-performance designs. However, the test process for such faults suffers from severe performance loss due to the presence of untestable M-pdfs. Thus, a framework to identify untestable M-pdfs is vital to the efficiency of the testing for such faults. In this dissertation, we presented a first-of-its-kind framework to identify untestable M-pdfs. This framework utilizes sequential implications and a 3-step methodology to incrementally identify untestable M-pdfs. Our approach was applied for large industrial designs, and the scalability of the technique was observed in identifying untestable M-pdfs for large designs. Our results can be of prime importance to functional test generation and silicon debug where significant resources are wasted while targeting untestable M-pdfs.

The unifying theme of this dissertation was the use of Boolean learning in terms of logic implications towards several applications in test and verification. In that process, we defined several novel concepts, presented new, scalable and efficient frameworks which can benefit the industry and academia equally.

Bibliography

[1] M. H. Schulz, E. Trischler, and T. M. Sarfert, “SOCRATES: A highly efficient automatic test pattern generation system,” *IEEE Trans. Computer-Aided Design*, vol. 7, Jan 1998, pp. 126137.

[2] A. El-Maleh, M. Kassab and J. Rajski, “A fast sequential learning technique for real circuits with application to enhancing ATPG performance,” in *Proc. of Design automation Conf.*, June 1998, pp. 625 631.

[3] P. Tafertshofer; A. Ganz and K. J. Antreich, “IGRAINE-an Implication GRaph-bAsed engINE for fast implication, justification, and propagation,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, August 2000, pp. 907 927

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

[5] M. Enamul Amyeen, W. K. Fuch, I. Pomeranz, V. Boppana, “Implication and evaluation techniques for proving fault equivalence,” in *Proc. IEEE Vlsi Test Symposium*, April 1999 pp. 201 207

[6] D. Paul, M. Chatterjee and Dhiraj K. Pradhan, “VERILAT: Verification Using Logic Augmentation and Transformations,” *IEEE Trans. On CAD of Integrated Circuits and Systems*, vol. 19, no. 9, Sept. 2000

[7] J. Marques-Silva and T. Glass, “Combinational equivalence checking using satisfiability and recursive learning,” in *Proc. Design, Automation and Test in Europe Conf.*,March 1999, pp.145 149

[8] R. Arora, and M. S. Hsiao, “Enhancing SAT-based bounded model checking using sequential logic implications,” in *Proc. Of Intl conference on VLSI Design*, Jan. 2004, pp. 784-787

[9] H. Ichihara, K. Kinoshita, “On acceleration of logic circuits optimization using implication relations,” in *Proc. Asian Test Symposium*, Nov. 1997 pp. 222 - 227

[10] W. Kunz, D. Stoffel, and P. R. Menon, “Logic optimization and equivalence checking by implication analysis,” *IEEE Trans. On CAD of Integrated Circuits and Systems*, Volume: 16 ,Issue: 3, March 1997, pp.266 281

[11] M. A. Iyer and M. Abramovici, “FIRE: a fault independent combinational redundancy algorithm,” *IEEE Trans. on VLSI*, June 1996, pp. 295-301.

[12] M.A. Iyer, D.E. Long and M. Abramovici, “Identifying Sequential Redundancies without Search,” in *Proc. of Design Automation Conf.*, 1996, pp. 457-462

[13] Qiang Peng, M. Abramovici and J. Savir, “MUST: Multiple-Stem Analysis for Identifying Sequentially Untestable Faults,” in *Proc. of Intl Test Conference*, Nov. 2000. pp. 839-846

[14] M. S. Hsiao, “Maximizing Impossibilities for Untestable Fault Identification,” in *Proc. Design Automation and Test in Europe Conf.*, March 2002, pp. 949-953

[15] M. Syal, M. S. Hsiao, “Untestable Fault Identification Using Recurrence Relations and Impossible Value Assignments,” of *Proc. Of Intl conference on VLSI Design*, Jan. 2004, pp. 481-486.

[16] M. H. Schulz and E. Auth, “Improved deterministic test pattern generation with applications to redundancy identification,” *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 811816, July 1989.

[17] W. Kunz and D. K. Pradhan, “Accelerated dynamic learning for test pattern generation in combinational circuits,” *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 684694, May 1993.

[18] J. Rajski and H. Kox, “A Method to Calculate Necessary Assignments in ATPG,” in *Proc. Intl. Test Conf.*, Nov. 1990, pp. 25-34

[19] S.T. Chakradhar and V. D. Agarawal, “A transitive closure algorithm for test generation”, *IEEE Transactions on CAD*, 1993, pp. 1015 - 1028

[20] W. Kunz and D. K. Pradhan, “Recursive Learning: A new Implication Technique for Efficient Solutions to CAD problems-test, verification, and optimization,” *IEEE Trans. on CAD*, pp. 1149-1158. Sept 1994,

[21] J. Zhao, J. A. Newquist and J. Patel, “A graph traversal based framework for sequential logic implication with an application to c-cycle redundancy identification,” in *Proc. VLSI Design Conf.*, 2001, pp. 163-169.

[22] A. D. Friedman, “Fault Detection in Redundant Circuits,” *IEEE Transactions on Electronic Computers*, vol. EC-16, February 1967.

[23] K.T. Cheng, “Redundancy Removal for Sequential Circuits without Reset States,” *IEEE Tran. On CAD*, vol. 12, no. 1, Jan. 1993, pp 13-24

[24] V. D. Agrawal and S. T. Chakradhar, “Combinational ATPG Theorems for Identifying Untestable Faults in Sequential Circuits,” *IEEE Trans. On CAD*, vol. 14, no. 9, Sept. 1995, pp. 1155-1160.

[25] S. M. Reddy, Irith. Pomeranz, X. Lim and Nadir Z. Basturkmen, “New procedures for identifying Undetectable and Redundant Faults in Synchronous Sequential Circuits,” in *Proc. VLSI Test Symposium*, 1999. pp. 275 -281.

[26] D. E. Long, M. A. Iyer, M. Abramovici, “FILL and FUNI: Algorithms to Identify Illegal States and Sequentially Untestable Faults,” in *ACM TODAES*, Pages 631-657

[27] V. Singhal, C. Pixley, A. Aziz, R. K. Brayton, “Theory of safe replacements for sequential circuits,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 20, no. 2, Feb. 2001, pp. 249 - 265.

[28] C. Pixley, “A theory and implementation of sequential hardware equivalence,” *IEEE Trans. Computer Aided Design*, vol. 11, Dec. 1992, pp. 1469-1494.

[29] K. -T. Cheng, “Redundancy removal for sequential circuits without reset states,” *IEEE Trans. on Computer Aided Design*, vol. 12, Jan. 1993, pp. 13-24.

[30] S. -Y. Huang, K. -T. Cheng, K. -C. Chen, “Verifying sequential equivalence using ATPG techniques,” *ACM TODAES*, vol. 6, no. 2, April 2001, pp. 244 - 275.

[31] O. Coudert, C. Berthet and J. C. Madre, “Verification of Synchronous Sequential Machines Based on Symbolic Execution,” *Automatic Verification Methods For Finite State Systems*, LNCS no. 407, Springer Verlag, 1990.

[32] H.J. Touati, H. Savoj, B. Lin, R. K. Brayton and A. Sangiovanni-Vincentelli, “Implicit State Enumeration of Finite State Machines using BDD,” in *Proc. ICCAD*, Nov. 1990, pp. 130-133.

[33] J. Hong, R. Jiang and R. K. Brayton, "On the Verification of Sequential Equivalence," in *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 22, no. 6, June 2003, pp. 686-697.

[34] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Trans. on Comp.*, vol. 48, no. 5, May 1999, pp. 506 - 521.

[35] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proc. of Design Automation Conf.*, June 2001, pp. 530 - 535

[36] E. Goldberg, and Y. Novikov, "BerkMin: A fast and robust SAT-solver," in *Proc. of Design Automation and Test in Europe*, March 2002, pp. 142 - 149.

[37] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking using Satisfiability Solving," *Formal Methods in System Design*, 19(1):7-34, July 2001, Kluwer Academic Publishers.

[38] J. P. Whittemore, J. Kim, and K. A. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," in *Proc. of Design Automation Conf.* , June 2001, pp. 542-545.

[39] L. Zhang, M. R. Prasad, and M. S. Hsiao, "Incremental Deductive and Inductive Reasoning for SAT-based Bounded Model Checking," in *Proc. of Intl' Conf. on CAD*, Nov. 2004, pp. 502-509.

[40] http://www.princeton.edu/∼chaff/zchaff.html

[41] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *Proc. of Design, Automation and Test in Europe*, Feb. 1998, pp. 618 - 623.

[42] K. Ng, M. R. Prasad, R. Mukherjee, and J. Jain, "Solving the latch mapping problem in an industrial setting," in *Proc. of Design Automation Conf.*, June 2003, 442-447.

[43] D. Anastasakis, R. Damiano, H. -K.T. Ma, and T. Stanion, “A practical and efficient method for compare-point matching,” in *Proc. Design Automation Conf.*, June 2002, pp. 305 - 310.

[44] K. Fuchs, M. Pabst and T. Rssel, “RESIST: A Recursive Test Pattern Generation Algorithm for Path Delay Faults Considering Various Test Classes,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 12, No. 12, December 1994, Pages 1550-1562

[45] K. Heragu, J. H. Patel and V. D. Agrawal, “Fast Identification of Untestable Delay Faults Using Implications,” in *Proc. Intl’ Conference on CAD*, November 1997, Pages 642-647.

[46] W. C. Wu, C. L. Lee and J. E. Chen, “Identification of Robust Untestable Path Delay Faults,” in *Proc. of Asian Test Symposium*, November 1995, Pages 229 - 235

[47] Z. Li, Y. Min and R. K. Brayton, “A New Low-Cost Method for Identifying Untestable Path Delay Faults,” in *Proc. of Asian Test Symposium*, December 1998, Pages 76-81

[48] M. Syal, M. S. Hsiao, “A Novel, Low-Cost Algorithm for Sequentially Untestable Fault Identification,” in *Proc. IEEE Design Automation and Test in Europe Conf.*, March 2003, pp. 316-321.

[49] I. Pomeranz and S. M. Reddy, “Classification of Faults in Synchronous Sequential Circuits,” *IEEE Trans. on Computers*, vol. 42, No. 9, Sept. 1993. Pages 1066-1077

[50] Hsing - Chung Liang, Chung Len Lee, Jwu E. Chen., “Identifying Untestable Faults in Sequential Circuits,” *IEEE Design and Test of Computers*, Fall 1995, Pages 14-23

[51] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, vol. C-30, no. 3, March 1981, pp. 215-222.

[52] L. H. Goldstein, "Controllability/Observability Analysis of Digital Circuits," *IEEE Trans. on Circuits and Systems*, vol. CAS-26, no. 9, Sept. 1979, pp. 685-693.

[53] Z. Barzilai and B. Rosen, "Comparison of AC Self-Testing Procedures," in *Proc. Intl Test Conf.*, Nov. 1983, pp. 89-94.

[54] J. L. Carter, V. S. Iyenger and B. K. Rosen, "Efficient Test Coverage Determination for Delay Faults," in *Proc. Intl Test Conf.*, Sept. 1987, pp. 418-427.

[55] K. T. Cheng and H. C. Chen, "Delay Testing for Non-Robust Untestable Circuits," in *Proc. Intl Test Conf.*, Oct. 1993, pp. 954-961.

[56] A. Krsti and K. T. Cheng, Delay Fault Testing for VLSI Circuits. Boston: Kluwer Academic Publishers, 1998.

[57] Z. Li, Y. Min and R. K. Brayton, "Efficient Identification of Non-Robustly Untestable Path Delay Faults," in *Proc. Intl Test Conf.*, Nov. 1997, pp. 992-997.

[58] S. Kajihara, T. Shimono, I. Pomeranz and S. M. Reddy, "Enhanced Untestable Path Analysis Using Edge Graphs," in *Proc. Asian Test Symposium*, Dec. 2000, pp. 139-144

[59] Y. Shao, S. M. Reddy, S. Kajihara and I. Pomeranz, "An Efficient Method to Identify Untestable Path Delay Faults," in *Proc. Asian Test Symposium*, Nov. 2001, pp. 233-238

[60] A. Krishnamachary and J. A. Abraham, "Effects of Multi-cycle Sensitization on Delay Tests," in *Proc. Intl Conf. on VLSI Design*, Jan. 2003, pp. 137-142

[61] S. Kundu, C. Tirumurti, R. Jayabharathi, P. Parvathala, “A Path Delay Fault Simulation System,” in *Proc. European Test Workshop*, May 2002.

[62] E. S. Park and M. R. Mercer, “Robust and Non-robust Tests for Path Delay Faults in Combinational Circuits,” in *Proc. Intl Test Conf.*, Sept. 1987, pp. 1027-1034.

[63] M. Syal, S. Chakravarty and M. Hsiao, “Identifying Untestable Transition Faults in Latch Based Designs with Multiple Clocks,” in *Proc. Intl Test Conf.*, Oct. 2004, pp. 1034-1043.