# EFFICIENT PATH DELAY TEST GENERATION WITH BOOLEAN SATISFIABILITY

A Dissertation

by

KUN BIAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2013

Major Subject: Computer Engineering

## ABSTRACT

This dissertation focuses on improving the accuracy and efficiency of path delay test generation using a Boolean satisfiability (SAT) solver. As part of this research, one of the most commonly used SAT solvers, *MiniSat*, was integrated into the path delay test generator *CodGen*. A mixed structural-functional approach was implemented in *CodGen* where longest paths were detected using the K Longest Path Per Gate (KLPG) algorithm and path justification and dynamic compaction were handled with the SAT solver.

Advanced techniques were implemented in *CodGen* to further speed up the performance of SAT based path delay test generation using the knowledge of the circuit structure. SAT solvers are inherently circuit structure unaware, and significant speedup can be availed if structure information of the circuit is provided to the SAT solver. The advanced techniques explored include: Dynamic SAT Solving (DSS), Circuit Observability Don't Care (Cir-ODC), SAT based static learning, dynamic learnt clause management and Approximate Observability Don't Care (ACODC). Both ISCAS 89 and ITC 99 benchmarks as well as industrial circuits were used to demonstrate that the performance of *CodGen* was significantly improved with *MiniSat* and the use of circuit structure.

DEDICATION

To those who love me

ACKNOWLEDGEMENTS

I would like to thank my advisor and committee chair, Dr. Duncan M. (Hank) Walker for his advice and support throughout my Ph.D. studies at Texas A&M University. His insights in this particular research area, his technical guidance and spiritual support were invaluable to this work. This dissertation would never have been completed without his advice and encouragement. I owe him lots of gratitude for making my research life enjoyable and rewarding. What I learned from him will benefit my future career.

I am also grateful to my co-chair Dr. Sunil Khatri, and my committee members, Dr. Scott Miller and Dr. Paul Gratz, for their valuable suggestions and personal encouragement. Also I want to thank Dr. Rabi Mahapatra for his guidance during the early stage of my PhD study.

At the end, I also want to thank those guys who tried very hard to make my life here tougher. Without them, the days in a small town of central Texas could be much more boring.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

# 1. INTRODUCTION

## 1.1 Delay Testing

Physical defects such as electrical opens and shorts may occur during the fabrication process of semiconductor chips. Most defects affecting circuit performance are functional defects that can be detected using traditional test methods. [1][2][3] However, some manufacturing defects, which are not “large” enough to cause functional failure in the circuit, may only affect the operating speed of the circuit. Delay tests target the small manufacturing defects, to ensure that the manufactured chips work within the specified timing constraints. As fabrication processes are becoming more complex and the system clock speeds are getting faster, delay testing has become essential in ensuring the proper operation of the manufactured chips. Based on their origins, delay faults can be classified as global delay faults, which are caused by global process parameter variations, and local delay faults, which are caused by local process disturbance. Delay fault models [4][5] have been developed as the abstraction of delay defects and have been implemented in software for the purpose of Automatic Test Pattern Generation (ATPG) [6][7] and fault coverage estimation [8]. Various delay fault models are presented in the Section 1.2. Among them, the path delay fault model is accurate enough to detect Small Delay Defects (SDDs) and therefore is the basis of this research. Section 1.3 presents circuit structures and approaches that are used to enable delay testing. Sections 1.4 and 1.5 introduce the idea of the K Longest Path Per Gate (KLPG) delay test approach, as well as pseudo-functional testing.

## 1.2 Delay Fault Models

A defect in a circuit is the unintended manufacturing difference between the actual circuit implementation and the specification. A fault is the representation of a defect at the abstracted function level. A fault model is an abstraction of the behavior of the circuit in the presence of a defect. Some popular delay fault models are discussed in the following sub-sections.

### 1.2.1 Transition Fault Model

The transition fault (TF) model [4] is the most commonly used delay fault model. It assumes that the delay fault affects only one place in the circuit. In this model, each gate input and output is assumed to have two transition faults: a slow-to-rise (STR) and a slow-to-fall (STF) delay fault. Thus the fault space of transition fault test is linear in the number of gates in the circuit. The extra delay introduced by the transition fault is assumed to be large enough to prevent the transition from reaching any observable primary outputs within the specified time. In other words, the transition fault effect can be observed through any path (whether long or short) to any observable primary output. This eliminates the need to consider circuit timing when generating transition fault tests.

Stuck-at fault test generation tools can be easily extended to generate tests for transition faults [2]. A transition fault test vector pair {v1, v2} can be composed by pairing stuck-at-0 and stuck-at-1 test patterns. The first vector v1 initializes the circuit and the second vector v2 sensitizes and propagates the fault effect to observable primary output(s). Any stuck-at fault is covered by a corresponding transition fault test, since a stuck-at fault can be considered as an asymptotically very slow transition fault.

The main disadvantage of the TF model is that the size (i.e. the value of the delay caused by the defect) of the fault is not considered. Transition fault test generators normally select the “easiest” path, which is the shortest one in most cases, to activate and propagate a transition. Thus the quality of TF test for small delay defects is a concern [9][10]. Another problem is that TF test often propagates a glitch from the fault site [11], which introduces a potential loss in the quality of the tests.

### 1.2.2 Gate Delay Fault Model

The gate delay fault model [6][7][8][12] assumes that a spot defect is lumped on a gate input or output and takes into account the size of the extra delay. Detecting such faults requires testing a long path through the fault site. It is necessary to specify the delay fault size in order to determine the quality of a test set, which is defined by how close the minimum detected delay fault sizes are to the minimum detectable fault sizes.

### 1.2.3 Line Delay Fault Model

The line delay fault model [13][14] is a variation of the gate delay fault model. It requires testing a rising or falling delay fault through the longest sensitizable path on every line in the circuit. Sensitizing the longest path through the target line can detect the smallest delay defect on the target line. However, this model may fail to detect some defects [15] with the increase of process variation in new technologies [16].

### 1.2.4 Path Delay Fault Model

The path delay fault model [5] models the cumulative delay on a path. It is the most conservative model since the fault space is all paths in the circuit. This model assumes that any path can have any delay. A circuit is considered faulty with a path delay fault if any one path is slow for a rising or falling transition. Thus tests for the path delay fault model can be used to detect Small Delay Defects (SDDs) in the circuit. The primary limitation of the path delay fault model is that the number of paths in the circuit can be an exponential function of the number of gates. For this reason it is often not considered practical to test all paths in the circuit and achieve high test coverage. For example, ISCAS85 benchmark circuit c6288, a 16-bit multiplier has close to 1020 paths [17]. However, it has been shown that for several circuits of interest, the number of paths is significantly less than exponential in the number of gates in the design.

## 1.3 Scan Based Delay Test

Testability is a relative measure of the effort or cost of testing a logic circuit. Testability analysis can be performed by calculating the controllability and observability of each signal line in the circuit. [18] Design-For-Test (DFT) techniques are used to improve the testability of a circuit. The most widely used DFT technique is scan design. In scan design, all or most storage elements in the design are converted into scan cells and the scan cells are “stitched” together to form one or more scan chains. The idea of scan design is illustrated in Figure 1.

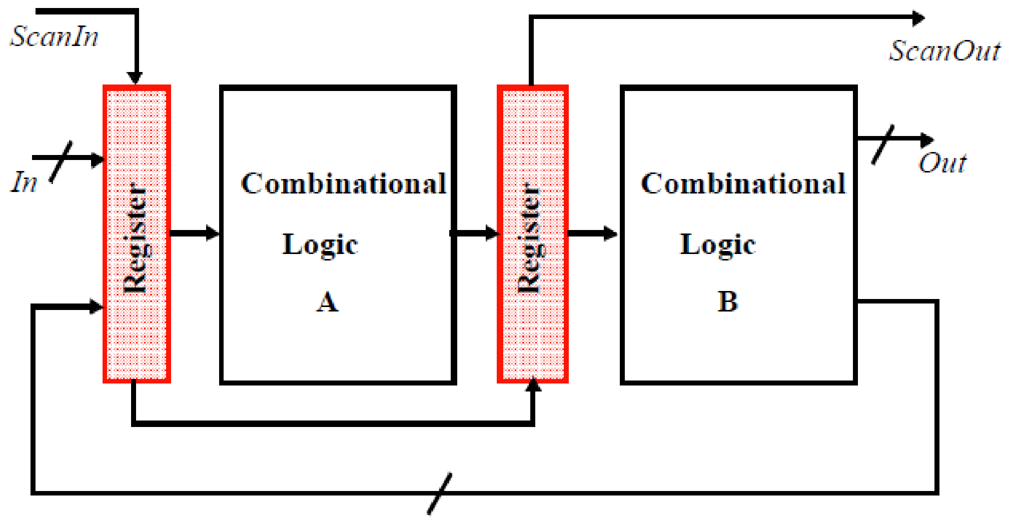


**Figure 1. Structure of scan design**

In scan based testing, first the Circuit-Under-Test (CUT) is set to scan mode and the test vectors are serially shifted into the scan registers (cells). Then the CUT is switched to test mode and the test vectors are applied to the combinational logic. The test results are captured into the scan cells at the next clock cycle. Then the CUT is switched back to scan mode and the test results are serially shifted out and compared with the expected responses. The next test vector can be shifted in at the same time. Scan design provides access to internal storage elements in the circuit which are not directly observable, and thus the testability of the circuit is improved. Another advantage of scan design is that the complexity of test generation is significantly reduced.

### 1.3.1 Scan Cell Types

#### 1.3.1.A Muxed-D Scan

Figure 2 shows an edge-triggered muxed-D scan cell design. The scan cell is composed of a multiplexer and a standard D flip-flop (FF). The scan enable (SE) signal controls the multiplexer to select between the data input (D) and scan input (SI). Clock signal (CP) is used to clock the flip-flop in both normal and test modes.

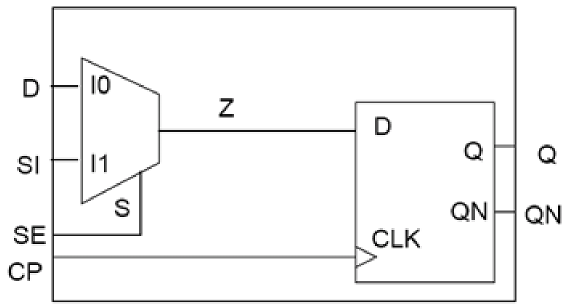


**Figure 2. Muxed-D Scan Cell**

#### 1.3.1.B LSSD Scan

A shift register latch (SRL) [19][20] can be used as a level sensitive scan design (LSSD) scan cell. This scan cell contains a pair of latches, a master two-port D latch L1 and a slave D latch L2. Clocks C, A and B are used to select between the data input D and the scan input I to drive +L1 and +L2, as shown in Figure 3. During test the SRLs are accessed by applying appropriate clock signal sequences. LSSD can be implemented using a single-latch design [19] or a double-latch design [21] based on different clock schemes.

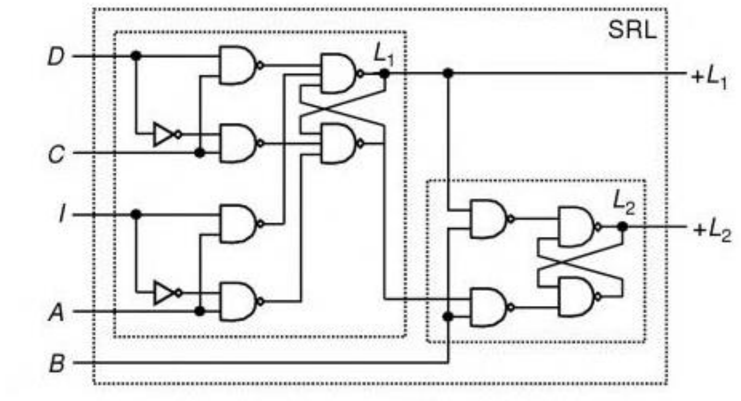


**Figure 3. LSSD Scan Cell**

1.3.1.C Enhanced Scan

Enhanced scan [22][23] allows storing two bits of data in the scan cell. Thus both the initialization and test vector can be loaded into a scan cell and applied consecutively to the circuit under test. For a flip-flop design, this is achieved by adding an extra holding latch to the output of each flip-flop. Since the two bits are independent of one another, a higher fault coverage can be achieved (since there is the ability to apply an arbitrary pair of test vectors). The main disadvantage of enhanced scan design is the extra area, timing and power introduced by the extra holding latch. An example of enhanced scan cell is shown in Figure 4.

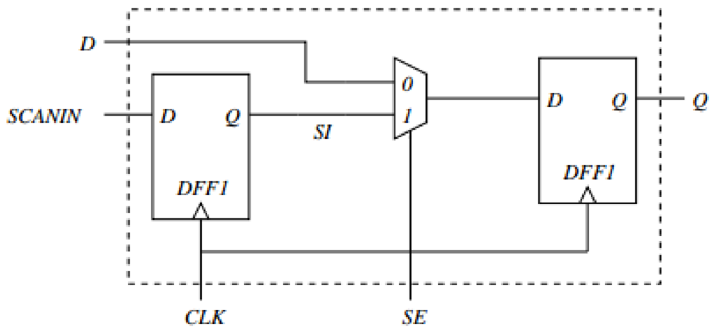


**Figure 4. Example of enhanced scan cell**

1.3.2 Scan Based Delay Testing

Delay testing requires launching transitions into the circuit: therefore there are two vectors in the test pattern of delay tests. The first vector initializes the circuit and is termed the initialization vector ($V_1$) and the second vector launches the transitions and is termed the test vector ($V_2$). According to the clocking scheme and the relationship between $V_1$ and $V_2$, delay tests can be classified into two types: Launch-On-Shift (LOS) [24][25] and Launch-On-Capture (LOC). [26] Figure 5 shows the clock diagram of both approaches. Experimental results show that LOS test can have higher delay fault coverage than LOC test. However, LOS test requires an at-speed Scan Enable (SE) signal, therefore it is more difficult to lay out a scan design in LOS style.

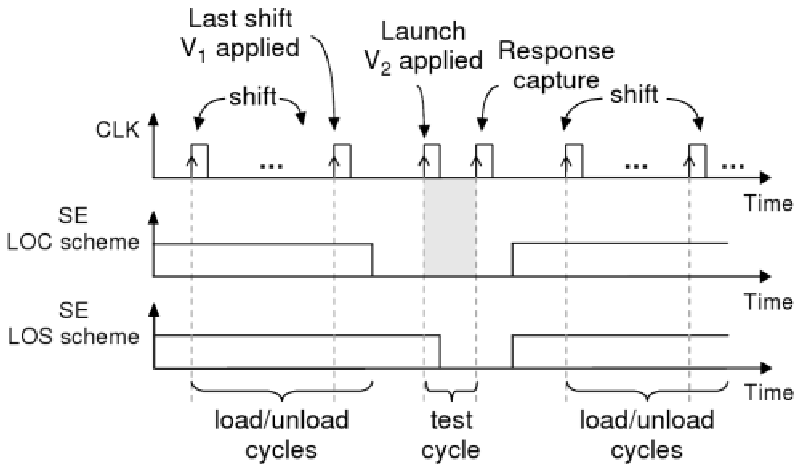


**Figure 5. Clock Diagram for Scan Based Delay Testing**

1.3.2.A Launch-On-Shift (LOS)

In the Launch-On-Shift scheme, the first vector $V_1$ is loaded to the scan cells for initialization and the second vector $V_2$ is shifted into the scan cells to launch the transition. In this case, $V_2$ is a one-bit shift of the first vector $V_1$. One capture clock cycle is then applied at-speed to capture the test response. The Scan Enable (SE) signal must switch during the test clock cycle, which is the reason why the SE signal must operate at speed in LOS design. Since this effectively requires the generation of a second clock network for the SE signal (which has tight timing constraints just like a clock signal does), LOS is not practical in high-speed designs.

#### 1.3.2.B Launch-On-Capture (LOC)

In the Launch-On-Capture scheme, two capture clocks are applied at speed to capture the test response into the scan cells. In this scenario, the second vector $V_2$ is the combinational circuit's response to the first vector $V_1$. The first capture clock is used to capture $V_2$ into the scan cells and launch transitions into the circuit, and the second capture clock is used to capture the test response. In LOC design, the SE signal is switched during the dead cycles between lowering the SE signal and applying the first capture clock so it can operate at lower speed. As a result, the timing constraints on the SE signal are less aggressive, and hence LOC is used in high-speed designs.

## 1.4 KLPG Algorithm and *CodGen*

Based on the path delay fault model, the *K Longest Path Per Gate* (KLPG) algorithm [27][28] has been proposed to efficiently test delay faults in combinational and sequential circuits. In KLPG based path delay test generation, delay tests for K longest rising and falling paths going through each gate (or line) are generated. The reason why more than one path has to be tested is that because of process variations, the longest path reported by timing analysis may not be the actual longest path in the fabricated circuit. [29] This is illustrated in Figure 6. As shown in this figure, because of process variation, each path has certain probability to be the longest path in the fabricated circuit as a result of process variations. All of these paths must be tested to detect a Small Delay Defect (SDD) at the targeted fault site. For example, path $P_1$ (which is not statically the longest path) may become the longest path in a fabricated design with a delay defect of size greater than $\Delta_1$.

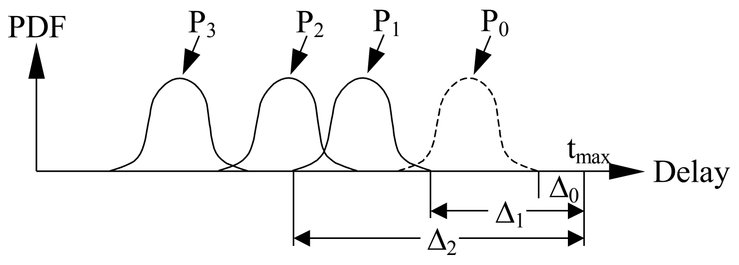


**Figure 6. Probabilistic distribution of path lengths**

This research is built on top of prior work with *CodGen*, which is a KLPG based path delay test generator supporting Pseudo Functional Test (PFT) [30] and dynamic compaction. [31] The basic *CodGen* algorithm is shown in Figure 7. In *CodGen*, path delay tests are generated in three steps:

- Path Search: For each targeted fault site, longest rising and falling paths are generated with the KLPG algorithm. The result of path search is the set of Necessary Assignments (NAs) to sensitize the path. An example of Necessary Assignment is a logic one value on the side input of an AND gate, in order to permit a transition to propagate from the other input to the output.

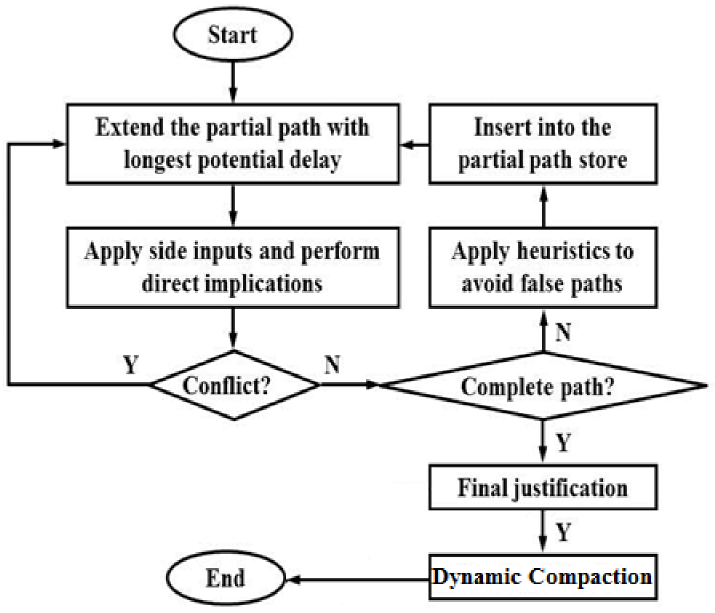

Start
Extend the partial path with longest potential delay
Insert into the partial path store
Apply side inputs and perform direct implications
Apply heuristics to avoid false paths
N
Y
Conflict?
N
Complete path?
Y
Final justification
Y
End
Dynamic Compaction

**Figure 7. Basic *CodGen* algorithm**

- Path Justification: The paths found in the first step are topological longest paths, but some of them can be false paths, which means that the transition cannot propagate along the path. Because of this the paths must be justified to check whether they are sensitizable. During justification, a set of input value assignments are found to set the necessary assignments. An example of a sensitizable path is shown in Figure 8. In this example, a=0 and c=1 are necessary assignments to sensitize the path from b to g, and X0 and S1 are the values on a and c to justify them. Here, “X0” on a means that the value is “X” or “don’t cares” (DC) in the first vector and 0 in the second vector. The value “S1” on c means “stable one”.

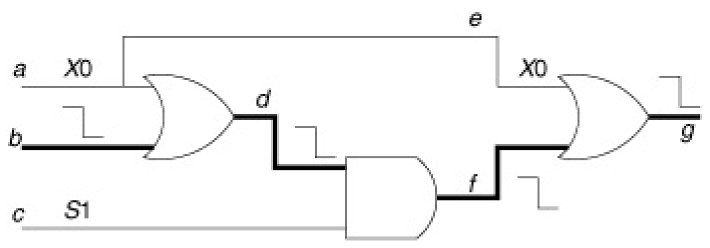


**Figure 8. Example of sensitizable path**

- Dynamic Compaction: Paths are compacted to reduce the total number of test patterns generated. In the case where test vectors have been generated for each path, the direct compaction of test vectors is called static compaction. Dynamic compaction is implemented in *CodGen* where the Necessary Assignments (NAs) of the paths are compacted to improve the compaction ratio. However, dynamic compaction takes significantly more CPU time than static compaction.

## 1.5 Pseudo Functional Test

The voltage level of the power grid in the circuit can significantly affect the accuracy of delay test. [32][33] The launching of the delay test causes a surge in the current drawn from the power grid. Because of the inductance on the power grid, this current surge will cause a large drop in power supply voltage, followed by inductive ringing, as shown in Figure 9. The temporarily lowered power supply voltage as a result of this ringing (marked on the left of Figure 9) will cause the circuit to operate more slowly than normal, and may cause a good chip to be rejected as bad. This is termed test overkill.

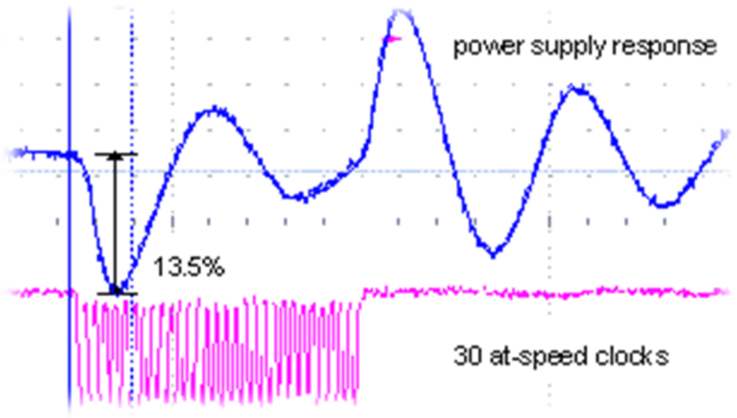


**Figure 9. Delay test induces drop of power supply voltage [32]**

The solution to this power supply voltage problem is called Pseudo Functional Test (PFT) where the delay test vectors are preceded by several medium-speed *preamble* cycles, after which the delay test vectors are launched at-speed. The timing diagram of PFT is shown in Figure 10. After the preamble cycles, the power supply voltage should have been stabilized such that it won't affect the accuracy of the delay test. However, introducing preamble cycles adds to the complexity of test generation. In this case, the test vectors scanned in from the tester are different from the actual delay test vectors applied during the launch cycle. So once the test generator has generated the test vectors for the delay tests, it must back-trace through the preamble time frames to determine the test patterns to scan in. This is termed *time frame expansion*, and is typically used during sequential test. It takes considerable CPU time to generate PFT patterns. Improving the performance of PFT test generation is the central focus of this dissertation.

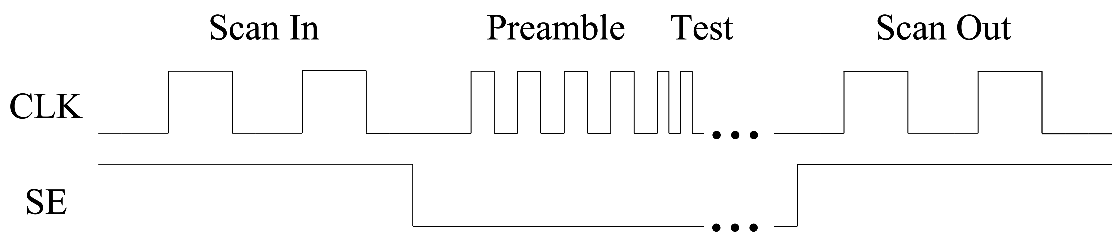


**Figure 10. Clock diagram of pseudo functional test**

## 1.6 Structure of This Dissertation

The rest of this dissertation is structured as follows: In Section 2, the design of a SAT based path delay test generator is presented along with its implementation using the *CodGen* structure. Section 3 discusses a number of advanced techniques to speed up the performance of SAT based ATPG and benchmark results are shown to justify the effectiveness of each technique. In Section 4, enhancements to *CodGen* are discussed which improve its usability. Section 5 concludes the research.

# 2. BOOLEAN SATISFIABILITY BASED DELAY TEST GENERATION

## 2.1 Motivation

Automatic Test Pattern Generation (ATPG) is a very time consuming process. In industry, it often takes several days or even weeks to generate test patterns for a newly designed digital chip. [34] This problem becomes more serious in the case of path delay testing, especially when Pseudo Functional Test (PFT) and dynamic compaction are used by the test generator, due to the increased complexity of these approaches. *CodGen* employs both of these technologies, and hence its runtime performance becomes a major concern. Besides runtime, accuracy is another concern of *CodGen* performance. Originally *CodGen* used the PODEM algorithm [35] in path justification and dynamic compaction. The built-in backtrack limitation in PODEM may cause the algorithm to give up some delay tests even though they can be justified with a more efficient method. This problem reduces the fault coverage of the delay tests.

Recently the performance of modern Boolean Satisfiability (SAT) solvers has significantly improved [36][37] and SAT based ATPG has been successfully implemented to generate test patterns for stuck-at faults and transition delay faults. [38][39] It is therefore natural to explore SAT based approaches for the purpose of generating test patterns for path delay faults.

The motivation of this research is to improve both the efficiency of path delay test generation and the fault coverage of delay tests with the help of a highly efficient SAT solver.

## 2.2 Boolean Satisfiability (SAT) Solver

### 2.2.1 Boolean Satisfiability Problem

Definition of Boolean Satisfiability Problem: Given a formula S, expressed in Conjunctive Normal Form (CNF), on Boolean variables $X = \{x_1, x_2 \ldots, x_m\}$, decide if there exists an assignment to X, such that S evaluates to true. In that case, S is said to be satisfiable. Otherwise, it is unsatisfiable.

### 2.2.2 Survey of SAT Solvers

Boolean Satisfiability (SAT) is one of the most well-known NP-complete problems. Despite its worst-case exponential runtime characteristics, general-purpose SAT solvers have found diverse applications in such areas as hardware and software verification, ATPG, scheduling and even machine intelligence. [40] Modern SAT solvers are able to solve large problems with over a million variables and several million constraints. Some of the most commonly used modern SAT solvers include: *Chaff*, [36] *Berkmin*, *Seige* and *MiniSat*. [37].

In essence, SAT solvers provide a generic combinational reasoning and search platform. SAT solvers can be divided into two categories: A complete SAT solver is one that, given the input formula F, eventually either produces a satisfying assignment for F or proves F is unsatisfiable. An incomplete SAT solver does not provide a guarantee that it will eventually either report a satisfying assignment or prove the given formula unsatisfiable. There is a preset backtrack limit in some incomplete SAT solvers, after which they may or may not produce a solution. Incomplete SAT solvers use many

Stochastic Local Search (SLS) strategies to improve their typical-case performance such that on many problems they significantly outperform their complete counterparts. However, these incomplete SAT solvers perform well on randomly generated SAT instances, but often perform much worse on circuit-derived structural SAT instances.

Most classic SAT solvers are built using the Davis-Putnam-Logemann-Loveland (DPLL) procedure, [41][42] which is a complete, systematic search process for finding a satisfying assignment for the given Boolean formula. The DPLL procedure performs backtrack in the space of partial truth assignments and efficiently prunes the search space. Most modern SAT solvers are Conflict-Driven Clause Learning (CDCL) solvers. The organization of CDCL solvers is primarily inspired by the GRASP SAT procedure. Since their inception in the mid-90s, CDCL SAT solvers have been applied, in many cases with remarkable success, to a number of practical applications including hardware verification and ATPG. The concept and organization of CDCL SAT solvers are discussed in Section 2.2.4.

### 2.2.3 Conjunctive Normal Form (CNF)

The inputs to the SAT solvers are in the format of Conjunctive Normal Form (CNF) or more colloquially, a “product of sum” expression. Some examples of CNF clauses are shown in Figure 11. A Boolean formula F in CNF format is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals, where each literal is either a Boolean variable or its negation.

To use SAT solvers in the applications of electronic circuit design and verification, the circuit structure must be translated to a Boolean formula in CNF format.

The basic idea of translating combinational gates into CNF was proposed in [44]. In Section 2.5, I will show how to generate CNF clauses for sequential elements in the circuit.

$$\overline{A} \cdot (B + C)$$
$$(A + B) \cdot (\overline{B} + C + \overline{D}) \cdot (D + \overline{E})$$
$$A + B$$
$$A \cdot B$$

**Figure 11. Examples of CNF clauses**

2.2.4 Conflict Driven Clause Learning (CDCL)

Besides using DPLL, CDCL SAT solvers implement a number of additional key techniques: [43]

- Learning new clauses from conflicts during backtrack search
- Exploiting structure of conflicts during clause learning
- Using lazy data structures for the representation of formulas
- Branching heuristics that have low computational overhead and receive feedback from backtrack search
- Periodically restarting backtrack search
- Additional techniques include deletion policies for learnt clauses, the actual implementation of lazy data structures, and the organization of unit clause (implication) propagation.

CDCL SAT solvers dynamically generate learnt clauses based on conflicts detected in searching a satisfying solution. For example, if a CDCL SAT solver made the assignments: A=0, B=0, C=0 when trying to solve a particular problem and this resulted in a conflict, the solver would construct a conflict graph, and, from this graph, it will dynamically generate a learnt clause which would ensure that this conflict would not recur. If the conflict graph only had variables A=0 and C=0 involved in the conflict, one possible format of this learnt clause is A+C. This would ensure that A=0 and C=0 would not be tried again in the SAT solving procedure.

The most common lazy data structure is the watch list. [43] In CDCL SAT solvers, each literal has a watch list which contains pointers to CNF clauses in which this literal is watched. Each clause has 2 watched literals (selected at random). By definition, both watched literals in a clause are unassigned. Whenever the value of one literal is asserted, the CNF clauses on its watch list are evaluated. If there is another unwatched, unassigned literal in these clauses, one of these unwatched unassigned literals is now watched instead of the asserted literal. If there is no remaining unwatched unassigned literal in the clause, then an implication is generated on the other watched literal of the clause being evaluated. The purpose of using watch lists in CDCL SAT solvers is to enable the solver to very rapidly locate the implications that are generated as a consequence of a variable assignment, and avoid useless evaluations of all the CNF in order to find implications.

CDCL SAT solvers use Boolean Constraint Propagation (BCP) [36] to identify the assignments of Boolean variables and propagate the assigned value. This method is based on the observation that if a n-literal clause consists of n-1 literals whose assigned value is 0, and one unassigned literal, then that unassigned literal must take on a value of

1 to make the CNF satisfiable. This can be illustrated with the example shown in Figure 12. This example shows the evaluation process of CNF clause A+B+C. At the beginning, this clause is added to the watch lists of A and B. If A is assigned to 0, this clause is evaluated. Since there are two unassigned literals in the clause, their assignments cannot be decided yet. Then this clause is removed from the watch list of A and added to the watch list of C. Later when the value of C gets assigned, this clause is evaluated again. If C=1, this clause has been satisfied. If C=0, according to the rule of BCP, B is assigned to the value of 1 such that the CNF can be satisfiable and the value of this assignment gets propagated by evaluating those clauses on the watch list of B.

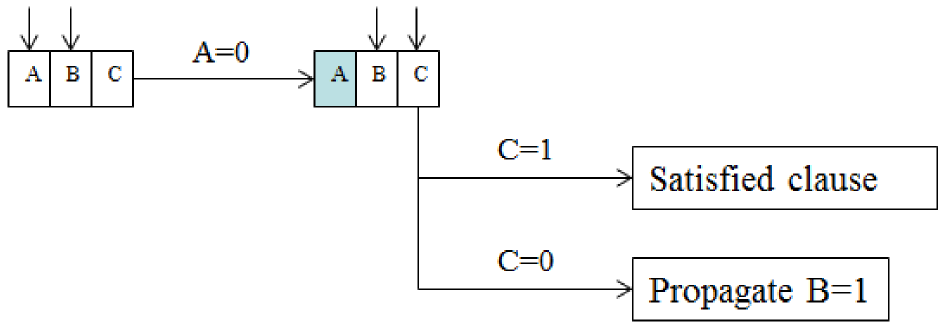


**Figure 12. Example of Boolean constraint propagation**

## 2.3 Previous Work on SAT Based ATPG

Previously SAT based ATPG has been successfully applied to stuck-at and transition delay fault testing [39][44]. In recent years, as the performance of SAT solvers has continued to improve [36][37], this approach has become increasingly practical for test generation.

There has been very limited previous work on SAT-based ATPG for path delay testing. [38][45][46] The fundamental challenge in applying SAT solvers to path delay test generation is in encoding path delay with Boolean variables. In [45] the path delay uses a unipolar encoding. This approach works well for unit gate delays, but the number of variables grows exponentially with delay resolution. For example, a precision of three digits requires 1000 Boolean variables for each logic gate. Binary encoding can be used to reduce the number of Boolean variables used for gate delays, but this approach introduces complex CNF clauses in describing the arithmetic operations of delay variables.

## 2.4 Mixed Structural-Functional Approach

To cope with the difficulty of encoding delay values in SAT-based path delay test, we propose a mixed structural-functional approach for path delay test generation. The entire process of path delay test generation contains two stages: the first stage is path searching, which is implemented using the KLPG algorithm. As discussed in Section 1, the KLPG algorithm generates K longest paths for each gate in the circuit. Since the KLPG algorithm is based on structural information, the delay encoding problem is avoided and real-valued delays can be used. In the second stage, the generated paths are justified and then compacted into patterns using a SAT solver.

By separating path delay test generation into two stages, we can fully exploit the advantages of both the KLPG algorithm and the SAT solver. With the SAT solver, we can generate path delay tests more efficiently and accurately. Using the KLPG algorithm for path search saves the trouble of encoding delays. The new *CodGen* flow chart after implementing mixed structural-functional method is shown in Figure 13. In this new flow chart, the shaded region, which consists of path justification and dynamic compaction, is the portion of the new *CodGen* implemented with the SAT solver.

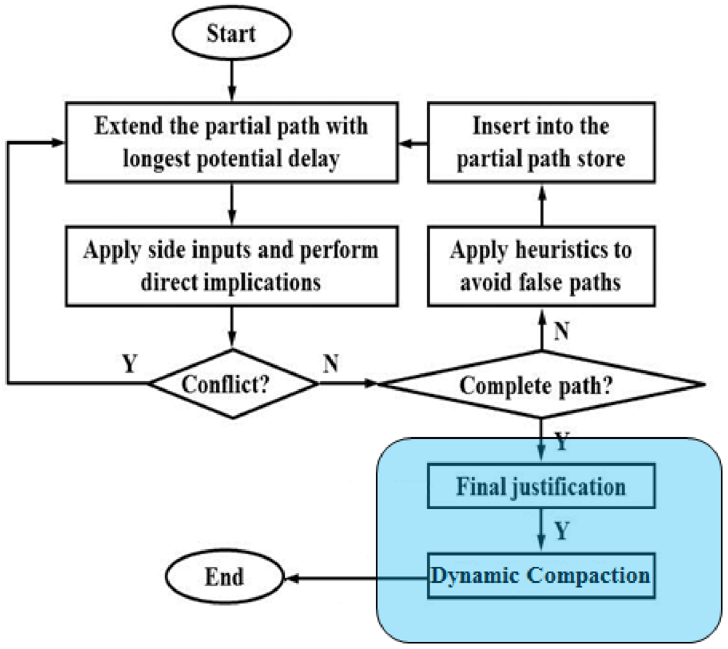

Start
Extend the partial path with longest potential delay
Insert into the partial path store
Apply side inputs and perform direct implications
Apply heuristics to avoid false paths
Y
Conflict?
N
N
Complete path?
Y
Final justification
Y
End
Dynamic Compaction

**Figure 13. Mixed structural-functional approach**

## 2.5 *MiniSat* Integration

### 2.5.1 *MiniSat*

The SAT solver used in this research is *MiniSat*. [37] *MiniSat* is a minimalistic, open-source SAT solver, developed to help researchers and developers alike to get started on SAT. [47] It is one of the most well developed SAT solvers at this time and was a winner of the international SAT competition. [48] The key features of *MiniSat* include:

- Easy to modify: *MiniSat* is small and well-documented, making it an ideal starting point for developers to adapt SAT based techniques
- Highly efficient: As a previous winner of the international SAT competition, *MiniSat* is well maintained to reflect the latest developments in SAT solvers.
- Designed for integration: *MiniSat* supports incremental SAT and has mechanisms for adding non-clausal constraints. It is also extensible.

Developers from multiple research areas have been using *MiniSat* to solve Boolean satisfiability (SAT) problems originating from diverse disciplines. There have been reports of using *MiniSat* in SAT-based ATPG to generate test patterns for structural tests [38][49]. It is reported that this approach can significantly improve the fault coverage and runtime efficiency of the ATPG process, especially for large designs from industry. SAT-based ATPG also provides a robust solution for some faults which are hard to test with traditional ATPG techniques.

### 2.5.2 CNF Generation

Boolean formulas written in Conjunctive Normal Form (CNF) format can be easily manipulated programmatically and modern SAT solvers, including *MiniSat*, require CNF as the input format. The basic idea of translating logic circuits into CNF is described in [44]. The idea is based on the fact that in a logic gate, the output variable and its logic function are implications of each other. The advantage of this approach is that the CNF can be generated locally, without considering any other gates in the circuit. The disadvantage is that the generated CNF is not always efficient and may contain many redundancies. The CNF of most commonly-used Boolean gates are summarized in

Table 1. In the table, X and Y are input signals of the gate and Z is the output. Following the idea described in [44], the CNF of gates with arbitrary truth table can also be generated. Over the years, there have been several efforts in improving the efficiency of CNF generation with chained ITE [50] or BDDs [51].

**Table 1. CNF of Common Boolean Gates**

| Type | CNF |
|---|---|
| BUF | $(Z+\bar{X})(\bar{Z}+X)$ |
| NOT | $(Z+X)(\bar{Z}+\bar{X})$ |
| AND | $(\bar{Z}+X)(\bar{Z}+Y)(Z+\bar{X}+\bar{Y})$ |
| NAND | $(Z+X)(Z+Y)(\bar{Z}+\bar{X}+\bar{Y})$ |
| OR | $(Z+\bar{X})(Z+\bar{Y})(\bar{Z}+X+Y)$ |
| NOR | $(\bar{Z}+\bar{X})(\bar{Z}+\bar{Y})(Z+X+Y)$ |

In delay test, we must generate test vectors for multiple time frames. For example, there are two test vectors in either Launch-On-Capture (LOC) or Launch-On-Shift (LOS) scan test. Pseudo-Functional Test (PFT) requires test vectors for more than two preamble time frames. In traditional delay test generation based on PODEM, we have to perform time frame expansion such that the algorithm can handle multiple time frames. To use CNF in delay test generation, we must find an efficient way to generate the CNF for multiple time frames. The most straightforward way to do this is to create separate Boolean variables for any logic signal in every time frame. For example, if there is a logic signal called X in the circuit, we create the Boolean variable $X^0$, $X^1$, ..., $X^{n-1}$ to represent the values of X in the time frames 0, 1, …, n-1, where n = 2 for either

LOC or LOS test and n > 2 for PFT. Following Table 1, the CNF of combinational gates can be generated using the Boolean variables with time frame annotation.

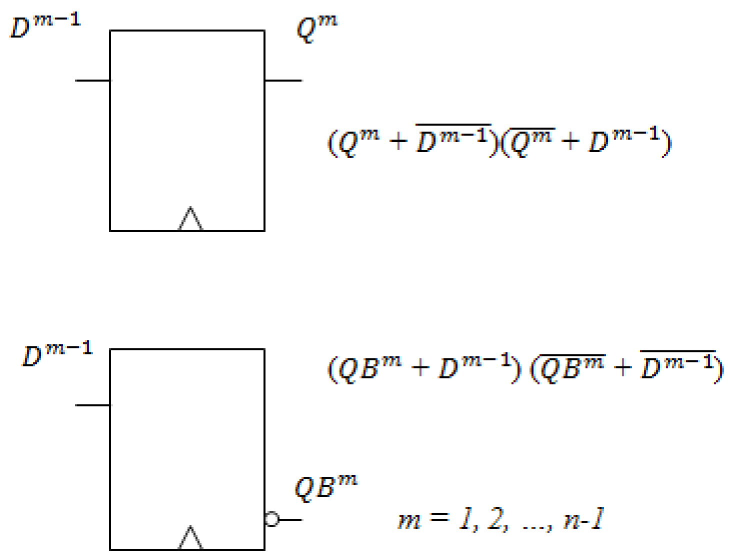

$D^{m-1}$
$Q^m$
$(Q^m + \overline{D^{m-1}})(\overline{Q^m} + D^{m-1})$
$D^{m-1}$
$(QB^m + D^{m-1})\,(\overline{QB^m} + \overline{D^{m-1}})$
$QB^m$
$m = 1, 2, \ldots, n\text{-}1$

**Figure 14. CNF for DFF (non-inverting and inverting)**

Special cautions must be taken when generating CNF for sequential elements in the circuit, since their input and output signals are Boolean variables in different time frames. Figure 14 shows both non-inverting and inverting D Flip-flops (DFFs) and their corresponding CNFs. The CNFs of inverting and non-inverting DFFs are in the same format as the CNFs of inverters and buffers except that the output Boolean variable is one time frame later than the input variable.

Another special case in CNF generation is in handling fixed Primary Inputs (PIs). Because of the limited number of high-speed pins on low-cost testers, the PIs of a chip are fixed during the application of a test pattern. In this case, for each PI, there is only one Boolean variable used for all time frames. Due to the same low-cost tester constraints, the Primary Outputs (POs) of the circuit are ignored.

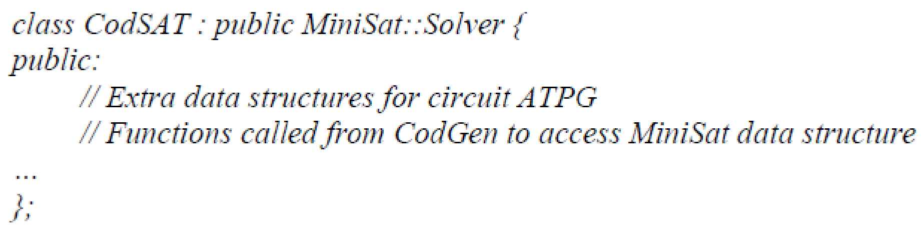
Since the Circuit-Under-Test (CUT) does not change during the process of delay test generation, we can create one CNF describing the entire CUT at the beginning of test pattern generation following the approach described above, and use it over and over again during the entire test generation process.

### 2.5.3 *MiniSat/CodGen* Interface

There are two options in implementing the interface between *MiniSat* and *CodGen*. Each of them has its advantages and disadvantages:

1. Private Inheritance/Inclusion: Better protection for the internal data structures of *MiniSat*. Less efficient since only the public interface of the SAT solver instance can be accessed.
2. Public Inheritance/Extension: *MiniSat* is designed to be extensible. The advantage of this option is that the code can be very efficient by directly accessing the internal data structures of *MiniSat*. But this option adds the danger of corrupting the internal data of *MiniSat*.

In this research, originally the *MiniSat/CodGen* interface was implemented using option 1 for its simplicity and security. Later for the purpose of improving the performance of SAT based test generation, the interface is modified to option 2. The code to implement this interface is illustrated in Figure 15.



```
class CodSAT : public MiniSat::Solver {
public:
        // Extra data structures for circuit ATPG
        // Functions called from CodGen to access MiniSat data structure
...
};
```

**Figure 15.** ***MiniSat/CodGen*** **interface**

Since we are using *MiniSat*, which is a general purpose SAT solver, in the particular application of delay test, we can define those data structures necessary to represent the circuit under test in this wrapper class. By calling the public functions defined in this wrapper class, *CodGen* will have the ability to access the internal data structures of *MiniSat* through this wrapper class. It is believed that this new approach of implementing the *CodGen-MiniSat* interface will greatly facilitate the future effort of improving SAT performance with the knowledge of circuit structure.

## 2.6 Benchmark Platforms

Two Windows servers are used in this research for the purpose of running benchmarking test cases and they are described as follows:

- Server 1 (S1): A Windows 7 PC with dual AMD Opteron Processors 252 (2.59 GHz) and 16.0 GB main memory
- Server 2 (S2): A Windows 7 PC with dual Intel E5-2603 Processors (4 core, 1.80 GHz) and 64.0 GB main memory

In this dissertation, unless specially mentioned, results presented in Section 2 are generated on Server 1 and results presented in Section 3 are generated on Server 2.

## 2.7 Experimental Results

Experiments are conducted to test the performance of *CodGen* with *MiniSat* and the results are compared with those of the original *CodGen* using PODEM. All experiments are performed for K=1 and K=5 where K is the number of rising and falling paths to be tested through each line in the circuit, using robust path constraints. Besides Launch-On-Capture (LOC) test, where number of time frames n=2, we also did Pseudo Functional Test (PFT) with number of time frames n=4, 6, 8, 10. Fixed primary inputs (PIs) are used in all the experiments. In the case of using PODEM, a backtrack limit of 200 is used for the runs.

**Table 2. Comparison of number of paths found with PODEM and SAT (K=5)**

| Circuit | n=2 | | n=4 | | n=6 | | n=8 | | n=10 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | PODEM | SAT | PODEM | SAT | PODEM | SAT | PODEM | SAT | PODEM | SAT |
| s1423 | 1323 | 1419 | 488 | 1088 | 199 | 1080 | 137 | 1080 | 135 | 1080 |
| s1488 | 189 | 261 | 121 | 261 | 113 | 261 | 104 | 261 | 95 | 261 |
| s1494 | 202 | 285 | 133 | 285 | 115 | 285 | 108 | 285 | 95 | 285 |
| s5378 | 5214 | 5226 | 1731 | 2753 | 1052 | 2310 | 826 | 2268 | 698 | 2257 |
| s9234 | 6605 | 7043 | 3342 | 4828 | 905 | 4516 | 779 | 4516 | 852 | 4516 |
| s13207 | 6886 | 6945 | 2513 | 3286 | 1042 | 2050 | 786 | 1349 | 592 | 1027 |
| s15850 | 6618 | 7073 | 4284 | 5548 | 3241 | 5027 | 3147 | 4991 | 3101 | 4991 |
| s35932 | 22274 | 22418 | 21890 | 22034 | 21506 | 21906 | 21314 | 21906 | 20988 | 21906 |
| s38417 | 50172 | 50949 | 42085 | 48417 | 30885 | 41266 | 23255 | 40204 | 17825 | 39750 |
| s38584 | 17314 | 18518 | 11031 | 15231 | 7072 | 14227 | 6428 | 13961 | 4590 | 13728 |
| **Average** | **1.00** | **1.11** | **1.00** | **1.57** | **1.00** | **2.53** | **1.00** | **2.98** | **1.00** | **3.19** |

Compared with PODEM, using the SAT solver in *CodGen* detects more sensitizable paths, achieves a higher compaction ratio and runs much faster. The number of paths found by *CodGen* in both scenarios are summarized in Table 2 for the case of

K=5. For n=2, on average *MiniSat* detects 11% more paths than PODEM. Also, the advantage of using *MiniSat* increases with increasing number of time frames.

**Table 3. Comparison of compaction ratios of PODEM and SAT (K=5)**

| Circuit | n=2 | | n=4 | | n=6 | | n=8 | | n=10 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | PODEM | SAT | PODEM | SAT | PODEM | SAT | PODEM | SAT | PODEM | SAT |
| s1423 | 4.32 | 4.64 | 2.82 | 4.17 | 2.73 | 4.14 | 2.74 | 4.15 | 3.29 | 4.17 |
| s1488 | 2.91 | 3.53 | 2.69 | 3.53 | 2.46 | 3.58 | 2.36 | 3.58 | 2.26 | 3.58 |
| s1494 | 3.11 | 3.61 | 2.66 | 3.61 | 2.40 | 3.61 | 2.35 | 3.61 | 2.16 | 3.56 |
| s5378 | 10.41 | 11.21 | 6.58 | 12.57 | 8.62 | 13.59 | 6.83 | 13.42 | 6.02 | 13.43 |
| s9234 | 8.70 | 9.40 | 8.38 | 12.10 | 9.84 | 12.41 | 10.12 | 12.31 | 12.17 | 12.48 |
| s13207 | 3.33 | 3.52 | 20.43 | 19.44 | 8.20 | 14.96 | 11.39 | 29.98 | 8.71 | 21.85 |
| s15850 | 6.18 | 6.64 | 4.58 | 5.27 | 3.90 | 6.01 | 3.69 | 6.09 | 3.96 | 6.09 |
| s35932 | 718.52 | 679.33 | 533.90 | 688.56 | 130.34 | 625.89 | 68.53 | 625.89 | 72.62 | 644.29 |
| s38417 | 50.22 | 56.74 | 34.50 | 50.07 | 26.02 | 40.14 | 18.85 | 33.50 | 14.31 | 29.62 |
| s38584 | 41.03 | 47.48 | 32.64 | 56.83 | 27.41 | 51.73 | 26.24 | 50.95 | 29.24 | 50.66 |
| **Average** | **1.00** | **1.10** | **1.00** | **1.41** | **1.00** | **1.89** | **1.00** | **2.49** | **1.00** | **2.45** |

.

**Table 4. Comparison of runtime performance of PODEM and SAT (K=5)**

| Circuit | n=2 | | n=4 | | n=6 | | n=8 | | n=10 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | PODEM | SAT | PODEM | SAT | PODEM | SAT | PODEM | SAT | PODEM | SAT |
| s5378 | 1:12 | 0:23 | 1:17 | 0:19 | 1:23 | 0:21 | 2:14 | 0:28 | 2:20 | 0:35 |
| s9234 | 4:36 | 2:50 | 6:07 | 2:04 | 4:07 | 2:13 | 7:49 | 2:56 | 13:37 | 3:38 |
| s13207 | 5:52 | 4:28 | 2:39 | 1:35 | 2:28 | 1:02 | 1:34 | 0:36 | 1:53 | 0:32 |
| s15850 | 4:54 | 3:28 | 12:16 | 4:02 | 18:51 | 5:12 | 29:53 | 7:03 | 38:28 | 9:06 |
| s35932 | 23:58 | 20:53 | 40:24 | 25:56 | 2:04:14 | 40:44 | 4:02:59 | 54:37 | 5:12:16 | 1:12:29 |
| s38417 | 4:26:02 | 32:49 | 9:54:13 | 1:19:10 | 11:00:21 | 2:08:14 | 13:42:00 | 3:15:21 | 16:42:38 | 4:38:01 |
| s38584 | 28:49 | 18:18 | 51:07 | 25:55 | 1:06:54 | 32:29 | 1:20:17 | 43:42 | 1:13:05 | 57:40 |
| **Average** | **1.00** | **0.38** | **1.00** | **0.31** | **1.00** | **0.32** | **1.00** | **0.28** | **1.00** | **0.28** |

Table 3 compares the compaction ratios of *CodGen* with PODEM and *MiniSat* for the case of K=5. For n=2, on average with *MiniSat* the compaction ratios are improved by 10%. The improvement increases with increasing number of time frames.

The total *CodGen* runtimes with PODEM and *MiniSat* are compared in Table 4 for the case of K=5. With *MiniSat*, on average *CodGen* is sped up by 60~70% compared with PODEM.

# 3. TECHNIQUES TO IMPROVE SAT EFFICIENCY

## 3.1 Dynamic SAT Solving

Dynamic SAT Solving (DSS) is based on the observation that to test the path delay faults through a particular target line (fault site), only those gates within the transitive fan-in of the fan-out cone of the target line are involved. This region is shown in Figure 16, as the two shaded areas. For most target lines, this region (called the fault region henceforth) is a small fraction of the entire circuit. This is particularly true for large industrial circuits.

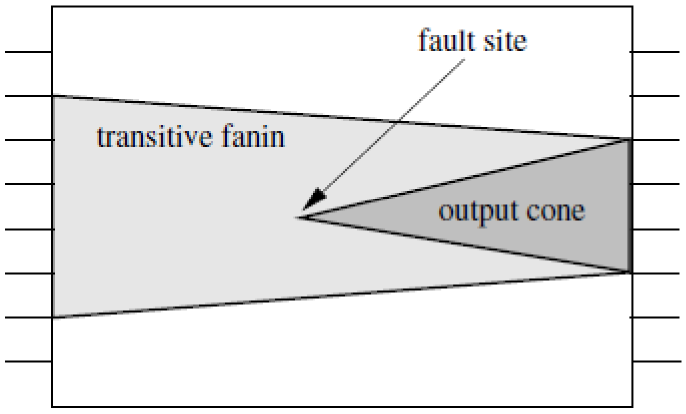


**Figure 16. Transitive fan-in cone of targeted fault site**

To reduce SAT solution time, the internal data structure of *MiniSat* has been modified such that CNF clauses can be dynamically turned on (for those clauses in the fault region) and off (for those clauses of the unshaded region). *MiniSat* uses optimized Boolean Constraint Propagation (BCP) [36] to evaluate CNF clauses and assign values to Boolean variables. The basic idea of optimized BCP is to "watch" no more than two literals of each CNF clause at any time. Whenever there is only one undecided literal left in the CNF clause, *MiniSat* assigns a value to the corresponding undecided Boolean variable and propagates the newly assigned value to other clauses as an implication. To dynamically disable the evaluation of a particular CNF clause, this clause has to be removed from all the watch lists. This is the key observation used in DSS. Adding the CNF clause back to the watch lists of its literals will enable the evaluation of this clause. With this technique, as delay tests are generated for a particular line, first the fault region is calculated by breadth-first search. Only those gates inside the resulting region are enabled for SAT solving. We call this new technique Dynamic SAT Solving (DSS). A similar approach of speeding up *MiniSat* based ATPG has been reported in [52].

**Table 5. Results of dynamic SAT solving (DSS) (n=2, K=5)**

| Circuit | CPU Time | | Path Count | | Pattern Count | | Compaction | |
|---|---|---|---|---|---|---|---|---|
| | Original | DSS | Original | DSS | Original | DSS | Original | DSS |
| s5378 | 0:00:53 | 0:00:27 | 5225 | 5226 | 493 | 466 | 10.60 | 11.21 |
| s9234 | 0:04:18 | 0:03:18 | 6869 | 7043 | 785 | 749 | 8.75 | 9.40 |
| s13207 | 0:06:01 | 0:05:14 | 6911 | 6945 | 2068 | 1971 | 3.34 | 3.52 |
| s15850 | 0:06:28 | 0:04:52 | 7054 | 7073 | 1068 | 1065 | 6.60 | 6.64 |
| s35932 | 0:51:41 | 0:26:41 | 22418 | 22418 | 34 | 33 | 659.35 | 679.33 |
| s38417 | 1:30:21 | 0:52:45 | 50618 | 50949 | 952 | 898 | 53.17 | 56.74 |
| s38584 | 0:34:53 | 0:24:57 | 17557 | 18518 | 398 | 390 | 44.11 | 47.48 |
| b14 | 5:45:56 | 5:26:17 | 52156 | 52747 | 27683 | 26500 | 1.88 | 1.99 |
| b15 | 7:12:17 | 9:46:49 | 22345 | 33065 | 7797 | 11288 | 2.87 | 2.93 |
| b17 | 48:54:43 | 48:22:06 | 91055 | 117737 | 10739 | 10251 | 8.48 | 11.49 |
| b20 | 8:09:00 | 7:37:49 | 109709 | 110140 | 31231 | 24064 | 3.51 | 4.58 |
| b21 | 9:39:33 | 10:58:07 | 111476 | 111966 | 31132 | 23806 | 3.58 | 4.70 |
| b22 | 38:38:51 | 38:41:16 | 162019 | 163400 | 37206 | 26825 | 4.35 | 6.09 |
| STC | 7:25:54 | 4:00:56 | 67861 | 70872 | 2762 | 2558 | 24.57 | 27.71 |
| tex1 | 60:00:21 | 37:36:47 | 158521 | 158750 | 2258 | 1802 | 70.20 | 88.10 |
| **Average** | **1.000** | **0.817** | **1.000** | **1.063** | **1.000** | **0.939** | **1.000** | **1.147** |

The effectiveness of DSS has been proven with ISCAS89 and ITC99 benchmarks. ITC99 benchmarks are larger than ISCAS89 benchmarks. The results for n=2, K=5 are shown in Table 5. As expected, DSS significantly reduces CPU time. Speedups of up to 2 times are observed. In most cases, the compaction ratios are significantly improved due to the smaller problem sizes to be solved. On average, DSS is going to reduce *CodGen* runtime by 18.3% and improve compaction ratio by 14.7%.

## 3.2 Circuit Observability Don't Cares

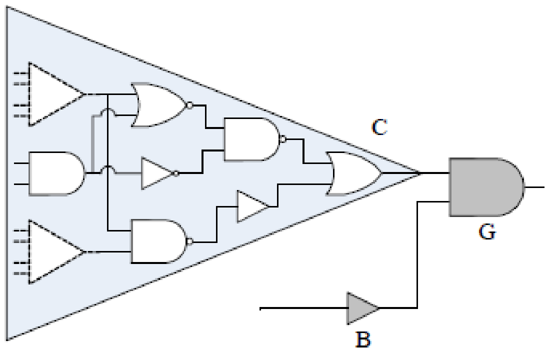


**Figure 17. Example of Cir-ODC**

Another approach to speed up SAT performance is to use Circuit Observability Don't Cares (Cir-ODCs). The definition of Observability Don't Care (ODC) is as follows: under certain logic conditions C, if one signal S in the design does not affect any output of the design, then C is the ODC of S. An example of Cir-ODCs is shown in Figure 17. In this example, B=0 is the Cir-ODC of those gates inside the fan-in cone of C.

In [53], the authors described an algorithm to quickly calculate Cir-ODCs. This algorithm has been implemented in *CodGen* to evaluate the effectiveness of using Cir-ODCs in path delay test generation. Cir-ODCs can be used in path justification and dynamic compaction based on the observation that no matter how many time frames there are in the delay test, all the gates along the path have to be observable in the last frame, in order to propagate the transition along the path, therefore their Cir-ODCs must be false. One advantage of the algorithm in [53] is that the Cir-ODCs calculated are in the form of sum of single literals, so they can be directly used as assumptions (restrictions on Boolean variable values) when SAT is invoked.

**Table 6. Results of delay test generation with Cir-ODC (*n*=*2*, *K*=*5*)**

| Circuit | CPU Time | | Path Count | | Pattern Count | | Compaction Ratio | |
|---|---|---|---|---|---|---|---|---|
| | Original | Cir-ODC | Original | Cir-ODC | Original | Cir-ODC | Original | Cir-ODC |
| s5378 | 0:00:53 | 0:00:48 | 5225 | 5225 | 493 | 493 | 10.60 | 10.60 |
| s9234 | 0:04:18 | 0:03:55 | 6869 | 6869 | 785 | 785 | 8.75 | 8.73 |
| s13207 | 0:06:01 | 0:06:15 | 6911 | 6911 | 2068 | 2068 | 3.34 | 3.34 |
| s15850 | 0:06:28 | 0:06:32 | 7054 | 7054 | 1068 | 1067 | 6.60 | 6.55 |
| s35932 | 0:51:41 | 0:48:25 | 22418 | 22418 | 34 | 34 | 659.35 | 659.35 |
| s38417 | 1:30:21 | 1:27:31 | 50618 | 50618 | 952 | 952 | 53.17 | 53.17 |
| s38584 | 0:34:53 | 0:36:18 | 17557 | 17557 | 398 | 405 | 44.11 | 42.82 |
| b14 | 5:45:56 | 6:01:16 | 52156 | 52156 | 27683 | 27683 | 1.88 | 1.88 |
| b15 | 7:12:17 | 4:53:47 | 22345 | 22345 | 7797 | 7797 | 2.87 | 2.87 |
| b17 | 48:54:43 | 51:20:23 | 91055 | 91055 | 10739 | 10722 | 8.48 | 8.49 |
| b20 | 8:09:00 | 7:33:24 | 109709 | 109709 | 31231 | 31205 | 3.51 | 3.51 |
| b21 | 9:39:33 | 9:43:26 | 111476 | 111476 | 31132 | 31111 | 3.58 | 3.58 |
| b22 | 38:38:51 | 43:18:08 | 162019 | 162019 | 37206 | 37216 | 4.35 | 4.35 |
| STC | 7:25:54 | 7:58:19 | 67861 | 67842 | 2762 | 2763 | 24.57 | 24.56 |
| tex1 | 60:00:21 | 59:12:22 | 158521 | 157916 | 2258 | 2258 | 70.20 | 70.20 |
| **Average** | **1.000** | **0.946** | **1.000** | **1.000** | **1.000** | **1.001** | **1.000** | **0.997** |

The effects of reducing total runtime with Cir-ODCs is demonstrated in Table 6 for the case of n=2, K=5. Both ISCAS89 and ITC99 benchmarks along with a couple of industry designs are also used to measure the effects of Cir-ODCs. It is shown that in some ITC99 test cases such as b15, Cir-ODCs can reduce the total SAT time by up to 30%.

## 3.3 SAT Based Static Learning

Learnt clauses generated with static learning based on circuit structure can be used to guide the operation of the SAT solver. *MiniSat* itself can be used to implement efficient recursive learning [54]. Assume there are two nodes *A* and *B* in the circuit and their corresponding values cannot be decided with direct implication. Then we can set the values of *A* and *B* and use the values as the assumption in calling *MiniSat*. Learnt clause can be generated if a certain combination of the assigned values is determined to be invalid. For example, as shown in Figure 18, if we set A=0 and B=1, after calling *MiniSat*, it is determined that this is an invalid state of the circuit and the corresponding learnt clause can be generated accordingly. In this particular example, the generated learnt clause has the form $A+\bar{B}$.

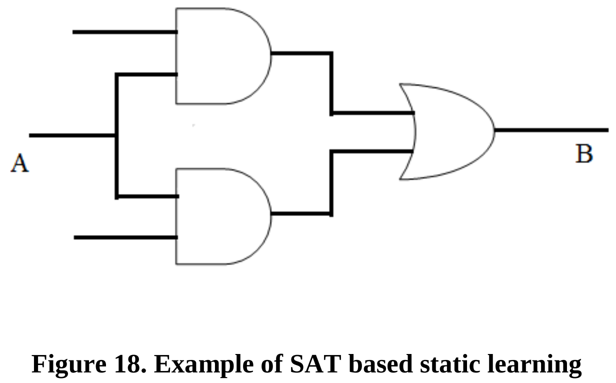


**Figure 18. Example of SAT based static learning**

SAT based recursive learning is performed at the beginning of the *CodGen* run and the generated learnt clauses are added to the CNF instance to guide the SAT solver in justification and dynamic compaction. Since these learnt clauses will prune the search space of the SAT solver, it is expected that the SAT solver will run more efficiently.

**Table 7. Results of SAT based static learning (n=2, K=5)**

| Circuit | CPU Time | | Path Count | | Pattern Count | | Compaction Ratio | |
|---|---|---|---|---|---|---|---|---|
| | Original | D=2 | Original | D=2 | Original | D=2 | Original | D=2 |
| s5378 | 0:00:53 | 0:00:52 | 5225 | 5225 | 493 | 493 | 10.60 | 10.60 |
| s9234 | 0:04:18 | 0:04:21 | 6869 | 6869 | 785 | 785 | 8.75 | 8.75 |
| s13207 | 0:06:01 | 0:06:08 | 6911 | 6911 | 2068 | 2068 | 3.34 | 3.34 |
| s15850 | 0:06:28 | 0:06:18 | 7054 | 7054 | 1068 | 1068 | 6.60 | 6.60 |
| s35932 | 0:51:41 | 0:37:24 | 22418 | 22418 | 34 | 34 | 659.35 | 659.35 |
| s38417 | 1:30:21 | 1:32:26 | 50618 | 50618 | 952 | 952 | 53.17 | 53.17 |
| s38584 | 0:34:53 | 0:35:28 | 17557 | 17557 | 398 | 398 | 44.11 | 44.11 |
| b14 | 5:45:56 | 6:10:47 | 52156 | 52156 | 27683 | 27683 | 1.88 | 1.88 |
| b15 | 7:12:17 | 6:48:35 | 22345 | 22349 | 7797 | 7797 | 2.87 | 2.87 |
| b17 | 48:54:43 | 48:37:28 | 91055 | 91055 | 10739 | 10739 | 8.48 | 8.48 |
| b20 | 8:09:00 | 7:57:46 | 109709 | 109709 | 31231 | 31231 | 3.51 | 3.51 |
| b21 | 9:39:33 | 9:38:07 | 111476 | 111476 | 31132 | 31132 | 3.58 | 3.58 |
| b22 | 38:38:51 | 38:56:09 | 162019 | 162019 | 37206 | 37206 | 4.35 | 4.35 |
| STC | 7:25:54 | 7:31:56 | 67861 | 67861 | 2762 | 2762 | 24.57 | 24.57 |
| tex1 | 60:00:21 | 58:10:12 | 158521 | 158521 | 2258 | 2258 | 70.20 | 70.20 |
| **Average** | **1.000** | **0.982** | **1.000** | **1.000** | **1.000** | **1.000** | **1.000** | **1.000** |

The results of running delay test generation with SAT based static learning are summarized in Table 7. In this table, the results of running the benchmarks with two level SAT based static learning (D=2) are compared with the results without any static learning. Here the depth of static learning D is defined as the number of multiple input combinational gates passed through when backtracking from the targeted node in the circuit. It can be shown that the average runtime of path delay test generation can be reduced with SAT based static learning, while the average number of test patterns and average compaction ratio remain almost the same.

## 3.4 Dynamic Learnt Clause Management

*MiniSat* generates learnt clauses during the SAT solving process. Several strategies to dynamically manage learnt clauses for better performance have been explored [55]. Generally speaking, *MiniSat* generates many learnt clauses. This increases memory usage and may slow down its operation. Therefore the dynamically generated learnt clauses have to be purged regularly. The difference between learnt clause management strategies is the criteria to decide which clauses to purge. These criteria are based on the activity factors of the learnt clauses (which reflect whether a particular learnt clause has been evaluated recently) or simply based on the length of the learnt clause, or both.

By default, *MiniSat* manages its learnt clauses based on both the activity factors and lengths of those learnt clauses. Whenever the learnt clause management function is triggered inside *MiniSat*, it reduces the number of learnt clauses based on the following criteria:

- Always keep learnt clauses of length are less than 3
- Reduce other learnt clauses based on their activity factor until the total number of dynamic generated learnt clauses drops to half of the original value before the management function is called.

Other dynamic learnt clause management policies based on the length or activity factor of the learnt clauses are implemented in *CodGen* for the purpose of determining the best learnt clause management strategy. The results of running ISCAS89 and ITC 99 benchmarks and industrial circuits with different learnt clause management policies are summarized in Table 8. In the experiment, learnt clause management function is call whenever the total number of learnt clauses reaches the 1% of the size of the CNF being solved. This threshold is intentionally set to be low such that the function can be called in all the benchmarks and therefore the effects of different learnt clause management policies can be compared. The five policies compared in Table 8 are as follows:

- Policy 1: Default *MiniSat* learnt clause management policy described above
- Policy 2: This policy is only based on activity factor of the learnt clauses and each time the function is called, only ¼ of the most used learnt clauses are kept and the other ¾ are removed.
- Policy 3: This policy is only based on the length of the learnt clauses. Each time the function is called, only those learnt clauses whose length is less than or equal to 3 are kept.
- Policy 4: In this policy, the nodes in the circuit are levelized before the run. For each generated learnt clause, calculate the average span parameter which is defined as ($maxLevel - minLevel + 1$) / $size$, where *maxLevel* and *minLevel* are the maximum and minimum levels of all the literals inside the learnt clause and *size* is the number of literals in the clause. Sort all the learnt clauses based on this parameter and remove half of the learnt clauses with smaller values of average span parameter.

- Policy 5: This policy is quite similar with Policy 4, but instead of using average span, Policy 5 uses absolute span which is defined as (*masLevel* – *minLevel* + 1).

**Table 8. Results of dynamic learnt clause management (n=6, K=5)**

| Circuit | SAT Time (s) | | | | |
|---|---|---|---|---|---|
| | Policy 1 | Policy 2 | Policy 3 | Policy 4 | Policy 5 |
| s5378 | 49.6 | 56 | 55.2 | 48.4 | 55.7 |
| s9234 | 284.5 | 306.5 | 330.7 | 283.2 | 594.3 |
| s13207 | 197.4 | 204 | 209.2 | 177.8 | 489.3 |
| s15850 | 533.6 | 578 | 572.8 | 534.9 | 747.5 |
| s35932 | 8708.2 | 8626.1 | 8442.23 | 8239.2 | 94643.3 |
| s38417 | 13666.9 | 14999.3 | 14541.3 | 13300.2 | 278444 |
| s38584 | 3623.9 | 3774.7 | 3831.1 | 3309.4 | 6199.1 |
| **Average** | **1.000** | **1.065** | **1.071** | **0.958** | **5.721** |

It can be concluded from Table 8 that the effect of each of the learnt clause management policies varies from one circuit to the other. In some benchmarks such as s9234 and s35932, Policies 2 and 3 run better than the default learnt clause management policy of *MiniSat,* while in the other cases, *CodGen* gets better results with the default policy. It is believe that *MiniSat* developers must have optimized their default dynamic learnt clause management policy such that the performance can't be easily improved by simply modifying some parameters as done in Policy 2 and 3. As shown in Policy 4, the runtime performance indeed can be improved by introducing the knowledge of circuit structure, but this has to be done in the correct way. As shown in Policy 5, if there is something wrong with the dynamic learnt clause management policy, the runtime performance can be significantly degraded. The problem with Policy 5 is that this policy is going to favor longer learnt clauses and at the end make the SAT solving process more and more complex.

Mathematically dynamic learnt clause management can be represented with an optimization problem with multiple parameters such as the threshold to trigger the dynamic learnt clause management function, limits of clause length and activity factor used in reducing the learnt clauses, and other clause characteristics used in their selection. In practice, the actual effect of each management policy depends on the characteristics of the circuit. We have implemented several built-in learnt clause management policies in *CodGen* and give the users options to choose from them, so that they can tune *CodGen* to their problems.

## 3.5 Buffer Reduction

Buffer insertion has been widely used in physical design to reduce interconnect delay, adjust timing and minimize power consumption of the circuit. [56][57][58] It is estimated that greater than 70% of a VLSI IC in the 32nm technology node will comprise of buffers. [59] But too many buffers in the circuit will increase the complexity of ATPG, especially in the case of path delay test generation where the longest paths are the targets of test generation. Buffer reduction has been implemented in SAT based *CodGen* to reduce the complexity of the circuit and therefore the runtime of path delay test generation. The two scenarios of buffer reduction are illustrated in Figure 19. In the first case shown in Figure 19(a), a chain of buffers, inverters or any combination of them can be reduced to a single buffer (or inverter) depending on the polarity of the entire chain. In the second case shown in Figure 19(b), a single buffer or inverter driven by a combinational gate can be reduced into a single combinational gate. The polarity of the combinational gate has to be reversed in the case where the reduced gate drives an inverter. In both cases, the delay of the resulting gate has to be adjusted to reflect the combined delay of the original gates.

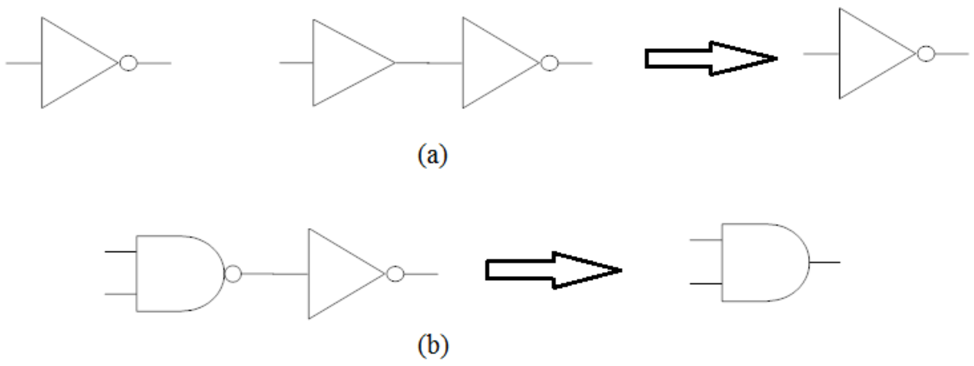


**Figure 19. Buffer reduction. (a) Reduce buf/inv chains to single buf/inv; (b) reduce combinational gate and buf/inv to single combinational gate**

**Table 9. Results of buffer reduction (n=2, K=5)**

| Circuit | CPU Time | | Path Count | | Pattern Count | | No of Clauses | |
|---|---|---|---|---|---|---|---|---|
| | Original | Reduction | Original | Reduction | Original | Reduction | Original | Reduction |
| s5378 | 0:00:53 | 0:00:40 | 5225 | 5200 | 493 | 510 | 14698 | 10614 |
| s9234 | 0:04:18 | 0:02:32 | 6869 | 6676 | 785 | 802 | 27980 | 18084 |
| s13207 | 0:06:01 | 0:02:58 | 6911 | 6595 | 2068 | 1969 | 40784 | 24456 |
| s15850 | 0:06:28 | 0:03:31 | 7054 | 6945 | 1068 | 1035 | 48970 | 29858 |
| s35932 | 0:51:41 | 0:44:50 | 22418 | 22450 | 34 | 33 | 95580 | 86148 |
| s38417 | 1:30:21 | 0:50:41 | 50618 | 44045 | 952 | 873 | 114958 | 79478 |
| s38584 | 0:34:53 | 0:29:58 | 17557 | 17012 | 398 | 398 | 109722 | 98762 |
| b14 | 5:45:56 | 6:10:22 | 52156 | 52156 | 27683 | 27681 | 58348 | 58340 |
| b15 | 7:12:17 | 7:20:50 | 22345 | 22346 | 7797 | 7795 | 53018 | 52998 |
| b17 | 48:54:43 | 50:07:50 | 91055 | 91051 | 10739 | 10768 | 190784 | 190708 |
| b20 | 8:09:00 | 8:24:40 | 109709 | 109709 | 31231 | 31211 | 117750 | 117746 |
| b21 | 9:39:33 | 9:57:30 | 111476 | 111476 | 31132 | 31114 | 120000 | 119992 |
| b22 | 38:38:51 | 39:25:27 | 162019 | 162019 | 37206 | 37198 | 174786 | 174774 |
| STC | 7:25:54 | 7:12:58 | 67861 | 67654 | 2762 | 2746 | 231180 | 225180 |
| tex1 | 60:00:21 | 44:45:19 | 158521 | 146374 | 2258 | 2018 | 1548504 | 1326872 |
| **Average** | **1.000** | **0.839** | **1.000** | **0.978** | **1.000** | **0.984** | **1.000** | **0.860** |

A two-pass buffer reduction algorithm has been implemented in *CodGen* in which each pass handles one of the scenarios shown in Figure 19. The effectiveness of buffer reduction has been proven with ISCAS 89 and ITC 99 benchmarks and several industry designs. It can be shown that in the case where there are large numbers of buffer/inverters, the buffer reduction mechanism reduces the total number of gates in the design by up to 47% and test generation times are also reduced accordingly. The benchmark results are summarized in Table 9.

## 3.6 Cross Time Frame Learning

There are multiple time frames in delay test, especially when Pseudo Functional Test (PFT) is used. Learnt clauses generated in one time frame can be reused in later time frames. One possible enhancement to increase *MiniSat* performance is to copy learnt clauses from one time frame to later ones. This may save *MiniSat*'s effort to generate the same learnt clause for each of the time frames. But this could also slow down *MiniSat*'s performance by introducing too many learnt clauses. So the benefit of doing this has to be explored.

Cross Time Frame Learning has been implemented in *CodGen* where every dynamically generated learnt clause is immediately copied to later time frames in the hope that the duplicated learnt clauses used in their time frames and the effort of generating the same learnt clauses there can be amortized. The results are tested with ISCAS 89 benchmarks. The benchmark results are summarized in Table 10. The conclusion is that in some test cases, such as s15850 and s38584, cross time frame learning can be used to reduce the runtime, while in the others the runtime slightly increases. Therefore, the effect of cross time frame learning depends on the characteristics of the design. It is also shown that the effectiveness of cross time frame learning is dependent on the depth of learning D, which is defined as the maximum number of time frames the generated learnt clause will be copied. The results in Table 10 shows that on average, the best results can be achieved by setting D = 4.

**Table 10. Results of cross time frame learning (n=6, K=5)**

| Circuit | SAT Time (s) | | | | | |
|---|---|---|---|---|---|---|
| | Original | D=1 | D=2 | D=3 | D=4 | No Limit |
| s5378 | 49.6 | 48.1 | 51.3 | 49.3 | 47 | 49.7 |
| s9234 | 284.5 | 298.5 | 284.9 | 298.1 | 282.2 | 289.4 |
| s13207 | 197.4 | 188.5 | 194.9 | 185 | 185 | 194.7 |
| s15850 | 533.6 | 538.9 | 522 | 525.2 | 516.1 | 554.9 |
| s35932 | 8708.2 | 8622.6 | 8277.9 | 8008.3 | 8011.4 | 8141.9 |
| S38417 | 13666.9 | 12239.1 | 13412.3 | 12609.4 | 12109.2 | 13515.1 |
| s38584 | 3623.9 | 3535.8 | 3776.1 | 3495.7 | 3298.9 | 3566.2 |
| **Average** | **1.000** | **0.978** | **0.997** | **0.967** | **0.937** | **0.993** |

## 3.7 SAT Based Approximate Observability Don't Cares (AODC) Calculation

One of the operations in logic optimization of a multi-level Boolean network is the calculation of multi-level Observability Don't Cares (ODCs) of the nodes in the network. ODCs are powerful tools in logic optimization, but the minimization of one node with respect to its ODCs can potentially change the ODCs of other nodes in the network. As a result, Compatible ODCs (CODCs) are used in actual optimization. Compared with ODCs, CODCs have the following property - if one node is minimized with respect to its CODC, the CODCs of all the other nodes are still valid. The calculation of exact CODCs in large circuits can be very time consuming. In [60], the authors described an efficient algorithm for calculating approximate CODCs (ACODCs) using BDDs.

ACODCs are more accurate compared with the Cir-ODCs described in [20]. But normally a BDD software package, such as CUDD, [61] must be used to calculate ACODCs and extra CNF clauses must be added to the SAT instance to represent the effects of the CODCs. One important observation is that in the scenario of ATPG, it is not necessary to use CODCs since the structure of the design can't be modified as in the case of logic synthesis. Therefore, the idea similar to the one presented in [60] can be used to generate Approximate ODCs (AODCs) to be used later in path delay test generation.

In *CodGen*, a SAT based approach of generating AODC has been implemented. The idea of SAT based AODC generation is shown in Figure 20. Since we are calculating approximate ODC (AODC), the input/output boundaries are not unique. The basic principle of SAT based AODC calculation is to launch a transition at the targeted fault site and test whether the propagation of this transition to the output boundary can be blocked by setting particular values on the input boundary. The algorithm for SAT based AODC calculation is as follows:

1. Build CNF instance: Since there is a transition, there should be two time frames in this problem to be solved by the SAT solver. The CNF instance is built by first searching the output cone, starting from the fault site forward to the output boundary and then backward from each node in the output cone to

the input boundary. The depth D of the forward and backward traversals is limited to keep the computation tractable. Two Boolean variables are used for each node in the search except for the nodes on the output boundary. Additional CNF clauses are added to force a transition on the fault site. Suppose the Boolean variables for the targeted fault site are $A$ and $B$ in the two time frames. The CNF clauses to create a transition on the fault site are in the format of $(A+B)\ (\bar{A}+\bar{B})$.

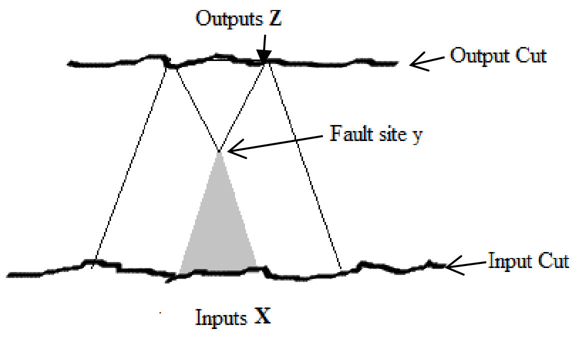


**Figure 20. SAT based approximate ODC calculation**

2. Solve the CNF instance by calling SAT solver. If the result is UNSAT, there is no AODC for the current input/output boundaries. Otherwise, go to the next step.
3. Figure out “don’t care” variables on the input boundary: Flip the Boolean value, which is assigned by SAT solver in last step, on one of the fixed nodes on input boundary and rerun SAT solver. If the result is still SAT, this node is a “don’t care” variable in the AODC clause and can be ignored in the calculated AODC, otherwise, restore the Boolean value on the node.

4. Attach the calculated AODC on the gate and use it later in delay test generation. Like Cir-ODC, when either justifying or compacting a path, each node along the path has to be observable in the last time frame and therefore its AODC has to be evaluated to false in that time frame. In this approach, AODC can be represented as the sum of single literals. If there is only one literal in the AODC clause, it can be directly applied as an assumption when calling SAT solver. Otherwise, the calculated AODC clause is attached to the node of the targeted fault site and the same mechanism developed for Dynamic SAT Solving (DSS) is used to dynamically turn on the AODC clause whenever the current path passes through the node where the AODC clause is attached.

**Table 11. Results of delay test generation with approximate ODC (AODC) (n=2, K=5)**

| Circuit | CPU Time | | Path Count | | Pattern Count | | Compaction Ratio | |
|---|---|---|---|---|---|---|---|---|
| | Original | AODC | Original | AODC | Original | AODC | Original | AODC |
| s5378 | 0:00:53 | 0:00:27 | 5225 | 5155 | 493 | 462 | 10.60 | 11.16 |
| s9234 | 0:04:18 | 0:03:12 | 6869 | 7003 | 785 | 749 | 8.75 | 9.35 |
| s13207 | 0:06:01 | 0:05:04 | 6911 | 6864 | 2068 | 1968 | 3.34 | 3.49 |
| s15850 | 0:06:28 | 0:04:40 | 7054 | 6923 | 1068 | 985 | 6.60 | 7.03 |
| s35932 | 0:51:41 | 0:28:54 | 22418 | 22418 | 34 | 33 | 659.35 | 679.33 |
| s38417 | 1:30:21 | 0:53:13 | 50618 | 49168 | 952 | 845 | 53.17 | 58.19 |
| s38584 | 0:34:53 | 0:24:54 | 17557 | 18482 | 398 | 390 | 44.11 | 47.39 |
| b14 | 5:45:56 | 5:03:47 | 52156 | 52747 | 27683 | 26500 | 1.88 | 1.99 |
| b15 | 7:12:17 | 4:57:05 | 22345 | 33065 | 7797 | 11288 | 2.87 | 2.93 |
| b17 | 48:54:43 | 40:35:45 | 91055 | 117606 | 10739 | 10275 | 8.48 | 11.45 |
| b20 | 8:09:00 | 6:08:36 | 109709 | 110140 | 31231 | 24064 | 3.51 | 4.58 |
| b21 | 9:39:33 | 8:44:29 | 111476 | 111966 | 31132 | 23806 | 3.58 | 4.70 |
| b22 | 38:38:51 | 31:00:43 | 162019 | 163400 | 37206 | 26825 | 4.35 | 6.09 |
| STC | 7:25:54 | 4:03:22 | 67861 | 70487 | 2762 | 2568 | 24.57 | 27.45 |
| tex1 | 60:00:21 | 38:07:44 | 158521 | 156829 | 2258 | 1788 | 70.20 | 87.71 |
| **Average** | **1.000** | **0.714** | **1.000** | **1.056** | **1.000** | **0.930** | **1.000** | **1.150** |

The results of calculating Approximate ODC (AODC) and delay test generation with AODC are summarized in Table 11. These results are based on AODC calculation where both forward search depth and backtrack depth are set to 3. The benchmark results show that with AODC, path delay generation can be sped up by up to ~40%. On average, with calculated AODC clauses, CodGen is going to detect more sensitizable paths, reduce the total number of test vectors and therefore achieve better compaction ratios in most of the benchmarks. It can be concluded that AODC is one of the effective techniques to speed up SAT based ATPG. Applying AODC in path delay test generation may reduce the path count. For example, when using AODC, the total path count in s5378 is reduced from 5225 to 5155. This reduction is caused by the fact that the AODC is approximate, and some knowledge of circuit structure may be lost because of the input cut when calculating AODC. For example, if there are two inputs A and B in the cut set, and they are equivalent signals in the original circuit, then A=B. But after the cut is made, they appear independent and the SAT solver may assign them different values. In this case, the AODC may erroneously identify a line in the path as not observable, and the path is skipped. Using a larger depth for the cut line would reduce this error.

## 3.8 Combined Techniques

The results of running path delay test generation with DSS, buffer reduction and AODC combined together are summarized in Table 12. With all three speedup techniques working together, it can be shown that the process of path delay test generation can be sped up by more than 70%. And the improvement of runtime performance is across the board. In every case of ISCAS 89 and ITC 99 benchmarks and industry designs tested, we see significant reduction in overall CPU time. This proves the overall effectiveness of the speedup techniques we have been exploring. Also, it is observed that combined together, these techniques improve the dynamic compaction ratio by 12% on average.

**Table 12. Results of delay test generation with DSS, buffer reduction and AODC (n=2, K=5)**

| Circuit | CPU Time | | Path Count | | Pattern Count | | Compaction Ratio | |
|---|---|---|---|---|---|---|---|---|
| | Original | Combined | Original | Combined | Original | Combined | Original | Combined |
| s5378 | 0:00:53 | 0:00:18 | 5225 | 5199 | 493 | 494 | 10.60 | 10.52 |
| s9234 | 0:04:18 | 0:01:40 | 6869 | 6831 | 785 | 757 | 8.75 | 9.02 |
| s13207 | 0:06:01 | 0:02:14 | 6911 | 6625 | 2068 | 1876 | 3.34 | 3.53 |
| s15850 | 0:06:28 | 0:02:25 | 7054 | 6952 | 1068 | 1029 | 6.60 | 6.76 |
| s35932 | 0:51:41 | 0:21:27 | 22418 | 22450 | 34 | 34 | 659.35 | 660.29 |
| s38417 | 1:30:21 | 0:26:19 | 50618 | 44297 | 952 | 797 | 53.17 | 55.58 |
| s38584 | 0:34:53 | 0:19:26 | 17557 | 17753 | 398 | 385 | 44.11 | 46.11 |
| b14 | 5:45:56 | 5:07:21 | 52156 | 52747 | 27683 | 26513 | 1.88 | 1.99 |
| b15 | 7:12:17 | 5:09:29 | 22345 | 33070 | 7797 | 11325 | 2.87 | 2.92 |
| b17 | 48:54:43 | 37:23:24 | 91055 | 117734 | 10739 | 10314 | 8.48 | 11.41 |
| b20 | 8:09:00 | 6:40:03 | 109709 | 110109 | 31231 | 24012 | 3.51 | 4.59 |
| b21 | 9:39:33 | 8:42:21 | 111476 | 111956 | 31132 | 23801 | 3.58 | 4.70 |
| b22 | 38:38:51 | 32:00:39 | 162019 | 163393 | 37206 | 26855 | 4.35 | 6.08 |
| STC | 7:25:54 | 3:49:35 | 67861 | 70675 | 2762 | 2556 | 24.57 | 27.65 |
| tex1 | 60:00:21 | 28:48:10 | 158521 | 146608 | 2258 | 1716 | 70.20 | 85.44 |
| **Average** | **1.000** | **0.576** | **1.000** | **1.039** | **1.000** | **0.930** | **1.000** | **1.132** |

The previous benchmark results show that these three techniques: DSS, buffer reduction and AODC are the most effective techniques in speeding up path delay test generation. On average, each of the three techniques can speed up CodGen by 18.3%, 16.1% and 28.6% respectively. Combined together, these techniques can speed path delay test generation by 42%, which is much better compared with the results of applying a single speedup technique. This is very important result since it shows the synergy of different speedup techniques when applied together correctly. In some cases, the path count for the combined techniques is less than the individual techniques. This is because the AODC is computed after buffer reduction, so the AODCs may be different than those identified for Table 11, and the different approximation may result in the loss of some paths.

**Table 13. Results of pseudo functional test generation with DSS, buffer reduction and AODC (*n=6;* K=5*)***

| Circuit | CPU Time (h:mm:ss) | | Path Count | | Pattern Count | | Compaction Ratio | |
|---|---|---|---|---|---|---|---|---|
| | n=2 | n=6 | n=2 | n=6 | n=2 | n=6 | n=2 | n=6 |
| s5378 | 0:00:18 | 0:00:16 | 5199 | 1912 | 494 | 203 | 10.52 | 9.42 |
| s9234 | 0:01:40 | 0:01:49 | 6831 | 4358 | 757 | 365 | 9.02 | 11.94 |
| s13207 | 0:02:14 | 0:01:17 | 6625 | 2586 | 1876 | 153 | 3.53 | 16.90 |
| s15850 | 0:02:25 | 0:04:13 | 6952 | 4964 | 1029 | 833 | 6.76 | 5.96 |
| s35932 | 0:21:27 | 1:01:35 | 22450 | 21889 | 34 | 36 | 660.29 | 608.03 |
| s38417 | 0:26:19 | 1:29:37 | 44297 | 34608 | 797 | 866 | 55.58 | 39.96 |
| s38584 | 0:19:26 | 0:43:13 | 17753 | 13838 | 385 | 270 | 46.11 | 51.25 |
| b14 | 5:07:21 | 1:11:45 | 52747 | 16358 | 26513 | 2176 | 1.99 | 7.52 |
| b15 | 5:09:29 | 0:02:05 | 33070 | 1924 | 11325 | 59 | 2.92 | 32.61 |
| b17 | 37:23:24 | 0:24:16 | 117734 | 8136 | 10314 | 116 | 11.41 | 70.14 |
| b20 | 6:40:03 | 9:06:47 | 110109 | 107504 | 24012 | 20362 | 4.59 | 5.28 |
| b21 | 8:42:21 | 10:06:11 | 111956 | 109353 | 23801 | 22020 | 4.70 | 4.97 |
| b22 | 32:00:39 | 19:28:06 | 163393 | 157234 | 26855 | 22875 | 6.08 | 6.87 |
| STC | 3:49:35 | 1:50:50 | 70675 | 22562 | 2556 | 1720 | 27.65 | 13.12 |
| tex1 | 28:48:10 | 84:50:18 | 146608 | 115786 | 1716 | 2749 | 85.44 | 42.12 |
| **Average** | **1.000** | **1.308** | **1.000** | **0.607** | **1.000** | **0.642** | **1.000** | **2.403** |

As mentioned earlier, one purpose of using SAT based delay test generation is to speed up the process of generating test patterns for Pseudo Functional Test (PFT), where more than 2 time frames are used in the delay test. To judge the effectiveness of those speedup techniques in the case of PFT, we also ran benchmarks and industry designs for the case of n=6 and K=5. The results are summarized in Table 13. It is shown that compared with the cases of n=2, on average the test generation times are increased by 31%. This result is better than expected since in theory the runtime can increase exponentially with the number of time frames. Part of the reason is that in some ITC99 benchmarks, the numbers of sensitizable paths drop significantly when n=6. It is also noticed that when n=6, the average compaction ratio is increased by 140% compared with the case of n=2.

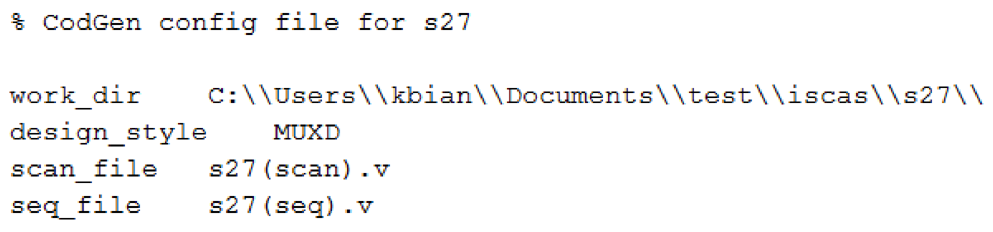
The results shown in Table 13 indicate that SAT based approach, along with the speedup techniques developed in this section, should be very powerful in improving the efficiency of path delay test generation, especially when Pseudo Functional Test (PFT) and dynamic compaction are used. In SAT based PFT, with increasing number of time frames (n), the runtime of delay test generation grows asymptotically slower than the linear functions of n. Meantime, the compaction ratio improves with increasing number of time frames. Compared to the original PODEM implementation, the SAT implementation combined with speedup techniques provides almost an order of magnitude speedup. The results in Table 4 were run on a different machine, so cannot be directly compared.

As mentioned in Section 2.1, the motivation of this research is to improve both the efficiency and the accuracy of path delay test generation in the scenario of using PFT and dynamic compaction. It can be concluded from the results shown in Table 12 and 13 that SAT based approach, along with advanced speedup techniques, is the correct solution to address the problems raised in Section 2.1.

# 4. CODGEN ENHANCEMENTS

## 4.1 Configure File

A configure file is a normal input used by Electronic Design Automation (EDA) tools to enhance their usability. By using a configure file, instead of specifying many parameters either on command line or interactively, the name of the configure file can become the only required input parameter to invoke the EDA tool and the software can process the parameters specified in the configure file automatically. The mechanism to support a configure file has been added to *CodGen*. Currently, the following parameters can be specified in the *CodGen* configure file: path of the working directory, design style, name of the scan file, name of the netlist, name of the dofile and name of the delay file. In the long run, all those parameters currently hardcoded in the *CodGen* can be included in configure file such that they can be easily configured. A sample *CodGen* configure file is shown in Figure 21.



```
% CodGen config file for s27

work_dir    C:\\Users\\kbian\\Documents\\test\\iscas\\s27\\
design_style    MUXD
scan_file   s27(scan).v
seq_file    s27(seq).v
```

**Figure 21. Sample *CodGen* configure file**

## 4.2 LSSD Support

Some industry designs used to evaluate SAT based *CodGen* use Level Sensitive Scan Designs (LSSD). [19][20] *CodGen* has been enhanced to support LSSD such that it can generate PFT KLPG tests for those industry designs.

### 4.2.1 LSSD Design Styles

LSSD based full-scan designs can be implemented in two styles, depending on which latch is used to drive the combinational logic as shown in Figure 22.

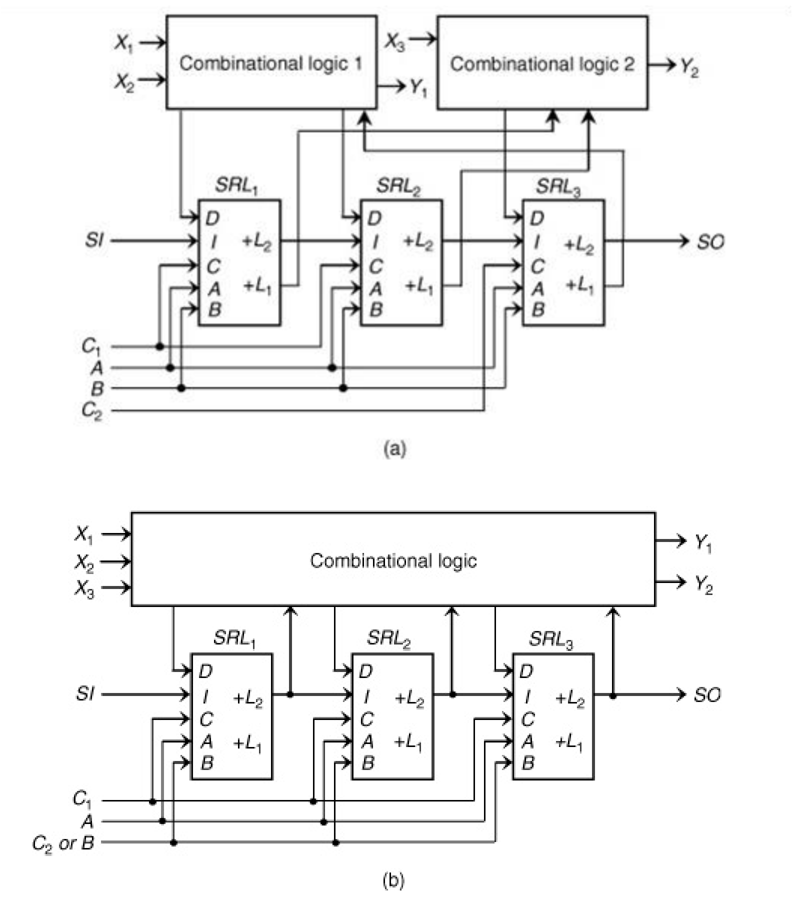


**Figure 22. LSSD design styles: (a) single-latch design, (b) double-latch design**

In single latch design, Level 1 latches (L1) are used to drive the combinational logic of the design. In this scenario, the design must be split into two clock domains which are driven by two different system clocks $C_1$ and $C_2$ respectively. One of the design rules for single latch LSSD design is that the combinational logic driven by LSSD of system clock $C_1$ has to be used to drive LSSDs of system clock $C_2$, and vice versa. This is shown in Figure 22 (a).

In double latch design, Level 2 latches (L2) are used to drive the combinational logic of the design. Since in this case, combinational logic is driven by L2 latches and the results are captured by L1 latches and the system clocks for L1 and L2 latches are nonoverlapping, the entire design can be implemented in single clock domain. This is shown in Figure 22 (b).

### 4.2.2 *CodGen* Enhancements to Support LSSD

Support for LSSD designs has been implemented in the *CodGen* for the purpose of running industrial designs from Advanced Micro Devices (AMD). These AMD test cases use full-scan design of the single latch design style. From the viewpoint of Automatic Test Pattern Generation (ATPG), delay tests of single latch LSSD designs can be generated with the same approach to generate delay tests for MUX-D flip-flop designs. But in the configure file, the design style has to be specified as LSSD so that the netlist of the scan chains can be processed correctly.

Another feature of SAT based *CodGen* in processing LSSD designs is to justify the validity of the clock network. This feature requires the user specify the name of the clock pin in the configure file. Knowing the name of the clock pin, *CodGen* can build the CNF instance corresponding to the clock network and then use the SAT solver to validate the correctness of the clock network. This is done by setting the Boolean value on the clock pin to be either 0 or 1 and then invoke the SAT solver to check whether the CNF can be satisfied. If the CNF can be satisfied in both cases, the clock network is valid. Meantime, the clock domain of each LSSD cell can be detected on the fly. After the CNF is satisfied, the LSSDs with clock signal C=0 belong to one clock domain while the LSSDs with C=1 belong to the other clock domain. This knowledge can be used later in delay test generation to generate the correct clock signal for the delay tests.

The results of running LSSD based AMD test cases are summarized in Table 14. These results are generated on Server 2 which is described in Section 2.6.

**Table 14. Results of generating pseudo functional test for AMD test case**

| Time Frame (n) | CPU Time | Path Count | Pattern Count | Memory (GB) |
|---|---|---|---|---|
| 2 | 19:10:12 | 66265 | 1079 | 3.0 |
| 4 | 44:17:46 | 55800 | 1924 | 4.6 |
| 6 | 72:18:31 | 52056 | 1688 | 6.2 |
| 8 | 106:02:41 | 51854 | 1884 | 7.9 |

# 5. CONCLUSIONS

This dissertation focuses on improving the accuracy and efficiency of path delay test generation with a Boolean satisfiability (SAT) solver. It is demonstrated with benchmark results that the performance of the path delay test generator can be significantly improved with the new mixed structural-functional approach where the KLPG algorithm was used in path search while path justification and dynamic compaction are handled with a SAT solver. The runtime of test generation was significantly reduced while more sensitizable paths were detected. The dynamic compaction ratio was also improved with SAT solver, so that fewer test patterns are generated.

A series of advanced techniques to improve SAT performance with the knowledge of circuit structure were explored and the effectiveness of those techniques was justified with both ISCAS 89 and ITC99 benchmark suites. Dynamic SAT Solving (DSS) can be used to reduce the runtime of SAT based path delay test generation by up to 60%. Other techniques, including Cir-ODC, SAT based static learning, dynamic learnt clause management, AODC and cross time frame learning can also be used to improve the performance of path delay test generation by 10~30%.

Several industrial designs from AMD and Texas Instruments were also used to justify the effectiveness of this research.

# REFERENCES

[1] R. D. Eldred, “Test Routines Based on Symbolic Logical Statements," *Journal of the ACM*, vol. 6, pp. 33-36, 1959.

[2] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, “Transition Fault Simulation," *IEEE Design and Test of Computers*, vol. 4, no. 2, pp. 32-38, 1987.

[3] K. -T. Cheng, “Transition Fault Testing for Sequential Circuits,” *IEEE Transactions on Computer Aided Design of Integrated Circuits and System*, vol. 12, no.12, pp. 1971-1983, Dec. 1993.

[4] Z. Barzilai and B. K. Rosen, “Comparison of AC Self-Testing Procedures,” in *Proceedings of IEEE International Test Conference*, pp. 89-94, Oct. 1983.

[5] G. L. Smith, “Model for Delay Faults Based Upon Paths”, *IEEE Int’l Test Conf.*, Philadelphia, PA, Oct. 1985, pp. 342-349.

[6] V. S. Iyengar, B. K. Rosen, and I. Spillinger, “Delay Test Generation 1 – Concepts and Coverage Metrics,” in *Proceedings of IEEE International Test Conference*, pp. 857-866, Sept. 1988.

[7] V. S. Iyengar, B. K. Rosen, and I. Spillinger, “Delay Test Generation 2 – Algebra and Algorithms,” in *Proceedings of IEEE International Test Conference*, pp. 867-876, Sept. 1988.

[8] J. Carter, V. Iyengar, and B. Rosen, “Efficient Test Coverage Determination for Delay Faults,” in *Proceedings of IEEE International Test Conference*, pp. 418-

427, Sept. 1987.

[9] E. S. Park, M. R. Mercer, and T. W. Williams, “Statistical Delay Fault Coverage and Defect Level for Delay Faults,” in *Proceedings of IEEE International Test Conference*, pp. 492-499, Sept. 1988.

[10] C. W. Tseng and E. J. McCluskey, “Multiple-Output Propagation Transition Fault Test,” in *Proceedings of IEEE International Test Conference*, pp. 358-366, Oct. 2001.

[11] X. Lin and J. Rajski, “Propagation Delay Fault: A New Model to Test Delay Faults,” in *Proceedings of IEEE Asian South Pacific Design Automation Conference*, pp. 178-183, 2005.

[12] A. K. Pramanick and S. M. Reddy, “On the Fault Coverage of Gate Delay Fault Detecting Tests,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 1, pp. 78-94, Jan. 1997.

[13] A. K. Majhi, J. Jacob, L. M. Patnaik, and V. D. Agrawal, “On Test Coverage of Path Delay Faults,” in *Proceedings of International Conference on VLSI Design*, pp. 418-421, Jan. 1996.

[14] A. K. Majhi, V. D. Agrawal, J. Jacob, and L. M. Patnaik, “Line Coverage of Path Delay Faults,” *IEEE Transactions on VLSI Systems*, vol. 8, no. 5, pp. 610-613, Oct. 2000.

[15] A. K. Majhi and V. D. Agrawal, “Tutorial: Delay Fault Models and Coverage,” in *Proceedings of International Conference on VLSI Design*, pp. 364-369, Jan. 1998.

[16] S. R. Nassif, “Modeling and Analysis of Manufacturing Variations,” in *Proceedings of IEEE Custom Integrated Circuits Conference*, pp. 223-228, May 2001.

[17] W. Qiu and D. M. H. Walker, “Testing the Path Delay Faults for ISCAS85 Circuit c6288,” in *Proceedings of IEEE International Workshop on Microprocessor Test and Verification*, pp. 38-43, May 2003.

[18] L. H. Goldstein and E. L. Thigpen, “SCOAP: Sandia Controllability/Observability Analysis Program,” in *Proceedings of ACM/IEEE Design Automation Conference*, pp. 190–196, Jun. 1980.

[19] E. B. Eichelberger and T. W. Williams, “A Logic Design Structure for LSI Testability,” in *Proceedings of ACM/IEEE Design Automation Conference*, pp. 462-468, June 1977.

[20] F. Motika, N. Tendolkar, C. Beh, W. Heller, C.Radke, and P. Nigh, “A Logic Chip Delay-test Method Based on System Timing,” *IBM Journal of Research and Development*, Vol. 34, No.2/3, pp. 299-312, March/May 1990.

[21] S. DasGupta, P. Goel, R. G. Walther, and T. W. Williams, “A Variation of LSSD and Its Implications on Design and Test Pattern Generation in VLSI,” in *Proceedings of IEEE International Test Conference*, pp. 63-66, Nov. 1982.

[22] C. T. Glover and M. R. Mercer, “A Method of Delay Fault Test Generation,” in *Proceedings of ACM/IEEE Design Automation Conference*, pp. 90-95, Jun. 1988.

[23] B. I. Dervisoglu and G. E. Strong, “Design for Testability: Using Scan Path Techniques for Path-delay Test and Measurement,” in *Proceedings of IEEE*

*International Test Conference*, pp. 365-374, Oct. 1991.

[24] J. Savir, "Skewed-Load Transition Test: Part I, Calculus," in *Proceedings of IEEE International Test Conference*, pp. 705-713, Sept. 1992.

[25] S. Patel and J. Savir, "Skewed-Load Transition Test: Part II, Coverage," in *Proceedings of IEEE International Test Conference*, pp. 714-722, Sept. 1992.

[26] J. Savir and S. Patel, "Broad-Side Delay Test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 8, pp. 1057-1064, Aug. 1994.

[27] W. Qiu and D. M. H. Walker, "An Efficient Algorithm for Finding the K Longest Testable Paths Through Each Gate in a Combinational Circuit", *IEEE Int'l Test Conf.*, Charlotte, NC, Sep. 2003, pp. 592-601.

[28] W. Qiu, J. Wang, D. M. H. Walker, D. Reddy, X. Lu, Z. Li, W. Shi and H. Balachandran, "K Longest Paths Per Gate (KLPG) Test Generation for Scan-Based Sequential Circuits", *IEEE Int'l Test Conf.*, Charlotte, NC, pp.223-231, Oct. 2004.

[29] D. M. H. Walker, "Tolerance of Delay Faults," *IEEE Int'l Workshop on Defect and Fault Tolerance in VLSI Systems,* 1992, pp. 207-216.

[30] S. Lahiri, D. M. H. Walker and K. Bian, "KLPG based Pseudo-functional Test with Dynamic Compaction", *SRC TECHCON,* Austin, TX, Sep. 2011.

[31] Z. Wang and D. M. H. Walker, "Dynamic Compaction for High Quality Delay Test", *IEEE VLSI Test Symp.*, San Diego, CA, Apr-May, 2008, pp.243-248.

[32] P. Pant and J. Zelman, "Understanding Power Supply Droop During At-Speed

Scan Testing”, *IEEE VLSI Test Symp.*, Santa Cruz, CA, May 2009, pp.227-232.

[33] B. Nadearu-Dostie, K. Takeshita and J.-F. Cote, “Power-Aware At-Speed Scan Test Methodology for Circuits with Synchronous Clocks”, in *Proceedings of IEEE International Test Conference*, pp.1-10, Oct. 2008.

[34] M. Abadir and T. Ambler, “Economics of Electronic Design, Manufacture and Test”, pp. 189, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1994.

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

[36] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, “Chaff: Engineering an efficient SAT solver”, *IEEE/ACM Design Automation Conf.*, San Francisco, Jun. 2001, pp. 530-535.

[37] N. Eén and N. Sörensson, “An extensible SAT solver”, *Int’l Conf. on Theory and Applications of Satisfiability Testing*, Vancouver, BC, May 2004, pp. 502-518.

[38] R. Drechsler et al., “Test Pattern Generation using Boolean Proof Engines”, Ch. 10, Springer, Berlin, Germany, 2009.

[39] R. Drechsler, S. Eggersglüβ, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel and D. Tille, “On Acceleration of SAT-based ATPG for Industrial Designs”, *IEEE Trans. on Computer-Aided Design*, vol. 27, no. 7. pp. 1329-1333, Jul. 2008.

[40] C. P. Gomes, H. Kautz, A. Sabharwal and B. Selman, “Satisfiability Solvers”, *Foundations of Artificial Intelligence*, vol.3, pp. 89-134, 2008.

[41] M. Davis, G. Logemann and D. Loveland, “A Machine Program for Theorem Proving”, *CACM*, vol.5, no.7, pp.394-397, 1962.

[42] M. Davis and H. Putnam, “A Computing Procedure for Quantification Theory”, *CACM*, vol.7, no.3, pp.201-215, 1960.

[43] J. Marques-Silva, I. Lynce and S. Malik, “Conflict-Driven Clause Learning SAT Solvers”, Handbook of Satisfiability, Ch.4, pp.127-149, IOS Press, Amsterdam, the Netherlands, 2008.

[44] T. Larrabee, “Test Pattern Generation Using Boolean Satisfiability”, *IEEE Trans. on Computer-Aided Design*, vol. 11, no.1, pp. 4-15, Jan. 1992.

[45] M. Sauer, A. Czutro, T. Schubert, S. Hillebrecht, I. Polian and B Becker, “SAT-based Analysis of Sensitisable Paths”, *Proc. of DDECS’11*, pp. 93-98, 2011.

[46] F. Lu, L.-C. Wang, K. T. Cheng and R. C.-Y. Huang, “A Circuit SAT Solver With Signal Correlation Guided Learning”, *Proc. of DATE’03*, pp. 892-897, 2003.

[47] N. Sorenssion and N. Een, “MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization”, http://www.minisat.se/, 2005.

[48] J. Chen, “Solvers with a Bit-Encoding Phase Selection Policy and a Decision-Depth-Sensitive Restart Policy”, *Proc. of SAT Competition 2013*, pp. 44-45, 2013.

[49] Y.-C. Lin, F. Lu and K.-T. Cheng, “Pseudofunctional Testing”, *IEEE Trans. on Computer-Aided Design*, vol. 25, no. 8, pp. 1535-1546, Aug. 2006.

[50] M. N. Velev, “Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors”, *IEEE ASP Design Automation Conf.*, Yokohama, Japan, Jan. 2004, pp. 310-315.

[51] D. Tille, S. Eggersglüβ, R. Krenz-Bååth, J. Schloeffel, R. Drechsler, “Improving CNF Representations in SAT-based ATPG for Industrial Circuits using BDDs”, *IEEE Europe Test Symp.*, Prague, Czech, May 2010, pp. 176-181.

[52] S. Eggersglüβ, D. Tille and R. Drechsler, “Speeding up SAT-based ATPG using Dynamic Clause Activation”, *ATS’09*, pp.177-182, 2009.

[53] Z. Fu, Y. Yu and S Malik, “Considering Circuit Observability Don’t Cares in CNF Satisfiability”, *Proc. of DATE’05*, pp. 1108-1113, 2005.

[54] 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 Computer-Aided Design*, vol. 13, no.9, pp. 1143-1158, 1994.

[55] S. Eggersglüβ and R. Drechsler, “Increasing robustness of SAT-based delay test generation using efficient dynamic learning techniques’, in *IEEE Eur. Test Symp.*, 2009, pp. 81-86.

[56] R. Chen and H. Zhou, “An Effective Algorithm for Buffer Insertion in General Circuits Based on Network Flow”, *IEEE Trans. on Computer-Aided Design,* vol. 26, no. 11, pp.2069-2073, 2004.

[57] Y. Ma, X. Hong, S. Dong, S. Chen, C. K. Cheng and J. Gu, “Buffer Planning as an Integral Part of Floorplanning with Consideration of Routing Congestion”, *IEEE Trans. on Computer-Aided Design*, vol. 24, no. 4, pp. 609-621, 2005.

[58] J. Lillis, C. K. Cheng and T. T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model", *IEEE J. Solid-State Circuit*, vol. 31, no. 3, pp. 437-447, 1996.

[59] P. Saxena, N. Menezes, P. Cocchini and D. A. Kirkpatrick, "The Scaling Challenge: Can Correct-by-Construction Design Help?", in *Proc. of ACM/IEEE ISPD*, pp. 51-58, 2003.

[60] N. Saluja and S. P. Khatri, "A Robust Algorithm For Approximate Compatible Observability Don't Care (CODC) Computation", Proc. of 41st DAC, San Diego, CA, 2004, pp.422-427.

[61] F. Somenzi, "CUDD: CU Decision Diagram Package Release 2.5.0", http://vlsi.colorado.edu/~fabio/CUDD/, 2012.