Integration of Learning Techniques into Incremental Satisfiability for Efficient Path-Delay Fault Test Generation *

Kameshwar Chandrasekar and Michael S. Hsiao ({*kamesh, hsiao*}@*vt.edu*)
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24061

Abstract

In recent years, several Electronic Design Automation (EDA) problems in testing and verification have been formulated as Boolean Satisfiability (SAT) instances due to the development of efficient general-purpose SAT solvers. Problem-specific learning techniques and heuristics can be integrated into the SAT solver to further speed-up the search for a satisfying assignment. In this paper, we target the problem of generating a complete test-suite for the path delay fault (PDF) model. We provide an Incremental Satisfiability framework that learns from (1) static logic implications, (2) segment-specific clauses, and (3) unsatisfiability cores of each untestable partial PDF. These learning techniques improvise the test generation for path delay faults that have common testable and/or untestable segments. The experimental results show that a significant portion of PDFs can be excluded dynamically in the proposed incremental SAT formulation for large benchmark circuits, thus potentially achieving speed-ups for PDF test generation.

1 Introduction

The increasing clock frequencies and reduced feature sizes have made delay testing a necessity. While various fault models have been proposed to capture the effect of delay defects, the path delay fault (PDF) model is the most accurate in characterizing the cumulative effect of distributed delays along each path in a circuit. However, the main bottleneck in the PDF model is the exponential number of paths in a circuit. As a result, Automatic Test Pattern Generation (ATPG) suffers from temporal explosion if each path delay fault needs to be targeted in the circuit. In order to reduce the number of PDFs that need to be considered for test generation, several methods have been proposed to identify the untestable PDFs *a priori*. In [1], static logic implications are used to identify many untestable path delay faults in a circuit. Other methods have been proposed in [2, 3] to quickly identify the untestable PDFs. In general, these techniques identify a set of PDFs that are untestable and predict a lower bound on the number of untestable PDFs. However, integrating these techniques into the test generation algorithm leads to an additional overhead for the ATPG engines that directly work on the circuit structure and use different logic systems [4–6]. More recently, in [7], the authors propose a Zero-suppressed Binary Decision Diagram (ZBDD) based technique to identify all the testable PDFs in a circuit. The PDFs are stored as a ZBDD and set operations are used to remove all the untestable PDFs from the ZBDD. Subsequently, an ATPG engine can be invoked to generate the test vectors for all the testable PDFs identified by their technique.

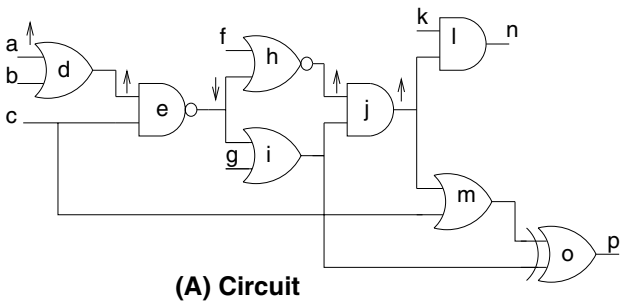
Boolean Satisfiability (SAT) was first used for delay testing in [8], in which all the paths in a circuit are enumerated, and hence it may not be applicable for large circuits. In [9], the authors suggest that incremental SAT is suitable for PDF testing. They provide a basic framework to generate non-robust test vectors for *all* the PDFs in the circuit. However, their framework is devoid of problem-specific learning techniques that can improve the performance of the SAT solver. To improve the performance of SAT, it has been suggested in [10–12] that circuit-related information can enrich the clause database of the SAT solver for equivalence checking and model checking problems. This opens an opportunity for problem-specific learning to discover the knowledge that could enhance the SAT solver.

In our work, we present efficient learning techniques to incrementally reduce the number of paths that need to be considered, thereby speeding up the Incremental SAT solver, for PDF test generation. We target the problem of generating a complete test suite for all the path delay faults in a circuit. The contributions of our work are as follows:

1. We convert the static logic implications in a circuit into clauses and add them to the clause database of SAT solver. Since implications were shown to be efficient in identifying the untestable PDFs in a circuit, they will help to quickly identify the untestable path delay faults during test generation.

2. For each fanout-free segment in the circuit, we try to sensitize the segment-fault constraints individually and store the conflict clauses generated during the search as *learned segment clauses*. Then, during the ATPG,

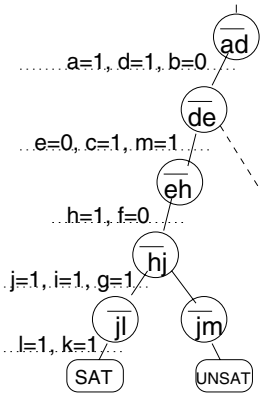
*supported in part by NSF Grants CCR-0196470 and CCR-0305881.



(A) Circuit

Path Recursion		Clauses Added
Depth	Segment	(On & Off path)
1	ad ↑	(a) (d) (–b)
2	de ↑	(–e) (c)
3	eh ↓	(h) (–f)
4	hj ↑	(j) (i)
5	jl ↑	(l) (k)
5	jm ↑	(m) (–c)

(C) Sensitization clauses



(D) Incremental SAT

(–a + d) (–b + d) (a + b + –d) (c + e) (d + e) (–c + –d + –e) (–e + –h) (–f + –h) (e + f + h)
(–e + i) (–g + i) (e + g + –i) (h + –j) (i + –j) (–h + –i + j) (j + –l) (k + –l) (–j + –k + l)
(–c + m) (–j + m) (c + j + –m) (–i + –m + –o) (i + m + –o) (–i + m + o) (i + –m +o)

(B) Circuit CNF

Figure 1. Incremental SAT for Non-Robust PDF testing

we add these clauses along with the sensitization constraints for each fanout-free segment.

3. When an untestable partial PDF is encountered during test generation, we extract the unsatisfiable core of clauses. These clauses are used to identify other untestable PDFs, on the fly, during test generation.

The experimental results show that a significant portion of PDFs can be excluded dynamically in the proposed incremental SAT formulation for large benchmark circuits, thus potentially achieving speed-ups for PDF test generation.

The rest of the paper is organized as follows. In Section 2, we provide a brief introduction to incremental SAT and path delay faults. In Section 3, we describe the mechanism to add static logic implications and *learned segment clauses* to the SAT solver. Section 4 presents the idea of extracting *unsatisfiable cores* for untestable PDFs and using them to avoid other untestable path delay faults. The experimental results on ISCAS 85 and ISCAS 89 circuits are reported in Section 5 and Section 6 concludes the paper.

2 Preliminaries

2.1 Incremental Satisfiability (ISAT)

The Boolean Satisfiability problem is to determine whether a satisfying variable assignment, V, exists for a given Boolean formula, $\Phi(V)$, commonly expressed in conjunctive normal form (CNF). A CNF is a conjunction of clauses, where each clause is a disjunction of literals. A literal is a variable occurring in its positive or negative polarity. The general form for the formula is as follows:

$$\Phi(v_1, v_2, ...v_n) = C_1 . C_2 ... . C_m \quad (1)$$

where,

- Φ is a propositional formula in CNF
- v_i is the i^{th} Boolean variable in the CNF
- C_j is the j^{th} clause in the CNF

In conventional SAT, all the clauses in the CNF are manipulated to find a satisfying variable assignment for Φ. In the case of incremental SAT, we partition the clauses into different groups. Initially, only a single partition of clauses is in the clause database. If a satisfying assignment is found for the existing set of clauses, we add the next partition of clauses iteratively. If no satisfying assignment is found in an iteration, then the SAT solver can stop and report that no satisfying assignment exists for Φ, since a subset of clauses in Φ cannot be satisfied.

Incremental SAT is advantageous if the SAT solver can conclude that Φ is unsatisfiable using a smaller set of clauses. In the worst case, we have to add all the partitions into the clause database of the SAT solver. It should be noted that the solver can proceed from its previous variable assignments when we add-in the next partition of clauses to the SAT solver in the subsequent iteration. Since a huge number of untestable PDFs exist in a circuit [13], and many untestable path delay faults have common untestable segments, incremental SAT can identify the untestable PDFs in groups and speed up the test generation.

2.2 Path Delay Fault Testing

For each structural path in the circuit from the primary input to the primary output, two path delay faults are associated - rising PDF and falling PDF. In order to detect a path delay fault, we require two test vectors $< t_1, t_2 >$, where a rising (falling) transition is asserted at the beginning of the path for the corresponding rising (falling) PDF. In addition to the initial transition, we sensitize the off-path inputs of the path to propagate the transition to the output. A test $< t_1, t_2 >$, for a given path delay fault, is said to be *robust* if the test can detect the fault independent of other path delay faults in the circuit. On the other hand, if the path delay fault can be masked by the presence of other faults in the circuit, then the test $< t_1, t_2 >$ is called a *non-robust* test. In our work, we target non-robust tests, since non-robust test generation

is a direct application of incremental SAT and the non-robust conditions are present in robust tests as well. Nevertheless, the proposed techniques can be extended to robust test generation or any other circuit problem that is an application of incremental satisfiability by modifying the incremental clauses for each iteration in the CNF.

For non-robust PDF test generation, it is sufficient to sensitize the off-path inputs in the second test vector t_2, such that the on-path transition propagates to the primary output. In Figure 1, we demonstrate the non-robust test generation for PDFs using incremental satisfiability. The circuit is illustrated in Figure 1 (A). The consistency clauses for the gates in the circuit is given in part (B) of the Figure. The sensitization clauses for each segment is shown in part (C). We start from segment $ad \uparrow$ and recursively add the sensitization clauses for the fanout segments. The recursive addition of segments is illustrated by the tree structure in Figure 1 (D). When a fanout stem is reached (eg. segment $hj \uparrow$), we add the segments in its branches recursively in a depth-first fashion. In this example, we first add $jl \uparrow$. If we reach a primary output during recursion, then all the sensitization clauses for the current PDF have been satisfied, and the variable assignments at the primary inputs is the test vector for that PDF. In this example, the rising path delay fault at $a-d-e-h-j-l$ is testable. Then, we backtrack in the segment recursion tree, delete the sensitization clauses of $jl \uparrow$ and try to sensitize the next fanout segment - $jm \uparrow$. If the SAT solver reports that a partial PDF is untestable, all PDFs with this partial PDF as a prefix are untestable. In this example, the partial rising PDF $a-d-e-h-j-m$ is untestable. So, the ATPG can backtrack immediately in the segment recursion tree without sensitizing the segment mp. In this way, the ATPG proceeds for all the paths in the circuit and generates test vectors for all the testable PDFs.

3 Static Learning

In general, for solving circuit problems using Boolean Satisfiability, the clauses for all the gates in the circuit are built to form the complete clause database. It has been shown in [10–12] that additional constraints can be added to the clause database to speed up the search process. In order to take advantage of static learning for path delay faults, we add the *static logic implications*, generated from the circuit, as well as certain *segment-specific clauses* to the clause database. It should be noted that these clauses are quickly generated in a preprocessing technique and are computed only once, before the actual test generation.

3.1 Static Logic Implications

The implications in a circuit provide useful information that helps to deduce relations between different lines in the circuit. They have been widely used to detect a huge number of untestable path delay faults in [1–3]. In the Boolean Satisfiability domain, implications can be seen as the circuit information that aid in accelerating the Boolean Constraint Propagation (BCP) (the most time consuming routine in the SAT solver [14]). In order to speed up the SAT solver and detect untestable path delay faults faster, during ATPG, we convert the implications into clauses and add them to the circuit CNF. As direct implications are already encoded in the circuit clauses, we add only the indirect implications and extended backward implications to the circuit CNF. These implications are generated using the techniques proposed in [15].

Each implication is converted into a clause as follows. Suppose we have a static implication: $a \rightarrow b$. The clause equivalent of this implication is $(\bar{a} + b)$. Similarly, for all the implications obtained from the circuit, we add the corresponding clauses to the circuit CNF. Because of the low computational overhead involved in generating static logic implications, this step is simple. Though simple, it can enrich the clause database with additional constraints that will increase the deductive power of the SAT solver.

3.2 Segment-specific learned clauses

In incremental satisfiability for path delay fault testing, we solve the clauses for partial PDFs first and then augment the clause database incrementally. The clauses added for each increment corresponds to the sensitization of a segment-fault (also referred as segments in the sequel). The initial segments (partial PDFs) that form the common prefix for various path delay faults (see Figure 2 (A)) are targeted once and are re-used while backtracking in the path recursion. Similarly, the segments in the latter part of the circuit (near the primary outputs) may be the common post-fixes for different path delay faults (see Figure 2 (B)). The existing test gen-

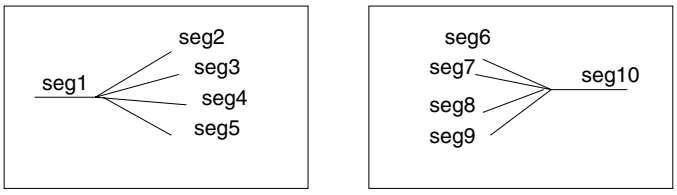


Figure 2. Prefix & Postfix partial PDFs

eration methods do not account for the latter scenario in the circuit. Moreover, in order to sensitize each segment, we add only the on-path and off-path clauses. It will be beneficial to the SAT solver if we add the clauses that prune the conflict search subspace for that segment. When a segment is encountered multiple times as a prefix/postfix, these clauses will guide the SAT solver to avoid the associated conflict-space and reach at a conclusion more quickly.

Motivated by these factors, we learn the clauses that can be added along with the sensitization constraints, for each

segment, to speed up the SAT solver. As a pre-processing step, we assert the sensitization constraints for each segment and perform an all-solutions SAT search [16]. We store the conflict-induced clauses, generated during the SAT search, as *learned segment clauses*. All these conflict-induced clauses constrain the solution space for the target segment. These clauses should be satisfied (a necessary condition) for detecting any PDF involving this segment. It should be noted that a proper subset of the conflict clauses define a solution super-space for that segment. Hence, these clauses represent constraints for the given segment and will preserve the satisfiability of the original problem.

In general, solving for the sensitization constraints of a single segment is very easy. Moreover, it is not necessary to complete the all-solutions SAT search for every segment, since each conflict clause can be individually stored as a *learned segment clause*. Considering the time constraints, it may be sufficient to learn the clauses only for certain segments, instead of all the segments. Due to the nature of segment recursion from input to output in incremental SAT, the segments with large number of fanout paths is more likely to be traversed fewer times (Figure 2 (A)) as compared to the segments with the same number of fanin paths (Figure 2 (B)). We use a scoring mechanism to determine the segments that will be traversed many times during PDF test generation. We assign the number of fanin paths in each segment (found using a linear time algorithm) as its score. Then, we compute the weighted average of all the scores and learn for segments whose scores are higher than the weighted average. Since the segments with higher scores will be encountered more times during ATPG, it is necessary to quickly traverse these segments to accelerate the ATPG.

4 Dynamic Learning from Unsatisfiable Cores of untestable partial PDFs

Definition 1 *Given an unsatisfiable CNF formula Φ, a subset of clauses, $\phi \subseteq \Phi$ that is unsatisfiable by itself is called an unsatisfiable core of clauses.*

The unsatisfiable cores were initially used to validate a SAT solver in [17, 18]. In their work, all the conflict clauses that are generated during SAT search and their resolvent clauses are stored externally. They propose an algorithm to generate an unsatisfiable core from these clauses. Starting from the final conflict clause, the algorithm backtraces through the resolvent clauses. Finally, the set of all original clauses reached during the backtrace is identified as the unsatisfiable core. We use the same procedure to generate the unsatisfiable core. However, we do not need to store the conflict clauses externally, since our objective is to extract the unsatisfiable core rather than validate the SAT solver.

In the segment recursion of incremental SAT, we obtain a partial PDF that is untestable. The corresponding clauses in the database of the SAT solver form an unsatisfiable CNF. The sensitization clauses in the unsatisfiable core for this CNF can be mapped to a group of segments that cannot be sensitized together. All the path delay faults that contain these segments are guaranteed to be untestable. They are called the *untestable core of segments* (or simply untestable core) in the sequel. A motivating scenario is given in Figure 3. Suppose we obtain an unsatisfiable CNF corresponding to an untestable partial PDF $(a-b-c-d-e-f-g-h-i)$. The untestable core of segments is found to be $\{ab, ef, hi\}$. It can be concluded that all the path delay faults that contain all these three segments are untestable. This will help to avoid testing for other untestable path delay faults and to speed up the test generator.

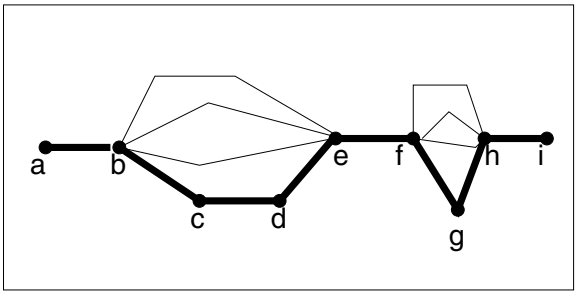


Figure 3. Unsatisfiable core of segments

After encountering each untestable partial PDF, we identify the untestable core of segments and store them in a table, T. If any partial PDF contains all segments of any untestable core, then that PDF is guaranteed to be untestable. Before choosing a segment for path recursion, we need to look up the table, T, to check if the current partial PDF will cover any of the untestable core of segments. If it covers any untestable core of segments, we ignore that segment and look for the next segment in the path recursion. This step helps to avoid one iteration of adding clauses, solving for SAT and deleting clauses in the incremental SAT framework.

Consider the untestable core of segments (UC) and the current partial PDF (PP) as *sets* of segments. If an untestable core of segments is a subset of the partial PDF, $UC \subseteq PP$, then we have to backtrack in the path recursion. However, checking for a subset in the table of unsatisfiability cores, for every iteration of ISAT, is time consuming. As an alternative, we introduce a *single watched segment* technique (motivated by the two-literal watching rule in zChaff [14]) to look for subsets on the fly.

While adding an untestable core to the table, T, we assign *one* segment, which is not yet chosen in the path recursion, as the watched segment for that row. For each segment, we store a watched list that contains the row numbers of the table, T, where it is a watched segment. Please note that each row corresponds to an untestable core. During test generation, before a segment is chosen, we traverse the rows of T in its watched list. For each row/untestable core, we search

Table 1. PDF ATPG for ISCAS '85 & '89 circuits

Circuit	#FF-seg	#PDFs	Incremental Satisfiability						
			No Learning	Static Learning Only		Static+Dynamic Learning			#Testable
			Time(s)	#Addtnl cls	Time(s)	#PDFs excl.	% excl.	Time(s)	PDFs
c432	236	167,852	0.59	480	0.46	7,044	**4.3**	0.49	2,477
c499	264	18,880	12.13	2,561	13	1,482	**14.7**	13.08	8,800
c880	451	17,284	4.51	137	4.59	135	**21.4**	4.58	16,652
c1908	995	1,458,114	309.25	2,070	297.2	911,046	**82.6**	257.21	355,168
c1355	776	8,346,432	1391.59	1,819	1467.94	1,654,492	**22.9**	1400.85	1,110,304
c2670	1,422	1,359,920	407.87	1,306	297.12	277,111	**22.5**	297.2	130,626
c3540	1,825	57,353,342	3709.58	1,483	3628.1	15,359,492	**27.4**	3441.87	1,202,584
c5315	2,885	2,682,610	911	1,645	971.93	482,286	**20.6**	919.36	342,117
s635	315	2,490	0.36	91	0.38	0	**0**	0.38	2,459
s641	260	3,488	1.08	54	1.09	248	**20.4**	1.09	2,270
s713	320	43,624	4.08	336	3.68	10,098	**26.1**	3.39	4,922
s991	436	14,920	5.12	89	5.04	2,282	**52.7**	4.92	10,590
s1512	648	6,972	3.95	434	3.81	1,233	**48.2**	3.66	4,414
s1269	646	79,140	17.19	237	17.64	14,246	**31.1**	16.9	33,382
s1423	704	89,452	25.49	285	26.32	11,370	**25.7**	26.04	45,198
s3271	1,661	38,388	51.81	5,854	48.29	4,434	**23.2**	47.59	19,292
s3384	1,688	39,582	61.3	1,818	57.15	367	**4.8**	57.03	31,966
s5378	2,395	27,084	130.35	3,861	123.76	1,271	**24.7**	126.01	21,928
s9234	3,480	489,708	678.04	3,603	641.5	159,031	**37.0**	635.06	59,854

Note a) #FF-segments - #fanout-free segments
Note b) #Addtnl cls - #additional clauses learned due to static learning
Note c) excl. - untestable PDFs identified by untestable cores

for the next available segment to be assigned as the watched segment for that core. If no free segment is available, then that core has been covered and we can conclude that the corresponding partial PDF is untestable.

The *watched segment* technique is explained in the following example. Let $P = \{s_1, s_2, s_3, s_4\}$ be the set of segments in the current partial PDF, and we have to choose s_5 as the next segment in the path recursion. Let us consider the following UTC, a table of untestable core of segments, where $*$ denotes the watched segment in each row.

$$\begin{aligned} UTC \quad = \quad & \{\{s_1, s_3, s_5*, s_7, s_9\}, && \text{(Row 1)} \\ & \{s_5, s_{10}, s_{15}*, s_{20}\}, && \text{(Row 2)} \\ & \{s_1, s_3, s_5*\}\} && \text{(Row 3)} \end{aligned}$$

The watched list for s_5 contains row 1 and row 3 in the table. For row 1, $\{s_1, s_3\}$ have already been chosen in P, and $\{s_7, s_9\}$ are free segments. So, we can change the watched segment and assign s_7 or s_9 as the next watched segment. However, for row 3, all the other segments in that row have already been chosen in P. This means that the untestable core in row 3 will become a subset of the partial PDF if we choose s_5. Therefore, we avoid s_5 and look for the next segment in the segment recursion to proceed with the test generation. Note that we did not consider row 2 during this entire manipulation and it is not necessary to update the watched segments for the backtracks in segment recursion.

5 Experimental Evaluation

The above techniques were implemented in C++ and integrated with zChaff [14], downloaded from the web-site [19]. However, the proposed techniques can be integrated into any DPLL based SAT solver. In fact, it may be beneficial if we use a SAT solver with sophisticated incremental SAT solving capabilities such as MINISAT [20]. zChaff was chosen since it has an inbuilt feature to extract unsatisfiable cores. The experiments were conducted on a 3 GHz Pentium 4 machine with 1 GB RAM running the Linux OS. A non-robust PDF ATPG was implemented to generate test vectors for *all* the path delay faults in ISCAS 85 and ISCAS 89 benchmark circuits with a time limit of 5000 seconds. We compare our technique with [9] only, since it is the most recent Boolean Satisfiability technique proposed for PDF test generation.

The experimental results are shown in Table 1. For each circuit, the number of fanout-free segments and the number of path delay faults are first reported. Next, the results of PDF ATPG using incremental SAT are listed for (1) no learning (similar to the ISAT in [9]), (2) static learning only, and (3) combined static and dynamic learning. The final column of the table reports the total number of detected PDFs by all three methods. Note that each method is able to complete detection of all testable PDFs in the circuit. Under the Static-learning ISAT, both the number of clauses added and the time taken for test generation are reported. For the combined static and dynamic learning, the number of PDFs excluded due to unsatisfiability cores during dynamic learning, the number of PDFs excluded as a percentage (%) of total

number of untestable PDFs, and the time taken for test generation are given.

From Table 1, it is observed that there is an improvement in time for many circuits, such as c1908, c2670, c3540, s5378 and s9234. The efficacy of dynamic learning is demonstrated by the number of untestable path delay faults identified by the untestable cores. The untestable cores are able to dynamically identify that more than 20% of the PDFs are eligible to be excluded during ATPG for most of the circuits. While there is some overhead in pruning the number of PDFs, the corresponding speedup is also obtained for larger circuits. For several small circuits, there is no appreciable improvement in time taken to complete the test generation. It is likely that the clauses added, during static learning for fanout-free segments, and the unsatisfiability cores, take more time than simply running the original SAT solver without learning for these cases. Finally, it should be noted that the implementation of the proposed techniques and the variable ordering are deciding factors for the performance of a SAT solver. In zCHAFF, the solver has to re-adjust its clause database and variables' status, every time a new clause is added incrementally. So the addition of many statically learned clauses may lead to a significant time-overhead as seen in c1355.

6 Conclusion and Future Work

In this work, we introduce static and dynamic learning techniques for an incremental satisfiability framework. We use the framework to generate a complete non-robust test suite for the path delay fault model in a circuit. We learn from static logic implications and segment-specific clauses, at each iteration of the increment, in order to enrich the clause database of the SAT solver. We use the unsatisfiability cores of unsatisfiable CNFs, generated for untestable PDFs, to identify other untestable path delay faults. Experimental results show that these unsatisfiable cores help to identify a large number of untestable PDFs and they are subsequently avoided during test generation. Potential directions for future research include:

- Represent all the PDFs as a ZBDD [7] and use the ZBDD structure for ISAT. This representation will help to physically remove the untestable PDFs on-the-fly.
- Generate test vectors for a particular subset of PDFs (e.g., critical paths only), by representing them as a ZBDD and proceed with ISAT.
- Provide a seamless integration of these techniques into a specialized incremental SAT solver like MINISAT [20].

References

[1] K. Heragu, J.H. Patel, V.D. Agrawal, "Fast Identification of untestable delay faults using implications", In *Proc. of ICCAD*, 1997, pp. 642-647.

[2] Z.C. Li, R.K. Brayton, Y. Min, "Efficient Identification of Non-Robustly Untestable Path Delay Faults", In *Proc. of ITC*, 1997, pp. 992-997.

[3] Y. Shao, S.M. Reddy, S. Kajihara and I. Pomeranz, "An Efficient Method to Identify Untestable Path Delay Faults", In *Proc. of ATS*, 2001, pp. 233-238.

[4] K. Fuchs, M. Pabst and T. Rossel, "RESIST:A Recursive Test Pattern Generation Algorithm for Path Delay Faults considering Various Test Classes", In *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 13, No. 12, 1994, pp.1550-1562.

[5] Y. Shao, I. Pomeranz and S.M. Reddy "Path delay fault test generation for standard scan designs using state tuples", In *Proc. of DAC*, 2002, pp. 767-772.

[6] D. Battacharya, P. Agrawal, V.D. Agrawal, "Test Generation for Path Delay Faults using Binary Decision Diagrams", In *IEEE Trans. on Computers*, Vol. 44, No.3, 1995, pp.434-447.

[7] S. Padmanaban and S. Tragoudas, "Using ZBDDs and BDDs for efficient identification of testable path delay faults", In *Proc. of DATE*, 2004, pp. 50-55.

[8] C. A. Chen and S. K. Gupta, "A Satisfiability-Based Test Generator for Path Delay Faults in Combinational Circuits", In *Proc. of DAC*, 1996, pp. 209-214.

[9] J. Kim, J. Whittemore, J.P. Silva and K. Sakallah, "On Applying Incremental Satisfiability to Delay Fault Testing", In *Proc. of DATE*, 2002, pp.380-384.

[10] M.K. Ganai *et al.*, "Combining strengths of circuit-based and CNF-based algorithms for a high performance SAT solver", In *Proc. of DAC*, 2002, pp. 747-750.

[11] F. Lu, L.C. Wang, K.T. Cheng, R. C. Huang, "A Circuit SAT solver with Signal Correlation Guided Learning", In *Proc. of DATE*, 2003, pp. 892-897.

[12] R. Arora and M.S. Hsiao, "Enhancing SAT based Equivalence Checking using static logic implications", In *Proc. of HLDVT*, 2003, pp. 63-68.

[13] K.T. Cheng and H. C. Chen, "Classification and Identification of Nonrobust Untestable Path Delay faults", In *IEEE Trans. CAD Integrated Circuits and Systems*, Vol. 15, No. 8, 1996, pp. 843-853.

[14] L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT solver", In *Proc. of DAC*, 2001, pp. 530-535.

[15] J.K. Zhao, E.M. Rudnick and J.H. Patel, "Static Logic Implications with application to redundancy identification", In *Proc. of VTS*, 1997, pp. 288-293.

[16] B. Li, M. S. Hsiao and S. Sheng, "A novel SAT all-solutions solver for efficient preimage computation", In *Proc. of DATE*, 2004, pp. 272-277.

[17] L. Zhang and S. Malik, "Validating SAT solvers Using an Independent Resolution-Based Checker: Practical Implementations and other Applications", In *Proc. of DATE*, 2003, pp. 880-885.

[18] E. Goldberg and Y. Novikov, "Verification of Proofs of Unsatisfiability for CNF Formulas", In *Proc. of DATE*, 2003, pp. 886-891.

[19] S. Malik, "Boolean Satisfiability Research Group at Princeton", http://www.princeton.edu/ chaff/zchaff.html.

[20] Niklas Een and Niklas Sorensson, "An Extensible SAT solver", In *Proc. of SAT*, 2003.