Compact Test Generation for Non-Robustly Testable PDFs

Maria K. Michael
ECE Department
University of Cyprus

Spyros Tragoudas
ECE Department
University Southern Illinois

ABSTRACT
We introduce a new Automatic Test Pattern Generation (ATPG) methodology for compact generation of test sets, to detect non-robustly testable path delay faults in combinational and fully enhanced scanned circuits. The proposed framework is non-enumerative with respect to the faults examined, and relies on the appropriate formulation and generation of functions that are used to derive the desired test set. Each generated function targets many faults and, implicitly, maintains a very large set of tests for the targeted faults. Such function-based approaches allow for further compaction using either static or dynamic test set compaction methods. One such dynamic test set compaction heuristic is also presented here. We evaluate the performance of the proposed methodology in terms of test efficiency. The reported results on a variety of benchmarks indicate that the method is very promising.

KEY WORDS
ATPG, delay test, delay faults, path delay faults, non-robust test, test compaction, test efficiency.

1 Introduction

Under the Path Delay Fault (PDF) model ([5], [16]), every fault is represented as a sequence of falling or rising transitions along a physical path, from a primary input to a primary output in the circuit. A PDF test consists of a pair of patterns $< v_1, v_2 >$, where v_1 is used to initialize and stabilize the targeted path to known values and v_2 is used to excite transitions on the path. Since the number of faults is, in the worst-case, exponential to the number of lines in the circuit, compact and non-fault enumerative ATPG is necessary for the PDF model.

A compact test set is also desirable for other reasons, including lower power dissipation and smaller test application time. We focus here on the problem of generating a small test set that provides high fault coverage, by explicitly avoiding fault enumeration. In this work, we measure the quality of the ATPG as the number of *single* path delay faults detected per generated test. We refer to this measure as the *test efficiency*.

There are two phases during the traditional test generation framework for path delay faults. The first one is the path sensitization phase which statically sensitizes a single PDF or a set of PDFs. Subsequently, during the line justification phase, the sensitized path(s) are tested provided that the remaining lines in the circuit are justified to desirable stable values or appropriate transitions.

A fault-by-fault (path-by-path) ATPG process is followed in traditional enumerative methods (see [1], [5], [9], [10], and [16], among others). To overcome the problem of examining all path delay faults in a circuit, many enumerative methods have proposed to only consider the longest topological paths (critical paths). Unfortunately, such restrictions remain enumerative because the examined paths in many circuits remain prohibitively many. Other approaches suggest not examining paths but instead segments. Such an approach was proposed in [7]. In the strict sense of the definition, this approach cannot be classified as path-enumerative since it does not proceed explicitly on a fault-by-fault basis. However, it does not guarantee a polynomial bound on the number of examined segments and, thus, the approach is practically enumerative.

To tackle the problem of targeting a huge number of faults on a path-by-path basis, [8] and [13] proposed non-enumerative ATPG approaches. Both approaches are using graph theoretic arguments and are building on top of structural-based fault propagation methods along selected paths in the circuit. In particular, they use graph theoretic arguments to simultaneously assign transitions to many paths during the path sensitization phase in order to identify subcircuits whose paths can be mutually sensitized. Unfortunately, the fault coverage from both of these methodologies is very low. Their test efficiency is also quite low. More importantly, none of these methods addresses scalability. In our context, we refer to scalability as the ability of the approach to maintain the test efficiency as the number of generated tests increases.

We use function-based techniques to generate the desired tests. This is a major difference between the proposed method and the non-enumerative approaches in [8] and [13] which use structural-based methods. Function-based ATPG methods for PDFs have also been proposed in [1], [4], and [6], but all these approaches are fault enumerative. The proposed method derives and manipulates boolean functions, we call them *test functions*, that guarantee the detection of several path delay faults at a time. We maintain and manipulate test functions using Binary Decision Diagrams (BDDs) [3].

Other procedures that explicitly target the generation of compact test sets for PDFs, but in an enumerative manner, were proposed in [2], [14], and [15]. The test compaction procedure of [2], as well as the most recent one included in [14], is using the concept of primary and secondary target faults. The level of compaction in both of these techniques depends greatly on the selection order of the primary and secondary faults. A slightly different concept is used in [15]. A maximal set of potentially compatible faults is derived by identifying a set of faults that can be tested together with some fault that has been arbitrarily selected. Even though they may not target all faults explicitly, the methods in [2], [14], and [15] are still enumerative in nature since they are based on the principle of first targeting a single fault and then attempting to find one or more faults that can be tested mutually with the original fault.

We propose a new non-enumerative ATPG methodology for non-robustly testable PDFs. A main characteristic of non-robust tests is that they can detect excessive delays on paths due to the propagation of hazards. A pair of vectors $< v_1, v_2 >$ is a non-robust test for one or more PDFs if v_2 statically sensitizes the targeted PDF(s) [10], [16]. There are no requirements on v_1 other than assigning the appropriate values on the primary inputs that are on the targeted PDF(s). Thus, ATPG for non-robust tests is centered in generating v_2. We note that the proposed approach could be used to also generate other types of tests, such as robust and functional (see [5], [9], and [10] for details on these or similar definitions), but it becomes considerably more complex. Generation of these types of tests will be the focus of future work.

The approach consists of a linear number (to the number of primary inputs) of topological circuit traversals. Specifically, there are two traversals per primary input, one per possible type of transition on the input. A list of a constant number of test functions is kept per circuit line during each circuit traversal. Each function is guaranteed to detect many segments (subpaths) sensitized from a specified primary input up to the line. The number of functions kept at a line is explicitly bounded to a small constant so that the non-enumerative property of the method is guaranteed. When a circuit traversal is completed, tests that detect several path delay faults originating from some primary input are generated. The tool is also designed such that the test efficiency is maintained at satisfactory levels as the number of generated tests increases by introducing a new compaction routine applied per generated list of test functions.

The rest of the paper is organized as follows. Section 2 presents basic terminology and describes the proposed ATPG method. Section 3 presents the new dynamic compaction routine that is incorporated in the ATPG framework, and Section 4 discusses how to condition the ATPG tool to run iteratively for additional fault coverage. Experimental results are presented in Section 5. Section 6 concludes.

2 The ATPG method

A circuit C is represented as a directed graph, denoted by G. The subgraph of G induced by primary input I is denoted by

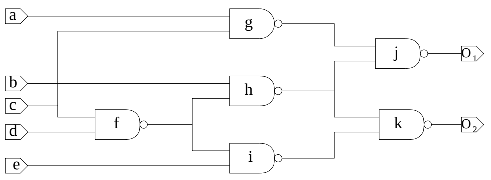

Figure 1. Circuit C17 (C)

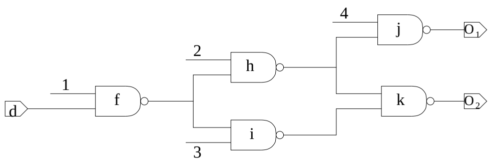

Figure 2. Subcircuit corresponding to G_d of C

G_I, which also contains all the lines of C that are independent of I but immediately drive some node in G_I. We call these lines the $supporting\ points$ of G_I. Figure 1 shows circuit C17 from the collection of the ISCAS'85 benchmark circuits, and Figure 2 shows the corresponding subcircuit represented by the subgraph G_d. Observe that G_d also contains lines $\{1, 2, 3, 4\}$ of C, which are the supporting points of G_d.

An input signal is either an *on-input* (on the targeted PDF) which assumes a certain transition to be propagated or an *off-input* (off the targeted PDF) which assumes a value to be justified. The controlling and non-controlling values of a gate g are denoted by $cv(g) \in \{0, 1\}$ and $ncv(g) \in \{0, 1\}$, respectively. A transition is designated by $tr \in \{r, f\}$, where $r = rising$ and $f = falling$. The $positive$ $(negative)$ $cofactor$ of a boolean function f with respect to variable x is denoted by f_x $(f_{\overline{x}})$, where $f_x = f_{|x=1}$ $(f_{\overline{x}} = f_{|x=0})$.

The proposed ATPG method consists of a linear number of topological circuit traversals. A primary input I with a transition type $tr \in \{r, f\}$ is considered per traversal. When considering some primary input I we first derive the corresponding G_I subcircuit. For each line in G_I, a list of functions is generated. Each function corresponds to the constraints that the non-robust test must satisfy in order for one or more PDFs to be detected from input I to the specific line. We refer to these functions as *test functions*. The test generation process for G_I terminates when a list of test functions is generated at each primary output of G_I. The test functions generated at a primary output O guarantee the detection of a number of PDFs from input I (for a specified transition on I) to output O.

Let $F_i()$ denote a test function for some line i, tr_i the type of transition on line i, and p_i the number of detected PDFs from a primary input I to line i. We maintain a vector $(F_i(), tr_i, p_i)$ to store this information. The transition on the targeted primary input I is denoted as $tr_I : t_1 \rightarrow t_2$, where $t_1, t_2 \in \{0, 1\}$. The functionality of a line i, expressed with respect to the primary input variables, is denoted by $f_i()$.

The non-robust test function formulation at some gate g with output line i is shown below:

$$F_i = \prod_{j \in ON(g)} F_j() \cdot \prod_{j \in OFF(g)} \left(f_j^{ncv(g)}() \right)_{|I=t_2} \cdot \prod_{j \in SP(g)} f_j^{ncv(g)}() \tag{1}$$

$ON(g), OFF(g)$, and $SP(g)$ denote the set on on-input, off-input, and supporting lines of gate g, respectively. Functions $f_j^{ncv(g)}()$ and $f_j^{cv(g)}()$ are used to denote the normal and complemented forms of function $f_j()$ depending (i) on the type of gate that line j is driving and (ii) whether line j is to be set to a controlling or non-controlling value. Precisely:

$$f_j^{ncv(g)}() = \begin{cases} f_j() & , g \in \{AND, NAND\} \\ \overline{f_j}() & , g \in \{OR, NOR\} \end{cases}$$

$$f_j^{cv(g)}() = \begin{cases} \overline{f_j}() & , g \in \{AND, NAND\} \\ f_j() & , g \in \{OR, NOR\} \end{cases}$$

The two pattern test is extracted from the test function $F_i()$. The input values for the second time frame (v_2) is fixed according to a selected minterm from the test function $F_i()$. The values for all inputs (other than the targeted input I whose values are determined by tr_I) for the first time frame (v_1) are actually don't cares (either 0 or 1). Since our goal is to generate compact test sets with as high fault coverage as possible, we extract a largest-size cube from the test function. This allows for a large number of inputs to be assigned transitions (these are inputs that are not fixed by the extracted cube) that may end up detecting additional PDFs. What is important to observe here is that a test extracted from a test function $F_i()$ will always cover at least as many as p_i PDFs.

Maintaining a list of test functions at each line is very critical to the performance of the proposed approach. Each function at a line detects a number of PDFs from a primary input to the line. Thus, a function at this point contains a collection of *potential* tests that may become invalid in future steps when additional constraints must be satisfied. For this reason, it is extremely important to generate several such functions. The list of functions for some line i can be formed by considering all the possible ways that the functions in the list of each of the immediate predecessors of i can be combined. A combination can be created by selecting a subset of the immediate predecessors of i to be the on-inputs. The remaining inputs are off-inputs. The supporting points (if any) are always viewed as off-inputs since no explicit sensitization is required through them.

Since the number of test functions per line can be prohibitive, the number of functions kept at a line is explicitly bounded to a small constant so that the non-enumerative property of the method is guaranteed. Thus, an intelligent decision must be made on how to select appropriate functions. We set bounds on the number of *considered* and the number of *selected* functions. The selection is made from the list of considered functions based on the number of PDFs detected per satisfiable function. Thus, those functions that sensitize the least number of paths are not selected.

3 Collapsing of Test Function Lists

The goal here is to further decrease the size of the generated test set by collapsing many lists of test functions into one, such that the total number of functions decreases while the total number of detected PDFs remains the same. Under the proposed framework, collapsing (also referred to as compaction) can be very effective since functions that implicitly represent many tests for a set of targeted PDFs are maintained. This gives us greater flexibility for compaction compared to traditional structural-based ATPG methods that generate one test per fault(s). In many structural-based ATPGs the generated test contains don't care values (thus, referred to as an incompletely specified test), which in turn can be expanded to represent many tests for the same fault(s). This is still limiting when compared to function-based ATPGs that can generate many incompletely specified tests per fault(s).

The motivation behind the compaction technique discussed in this Section is to further take advantage of the function-based ATPG framework. It is important to note here that compaction is inherent in the proposed method since tests are generated in a non fault enumerative manner. However, since the ATPG method presented in Section 2 produces a list of test functions per primary output (O) on a primary input (I) by primary input basis, a generated test will primarily target paths in an I/O subgraph. Intuitively, we can achieve a higher degree of compaction if we try to collapse the generated lists of test functions such that paths that span between different Os and Is can be tested together with a single test.

The flow of the proposed overall compaction scheme is illustrated in Figure 3. It follows a systematic levelized scenario based on dynamic programming. Assume a circuit with n primary inputs. At Level 1 we have the test function lists that are generated for every O, on an I-by-I basis, using the basic approach presented in Section 2. A generated test, up to this point, will detect PDFs within the specific I/O cone. At the next level (Level 2), the O lists for each I are compacted into a single list (we elaborate on how this compaction is actually performed at the end of this Section). These new lists are indicated by O_compact for each $I_i, i = 1 \ldots n$, in Figure 3. This step allows the generation of tests that detect PDFs between different Os (but still a single I).

An additional compaction step is performed for every I (other than I_1) to apply compaction between two O_compact lists. For example, Level 3 shows the resulting list after the O_compact lists for I_1 and I_2 have been compacted. In Level 4, the list from Level 3 and the O_compact list of I_3 are considered. This scenario continues until the list of input I_n is compacted with the list that was generated by compacting the lists for all previous $n-1$ inputs (Level n), to produce the final compacted list at Level $n+1$. Compaction at Levels $3 \ldots n+1$ is invoked once per level

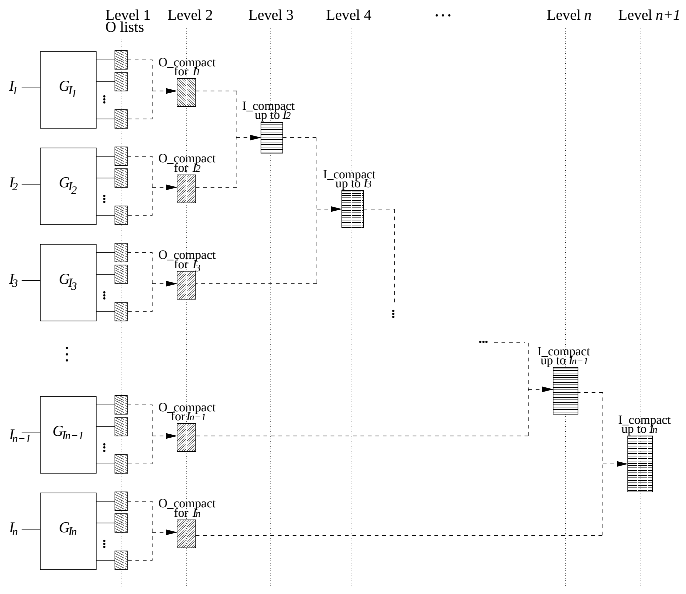

Figure 3. Flow of list compaction approach

whereas at Level 2 is invoked exactly n times (for each I_i). Thus, the compaction algorithm is performed $2n - 1$ times.

Observe that, as in the case of many compaction approaches [2, 14, 15], the compaction results are sensitive to the ordering on which the primary inputs are processed. Thus, a different order than the one shown in Figure 3 can lead to a final list with different number of test functions. However, the fault coverage remains the same among different input orderings.

The same list compaction algorithm is used for all levels. The input is two or more lists of test functions. Let $\mathcal{T}$ denote the total number of test functions (in all lists), and $\mathcal{P}$ denoted the total number of PDFs detected by the $\mathcal{T}$ functions. The desired output of the algorithm is a single compacted list that contains at most $\mathcal{T}$ test functions which still detect at least $\mathcal{P}$ PDFs. Two test functions are said to be *compatible* if their product (AND operation) is satisfiable. This implies that the PDFs detected by the two test functions can all be detected by a single test. The compaction algorithm follows a greedy scenario based on which it generates compatible functions iteratively. We start by assigning a unique identifier to each list. During the first iteration, functions from different lists are checked for compatibility pairwise (two at a time). We insist on selecting functions from different lists because the paths detected by each function are guaranteed to be different. Compatible functions are removed from their original lists, and their product is stored in a new list along with the original list identifiers. *Non-compacted* functions are those that are not compatible with any other function, and can be determined after all possible pairwise combinations are considered. Such functions are also stored in the new list and are marked as non-compacted so that future iterations will not consider them further. At the next iteration, the new list (generated during the previous iteration) is compacted in a similar manner. Again, only functions that were originally in different lists are allowed to be considered as a pair. This can be achieved by checking the saved list identifiers per function. The iterations continue until the size of the new list is 1 (all functions can be compacted into a single function) or is equal to the size of the list in the previous iteration. This way, it is guaranteed that no more than $\mathcal{T} - 1$ iterations will occur.

There is one extra operation that is necessary when checking for compatible functions that detect PDFs originating from different inputs (Levels $3 \ldots n + 1$). Assume two test functions $F_I()$ and $F_J()$, that detect PDFs from inputs I and J, respectively. Functions $F_I()$ and $F_J()$ have both been generated based on transitions $tr_I : tr_{I1} \rightarrow tr_{I2}$ and $tr_J : tr_{J1} \rightarrow tr_{J2}$ independently. In order to ensure that both transitions are taken into consideration when trying to compact the two functions into a single one, each of the test functions must be cofactored with respect to the desired transition on the other function. Thus, $F_{I\,|\,J=tr_{J2}}()$ and $F_{J\,|\,I=tr_{I2}}()$ are computed and the resulting collapsed function is the product of these two.

4 Fault Coverage Improvement using an Iterative method

When it is desired to increase the fault coverage (by generating additional tests), the approach described up to Section 3 can be repeated with slight modifications. Let all the activities described up to Section 3 constitute a single pass of the proposed methodology. The challenge here is to be able to condition the ATPG tool so that when a new pass is invoked it will try to target PDFs that have not been detected during the previous passes of the tool. This is a particularly difficult problem when PDF enumeration is not allowed. An iterative improvement scenario can be followed where the ATPG method is performed on several passes to generate tests that target new PDFs, while maintaining satisfactory test efficiency.

To assist to the above, some additional information is kept for every generated test function. Specifically, the identifiers of the on-input lines that are considered when forming the test function are kept. (Recall the discussion at the end of Section 2 on how these combinations are formed). This information is updated over all passes. Since we specifically limit the number of functions kept at each node and, in practice, the number of passes is kept to a small constant, this information is not space prohibitive.

These on-input line identifiers are checked when attempting to generate a new on-input combination for a test function. A higher priority is given to those combinations that were not considered in previous passes. We observe that, since we only maintain the *immediate* on-input line information, a combination that was considered during a previous pass does not necessarily target the exact same PDFs as the one considered in the current pass. Therefore, we do not drop such combinations, but we assign a lower priority to them.

5 Experimental Results

The approach was implemented in C language and run on a 900MHz SunBlade 1000 workstation. The package of [17] was used to generate the BDDs as well as the optimal initial ordering of variables that is also provided along with [17]. No variable reordering was necessary (even though it could have been invoked to provide with space savings, in the expense of additional CPU time). Fault coverages were obtained using the recent exact non-enumerative method in [12].

In our implementation, we restricted the on-input line combinations considered when generating a list of test functions per line to single and pairwise combinations. Up to 500 test functions were kept per function list. Since we are specifically interested in the generation of compact test sets, we primarily evaluate the proposed ATPG tool in terms of *test efficiency* (TE) per test, which is the number of PDFs detected by the test. The average test efficiency (TE_ave) of a test set is calculated by dividing the total number of PDFs detected by the total number of tests generated.

It is well known that certain functions cannot be represented efficiently by BDDs. For example, functions that express operations such as arithmetic multiplication have BDDs with exponential size regardless of the variable order [3]. We were able to create the BDDs for all the required functions for all the circuits besides c6288. Only the circuit induced by the first 12 primary outputs of c6288 can be represented efficiently by BDDs. In order to be able to target faults in the remaining part of the circuit (this part is the one induced by the remaining 20 primary outputs of c6288), we generate BDDs for simplified functions at the circuit lines instead of the original line functions. Simplified line functions are those obtained after preassigning a small cardinality subset $\mathcal{I}'$ of the input variable set $\mathcal{I}$ to either 0 or 1. The variables in subset $\mathcal{I}'$ are selected randomly. The only restriction we apply is that the cardinality of $\mathcal{I}'$ is equal or less than a small constant k. Once $\mathcal{I}'$ is selected, we assign the variables in $\mathcal{I}'$ to some combination of values and try to generate the BDDs that correspond to the simplified functions at the circuit lines. If the BDDs are constructed successfully, we proceed with the ATPG process. Observe that the ATPG process is performed on the original structure of the circuit, with the difference that the line functionalities are now modified (simplified) according to the selected values on the variables in $\mathcal{I}'$. Clearly, these simplified line functions do not contain any minterms that were not contained in the original line functions (after taking into consideration the fixed values on the variables in $\mathcal{I}'$). To be complete, the ATPG process must be repeated for all possible combinations of values for the variables in $\mathcal{I}'$, that is 2^k times. Up to this point, only PDFs that originate at any primary input $I \notin \mathcal{I}'$ are targeted. A second phase is required to target the remaining of the faults. In this second phase, a small cardinality subset $\mathcal{I}'' \in \mathcal{I} \setminus \mathcal{I}'$ is selected and the technique described for the first phase is applied again. Approximately 24% of the inputs of c6288 had to be preassigned, during each of the two phases, in order to construct the BDDs for the simplified line functions.

We present experimental results for the ISCAS'85 and ISCAS'89 full-scanned circuits. We note here that direct comparison with existing methods is not possible since all existing compact ATPG methods do not report results for non-robust tests. To our knowledge, we report the best, by far, TE_ave values for non-robust tests.

Table 1 lists the obtained results for five passes. For each pass, we list the number of tests generated (tests), the number of PDFs detected (PDFs), and the corresponding average test efficiency per circuit (TE_ave). Entries with "–" in Table 1 indicate that a 100% fault coverage was achieved in a previous pass and, therefore, no more passes were necessary. All non-robustly testable PDFs in circuits s298 and s386 were detected after the first pass. For the ISCAS'89, only circuits s713, s1196, and s1423 required a third pass, and the fourth and fifth passes were activated only for circuit s1423. Observe that the TE_ave per circuit is maintained at a satisfactory level (drops a little) as the number of passes increases.

In rows total_89 and total_85 (last two rows of Table 1) we show the total number of tests generated, the total number of faults detected, and the average number of faults per test for all the ISCAS'89 and ISCAS'85 circuits, respectively. To fairly compare the total TE_ave between the first and second pass for the ISCAS'89, we do not include circuits s298 and s386 in the total calculations for the first pass. (No totals are provided for passes 3, 4, and 5 for the ISCAS'89 since they were only required for very few circuit s1423.) The total TE_ave for the ISCAS'89 is almost the same for the first and second passes (it drops by 0.05% in the second pass). For the ISCAS'85, observe that even though the TE_ave per circuit always drops in the next pass (as it was expected), the total TE_ave appears to increase at the second pass. This occurs because the number of detected faults for c6288 is considerably larger than the sum of detected faults for all the remaining circuits, and, also, because the TE_ave decrease for c6288 from the first pass to the second is much smaller (1.69%) than in any other circuit.

Column 17 of Table 1 reports indicative results for the average CPU time per pass (in seconds), where ATPG time is the time required for the derivation of the test functions at the circuit lines, and Comp. time is the time required for the dynamic compaction method discussed in Section 3. The total execution time per pass (on the average) is the sum of ATPG and Comp. time.

6 Conclusions

A new function-based ATPG approach for non-robust tests that explicitly avoids the enumeration of paths, and can generate small tests sets that have high fault coverage, is presented. The proposed framework is especially attractive for dynamic test-set compaction, since a large number of

Table 1. Results for the ISCAS’85 and ISCAS’89 benchmark circuits.

Circuit	1st pass			2nd pass			3rd pass			4th pass			5th pass			CPU (secs)
	tests	PDFs	TE_ave	tests	PDFs	TE_ave	tests	PDFs	TE_ave	tests	PDFs	TE_ave	tests	PDFs	TE_ave	ATPG / Comp.
s298	64	364	5.68	-	-	-	-	-	-	-	-	-	-	-	-	39 / 5
s344	94	627	6.33	102	632	6.19	-	-	-	-	-	-	-	-	-	56 / 7
s349	90	619	6.87	97	634	6.53	-	-	-	-	-	-	-	-	-	55 / 7
s382	113	717	6.35	118	734	6.22	-	-	-	-	-	-	-	-	-	55 / 8
s386	101	414	4.10	-	-	-	-	-	-	-	-	-	-	-	-	31 / 8
s400	102	729	7.15	107	755	7.05	-	-	-	-	-	-	-	-	-	58 / 8
s420	275	840	3.05	310	932	3.01	-	-	-	-	-	-	-	-	-	92 / 8
s444	89	574	7.17	83	586	7.06	-	-	-	-	-	-	-	-	-	56 / 9
s526	111	719	6.47	116	720	6.21	-	-	-	-	-	-	-	-	-	56 / 8
s641	161	2089	12.97	181	2270	12.54	-	-	-	-	-	-	-	-	-	133 / 19
s713	224	4577	20.43	234	4706	20.11	259	4922	19.00	-	-	-	-	-	-	296 / 16
s820	199	961	4.83	209	984	4.71	-	-	-	-	-	-	-	-	-	55 / 12
s832	201	965	4.80	210	996	4.74	-	-	-	-	-	-	-	-	-	55 / 12
s953	332	2212	6.66	361	2312	6.40	-	-	-	-	-	-	-	-	-	55 / 15
s1196	379	3177	8.38	435	3534	8.12	477	3759	7.88	-	-	-	-	-	-	77 / 14
s1238	371	3368	9.08	416	3684	8.85	-	-	-	-	-	-	-	-	-	78 / 14
s1423	1789	29824	16.67	2211	35857	16.21	2396	37148	15.50	2524	38111	15.01	2637	38501	14.60	967 / 106
c880	579	9545	16.48	820	10744	13.90	1134	11592	10.22	1343	13190	9.82	1573	14539	9.24	719 / 88
c1355	4339	401158	92.45	5712	484985	84.91	6811	535021	78.55	7999	575891	70.00	9859	611605	62.04	6164 / 204
c1908	2283	86848	38.04	3704	115814	31.27	4946	133583	27.01	6127	148421	24.22	8635	160470	18.58	5214 / 853
c2670	644	62426	96.93	1047	73795	70.48	1497	79465	53.08	2179	82544	37.88	2822	85845	30.42	1302 / 999
c3540	1317	389064	295.42	2024	478681	236.5	2574	529824	205.84	3102	562692	181.40	3589	589390	164.22	5074 / 2105
c5315	841	164607	195.73	1397	206664	147.93	3163	226184	71.51	3780	239728	63.42	4247	245401	57.78	1724 / 1011
c6288	1001	$11.2\text{x}10^6$	11189	1991	$21.9\text{x}10^6$	10999	2713	$28.0\text{x}10^6$	10321	3512	$34.6\text{x}10^6$	9852	5101	$45.9\text{x}10^6$	8998	2020 / 491
c7552	2950	215649	73.10	4181	238933	57.15	5223	249422	57.15	6412	257123	40.10	7711	265888	34.48	1959 / 1699
total_89	4530	51998	11.48	5190	59336	11.43	-	-	-	-	-	-	-	-	-	
total_85	13954	12529297	897.9	20876	23509616	1126.15	28061	29765091	1060.73	34454	36479589	1058.79	43537	47873138	1099.59	

tests for sets of PDFs are implicitly maintained in the form of Boolean functions. The test efficiencies obtained from the experimental results for a variety of benchmark circuits demonstrate that the proposed framework is very promising.

References

[1] D. Bhattacharya, P. Agrawal and V. D. Agrawal, “Test Pattern Generation for Path Delay Faults using Binary Decision Diagrams”, *IEEE Trans. on Computers*, Vol. 44, No. 3, pp. 434-447, March 1995.

[2] S. Bose, P. Agrawal, and V. D. Agrawal, “Generation of compact delay tests by multiple path activation”, *Proc. ITC*, pp. 714-723, 1993.

[3] R. Bryant, “Graph-based algorithms for boolean function manipulation”. *IEEE Trans. on Computers*, Vol. C-35, No. 8, pp. 677-691, August 1986.

[4] C. A. Cheng and S. K. Gupta, “Test generation for path delay faults based on satisfiability”, *Proc. DAC*, 1996.

[5] K. T. Cheng and H. C. Chen, “Classification and Identification of Nonrobust Untestable Path Delay Faults”, *IEEE Trans. on CAD*, Vol. 15, pp. 845-853, August 1996.

[6] R. Drechsler, “BiTes: A BDD based test pattern generator for strong robust path delay faults”, *Proc. EDTC* pp. 322-327, 1994.

[7] K. Fuchs, M. Pabst, and T. Roessel, “RESIST: A Recursive Test Pattern Generation Algorithm”, *IEEE Trans. on CAD*, Vol. 13, No. 12, pp. 1550-1561, December 1994.

[8] D. Karayiannis and S. Tragoudas, “A Fast Non-enumerative Automatic Test Pattern Generator for Path Delay Faults”, *IEEE Trans. on CAD*, Vol. 18, No. 7, pp. 1050-1057, July 1999.

[9] A. Krstic, K. T. Cheng, *Delay Fault Testing for VLSI Circuits*, Kluwer Academic Publishers, Boston, MA, 1998.

[10] C. J. Lin and S. M. Reddy, “On delay fault testing in logic circuits”, *IEEE Trans. on CAD*, Vol. CAD-6, No. 5, pp. 694-703, September 1987.

[11] M. Michael and S. Tragoudas, “ATPG for Path Delay Faults without Path Enumeration”, *Proc. ISQED*, pp. 384-389, March 2001.

[12] S. Padmanaban, M. Michael, and S. Tragoudas, “Exact Path Delay Fault Coverage with Fundamental Zero-Suppressed BDD Operations”, *IEEE Trans. on CAD*, March 2003.

[13] I. Pomeranz, S. M. Reddy and P. Uppaluri, “NEST: A Non-enumerative Test Generation Method for Path Delay Faults in Combinational Circuits”, *IEEE Trans. on CAD*, Vol. 14, No. 12, pp. 1505-1515, December 1995.

[14] I. Pomeranz, and S. M. Reddy, “Test Enrichment for Path Delay Faults Using Multiple Sets of Target Faults”, *IEEE Trans. on CAD*, Vol. 22, No. 1, pp. 82-89, January 2003.

[15] J. Saxena and D.K. Pradhan, “A method to derive compact test sets for path delay faults in combinational circuits”, *Proc. ITC*, pp. 724-733, 1993.

[16] G. L. Smith, “Model for Delay Faults Based upon Paths”, *Proc. ITC*, pp. 342-349, November 1985.

[17] F. Somenzi, “CUDD: CU Decision Diagram Package”, ECE Dept., The University of Colorado at Boulder, release 2.3.0, 1999.

[18] P. Tafertshofer, A. Ganz, and K. J. Antreich, “IGRAINE–An Implication GRaph-bAsed engINE for Fast Implication, Justification, and Propagation”, *IEEE Trans. on CAD*, Vol. 19, No. 8, pp. 907-927, August 2000.