JOURNAL OF ELECTRONIC TESTING: Theory and Applications 21, 495–502, 2005

Functional Fault Equivalence and Diagnostic Test Generation in Combinational Logic Circuits Using Conventional ATPG

ANDREAS VENERIS
Department of Electrical and Computer Engineering and Department of Electrical and Computer Science, University of Toronto, Toronto, ON M5S 3G4, Canada
veneris@eecg.toronto.edu

ROBERT CHANG
Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada
robert.chang@utoronto.ca

MAGDY S. ABADIR
Freescale Semiconductor, Inc., Austin, TX 78729, USA
m.abadir@freescale.com

SEP SEYEDI
Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada
sep@eecg.toronto.edu

Received September 20, 2003; Revised March 28, 2004

Editor: A.J. van de Goor

Abstract. Fault equivalence is an essential concept in digital design with significance in fault diagnosis, diagnostic test generation, testability analysis and logic synthesis. In this paper, an efficient algorithm to check whether two faults are equivalent is presented. If they are not equivalent, the algorithm returns a test vector that distinguishes them. The proposed approach is complete since for every pair of faults it either proves equivalence or it returns a distinguishing vector. The advantage of the approach lies in its practicality since it uses conventional ATPG and it automatically benefits from advances in the field. Experiments on ISCAS'85 and full-scan ISCAS'89 circuits demonstrate the competitiveness of the method and measure the performance of simulation for fault equivalence.

Keywords: VLSI, test generation, diagnostic test generation, fault

1. Introduction

Computing the complete set of fault equivalence classes in a combinational circuit is a classic problem in digital circuit design [10]. Two faults are *functionally equivalent* (or *indistinguishable*) if no input test vector can distinguish them at primary outputs. Functional fault equivalence is a relation that allows faults in a circuit to be collapsed into disjoint sets of *equivalent fault classes*.

There are many reasons why equivalent fault classes are important to a designer. Fault diagnosis and diagnostic Automated Test Pattern Generation (ATPG) benefits from this knowledge. The effectiveness of diagnosis depends on its resolution, that is, its ability to identify a small set of lines that contain fault(s) [5]. A shorter fault list is typically preferred by the test engineer who probes the circuit to search for the failure. Fault equivalence information may aid diagnosis since only a single fault from each class needs to be identified [10]. Additionally, knowledge of fault equivalence classes and distinguishing input test vectors improves diagnosis by building compact fault dictionaries [10].

Another use of fault equivalence is in the field of testability analysis. By definition, each vector that detects a fault in an equivalence class is guaranteed to detect all faults in the same class since their *detectability* is the same. In turn, this information can be used by testability enhancement measures such as observation point and control point calculations [10]. Finally, fault equivalence is important in logic optimization. Existing tools optimize a design through an iterative sequence of logic rewiring transformations that reduce power, increase performance, etc [4,11,15]. It has been shown [4, 15], that logic rewiring operations can be modeled with a set of equivalent stuck-at faults. Therefore, knowledge of fault equivalence may improve design rewiring algorithms.

Methods to compute fault equivalence are classified as *structural* and *functional* [10]. Structural methods operate on the circuit graph to identify fault equivalence. These methods are fast but they have pessimistic results since they operate on fan-out free circuit regions only. Functional fault equivalence methods are more expensive but they typically identify more classes [1,2,7,9,12]. Some of these methods [1,2,12] use logic implications and/or dominator information to prove equivalence. Since identifying logic implications is NP-hard [11], these methods do not utilize the complete set of logic implications and they may not return the complete set of fault equivalence classes. Other functional methods [7,9] develop ATPG specific engines to check fault equivalence. In theory, these engines may be able to prove the equivalence of any pair of faults but they require significant engineering effort to implement and update with the rapid advances in ATPG.

In this paper, we present an efficient method that proves the equivalence of a pair of stuck-at faults. The method also returns a distinguishing vector if the faults are proven to be not equivalent. To simplify the discussion, we use the term "fault" to indicate a single stuck-at fault hereafter. To prove *fault equivalence* or perform *Diagnostic Automated Test Pattern Generation (DATPG)*, the method performs a sequence of simulation-and ATPG-based steps. It should be noted, ATPG is not exclusive to the method and other test generation and redundancy checking techniques (BDDs, SAT solvers etc) can be utilized. However, our selection of conventional ATPG as the underlying test generation engine comes from the fact that it has been shown [15] to be a computationally efficient platform for this type of problem.

The proposed methodology, originally presented in [3, 16], has a number of characteristics that make it both efficient and practical. Unlike methods that alter existing ATPG tools [7, 9], it uses conventional ATPG [6, 11] and a novel hardware construction to either prove equivalence of the fault pair or return a distinguishing vector. Therefore, it remains straightforward to implement in an industrial environment and it automatically benefits from potential advances in the field of ATPG.

When compared to non-ATPG based techniques [1, 2, 10, 12], the proposed approach is *complete*. In other words, for every fault pair it guarantees to prove equivalence or return a distinguishing vector, provided completeness of the underlying test generation engine. Although existing techniques may not be complete, they are computationally efficient because they use structural information to find implications and prove fault equivalence. This structural information is usually collected in linear time. Therefore, these methods can be used as pre-processing to the proposed approach to screen some fault pairs and improve overall run-time.

An extensive suite of experiments confirm the competitiveness of the approach when compared with existing DATPG tools [7]. The completeness of the method also allows us to measure the performance of test vector simulation in fault equivalence and DATPG. The results indicate that a small set of random input test vectors provide sufficient resolution to the fault equivalence problem with relatively small error bounds.

The paper is organized as follows. Section 2 presents the two steps of the proposed functional fault collapsing and DATPG algorithm in terms of single stuck-at faults. Section 3 presents experiments and discusses future work. Section 4 concludes this work.

2. Fault Equivalence and DATPG

In this Section we present the *two* steps of the fault equivalence/DATPG method. The *first step* computes an *approximation* of the final set of fault equivalence classes using structural fault collapsing and input test vector simulation. Faults in the same class may or may not be equivalent, but faults in different classes are *guaranteed* to be not equivalent. The *second step* uses conventional ATPG and a novel hardware construction on pairs of faults in the same class as computed in Step 1 to either formally prove the faults are equivalent or perform DATPG.

Since Step 2 computes the exact set of fault equivalence classes, it is of interest to know the *quality* of test vector simulation (Step 1) for fault equivalence and DATPG. Experiments presented in Section 3 suggest that usually a relatively small set of test vectors provides sufficient resolution to the problem. We now describe the procedures in detail.

2.1. Parallel Vector Simulation

The implementation starts with structural fault collapsing [10] to prove faults that are structurally proximal as equivalent so that only one representative fault from each such set needs be considered in later steps of the algorithm. Let set F be the complete set of representative faults. The faults in F are examined for equivalence by *Parallel Vector Simulation (PVS)*.

PVS is a simulation-based procedure that classifies these faults into potentially equivalence classes $F_1, F_2, \ldots, F_n$ with respect to an input test vector set T. PVS identifies two faults f_A and f_B as *potentially equivalent* and places them in the same class if and only if f_A and f_B give the exact same primary output responses for each test vector from T. Faults in different classes are guaranteed to be not equivalent since they have different responses for some vector(s) in T which also acts as a distinguishing vector(s) for these faults.

Pseudocode for PVS is given in Fig. 1. The input to PVS is a circuit C, the collapsed set of faults F and a set of input test vectors T. In experiments, T is a relatively small set of 100–1000 test vectors. This set of vectors consists of random vectors and vectors with high stuck-at fault coverage [8]. The output of PVS is a set of fault classes $F_1, F_2, \ldots, F_n$ such that two faults f_a and f_b are in the same class F_i if and only if they have the exact same responses for all vectors in T.

```
Parallel_Vector_Simulation(C, F, T)

(1) Simulate test vectors in T and create
    indexed bit-lists at every circuit line
(2) For every fault f s-a-v on line l do
(3)     fault_signature=0
(4)     set bit-list of l to value v
(5)     simulate at fan-out cone of l
(6)     update fault_signature
(7)     restore bit-lists at l fan-out cone
(8) Group faults with same signatures in
    same class F_i, i=1 ... n
```

Fig. 1. Pseudocode for PVS (Step 1).

At first, PVS simulates in parallel all test vectors in T and creates an indexed bit-list on every line l in the circuit as in [15] (Fig. 1, line (1)). The i-th entry of this bit-list for l contains the logic value of l when the i-th input test vector is simulated. Since the test set contains only well defined logic values (0 and 1), these bit-lists are conveniently stored as arrays of single 32-bit `unsigned long int` values.

Next, for every stuck-at v fault $f \in F$ on line l, value v is injected on l and simulated at the fan-out cone of l. The primary output bit-lists are treated as integers and added to produce the *signature* of fault f for test set T (lines 2–6). Once the algorithm computes the signature of f, bit-list values are restored at the fan-out cone of l in line 7. This process is repeated for every fault in F. Faults that have the exact same signatures are grouped together in line 8 and PVS terminates. Signature matching is implemented efficiently with the use of hash tables.

2.2. Fault Equivalence and DATPG

Step 1 of the algorithm is a fast procedure that screens the initial set of faults F into a set of potentially fault equivalent classes $F_1, F_2, \ldots, F_n$. As explained earlier, faults from the same class *may* or *may not* be equivalent.

Given a pair of faults $(f_A, f_B) \in F_i$, Step 2 performs a simple hardware construction and employs conventional ATPG to formally prove their equivalence or return a test vector that distinguishes them (DATPG). It should be noted, formal fault equivalence or DATPG is performed in a *single (atomic) step* for each pair of faults. As it is shown, if the ATPG tools exhausts the solution space and proves a redundancy, then the two fault are formally proven to be equivalent. On the other

hand, if it returns with a test vector(s), this is also a vector that distinguishes the faults. We now outline the theory and implementation of this step in more detail.

Assume, without loss of generality, faults f_A s-a-0 and f_B s-a-1 on lines l_A and l_B of a circuit. To examine their equivalence, the algorithm attaches two multiplexers, MUX_A and MUX_B, with common select line S. The 1-input of MUX_A is line l_A while its 0-input is connected to constant logic 0. Intuitively, logic 0 indicates a s-a-0 fault on l_A. Similarly, the 0-input of MUX_B is the original line l_B and the 1-input is a constant 1.

This construction allows us to simulate the original circuit under presence of fault f_A when $S = 0$ and the original circuit under presence of f_B when $S = 1$. Therefore, if ATPG for select line S s-a-0[1] exhausts the solution space returning no test vector to report that the fault on S is redundant, then (f_A, f_B) is an equivalent fault pair [15]. In other words, the original circuit under the presense of each of the (f_A, f_B) faults behaves identically. Otherwise, the input test vector returned is a vector that distinguishes the two faults.

To illustrate this process, we need to employ Roth's 9-valued alphabet [13] with pairs of logic values taken from the set $\{0/0, 1/1, 0/1, 1/0, 0/X, X/0, 1/X, X/1, X/X\}$.

Pair of values indicate simulation of *two* faulty circuits; one under the presence of f_A and the other under the presence of f_B. If the stuck-at fault on select line S is redundant, it implies that no $0/1$ and no $1/0$ value propagate(s) to any primary output. In other words, the two faulty circuits produce the same response for *all* input test vectors and the two faults are indistinguishable.

On the other hand, if a single $0/1$ ($1/0$) difference is propagated to a primary output, then one (and only one) of the two faults is *guaranteed* to be detected. Which fault is detected depends on the logic simulation value; if logic simulation gives 0 then f_B (f_A) is detected at the primary output since circuit under presence of f_A (f_B) and logic simulation have identical values.

Similar reasoning shows that if logic simulation gives 1, then fault f_A (f_B) is detected. Both faults are detected if appropriate $0/1$, $1/0$ and simulation values propagate to different primary outputs. This is a desirable phenomenon because a single vector distinguishes both faults at different primary outputs. Such vectors can be utilized to compile compact fault dictionaries [10] and aid diagnosis [5].

The examples that follow illustrate the above procedure. The first example illustrates the case where

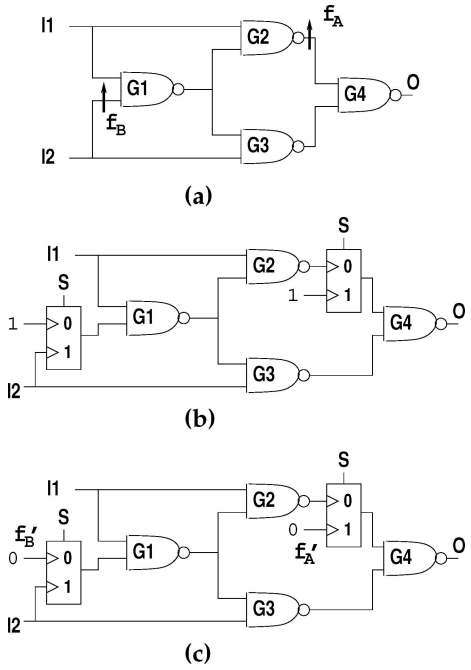

Fig. 2. Circuits for Examples 1 and 2.

the two faults are equivalent. The second example describes a situation where the faults are not equivalent and a distinguishing vector is returned.

Example 1: Consider the circuit in Fig. 2(a) and assume faults from the same class F_i $f_A = G_2 \rightarrow G_4$ and $f_B = I_2 \rightarrow G_1$ both s-a-1. To test their equivalence, we place two multiplexers, shown as boxes in Fig. 2(b), with common select line S. The 0-input to the first multiplexer is line G_2 and the 1-input of the multiplexer is 1 to represent the presence of a s-a-1 fault. Similarly, the 0-input of the second multiplexer is a logic 1 while I_2 feeds the other input. In both cases, the output of each multiplexer connects to the original output in the circuit. We observe, when $S = 0$, we operate on a circuit equivalent to the one in Fig. 2(a) under the presence of f_A and when $S = 1$ the circuit is equivalent to the one in Fig. 2(a) under the presence of f_B. ATPG for select line S s-a-0 declares that the fault is redundant. This indicates that the two circuits are functionally equivalent and confirms that (f_A, f_B) is an equivalent fault pair.

Example 2: Consider again circuit in Fig. 2(a) and faults $f_A' = G_2 \rightarrow G_4$ and $f_B' = I_2 \rightarrow G_1$ this time both stuck at logic 0. A similar multiplexer construction as in Fig. 2(b) gives circuit shown in Fig. 2(c). The difference is that a logic 0 is placed on appropriate

multiplexer inputs to indicate a stuck-at-0 fault. ATPG on common select line S s-a-0 returns test vector $(I_1, I_2) = (0, 0)$. This proves that the fault on S is not redundant and faults f'_A and f'_B are not equivalent. This is true since the test vector returned detects fault f'_A but does not even excite fault f'_B. In this case, the construction returns a distinguishing vector for fault pair (f'_A, f'_B).

Fig. 3 contains pseudocode for Step 2. For each class F_i $(i = 1 \ldots n)$, a representative f is randomly selected. For each other member $f' \in F_i$, we perform the multiplexer construction to check whether f and f' are equivalent (lines 4–5). If they are not equivalent, f' (and all other such non-equivalent faults from F_i) is placed in new class F_{n+1} (lines 6–9) which will be examined later and the distinguishing vector is recorded.

Observe, any such fault f' is guaranteed to be not equivalent with faults in any class $F_1, \ldots, F_{i-1}, F_{i+1}, \ldots, F_n$ by PVS. Faults placed in F_{n+1} may or may not be equivalent. Therefore, class F_{n+1} may get decomposed into new classes when it is examined later. Moreover, parallel fault simulation is performed with

```
Fault_Equivalence_DATPG(C, F_1, ..., F_n)

( 1) flag=0
( 2) for i=1 to n do
( 3)     randomly select f from F_i
( 4)     for every f' in Fi do
( 5)         perform the MUX construction
( 6)         if f' not equivalent to f do
( 7)             flag=1
( 8)             place f' in F_n+1
( 9)             store distinguishing vector
(10)      if flag=1
(11)         flag=0
(12)         n=n+1
```

Fig. 3. Fault equivalence and DATPG.

the distinguishing vector (line 9) for fault f' to identify any additional faults in F_i that are not equivalent. This may speed the process of decomposing F_i into new classes and it may save the user from invoking DATPG explicitly for these faults. We omit these details that can be found in [10].

The set of classes returned are the exact fault equivalence classes for circuit C and fault set F. DATPG

Table 1. Parallel vector simulation (Step 1).

ckt name	# of initial faults	faults after collaps	# ATOM vectors	# of fault classes after PVS				CPU time (sec)
				ATOM vectors	ATOM and 500 random	500 random	1000 random	
c432	798	419	110	371	417	413	418	0.11
c499	2434	1314	127	901	1076	1027	1092	0.59
c880	1770	940	133	857	889	853	868	0.17
c1355	2412	1302	192	1046	1088	1010	1079	0.89
c1908	1802	975	210	714	748	684	767	0.50
c2670	3177	1627	242	1141	1178	1160	1184	0.42
c3540	4116	2143	264	1408	1548	1541	1580	10.9
c5315	7042	3743	216	2971	3404	3381	3415	2.06
c6288	14303	7479	64	6397	6597	6593	6597	0.99
c7552	10081	5321	393	4186	4280	4180	4273	5.98
s820	1470	769	190	553	661	628	670	0.09
s1196	2409	1250	227	840	995	964	1030	0.29
s1238	2257	1185	240	740	929	896	953	0.29
s1494	2786	1458	191	1136	1314	1286	1348	1.22
s5378	6052	3254	358	2235	2521	2192	2552	1.70
s9234	18468	6927	660	3329	3558	3438	3611	878.30
s35932	48520	27106	129	24159	24364	24363	24364	62.20
s38417	76522	31024	1458	23293	23458	23460	23467	32921.00

can be invoked explicitly for classes not treated by PVS by using a similar multiplexer/ATPG construction to get test vectors that distinguish between these faults.

3. Experiments

We tested the proposed method on an Ultra 5 SUN workstation with 128 Mb of memory. The details of the ATPG engine employed can be found in [6, 11]. We use a low level 1 for recursive learning to provide a fair comparison with DIATEST [7]. Both algorithms are executed on the same workstation using the same set of circuits. Using this approach, the comparsion results accurately measure the competitiveness of the proposed approach.

We evaluate the proposed technique on ISCAS'85 combinational and full-scan ISCAS'89 sequential benchmarks optimized for area using `script.rugged` in SIS [14]. Test vectors with high stuck-at fault coverage (ATOM vectors) are computed as in [8]. Run-times reported are in seconds.

Table 1 contains information about PVS. The first column shows the circuit name and the second column has the total number of stuck-at faults which is roughly twice the number of lines, including branches. The third column shows the faults after structural fault collapsing [10]. We examine the performance of PVS with a set of random and stuck-at fault input test vectors. The number of ATOM vectors is shown in column 4. Columns 5–8 of Table 1 show the number of distinct fault classes upon termination of PVS for four different cases with respect to the test vector set T used: (i) ATOM vectors, (ii) ATOM vectors and 500 random vectors, (iii) 500 random vectors, and (iv) 1000 random vectors.

Intuitively, the more vectors we simulate the more accurate the results in terms of the final number of classes, as discussed earlier. A study of the numbers indicates that a relatively small set of random vectors (case (iv)) gives sufficient resolution and there is little to gain with a pre-computed set of stuck-at fault test vectors. This is also illustrated in Fig. 4 that depicts the number n of fault classes $F_1, F_2, \ldots, F_n$ versus the number of random vectors simulated. It is seen that,

Table 2. Fault equivalence and DATPG (Step 2).

	ATPG			DATPG				CPU comparison	
					faults detected		CPU (sec)		
ckt name	# pairs checked	# final classes	% err PVS	# pairs checked	one	both	fault equivalence	DATPG proposed	DIATEST [7]
c432	1	419	0.2	111	70	41	0.03	0.00	0.00
c499	17	1106	1.3	84	28	56	0.14	0.02	0.08
c880	104	892	2.7	128	45	83	0.02	0.01	0.01
c1355	18	1094	1.4	80	25	55	0.22	0.07	0.05
c1908	335	830	7.6	129	39	90	0.06	0.05	0.01
c2670	6203	1443	18.0	127	73	54	0.07	0.05	0.05
c3540	8914	1839	14.1	116	62	54	0.10	0.09	0.09
c5315	184	3480	1.9	129	42	87	0.11	0.09	0.09
c6288	892	6973	5.4	94	77	17	0.42	0.05	0.09
c7552	8768	4737	9.8	130	46	84	0.50	0.16	0.13
s820	869	754	11.1	82	57	25	0.06	0.00	0.01
s1196	5897	1214	15.1	63	49	14	0.10	0.01	0.06
s1238	8600	1147	16.9	70	60	10	0.09	0.00	0.14
s1494	481	1450	7.0	50	41	9	0.08	0.02	0.08
s5378	5398	2937	13.1	94	36	58	0.08	0.02	0.02
s9234	5090	4103	12.0	108	14	94	0.42	0.04	0.25
s35932	2742	24748	1.5	100	15	85	0.75	0.16	0.71
s38417	6419	29794	21.2	125	18	107	1.28	0.14	0.31

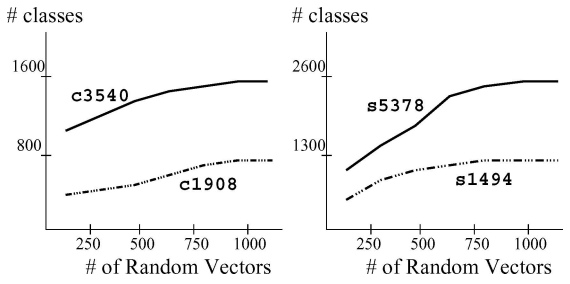

Fig. 4. Performance of PVS.

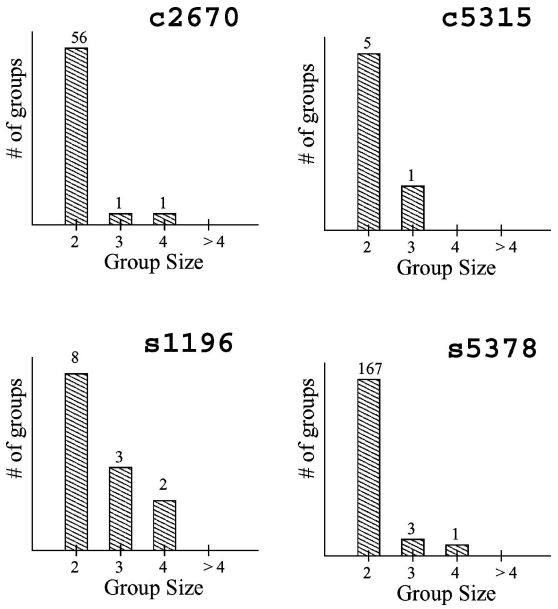

Fig. 5. Fault class distribution.

the number of fault classes converges with a relatively small number of vectors. We use the classes from case (iv) as input to Step 2. The last column of the table contains the total run-time for Step 1. These times can be improved if PVS is performed by re-using simulation results, as in critical path tracing [10], or if a compiled simulator is used.

Table 2 contains information that pertain to Step 2. Given fault pair (f_A, f_B), it tests whether the two faults are equivalent and returns a distinguishing vector if they are not. Columns 2–4 of Table 2 show values that pertain to the case when the faults are equivalent. The total number of fault pairs checked and the number of final (complete) fault classes are found in columns 2 and 3.

The relative error for PVS (Step 1), a simulation-based process, when compared to the formal engine of Step 2 is found in column 4. In many cases the relative error is rather small (less than 10%). This suggests that simulation of random vectors provides in most cases sufficient resolution to compute fault equivalence. Therefore, the designer is presented with a relatively small trade-off between time and accuracy.

DATPG results are found in columns 5–7. Column 5 contains the number of distinguishable fault pairs checked. Columns 6 and 7 contain the *manner* in which these faults are detected. Recall from Section 2.2, DATPG guarantees to detect one fault but it may detect both faults at *different* primary outputs. The numbers in these columns indicate that in as many as half of the cases, DATPG returns a test vector to distinguish *both* faults at different primary output (column 7). This may reduce the load for test vector compaction and the size of fault dictionaries and/or aid diagnosis [10].

Column 8 of Table 2 contains the average run-time for ATPG when the faults are equivalent (redundancy checking). Since ATPG tests pairs of faults in the same class, the less faults per class in the circuit after PVS, the less effort ATPG is expected to spend. This circuit-dependent property is depicted for four benchmarks in Fig. 5. The graph in that figure contains statistics on the size of the fault groups after Step 2 (excluding groups of size 1). We observe most benchmarks favor fault equivalence groups of size two or three.

Finally, the last two columns in Table 2 provide comparison results with DIATEST [7] for the same list of non-equivalent fault pairs. The comparison reveals that conventional ATPG and the hardware construction presented here provides an attractive alternative. In fact, larger values of implication learning [11] will speed up the ATPG tool and further improve the performance of the method. This confirms the practicality of the approach since it automatically benefits from advances in the field.

In the future, we intend to enhance PVS to re-use simulation results as in critical path tracing [10] and explore fault equivalence and DATPG for other fault types.

4. Conclusion

Fault equivalence is important in digital circuit design. A method for fault equivalence and diagnostic

ATPG using simulation-based and ATPG-based techniques was presented. It uses conventional ATPG and a simple hardware construction to prove equivalence or return distinguishing input test vectors. Experiments demonstrate its effectiveness and competitiveness.

Acknowledgments

The authors thank Dean W. K. Fuchs and Dr. E. Amyeen for their technical comments and support in this work.

Note

1. ATPG for select line *S* s-a-1 produces similar results.

References

1. M.E. Amyeen, W.K. Fuchs, I. Pomeranz, and V. Boppana, "Fault Equivalence Identification using Redundancy Information and Static and Dynamic Extraction," in *Proc. of IEEE VLSI Test Symposium*, pp. 124–130, 2001.
2. M.E. Amyeen, W.K. Fuchs, I. Pomeranz, and V. Boppana, "Implication and Evaluation Techniques for Proving Fault Equivalence," *Proc. IEEE VLSI Test Symp.*, 1999, pp. 201–207.
3. R. Chang, S. Seyedi, A. Veneris and M.S. Abadir, "Exact Functional Fault Collapsing in Combinational Logic Circuits," in *IEEE Latin American Test Workshop*, 2003, pp. 40–46.
4. S.C. Chang and M. Marek-Sadowska, "Perturb and Simplify: Multi-Level Boolean Network Optimizer," in *Proc. Int'l Conference on Computer-Aided Design*, 1994, pp. 2–5.
5. G. Fey and R. Drechsler, "Finding Good Counter-Examples to Aid Design Verification," in *ACM and IEEE Int'l Conference on Formal Method and Models for Codesign (MEMOCODE)*, 2003, pp. 51–52.
6. H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," in *IEEE Trans. on Computers*, vol. C-32, no. 12, December 1983.
7. T. Gruning, U. Mahlstedt, and H. Koopmeiners, "DIATEST: A Fast Diagnostic Test Pattern Generator for Combinational Circuits," in *Proc. Int'l Conf. on Computer-Aided Design*, 1991, pp. 194–197.
8. I. Hamzaoglu and J.H. Patel, "New Techniques for Deterministic Test Pattern Generation," in *Proc. of VLSI Test Symposium*, 1998, pp. 446–452.
9. I. Hartanto, V. Boppana, W.K. Fuchs, and J. H. Patel, "Diagnostic Test Pattern Generation for Sequential Circuits," *Proc. of IEEE VLSI Test Symp.*, 1997, pp. 196–202.
10. N. Jha and S. Gupta, *Testing of Digital Systems*, Cambridge University Press, 2003.
11. W. Kunz and D.K. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems—Test, Verification, and Optimization," in *IEEE Trans. on Computer-Aided Design*, vol. 13, no. 9, 1994, pp. 1143–1158.
12. A.V.S.S. Prasad, V.D. Agrawal, and M.V. Atre, "A New Algorithm for Global Fault Collapsing into Equivalence and Dominance Sets," in *Proc. IEEE Int'l Test Conf.*, 2002, pp. 391–397.
13. J.P. Roth, "Diagnosis of automata failures: A calculus & a method," *IBM Journal of Research Development*, vol. 10, pp. 278–291, June 1966.
14. E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. of Int'l Conference on Computer Design*, 1992, pp. 328–333.
15. A. Veneris and M.S. Abadir, "Design Rewiring Using ATPG," in *Proc. IEEE Trans. on Computer-Aided Design*, vol. 21, no. 12, 2002, pp. 1469–1479.
16. A. Veneris, R. Chang, M.S. Abadir and M. Amiri, "Fault Equivalence and Diagnostic Test Generation Using ATPG," in *IEEE Int'l Symposium on Systems and Circuits,* 2004.

Andreas Veneris received a Diploma in Computer Engineering and Informatics from the University of Patras in 1991, an M.S. degree in Computer Science from the University of Southern California, Los Angeles in 1992 and a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1998. In 1999 he joined the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto as an Assistant Professor. His research interests include CAD for synthesis, diagnosis and verification of digital circuits and systems.

Robert Chang received his B.A.Sc. degree in 2001 and M.Eng. degree in 2005 both in Computer Engineering from the University of Toronto. He is currently part of the software development team in iLogic Inc., Toronto, Canada.

Magdy S. Abadir received the B.S. degree in CS from the University of Alexandria, Egypt, an M.S. degree in CS from the University of Saskatchewan, Canada, and a Ph.D. in EE from the University of Southern California, Los Angeles.

Currently he is the manager for the Global Strategy, Tools and Methodology, NCSG, Freescale Semiconductor. Prior to that he was the group manager for tools and methodologies of Motorola's high-performance design group in Austin, Texas. Prior to joining Motorola he was the General Manager of Best IC Labs, and senior member of the technical staff at MCC. Technical interests include EDA tools, delay test, Test economics, Microprocessor test and verification.

Sep Seyedi received a Bachelor's in Electrical and Computer Engineering from the University of Toronto in 2004. He is currently pursuing his Master's degree at the same university in AI and robotics.