Incremental Diagnosis and Correction of Multiple Faults and Errors

Andreas Veneris, J. Brandon Liu, Mandana Amiri
University of Toronto
Department of ECE
Toronto, ON M5S 3G4
{veneris, liuji, amiri}@eecg.utoronto.ca

Magdy S. Abadir
Motorola
7700 W. Parmer
Austin, TX 78729
m.abadir@motorola.com

Abstract

An incremental simulation-based approach to fault diagnosis and logic debugging is presented. During each iteration of the algorithm, a single suspicious location is identified and fault modeled such that the functionality of the new design becomes "closer" to its specification. The method is based on a simple and, at a first glance, counter-intuitive theoretical result along with a number of heuristics which help avoid the exponential complexity inherent to the problems. Experiments on multiple design errors and multiple stuck-at faults confirm its effectiveness and accuracy, which scales well with increasing number of errors.

1. Introduction

With the transistor size continuing to shrink and the chip density and complexity to rise, errors and faults are more likely to occur during chip design and manufacturing. Although testing prevents a malfunctioning product from being shipped, *failure analysis* [4] is an equally important process as it helps tune the manufacture and/or design cycle accordingly to improve the product yield. An integral part of fault analysis is the identification of failing portions of the chip, known as **fault diagnosis**. Since the quality of fault analysis is tightly coupled with the resolution of fault diagnosis, the research community has shown significant interest in the development of efficient yet accurate fault diagnosis algorithms [4] [9] [12] [11].

Design Error Diagnosis and Correction (DEDC) is a similar problem but it occurs at an earlier stage of the design cycle. Logic design errors are functional mismatches between a design (implementation) and a specification [1]. Sources of these errors usually include the human factor, specification changes and bugs in CAD tools [2]. Formally, DEDC is defined as follows: given an erroneous design and a specification we need to identify candidate erroneous lines (diagnosis) and propose corrections to rectify the design [5] [6] [7] [10]. These corrections are usually selected from a design error model, such as the one by Abadir et al. [1]. That model contains ten different types of frequently occurring errors [2] such as gate type replacement, missing inverter, missing input wire etc., all of which can be modeled in terms of stuck-at faults [1].

It is notable that fault diagnosis and DEDC are two similar and inherently difficult problems because the space of suspects grows exponentially with the number of errors: $error\ space = \#\ lines^{\#\ errors}$ [10]. Moreover, the fault/correction space is even larger due to the vast number of candidate faults and errors (fault/error modeling).

In this work, we propose a simulation-based incremental approach to *multiple* fault diagnosis and logic debugging for combinational and full-scan sequential digital circuits. Given an erroneous design, its specification and a set of input test vectors $\mathcal{V}$, the method works in an iterative manner. It repeatedly identifies a single candidate error line and fault/error models this line to bring the behavior of the faulty device "closer" to the specification.

Here the notion of incremental design rectification is used for both fault diagnosis and DEDC but in theory it operates in the exact opposite directions. Rectification in the context of DEDC requires a set of corrections from a design error model, which make the erroneous netlist behave as the golden model (specification). Opposite to this process is fault diagnosis, where a set of lines and faults are identified and modeled so that the behavior of the correct netlist finally matches that of the faulty chip. For the sake of simplicity, in this paper the term "rectification" will refer to the process where the netlist available is modified to match the behavior of the device which can only be simulated. Also, we will not distinguish between faults and design errors or between fault models and corrections unless stated otherwise.

This work differs from previous incremental rectification approaches in the methodology used to identify the error location(s) and correction(s) at every iteration. In more detail, in subsection 3.2 we prove a theorem that stipulates a lower bound on the number of failed vectors each "valid" correction *must* rectify. This result, along with a number of different heuristics, allows efficient exclusion of corrections that cannot lead to an optimal solution. In practice, it helps avoid the exponential explosion of the problem. We also show that during rectification, it may be necessary to consider corrections that do not look optimal locally but have the potential to lead ultimately to an optimal solution. In addition, unlike previous approaches, the incremental framework presented here can handle erroneous designs with *any* number of errors.

To exhibit the effectiveness of the proposed approach, we run experiments on designs corrupted with multiple stuck-at faults or multiple design errors. In both cases, the results exhibit excellent resolution as the method can fault model designs within seconds of CPU time while it scales well with increasing number of errors. In addition, it can return nearly all possible sets of multiple stuck-at faults that explain the faulty behavior which, to the best of our knowledge, makes this work the first *exact* multiple stuck-at fault diagnosis algorithm. Finally, the algorithm is simple to code and can tackle additional real-life industrial problems as it can be easily extended to diagnose erroneous implementations at different levels of the design cycle (RTL, logic, or physical), either by devising appropriate error/fault models or through an efficient mapping algorithm [8].

In the section that follows we give relevant definitions. Diagnosis, correction and the overall algorithm flow are described in Section 3. Experiments on multiple design errors and multiple stuck-at faults can be found in Section 4 and Section 5 contains the conclusions.

2. Preliminaries

In this work, we investigate incorrect netlists with logic `NOT`, `BUFFER`, `AND`, `NAND`, `OR` and `NOR` gates. Although the algorithm can handle `XOR` and `XNOR` gates, we do not consider them explicitly. Further, we assume that both the specification and the design are completely simulatable. This assumption can be relaxed as discussed in [1]. Throughout our discussion, we say that a line l, fan-in to an `AND` or `NAND` (`OR` or `NOR`) gate has *controlling value* for input vector v if the value of l is 0 (1). If l drives a `NOT` or a `BUFFER` it always has controlling value. Finally, a line whose value changes during simulation under the presence of some fault(s) is called a *sensitized line* and a path of sensitized lines is called a *sensitized path*.

Prior to the execution of the algorithm, we simulate a number of random input test vectors $\mathcal{V}$ and create two bit-lists, $\mathcal{V}^l_{err}$ and $\mathcal{V}^l_{corr}$, on every line l in the circuit. The i-th entry of the $\mathcal{V}^l_{err}$ ($\mathcal{V}^l_{corr}$) list contains the logic value of l when we simulate the i-th input test vector from $\mathcal{V}$ with erroneous (correct) primary output responses. In other words, $\mid \mathcal{V} \mid = \mid \mathcal{V}^l_{corr} \mid + \mid \mathcal{V}^l_{err} \mid$. These bit-lists are used and properly updated, as it will be explained, during diagnosis and correction. Deterministic test generation for the errors injected is not the subject of this work.

Given a faulty device, the set of corrections that rectify it can be classified as either *actual* or *equivalent*. In our discussion, we will call any such correction a *valid* correction. Finally, in diagnosis we use *path–trace*, a line marking procedure developed for fault diagnosis by Venkataraman et al. [12], similar to critical path tracing. For an erroneous vector v, path–trace starts from an *erroneous* primary output for v and traces backwards toward the primary inputs of the circuit, while marking lines of interest. Details and examples of this linear time algorithm can be found in [12]. Its importance to fault diagnosis stems from the fact that it always marks *at least* one line from every set of valid corrections [10].

3. Incremental Rectification

The *input* to the method is an erroneous design, its specification and a set of input test vectors $\mathcal{V}$. In our experiments, we simulate vectors from [3] along with 6,000-10,000 random vectors to compile $\mathcal{V}^l_{corr}$ and $\mathcal{V}^l_{err}$ on every line l of the circuit. The *output* of the algorithm is a list of lines associated with valid corrections. Considering the existence of equivalent corrections, it is evident that an accurate fault diagnosis algorithm should not only return results that localize *some* faulty areas but also the *maximum* possible set of (equivalent) fault locations that explain the fault effects. This is not the case with DEDC where a single set of valid corrections is sufficient to provide a solution. In subsection 3.3 we explain how to tailor the method to handle both cases efficiently.

The algorithm works in an iterative manner. It bases its operational flow on a *decision tree*. Every iteration of this process is represented by a different level in the tree where different options (decisions) are available. At each iteration, it identifies a set of lines (diagnosis) and devises a set of corrections for these lines according to certain parameters described in the next two subsections. In most of the cases, each correction helps reduce the number of erroneous primary outputs for *all* vectors in $\mathcal{V}$. It should be noted that the algorithm does not discard intermediate corrections that may not look attractive. As reported in [7],the number of erroneous responses does not necessarily increase with the number of injected errors. We develop heuristics, presented in subsection 3.2, that prevent the algorithm from deleting such valid corrections.

It is evident that the *global flow* of the incremental process is crucial for its success. Considering the vast amount of corrections that can potentially model the error effects at each iteration, run-time and resolution are greatly affected by early decisions. If a greedy Depth-First-Search (DFS) approach is used, a wrong decision at the top level of the tree may lead to the exploration of a portion of the search space with no solution. On the other hand, a naive Breadth-First-Search (BFS) approach may result in excessive computation. Consequently we use a global flow approach which is a trade-off between DFS and BFS, described in subsection 3.3. In the next two subsections, we present the diagnosis and correction procedures.

3.1. Diagnosis

The objective of diagnosis is to quickly reduce the error space and eliminate lines that have no potential to lead towards an optimal solution. This process is done in two steps. During the first step, path-trace is executed to mark suspect lines in the circuit. It has been shown [10], that path-trace always marks at least one location from every set of locations where valid corrections exist. Therefore, we allow lines that have a high path-trace count to qualify for the second diagnosis step. During our experiments, we select the top 15-20% of these lines.

In the second step, for each line l, we invert the logic values in its $\mathcal{V}_{err}^{l}$ bit-list and propagate this difference throughout the fan-out cone of l. Recall that $\mathcal{V}_{err}^{l}$ contains the logic values of line l for the subset of vectors that activate the errors. Inversion and propagation of all of its values emulate the *maximum* effect any modification to this line can have on the circuit. Once done, **we count the number of erroneous primary outputs that are rectified and sort all lines according to these counts (heuristic 1)**. In a sense, the suspect lines are ranked by how much correcting potential they have. During correction, we visit these candidate lines in decreasing order of the counts. Our experiments indicate that lines with high counts often lead to valid corrections.

3.2. Correction

A correction attaches a fault/error model to a candidate line. In the stuck-at fault diagnosis, either a stuck-at-0 or a stuck-at-1 fault model is used. In DEDC, many of the ten error models devised by Abadir et al. [1] are attempted. Although diagnosis reduces the error space rapidly, the ability to select valid corrections at each iteration can influence the overall performance dramatically as incorrect decisions may cause the underlying decision tree to explode exponentially. Further, the selection of such valid corrections is not an easy task in DEDC considering the vast amount of corrections that can partially *model* the fault(s) effects.

In this section we present a number of techniques that guide the correction procedure and avoid exponential explosion. The following theorem gives a necessary condition which all valid corrections *must* satisfy. We use this theorem to screen corrections that cannot lead to an optimal solution. It is surprising, as indicated in our experiments, that this simple theorem provides a reliable guide to the correction procedure.

Theorem 1: Let $l_1, l_2, \ldots, l_N$ be the set of lines where a set of valid corrections can be applied and rectify a design. Let V be a set of input test vectors with *failing primary output* responses. Define V_i to be the maximum subset of vectors from V that produce an erroneous logic value on l_i (*i.e.,* excite the error) and propagate this difference to some primary output(s). The minimum size of V_i that has the maximum cardinality among all sets $V_1 \ldots V_N$ is no less than $\frac{|V|}{N}$.

The proof of the theorem, which is an application of the pigeon hole principle, is omitted due to the lack of space.

The correction algorithm proceeds as follows. Given an error location l that qualified, the algorithm *exhaustively* compiles a list of corrections from the design error or fault model as in [6] [10]. From these corrections, we keep the ones that satisfy both of the following screening tests after they are applied to the gate driving l:

Screening test on $\mathcal{V}_{err}^{l}$ vectors: Any qualifying correction must complement at least $\frac{|\mathcal{V}_{err}^{l}|}{N}$ bits in $\mathcal{V}_{err}^{l}$ (heuristic 2). This screening test is a direct application of Theorem 1 on the erroneous bit-list of l. This test can be performed efficiently with a single simulation step on the gate driving l and the fan-ins to that gate. The newly obtained logic values are then compared to $\mathcal{V}_{err}^{l}$. Our experiments indicate that this inexpensive simulation step disqualifies the majority of inappropriate corrections and helps reduce the size of the correction space significantly.

Since the number of errors N is unknown, in implementation we follow an aggressive approach in applying this screening test as we necessitate that a correction complements a high number of bit entries. We empirically set this limit to the initial value of 70% and reduce it progressively when the algorithm returns with no corrections. There are two reasons that justify such an approach. First, certain classes of faults and errors, such as gate related errors, are easier to excite than others such as wire related errors [1] [5] [10]. For instance, the design error "extra inverter at the output of a gate G" is excited for *every* input test vector (100%) even if this difference is not observed at a primary output. Similarly, a stuck-at-1 fault at an output of an AND gate is easier to excite as this gate is expected to have most of the times the logic value 0. Second, Theorem 1 provides a conservative *lower bound* on the size of V_i with the maximum cardinality. It does not account for vectors that simultaneously excite many errors and, consequently, can be attributed simultaneously to different V_i sets. For example, in a design with two missing inverters, the error sites are excited for every input vector with failing responses.

Screening Test on $\mathcal{V}_{corr}^{l}$: Any qualifying correction may sensitize only a small number of new paths to previously correct primary outputs (heuristic 3). We elaborate on the rationale of this heuristic with an example.

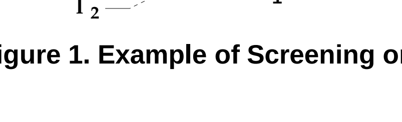

Figure 1. Example of Screening on $\mathcal{V}_{corr}^{l}$

Example 1: Fig. 1 depicts the situation where the effects of two design errors on lines l_1 and l_2 sensitize two paths that merge in gate G with logic values 0 and 1 respectively. Also assume that this vector v produces correct primary output responses. It can be seen that in the correct implementation, l_1 produces a logic 1 in the first input of G and l_2 produces a logic 0 in the second input of that gate. Therefore, when a valid correction is applied to l_1, under the presence of the second error, the logic value at the fan-out of G switches to 1. If a primary output is sensitized to the fan-out of G, v becomes erroneous. It remains erroneous until a valid correction is applied to l_2 and the fan-out of G switches back to its correct logic value.

The above example suggests that sometimes we should qualify corrections that sensitize a new number of erroneous primary outputs. In our experiments, we require a correction to create no more than 3-8% new erroneous vectors on the average. Empirically, this number has been shown to be sufficiently large to allow valid corrections to qualify with the only exception of multiple faults introduced into a NAND-based XOR structure [6]. In this case, as our experiments show, we need to allow for a larger number (15-20%) of new erroneous vectors to be created ensure success. We perform the screening test on $\mathcal{V}^l_{corr}$ efficiently through simulation on the $\mathcal{V}_{corr}$ bit-lists on the lines in the fan-out cone of l.

3.3 Algorithm Flow

As explained in the beginning of the previous Section, the algorithm bases its flow on a decision tree. At every level of this tree, it computes a set of corrections. It then ranks these corrections according to criteria described below, and selects a number of corrections to apply in subsequent execution. As indicated earlier, the general flow of this process has a significant impact on the final performance and resolution. To avoid the pitfalls of stand-alone BFS and DFS the flow of the algorithm works in a way which can be best described as a BFS/DFS *trade-off*.

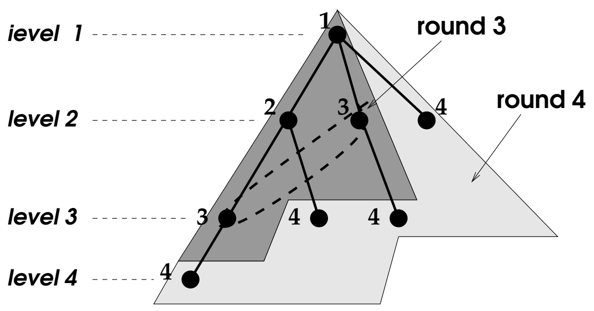

Figure 2. Decision Tree and Flow: A BFS/DFS trade-off

We explain the overall flow with the use of the decision tree shown in Fig. 2. Every node in this tree indicates a set of potential corrections—a correction is one error/fault model applied to a single candidate line—returned by a single iteration of the algorithm; an edge represents the application of a single (highly-ranked) correction to enter the next execution level; the level of a node indicates the number of corrections performed on the implementation so far. In other words, a path in the tree from the root to a leaf represents a set of corrections that can potentially rectify the implementation. The cardinality of the set equals to the length of the path. Instead of visiting nodes in the tree in a strictly BFS or DFS manner, the algorithm visits them in *rounds*. During each round, a single (highly-ranked) correction is selected from *every* node currently present. The correction is applied to obtain a new node in the next level of the tree. The number of nodes in the tree at most doubles with each round as the tree grows both in depth and breadth. For example, the numbers on the nodes in Fig. 2 indicate the round number in which the node is visited. The experiments show that this method of traversal overcomes the pitfalls of BFS and DFS.

In the remainder of this section we elaborate on the parameters involved with the execution of the algorithm and the three heuristics presented in subsections 3.1 and 3.2. On each candidate line under consideration, let h_1 indicate the percentage of erroneous primary outputs rectified during heuristic 1; h_2 indicate the percentage of bit-entries that need to be complemented in heuristic 2; and h_3 the percentage of correct primary outputs required to stay correct by heuristic 3. Runs of the algorithm initiate with values $h_1/h_2/h_3 = 1/1/1$ (single error case) and they are reduced progressively if it returns with no corrections. As the algorithm moves to diagnose designs with high cardinality of errors, h_1 reduces first before h_2 and h_3 do since these two parameters are *error independent*. For example, a typical run of the algorithm when the single and double error case has failed to return results is $h_1/h_2/h_3 = 0.3/0.7/0.95$, and followed by $h_1/h_2/h_3 = 0.3/0.5/0.85$ if it fails to return a correction. We also set $0.1/0.3/0.5$ as a lower limit for these parameters where the correction is declared as a leaf with failure and it is no longer attempted.

The corrections returned at level i are ranked according to the formula:

$$(1 - V_{ratio})h_3 \; + \; V_{ratio}h_1$$

and they are visited in the decreasing order of ranks during execution. In this formula, V_{ratio} indicates the percentage of vectors with erroneous output responses in $\mathcal{V}$ prior to the correction. The experiments show that this formula provides excellent guidance to the incremental algorithm. In all of the cases valid corrections rank in the top 5% in their respectively node and a solution is found before much of the decision tree is explored.

4. Experiments

We implemented the algorithm of Section 3 in C language and ran it on a SUN Ultra 5 workstation with 128 MB

Table 1: Results on Stuck-At Faults (time in sec.)

ckt	ckt	1 fault		2 faults			3 faults			4 faults		
name	lines	**# sites**	time	# tuples	**# sites**	time	# tuples	**# sites**	time	# tuples	**# sites**	time
C499	1265	**1.1**	0.70	2.8	**3.4**	0.72	7.3	**5.8**	0.91	3.3	**5.9**	29.6
C880	910	**2.7**	0.18	6.9	**4.9**	0.21	1	**6.1**	0.59	9.4	**7.3**	4.16
C1355	1211	**3.7**	0.17	2.3	**3.0**	0.81	4.5	**5.5**	10.89	9.3	**7.3**	44.9
C1908	928	**1.0**	0.98	2.3	**3.2**	2.76	12.1	**6.5**	0.85	7.3	**6.8**	16.7
C2670	1459	**2.2**	0.69	6.2	**5.6**	0.44	4.5	**5.0**	2.33	31.3	**9.8**	13.8
C3540	2281	**1.2**	1.96	4.9	**4.5**	0.67	7.2	**6.1**	29.72	14.3	**8.3**	67.6
C5315	3698	**1.7**	1.57	2.9	**3.3**	3.06	6.4	**5.1**	15.93	4.1	**6.1**	276
C6288	6264	**1.1**	3.43	2.2	**2.8**	2.94	5.9	**4.7**	14.58	1.5	**4.5**	61.3
C7552	5251	**2.7**	1.61	5.2	**3.9**	8.16	13.4	**6.2**	6.32	9.3	**8.3**	175
S838	731	**2.2**	0.16	4.0	**3.3**	0.35	7.2	**5.0**	1.80	214.7	**15.3**	1.01
S953	1082	**1.9**	0.26	2.6	**3.4**	0.55	2.3	**2.3**	13.81	96.1	**17.2**	49.38
S1196	1246	**2.4**	0.36	2.7	**3.0**	0.86	6.8	**6.5**	3.47	5.5	**7.2**	16.89
S1494	1421	**2.9**	0.21	7.0	**5.3**	0.25	18.9	**9.0**	1.31	5.3	**6.3**	17.17
S9234	4640	**4.1**	0.73	31.3	**8.4**	0.36	116	**12.5**	2.89	17.3	**10.2**	33.87
Average		**2.2**			**4.1**			**6.2**			**8.6**	

of memory for ISCAS'85 and full-scan versions of the ISCAS'89 benchmark circuits corrupted by multiple stuck-at faults and multiple design errors. The locations of the faults and errors were selected at random. The type of stuck-at faults was also selected at random while the types of design errors [1] were selected according to the distribution presented in [2]. In the following paragraphs we present and discuss the results. All run–times are in seconds and do not include the initial random simulation step for $\mathcal{V}$ which is performed only once. However they include the time for updating the bit-lists at subsequent steps.

4.1. Results on Stuck-at Faults

In order to simulate a realistic diagnosis environment, in the case of multiple stuck-at faults we first optimize the ISCAS'85 and ISCAS'89 circuits for area. Then we inject 1, 2, 3 or 4 faults and run the algorithm *exhaustively* (i.e. the search tree is fully traversed) as we are interested in *all* possible sets of faulty locations (tuples) and associated stuck-at faults that explain the faulty behavior of the device. In other words, we require the algorithm to have an *exact* performance and return the maximum set of minimal-size equivalent stuck-at fault sites that justify the faulty behavior. This information is the most useful to a test engineer. We ran 10 experiments for each fault case where every experiment has a different set of randomly injected faults. The average performance of the algorithm can be found in Table 1.

The first two columns of Table 1 contain the circuit name and circuit lines, respectively. The next columns contain the information pertaining to each different fault case: the number of tuples (pairs, triples, quadruples) returned that fully account for the faulty behavior (*i.e.*, *equivalent fault classes* [12]), the number of distinct lines in all these tuples the test engineer needs to probe, and the average run-time required to discover one tuple. Obviously, the number of tuples and distinct locations coincide for single faults. In all cases, unless fault masking occurred as described next, the algorithm always returned a tuple with the actual set of fault locations.

As a final note, fault masking, that is, the situation where some stuck-at fault(s) is not observable due to the presence of others, did not occur for the ISCAS'85 combinational circuits. This was not the case with the ISCAS'89 sequential circuits where fault masking occurred in more than 30% of the cases when 4 faults were injected, on the average. In these cases, the algorithm returned with triples or pairs that fully explain the fault behavior.

The average numbers shown in the last row of Table 1 exhibit the excellent resolution and justify the effectiveness of the proposed incremental diagnosis and fault modeling approach. In the future, we plan to apply this approach to other types of physical faults. The advantage of the algorithm lies in the fact that it can be adapted to other faults by adopting a suitable fault model in the correction stage.

4.2. Results on Design Errors

Table 2 shows the performance of the algorithm for 3 and 4 design errors. To simulate a realistic diagnostic environment, we use the original versions of all benchmark circuits that include redundancies (that is, circuit c1908 has 1908 lines, circuit c5315 has 5315 lines etc). These versions are also the hardest to diagnose and correct. In addition, all errors considered are observable. We ran 10 experiments for each error case per circuit where each time the types and locations were selected at random.

The first column of Table 2 contains the circuit name. Columns 2 and 6 of Table 2 contain the average run–time for diagnosis in a single execution of the algorithm when 3 and 4 errors are present, respectively. As explained in subsection 3.1, the purpose of diagnosis is to quickly eliminate the candidates that cannot serve as potential error locations for a valid correction. The CPU times reported in these columns show that heuristic 1 successfully eliminates 70-90% of these lines in roughly a second. The experimental results on correction, discussed next, indicate that error modeling and ranking according to heuristics 1, 2 and 3 also favors valid corrections.

The average time spent to return and rank the various corrections at a single execution of the algorithm is listed

Table 2: Results on Design Errors

ckt	3 error time (sec.)				4 error time (sec.)			
name	diag.	corr.	**nodes**	total	diag.	corr.	**nodes**	total
C499	0.43	0.44	**13.5**	11.70	0.43	0.50	**25.1**	23.35
C880	0.14	0.53	**34.9**	23.45	0.15	0.57	**120.8**	87.18
C1355	0.28	0.72	**41.0**	41.08	0.31	0.86	**193.3**	226.11
C1908	0.32	1.31	**20.6**	33.60	0.35	1.38	**50.6**	88.16
C2670	0.87	0.57	**16.2**	23.33	0.91	0.62	**44.3**	67.66
C3540	0.59	2.03	**11.8**	30.95	0.63	2.26	**28.4**	82.21
C5315	0.96	3.44	**29.9**	131.53	1.11	3.80	**65.2**	320.30
C6288	0.72	3.53	**14.5**	61.68	0.82	4.21	**23.0**	115.72
C7552	1.02	8.13	**12.2**	111.65	1.13	9.16	**20.6**	212.12
S838	0.11	0.68	**7.9**	6.27	0.12	0.79	**14.0**	12.82
S953	0.15	0.55	**6.3**	4.42	0.17	0.56	**11.9**	8.81
S1196	0.13	1.14	**5.7**	7.24	0.16	1.20	**10.5**	14.32
S1494	0.38	0.49	**3.1**	2.71	0.41	0.51	**8.3**	7.65
S9234	1.38	10.03	**31.4**	358.34	1.52	10.81	**55.7**	686.93

in columns 3 and 7. The next columns (columns 4 and 8) contain the average total number of algorithm executions for each error case, that is, the average number of nodes in the final tree of Fig. 2. Consider this tree, the leftmost path in any subtree consists of corrections ranked the highest in their respective node. So the first possible solution triple is found in a tree with 3 nodes (i.e. completed half the way through the 3rd round) and the second possible one in a tree with 6 nodes (i.e. completed half the way through the 4th round). In most cases, the algorithm completes in under 6 rounds with a maximum of 32 nodes. In other words, the first valid correction tuple is often found after exploring the first several leftmost paths of the decision tree. So a simple DFS approach will suffice for most of the cases. However, circuits c1355 and c880 often require 9 rounds, which allows a maximum of 256 nodes to be explored. Due to the large number of equivalent corrections, the first set of correction does not always contain the introduced errors.

The number of corrections returned during a single execution varies from 1 (the original) to a few thousand. Considering the huge amount of potential corrections at each step, these numbers indicate that heuristics 2 and 3 provide accurate error modeling resolution. They also favor the correction ranking of subsection 3.3 as it gives good guidance to the incremental algorithm flow. The total run–time to return a valid set of corrections is the product of columns 4 and 8 with the average single execution time of the algorithm and it can be found in the next columns. These numbers demonstrate the robustness of this approach, which can rectify benchmarks such as the 16-bit multiplier C6288, a traditionally hard to diagnose and correct circuit in the presence of many errors, in a few minutes of CPU time.

All in all, the results presented here support the effectiveness of an incremental approach and they confirm the potential of the underlying theory. Although experiments are performed on combinational and full-scan sequential circuits, the algorithm can be adapted to the diagnosis and correction of sequential circuits through time-frame expansion. It is also observed, as Theorem 1 suggests, that the method is heavily biased towards errors with high excitation measures. In other words, the method is more likely to perform well in the situations where the design is corrupted with errors that are easy to excite (even if they are not observable to the primary outputs) and their V_i sets (Theorem 1) overlap. It is among our future plans to further experiment and measure the parameters involved with these heuristics so that we can improve on the overall performance.

5. Conclusions

We presented a simulation-based method for diagnosing and fault modeling of physical faults and errors. The method rectifies the design through a sequence of interleaving diagnosis and correction steps that bring it “closer” to its specification. Experiments on multiple stuck-at faults and design errors are used to demonstrate its effectiveness. In the future we plan to investigate the application of this methodology to diagnosis of other physical faults and experiment with partial-scan devices.

References

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland. Logic verification via test generation. *IEEE Trans. CAD*, 7:138–148, January 1988.

[2] D. V. Campenhout, J. P. Hayes, and T. Mudge. Collection and analysis of microprocessor design errors. *IEEE Design and Test of Computers*, pages 51–60, Oct.-Dec. 2000.

[3] I. Hamzaoglu and J. H. Patel. New techniques for deterministic test pattern generation. *Proc. IEEE VTS*, pages 138–148, 1998.

[4] S. Y. Huang. Towards the logic defect diagnosis for partial-scan designs. *Proc. IEEE ASP-DAC*, pages 313–318, 2001.

[5] S. Y. Huang and K. T. Cheng. Errortracer: Design error diagnosis based on fault simulation techniques. *IEEE Trans. CAD*, 18(9):1341–1352, September 1999.

[6] D. Nayak and D. M. H. Walker. Simulation-based error diagnosis and correction in combinational digital circuits. *Proc. IEEE VTS*, pages 70–78, 1999.

[7] I. Pomeranz and S. M. Reddy. On correction of multiple design errors. *IEEE Trans. CAD*, 14:255–264, February 1995.

[8] S. Ravi, I. Ghosh, V. Boppana, and N. K. Jha. A technique for identifying rtl and gate-level correspondences. *International Conf. on Computer Design*, pages 591–594, September 2000.

[9] H. Takahashi, K. O. Boateng, and Y. Takamatsu. A new method for diagnosing multiple stuck-at faults using multiple and single fault simulations. *Proc. IEEE VTS*, pages 64–69, 1999.

[10] A. Veneris and I. N. Hajj. Design error diagnosis and correction via test vector simulation. *IEEE Trans. CAD*, 18(12):1803–1816, December 1999.

[11] S. Venkataraman and S. B. Drummonds. Poirot: Applications of a logic fault diagnosis tool. *IEEE Design and Test of Computers*, pages 19–30, Jan.-Feb. 2001.

[12] S. Venkataraman and W. K. Fuchs. A deductive technique for diagnosis of bridging faults. *Proc. IEEE ICCAD*, pages 562–567, 1997.