Incremental Design Debugging in a Logic Synthesis Environment

Andreas Veneris
University of Toronto
Dept ECE and CS
Toronto, ON M5S 3G4
veneris@eecg.toronto.edu

Jiang Brandon Liu
Freescale Semiconductors
High Performance Tools Group
Austin, TX 78729
brandon.liu@freescale.com

Abstract

In today's complex and challenging VLSI design process, multiple logic errors may occur due to the human factor and bugs in CAD tools. The designer often faces the challenge of correcting an erroneous design implementation. This study describes a simulation-based logic debugging solution for combinational circuits corrupted with multiple design errors. Unlike other simulation-based techniques that identify all errors at once, the proposed method works incrementally. At each iteration of incremental debugging, a single candidate location is rectified with linear time algorithms. This is done so that the functionality of the erroneous design gradually matches the correct one. A number of theorems, heuristics and data structures help identify a single candidate solution at each iteration and they also guide the search in the large solution space. Experiments on benchmark circuits confirm the effectiveness of incremental logic debugging.

1 Introduction

The digital VLSI design cycle commonly starts with a behavioral description coded in some Hardware Description Language (HDL). This description is next translated to a Register-Transfer Level (RTL) representation and synthesized to a gate-level (logic) implementation. Design validation and optimization steps guarantee the correctness and performance of the final product.

Although automated Computer-Aided Design (CAD) tools for synthesis and optimization are commonly used, circuit designers often need to manually modify the netlists generated by these tools to achieve specific optimization constraints and/or to make small specification changes. As the circuits grow in size and complexity, this manual resynthesis process becomes increasingly prone to error [1, 8]. Additionally, software bugs in CAD tools may introduce errors that alter the functionality of the synthesized design [1, 8]. These errors, known as *design errors*, are functional mismatches between a logic netlist and its functional specification. Experimental data has revealed that the nature of these errors usually involves the functional misbehavior of some gate elements and/or wire interconnection errors [1, 8]. The same experiments show that the number of errors is usually small (less than or equal to 4 errors).

Given a correct implementation of a specification and an erroneous logic netlist, **Design Error Diagnosis and Correction (DEDC)** is the automated process that identifies erroneous lines and proposes corrections on these lines to rectify the netlist [5, 7, 10, 11, 12, 15, 16, 17, 18]. DEDC methods are broadly classified as either simulation- or BDD-based [6]. Corrections proposed by a DEDC method are usually selected from a predetermined set of different types of permissible logic transformations, also known as *design error model*. A design error model should be simple to preserve the engineering effort invested on the design.

It is notable that DEDC is an inherently difficult problem. Because the implementation of the specification (in the form of HDL, RTL, etc) is an arbitrary one, it has to be treated as a "black box" controllable at primary inputs and observable at primary outputs. Consequently for the erroneous netlist, the solution (search) space for diagnosis grows exponentially with circuit lines and increasing number of errors [18]:

$$diagnosis\ space\ =\ (\#\ circuit\ lines)^{(\#\ errors)}$$

The correction space is further compounded by the large number of potential corrections applicable to each line, which is usually determined by the cardinality of the design error model that is used. Thus, the development of efficient DEDC tools to rectify designs with multiple errors is a challenging task.

The exponential growth of the solution space in terms of the number of errors also avails DEDC more options to correct a design [18]. This is because an error may be corrected by the *actual* or another functionally *equivalent* transformation [7]. Equivalent corrections exist because there may be many ways to synthesize a function and correct the design. We call an actual or an equivalent correction a *valid* correction. Since design errors and corrections are symmetric terms in this context, we do not make a distinction between them in our presentation.

In this paper, we describe a simulation-based **incremental DEDC (debugging) approach** for combinational circuits. Given an erroneous design, an implementation of its specification and a set of random input test vectors $\mathcal{V}$, the method iterates to identify and correct one error at a time and bring the design functionally "closer" to its specification. Incremental debugging terminates when the design is fully rectified in terms of the test vector set used. Test vector generation for these errors and design verification following a simulation-based DEDC method are not topics of this work [2, 4, 6].

The main difference between this approach and other incremental DEDC approaches [11, 15, 16] lies in the methods used to identify each single error location and its correction(s). In detail, we prove a theorem that asserts a lower bound on the number of failed vectors each valid correction *must* partially rectify. This result allows us to exclude corrections that might not lead to a solution and curtails the exponential explosion for the problem in practice.

Another advantage of the proposed incremental DEDC process lies its simplicity and efficiency. All steps at each iteration are adapted from *single error* DEDC algorithms. Single error DEDC is a well examined problem with linear time solutions [5, 7, 10, 11, 12, 15, 16, 17, 18]. We also show that during the execution of an incremental DEDC algorithm, it may be necessary to consider corrections that do not look attractive but may have the potential to lead to a solution in later stages. Finally, a novel decision making process on a search tree is presented to explore the solution space efficiently.

Experiments on combinational ISCAS'85 and full-scan sequential ISCAS'89 benchmark circuits confirm that the approach returns accurate corrections and it scales well with increasing number of errors. In fact, in all experiments it returns with a set of corrections that rectify the design for all test vectors used. Intuitively, its success is attributed to the fact that the solution space usually contains many equivalent corrections. Theoretically, since it utilizes a set of heuristics, the method may not return a solution and/or it may not find the complete set of all actual and equivalent solutions.

The paper is organized in the following. In the next section, we give relevant definitions and describe the error model. Incremental DEDC approach is presented in Section 3. Experiments are found in Section 4 and Section 5 contains the conclusions.

2 Preliminaries

We investigate erroneous structural netlists implemented with logic NOT, BUFFER, AND, NAND, OR and NOR primitive gates. The algorithm can accommodate XOR and XNOR gates but we do not consider them explicitly. Further, we assume that both the correct implementation of the specification and the design are completely simulatable. This assumption can be relaxed as discussed in [2, 20].

Throughout the discussion, line l, fan-in to an AND or NAND (OR or NOR) gate has a *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 a controlling value. 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*.

This work considers logic error types similar to those from the model of [2], shown in Fig. 1. This model contains a collection of simple logic gate or wire interconnection errors, such as gate replacement, missing input wire, extra input wire, etc, that may occur in logic synthesis [1, 8]. Most DEDC techniques [5, 7, 10, 11, 12, 15, 16, 17, 18, 19] use the er-

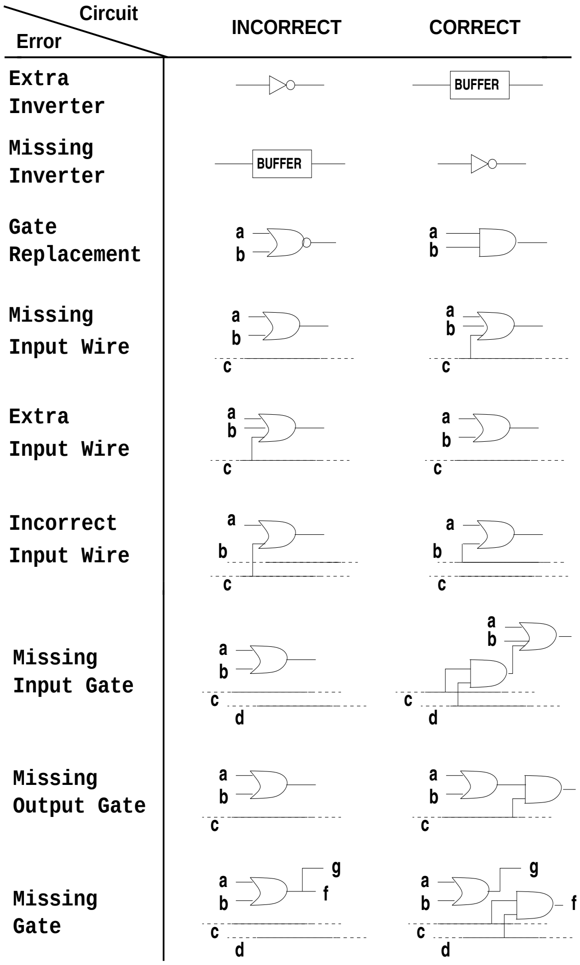


Figure 1: Common design error types

ror model by Abadir et al. [2] due to its practicality and its simplicity. Complex resynthesis algorithms during correction may alter the design globally and jeopardize the engineering effort already invested in it.

A simple design error model is desirable but it may not be adequate when a DEDC method is used as a platform to perform engineering changes. In the problem of *engineering change* one is required to modify a netlist to meet a new specification. Its relation to DEDC is discussed in detail in [14, 18]. Although the proposed DEDC method may be able to tackle some simple instances of the engineering change problem, a solution to the general problem is not the topic of this work.

Prior to the execution of the algorithm, we simulate a set $\mathcal{V}$ of random input test vectors and vectors for stuck-at vectors [9]. These types of vectors have been shown to detect more than 92% of design errors [2, 4, 18]. We use these vectors to create two indexed 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 the i-th failing (non-failing) input test vector from $\mathcal{V}$ is simulated. These bit-lists are properly updated and utilized during diagnosis and correction.

Most methods use simulation because it has been shown [2, 8, 18] that simulation with a small (100 to 3000) number of input vectors provides a reliable guide to DEDC. On average, 92% of corrections qualified this way are valid ones and pass formal verification.

In diagnosis we use *path-trace*, a line marking routine developed for fault diagnosis in [21] and similar to critical-path tracing [3]. 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 are found in [21]. Path-trace is important for multiple fault diagnosis because it *always marks* one line from every set of valid corrections [18].

3 Incremental Debugging

The *input* to incremental debugging is an erroneous design, a high-level implementation of the specification, the maximum number of possible design errors N and a set of input test vectors $\mathcal{V}$. The *output* of the algorithm is a set of circuit lines associated with the corrections that rectify the design for the vector set $\mathcal{V}$. For an erroneous circuit, we can estimate N empirically if we simulate vectors and record the percentage of failing primary output responses [18]. Usually the algorithm starts with a small value for N and increases this user-specified parameter if it fails to return with a correction(s).

The algorithm works in an iterative manner shown in Fig. 2. It bases its operational flow on a *decision tree.* Each iteration is represented by a different level in the tree where different options (*decisions*) are available. At each iteration, it identifies a set of suspect lines (diagnosis) and devises a set of corrections for these lines according to the parameters described in subsections 3.1 and 3.2.

In most cases, each correction reduces the number of erroneous primary outputs for *all* vectors in $\mathcal{V}$.

However, the algorithm does not discard intermediate corrections that may not look attractive. As reported in [16], the number of erroneous primary output responses does not increase monotonically with the number of injected errors. Subsection 3.2 presents heuristics that prevent deleting such corrections.

The *decision flow* of the incremental algorithm (that is, the sequence of decisions made at different levels of the tree) is crucial for its success. At each iteration, there may be a large amount of corrections that can model the error effects and their interaction. Total run-time and final 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 us to explore portion of the search space with no solution. A naive Breadth-First-Search (BFS) approach may result in excessive computation. Consequently we use an approach which is a *trade-off between DFS and BFS*, described in subsection 3.3.

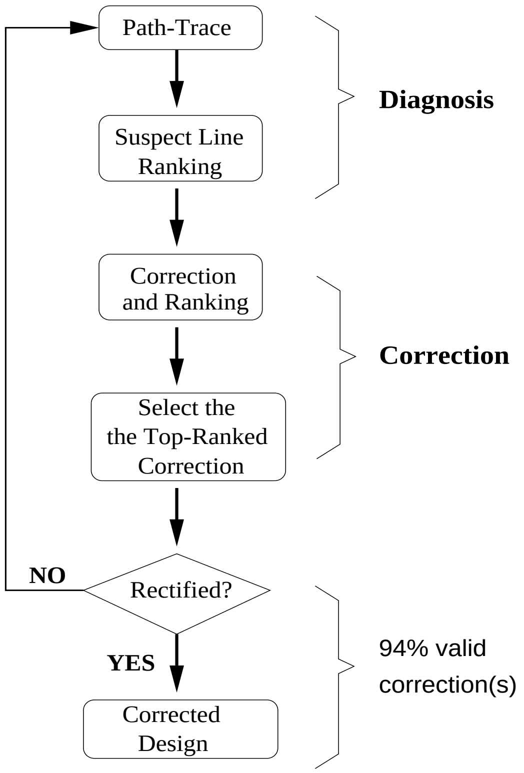


Figure 2: Algorithm description.

3.1 Diagnosis

Diagnosis quickly reduces the error space and eliminates lines with no potential to lead to a solution. This process is done in *two steps*. In the *first* step, path-trace marks suspect lines in the circuit. Path-trace always marks at least one location from every set of locations where valid corrections exist. We allow lines with high path-trace counts to qualify for the subsequent diagnosis, usually 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, a process known as *error simulation* [18]. Recall that $\mathcal{V}_{err}^{l}$ contains the logic values of line l for the subset of vectors that activate the errors. Once done, we count the number of erroneous primary outputs that are now rectified and rank all the 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 lines in a decreasing order of the counts. Experiments indicate that lines with high counts often lead to valid corrections.

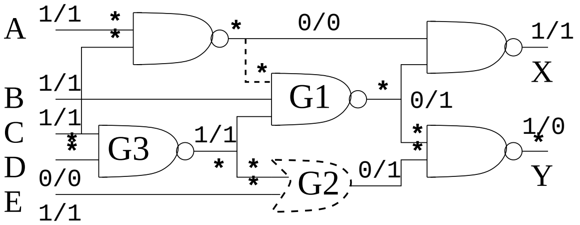


Figure 3: Line ranking during diagnosis

Example 1: Fig. 3 shows an erroneous implementation for ISCAS'85 benchmark $C17$ circuit. Gate $G1$ has an extra input wire and $G2$ is a `NAND` gate replaced by an `OR`. Test vector $v = (1, 1, 1, 0, 1)$ is simulated and pairs of fault-free/faulty values are shown next to each lines. Path-trace marks the lines with an asterisk " * ". Recall, error simulation inverts the logic value at a line and simulates at its fan-out cone for many vectors in parallel. Table 3.1 summarizes results of the ranking process performing error simulation with the vector v and two more vectors. Letter "C" ("E") in this table indicates that error simulation value agrees (does not agree) with the logic simulation one. We observe, line $G1$ matches more correct values for three vectors. For this reason, it ranks higher than $G3$ for these vectors in this stage of the algorithm.

test vector	err. design		inv $G3$		inv $G1$	
$(A, \ldots, E)$	X	Y	X	Y	X	Y
1 1 1 0 1	C	E	C	E	C	C
1 0 1 1 0	C	E	C	C	C	E
1 0 1 0 1	C	E	C	E	C	C

Table 1: Error Simulation on Suspect Lines

3.2 Correction

In correction, an instance of design error model is assigned to each suspect line that qualified diagnosis. This is done exhaustively for each correction as in [18]. As discussed, the ability to base decisions on valid corrections at each iteration influences the overall performance dramatically. The search for valid corrections is not an easy task if we consider the vast amount of possibilities that can model various error effects and different sensitized paths or co-sensitized paths (i.e. sensitized to more than one error) at each iteration of incremental diagnosis.

In this section we present a number of techniques and heuristics to prune the large solution space and guide the search. The following theorem gives a necessary condition all valid corrections *must* satisfy. We use this theorem to screen corrections that cannot lead to any optimal solution. As indicated in the experiments, that this simple theorem provides a reliable guide to correction.

Theorem 1: Let $\mathcal{L} = \{l_1, l_2, \ldots, l_N\}$ be the set of lines where a set of valid corrections exists. Let V_{err} be a set of input test vectors with *failing primary output* responses and let $V_{err}^{l_i}$ be the set of vectors from V_{err} that produce an erroneous logic value on l_i (*i.e.,* excite the error) and propagate this difference to some primary output(s). Then there exists a set $V_{err}^{l_i}$, $1 \leq i \leq N$, whose size is at least $\frac{|V_{err}|}{N}$.

Proof: The theorem is a direct application of the pigeonhole principle of counting. By definition, each vector in V_{err} excites at least one error to create sensitized path(s) to some primary output(s). In other words, each vector in V_{err} is attributed to *at least* one $V_{err}^{l_i}$ for some l_i, $1 \leq i \leq N$. Using the pigeonhole principle and the above observations, it becomes evident that there exists at least one set $V_{err}^{l_i}$ with at least $\frac{|V_{err}|}{N}$ test vectors from V_{err}.

The correction algorithm proceeds as follows. Given a suspect location l, it *exhaustively* compiles a list of corrections from the design error model. From these corrections, it keeps the ones that satisfy the following screening tests:

Screening test on $\mathcal{V}_{err}^l$ vectors: *Any qualifying correction must complement at least $\frac{|\mathcal{V}_{err}^l|}{N}$ bits in $\mathcal{V}_{err}^l$ when the correction is present on line l* **(heuristic 2)**. This screening test is a direct application of Theorem 1 as it implies that for every set of lines $\mathcal{L}$ where valid correction(s) exist, there is *at least* one location $l \in \mathcal{L}$ responsible for some erroneous primary output(s) in at least $\frac{|\mathcal{V}_{err}^l|}{N}$ failing primary output test vectors. Since these erroneous primary outputs are sensitized to respective $\mathcal{V}_{err}^l$ bit-list entries, a valid correction should complement the bit-list entries.

Evidently, the nature of this screening test may qualify corrections on lines that do not belong in any such set $\mathcal{L}$. It may also discard corrections on lines from a set $\mathcal{L}$ just because the number of failing output vectors attributed to them is not large enough to qualify the requirement of the theorem. The former set of (false) corrections will simply degrade the performance of the algorithm. On the other hand, since it always qualifies one element from $\mathcal{L}$, valid corrections discarded at the present iteration will be discovered at later iterations of the algorithm. This is because the algorithm examines incrementally the complete error space, as discussed later in subsection 3.3.

The test required by heuristic 2 can be performed efficiently with a single local simulation step on the driver gate of l. The newly obtained logic values are compared to $\mathcal{V}_{err}^l$. This set $\mathcal{V}_{err}^l$ is also updated appropriately once a line l qualifies to accommodate future iterations of the algorithm. Experiments indicate that this inexpensive simulation step disqualifies many inappropriate corrections improving the correction selection process dramatically.

Our implementation follows an aggressive approach and initially necessitates that a correction complements a high number of bit entries. This limit is empirically set to a value of 70% and reduced progressively when the algorithm returns with no corrections. This is because some errors are easy to excite such as errors with inversion. A vector with erroneous primary output response from in $\mathcal{V}$ may also excite more than one error and attribute to multiple V^l_{err} sets in Theorem 1.

Screening Test on $\mathcal{V}^l_{corr}$: *Any qualifying correction can sensitize a small number of new paths to previously correct primary outputs* **(heuristic 3)**. The number of erroneous primary outputs does not necessarily increase with the number of injected errors. This heuristic accounts for corrections that may increase the number of erroneous outputs and performs a screening with simulation of the $\mathcal{V}^l_{corr}$ bit-list at the fan-out cone of l. We elaborate on its rationale with the following example.

Example 2: Fig. 4 depicts the situation where the effects of two design errors on lines l_1 and l_2 have two sensitized 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, it now becomes erroneous. It remains so until a valid correction is applied to l_2 and the fan-out of G switches back to its correct logic value.

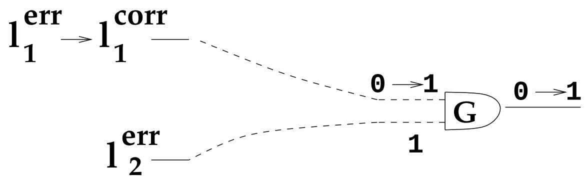


Figure 4: Example of Screening on $\mathcal{V}^l_{corr}$

This example suggests we may need to qualify corrections that sensitize a number of new erroneous primary outputs. Experiments permit a correction to create no more than 3-8% new erroneous outputs on the average. Empirically, this number has shown to be sufficiently large to allow valid corrections to qualify in most cases.

An exception is the case for `NAND`-implemented `XOR` circuits (e.g. C499, C432, etc) [15]. Consider `XOR` gate implemented with four `NAND` gates, as in Fig. 5, for example. Assume errors that replaces gates $G1$ and $G2$ with `AND`. If we correct any single error, it gives a larger number ($> 20\%$) of new erroneous primary outputs. In experiments for such circuits, the algorithm relaxes this constraint rapidly as lower values fail to qualify any corrections.

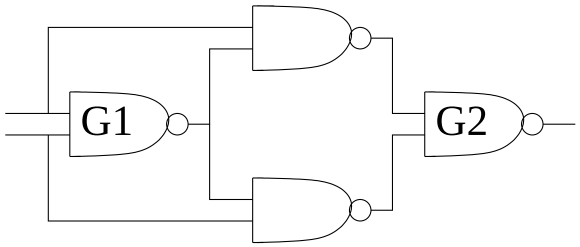


Figure 5: A `NAND`-based `XOR`

3.3 Decision Flow

Incremental debugging uses an underlying data structure based on a *decision tree*. At every node of this tree, the algorithm computes a set of corrections. It then ranks these corrections according to some criteria described later in this subsection. Next, it selects some of them to build its children nodes and it also updates the $\mathcal{V}^l_{err}/\mathcal{V}^l_{corr}$ bit-lists of each line appropriately. The decision making process has an important impact on the performance and resolution of the algorithm. To avoid pitfalls of stand-alone BFS or DFS, it makes decisions using a BFS/DFS *trade-off*.

We explain the decision making with the decision tree shown in Fig. 6. Every node in this tree represents a set of potential corrections—a correction is a single error model applied to one suspect line—returned by a single iteration of the algorithm; an edge represents the application of a single (highly-ranked) correction to enter the next tree 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 potentially rectify the design.

To make a decision, we visit these nodes in *rounds*. At 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. This tree traversal guarantees to reach a solution in finite time [13].

The round in which a node is created is shown next to the node in Fig. 6. Observe that the number of nodes in the tree at most doubles with each round as the tree grows both in depth and breadth. However,

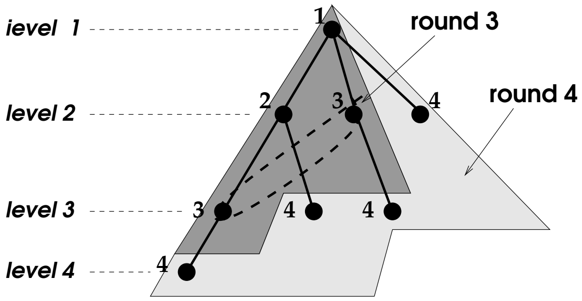


Figure 6: Decision Flow: A BFS/DFS trade-off

experiments indicate that the number of tree nodes visited before reaching a solution remains small in all cases.

Finally, we elaborate on the parameters of the three heuristics introduced with the algorithm. On each suspect line, let h_1 be the percentage of erroneous primary outputs rectified in heuristic 1; h_2 the percentage of bit-entries that are complemented in heuristic 2; and h_3 the percentage of correct primary outputs required by heuristic 3. Runs of the algorithm start with $h_1/h_2/h_3 = 1/1/1$ and they are reduced progressively if it returns with no corrections. Observe, incremental debugging with parameters $h_1/h_2/h_3 = 1/1/1$ equals to the single error case in traditional DEDC.

When many errors are present, h_1 reduces before h_2 and h_3, since h_2 and h_3 are error type dependent. An aggressive relaxation of the parameters can lead to numerous correction candidates. A combination which is typical for three suspected errors when higher values do not work is $h_1/h_2/h_3 = 0.3/0.7/0.95$. If this also fails, parameters are relaxed as $h_1/h_2/h_3 = 0.3/0.5/0.85$ etc. We also set 0.1/0.3/0.5 as a lower limit where a correction path is declared to terminate with failure and it is no longer attempted. For `NAND`-based `XOR` gates, stringent values of $h_1/h_2/h_3$ fail to return corrections until $h_1/h_2/h_3 = 0.4/0.5/0.5$ is reached.

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 a good guidance. In all of the cases valid corrections rank in the top 5-10% in their node and a solution is found before much of the decision tree is explored.

4 Experiments

We implemented the algorithm of Section 3 using C language and ran it on a SUN Ultra 5 workstation with 128 MB of memory for ISCAS'85 and full-scan versions of the ISCAS'89 benchmark circuits corrupted by up to 4 design errors. The locations of the design errors are selected at random. Error types are also selected at random from Fig. 1. We perform twenty experiments per circuit and per error case. Ten of these experiments involve gate related errors and the remaining ten are wire related errors.

This section presents and discusses results of the above experiments. All run-times are in CPU seconds and do not include the initial random simulation step for $\mathcal{V}$ which is performed only once with parallel vector simulation. Run-times include the time to update the bit-lists at each step of the incremental approach.

Table 2 shows average values for the performance of the algorithm for 2, 3 and 4 design errors. Detailed results for single errors are found in [18]. To emulate a realistic diagnostic environment, we use the original versions of all benchmarks with redundancies (that is, circuit c1908 has 1908 lines, c5315 has 5315 lines etc). Redundancies increase the solution space and the complexity of the problem.

The first column of Table 2 contains the circuit name. Columns 2, 6 and 10 of the table contain the average run-time for diagnosis in a single algorithm execution when 2, 3 and 4 errors are present, respectively. As explained in subsection 3.1, diagnosis tries to eliminate lines that cannot serve as potential error locations for a valid correction. The CPU times reported in these columns show that heuristic 1 successfully eliminate 70-90% of the 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 favors valid corrections.

The average time spent to compile and rank the corrections in a single algorithm execution is listed in columns 3, 7 and 11. The number of corrections in single algorithm iteration varies from 1 (the original) to a few thousand and it does not seem to give a pattern. Columns 4, 8 and 12 contain the total number

Table 2: Results for multiple errors

ckt	2 errors				3 errors				4 errors			
name	diag.	corr.	**nodes**	time	diag.	corr.	**nodes**	time	diag.	corr.	**nodes**	time
C499	0.42	0.41	**8.9**	7.36	0.43	0.44	**13.5**	11.75	0.43	0.5	**25.1**	23.35
C880	0.14	0.47	**29.3**	17.87	0.14	0.53	**34.9**	23.38	0.15	0.57	**120.8**	87.18
C1355	0.26	0.66	**34.9**	32.11	0.28	0.72	**41**	41	0.31	0.86	**193.3**	226.11
C1908	0.3	1.22	**16.3**	24.78	0.32	1.31	**20.6**	33.58	0.35	1.38	**50.6**	88.16
C2670	0.81	0.52	**10.2**	13.57	0.87	0.57	**16.2**	23.33	0.91	0.62	**44.3**	67.66
C3540	0.61	1.98	**8**	20.7	0.59	2.03	**11.8**	30.91	0.63	2.26	**28.4**	82.21
C5315	0.92	3.15	**25.8**	105.01	0.96	3.44	**29.9**	131.56	1.11	3.8	**65.2**	320.3
C6288	0.68	3.43	**10.2**	41.92	0.72	3.53	**14.5**	61.63	0.82	4.21	**23**	115.72
C7552	0.95	7.56	**10.9**	92.76	1.02	8.13	**12.2**	111.63	1.13	9.16	**20.6**	212.12
S838	0.11	0.59	**6.1**	4.27	0.11	0.68	**8.9**	7.031	0.12	0.79	**14**	12.82
S953	0.14	0.54	**5.9**	3.98	0.15	0.55	**9.3**	6.51	0.17	0.56	**11.9**	8.81
S1196	0.11	1.07	**4.6**	5.43	0.13	1.14	**8.7**	11.05	0.16	1.2	**10.5**	14.32
S1494	0.31	0.38	**3.9**	2.69	0.38	0.49	**7.4**	6.44	0.41	0.51	**10.1**	7.65
S9234	1.37	9.24	**25.9**	274.80	1.38	10.03	**31.4**	358.27	1.52	10.81	**55.7**	686.93

of algorithm executions (procedure invocations) for each error case. This corresponds to the number of nodes in the final decision tree of Fig. 6.

Consider this tree, the leftmost path in any subtree consists of decisions for corrections with the highest ranks in each node. For example, the first possible triple solution is found in a tree with 4 nodes. The second solution can be found in a tree with 6 nodes, which completed half the way through the 4th round. In most cases, the algorithm completes in under 6 rounds with a maximum of 32 nodes after exploring the first several leftmost paths of the decision tree. Some circuits such as c1355 and c880 often require 9 rounds and explore a maximum of 256 nodes. For these circuits, DFS would explore much of the tree before it returns with a solution.

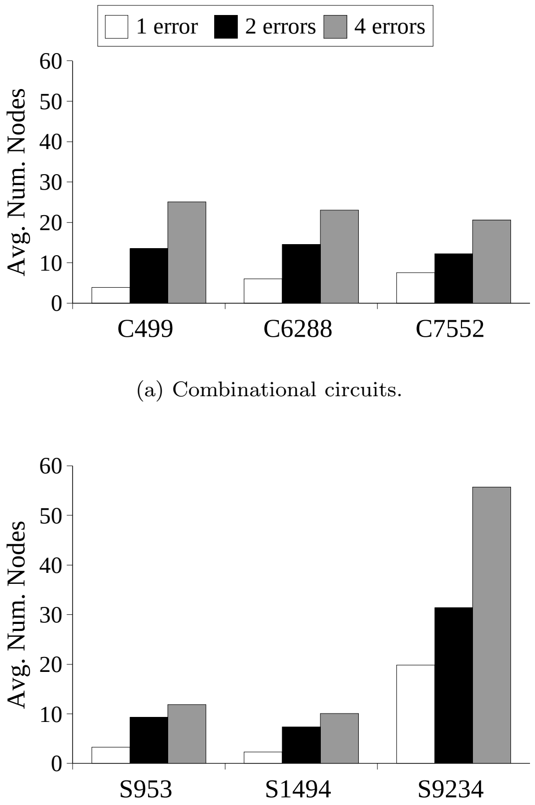
Figure 7 plots the average number of nodes in the tree data structure to diagnose some combinational and sequential circuits with 1, 2, and 4 errors. It is seen, the number scales well and it roughly doubles as the number of errors doubles. This means that the method is *space efficient*.

The total run-time to return a valid set of corrections is found in columns 5, 9, and 13. These runtimes demonstrate the robustness of this approach that rectifies benchmarks such as the 16-bit multiplier C6288, a hard to diagnose and correct circuit, in a few minutes of CPU time.

Since the proposed algorithm uses a set of heuristics to search efficiently in the search space for corrections, in theory, it cannot guarantee that it will return with a set of corrections. Nevertheless, in all experiments of Table 2 it always returned with a set of modifications that rectify the design for the set of test vectors utilized. This confirms the effectiveness of the theorems and heuristics presented in Section 3.

To demonstrate the effectiveness of the correction pruning heuristic 2, Figure 8 plots the average number (for 10 runs) of correction candidates returned by the correction algorithm in each iteration (white bars) versus the ones satisfying heuristic for two circuits. For each circuit, data for correcting 2 errors (left pair) and 4 errors (right pair) are given. It is seen, the heuristic reduces the correction candidates one order of magnitude (for over 1000 corrections, to less than 100). This, in effect, reduces both the runtime and memory usage significantly.

Figure 9 shows the average run-time performance for all combinational and all sequential circuits used in the experiments as the number of errors increases. Performance shows to scale well and the algorithm remains *time efficient*. It is also seen, incremental debugging for full-scan sequential circuits outperforms this for combinational circuits. This is because in full-scan mode, memory elements behave as pseudo outputs/inputs to reduce the structural level of the circuit. Path-trace is very effective in these cases and diagnosis benefits from this fact.



(a) Combinational circuits.

(b) Sequential circuits.

Figure 7: Search tree size

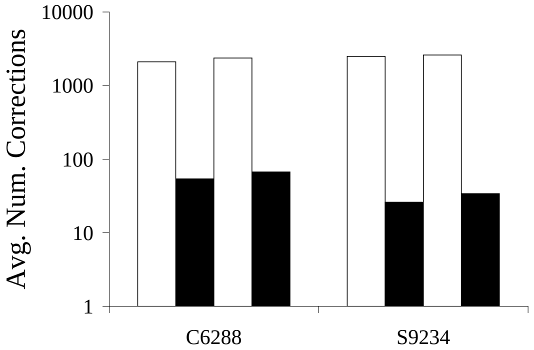


Figure 8: Effectiveness of Heuristic 2

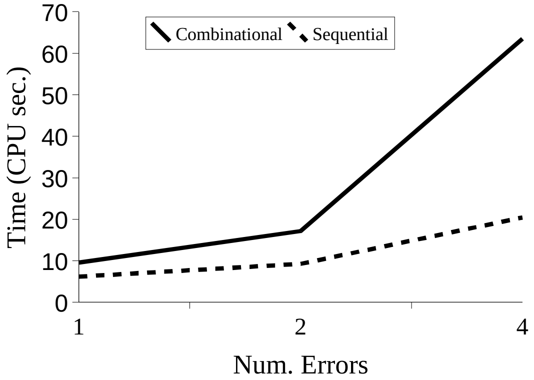


Figure 9: Run-time behavior

5 Conclusion

Design errors may occur in a logic synthesis environment. The designer often needs to debug a design yet preserve the engineering effort. We propose a simulation-based incremental debugging method for multiple design errors. The method rectifies an erroneous design through a sequence of interleaving diagnosis and correction steps. Each such step is fast and brings the functionality of the design "closer" to its specification. Experiments on benchmark circuits corrupted by many errors demonstrate the efficiency and practicality of the approach.

References

[1] E. J. Aas, K. Klingsheim and T. Steen, "Quantifying design quality: a model and design experiments," in *Proc. of EURO–ASIC*, pp. 172–177, 1992.

[2] M. S. Abadir, J. Ferguson and T. E. Kirkland, "Logic verification via test generation," in *IEEE Trans. on CAD*, vol. 7, pp. 138–148, January 1988.

[3] M. Abramovici, P. R. Menon and D. T. Miller, "Critical path tracing: an alternative to fault simulation," in *IEEE Design and Test of Computers*, vol. 1, pp. 89–93, February 1984.

[4] H. A. Asaad, and J. Hayes, "Logic Design Validation via Simulation and Automatic Test Pattern Generation," in *Journal of Electronic Testing: Theory and Applications, Kluwer Academic Publishers, vol. 16, pp. 575-589,* 2000.

[5] V. Boppana and M. Fujita, "Modeling the unknown!Towards model-independent fault and error diagnosis," in *Proc. of Int'l Test Conference, pp. 1094-1101,* 1998.

[6] R.E.Bryant, “Graph–Based Algorithms for Boolean Function Manipulation,” in *IEEE Trans. on Computers, vol.C–35, no.8, pp.677–691,* 1986.

[7] P. Y. Chung, and I. N. Hajj, “Diagnosis and correction of multiple design errors in digital circuits,” in *IEEE Trans. on VLSI Systems*, vol. 5, no. 2, pp. 233-237, June 1997.

[8] D. V. Campenhout, J. P. Hayes and T. Mudge “Collection and Analysis of Microprocessor Design Errors,” in *IEEE Design and Test of Computers, pp. 51-60,* Oct.-Dec. 2000.

[9] I. Hamzaoglu and J. H. Patel “New Techniques for Deterministic Test Pattern Generation,” in *Proc. IEEE VLSI Test Symposium, pp. 446-452,* 1998.

[10] N. Sridhar and M. S. Hsiao, “On efficient error diagnosis of digital circuits,” in *Proc. of Int’l Test Conference, pp. 678-687,* 2001.

[11] S. Y. Huang, K. C. Chen and K. T. Cheng, “Error correction based on verification techniques,” in *Proc. of ACM/IEEE Design Automation Conf.*, pp. 258–261, 1996.

[12] S. Y. Huang and K. T. Cheng, “Error-Tracer: Design Error Diagnosis Based on Fault Simulation Techniques,” in *IEEE Trans. on Computer–Aided Design*, vol. 18, no. 9, pp. 1341-1352, September 1999.

[13] H. R. Lewis and C. Papadimitriou, “Elements of Theory of Computation,” *Prentice-Hall,* 1981.

[14] C. C. Lin, K. C. Chen, S. C. Chang, M.M-.Sadowska and K. T. Cheng, “Logic synthesis for engineering change,” in *Proc. of ACM/IEEE Design Automation Conf.*, pp. 647–652, 1995.

[15] D. Nayak and D. M. H. Walker, “Simulation-Based Error Diagnosis and Correction in Combinational Digital Circuits,” in *Proc. IEEE VLSI Test Symposium*, pp. 70-78, 1999.

[16] I. Pomeranz and S. M. Reddy, “On correction of multiple design errors,” in *IEEE Trans. on CAD*, vol. 14, pp. 255–264, February 1995.

[17] A. Smith, A. Veneris and A. Viglas, “Design Diagnosis Using Boolean Satisfiability,” in *Proc. of IEEE Asian-South Pacific Design Automation Conference, pp. 218-223* January 2004.

[18] A. Veneris, and I. N. Hajj, “Design Error Diagnosis and Correction Via Test Vector Simulation,” in *IEEE Trans. on Computer–Aided Design,vol. 18, no. 12, pp. 1803–1816,* December 1999.

[19] A. Veneris, J. Liu, M. Amiri and M. S. Abadir, “Incremental Diagnosis and Debugging of Multiple Faults and Errors,” in *Proc. of Design and Test in Europe, pp. 716-721,* 2002.

[20] J. B. Liu, A. Veneris and H. Takahashi, “Incremental Diagnosis for Multiple Open-Interconnects,” in *Proc. of Int’l Test Conference, pp. 1085-1092,* 2002.

[21] S. Venkataraman and W. K. Fuchs, “A deductive technique for diagnosis of bridging faults,” in *Proc. IEEE/ACM Int’l Conf. on Computer Aided Design*, pp. 562–567, 1997.