Design Rewiring Using ATPG

Andreas Veneris, *Member, IEEE* and Magdy S. Abadir, *Senior Member, IEEE*

***Abstract*—Logic optimization is the step of the very large scale integration (VLSI) design cycle where the designer performs modifications on a design to satisfy different constraints such as area, power, or delay. Recently, automated test pattern generation (ATPG)-based design rewiring techniques for technology-dependent logic optimization have gained increasing popularity. In this paper, the authors propose a new operational framework to design rewiring that uses ATPG and diagnosis algorithms. They also examine its complexity requirements and discuss different implementation tradeoffs. To perform this study, the authors reduce the problem of design rewiring to the process of injecting a redundant set of multiple pattern faults. This formulation arrives at a new set of results with theoretical and practical applications. Experiments demonstrate the competitiveness of the approach and motivate future work in the area.**

***Index Terms*—CAD, diagnosis, synthesis, testing, VLSI.**

I. Introduction

DURING logic optimization, the gate level implementation (netlist) obtained by high-level synthesis tools is modified to achieve different constraints such as minimizing the area, minimizing power consumption, satisfying timing constraints, reducing switching noise, or improving the testability of the final circuit. Traditionally, logic optimization is carried out in two phases. First, technology-independent optimization returns an optimum logic network in terms of some general criteria such as gatecount, literal count, etc. In this phase, symbolic-based (Boolean or algebraic) methods [2], [19] are particularly efficient. Next, the netlist gates are mapped to a technology library and the design is further optimized under a new set of technology dependent constraints.

Recently, automated test pattern generation (ATPG)-based optimization techniques [3]–[7], [9], [10], [15], [22], [17] have gained increasing popularity for technology-dependent logic optimization. Their main strength lies in their performance since they are memory efficient. They have good failure characteristics, and, although theoretically they can be exponential in time, in practice this is rarely the case. Due to these facts, ATPG-driven techniques have successfully tackled problems such as area minimization [5], [10], [15], power reduction [17], performance optimization [16], [22], routing [4], and design for testability [7] as well as aiding devise solutions to other important problems such as logic verification [14], [15].

Manuscript received February 24, 2002; revised May 23, 2002. The work of the first author was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under Contract 227044-00. This paper was recommended by Associate Editor N. K. Jha.

A. Veneris is with the Department of Electrical and Computer Engineering and Department of Computer Science, University of Toronto, Toronto, ON M5S 3G4 Canada (e-mail: veneris@eecg.toronto.edu).

M. S. Abadir is with the High Performance Tools and Methodology Group at Motorola, Austin, TX 78729 USA (e-mail: m.abadir@motorola.com).

Digital Object Identifier 10.1109/TCAD.2002.804388

In general, existing techniques optimize a design through an iterative sequence of simple structural *design rewiring* operations. During each iteration, a single *target wire* is identified for removal because it violates some optimization constraint(s). Next, some simple redundant logic is added so that the target wire itself becomes redundant and it can be removed. Finally, the target wire is removed as well as other newly generated redundancies. This process is repeated until the desired optimization constraints are satisfied. Due to the nature of their operations, these techniques [3]–[7], [9], [10], [15], [22], [17] are also known as *redundancy addition/removal (RAR)* techniques.

In this work, we present a new operational framework to ATPG-based design rewiring that combines existing *design error diagnosis and correction (DEDC)* techniques [1], [23] with advances in ATPG [11], [13], [20], [14]. The proposed ATPG/diagnosis-based design rewiring (ADDR) methodology works in the opposite direction to existing procedures. It first introduces a design error and then it corrects it with a simulation-based DEDC algorithm and ATPG. We also describe efficient implementations for this method, study its complexity requirements, and present experiments that demonstrate its robustness.

It should be noted that in this paper we do not propose an algorithm that targets a specific optimization goal, but we introduce a new methodology to design rewiring that adds and complements existing techniques [24]. It is among our research plans to apply the presented results to specific optimization problems [25]. Also, we do not examine algorithms that identify newly generated redundancies after a series of logic transformations but the work in [4]–[6], [9], [10] applies here as well.

ADDR has several unique features that make it attractive when compared to existing approaches. First, it is not limited in the amount and type of target logic it eliminates. Since it formulates the elimination of the target logic through the introduction of an error, it can perform a wide variety of logic transformations. In general, the approach allows one to arbitrarily modify part of the circuit to introduce an error and correct it in a less crucial part of the design. This opens a new range of opportunities and applications to ATPG-based design rewiring.

The next novelty lies in the fact that the set of *logic (structural) transformations* it returns is always a superset of the one returned by existing ATPG-based design rewiring techniques. In fact, since there exist efficient DEDC approaches exhaustive on the correction space [23], it returns *all* permissible logic transformations that correct a target error. In other words, it makes better use of the internal don't cares of the design during optimization.

We also expect that this new view to design rewiring will aid in obtaining efficient solutions for multiple (simultaneous) logic

transformations. This has traditionally been computationally intensive for existing techniques [6], [9]. Multiple logic transformations are important for target logic with no single alternatives as they add to the solution space to help meet optimization goals [4], [5]. They also take further advantage of the internal don't cares of a design since the number of pairs of corrections for a circuit corrupted with two errors, for example, is usually larger than the product of single corrections when the circuit is corrupted by each error independently [23].

It is of importance to notice that the unique features of this new approach come at no additional complexity cost when compared to traditional ATPG-based techniques. To prove this, in Section V we reduce the problem of design rewiring to the problem of *multiple and simultaneous self-masking pattern faults* [21] injection. This formulation allows us to draw the conclusion that the complexity of the method equals to this of existing techniques. Moreover, the presented theory leads to the more general conclusion that the complexity of redundancy checking for a set of multiple pattern faults [21] is no harder than that for a single pattern fault, an interesting result that stands on its own. Therefore, recent advances in ATPG provide computationally efficient implementations for the method. Finally, experimental data confirm the effectiveness of simulation-based DEDC as it helps design rewiring avoid unnecessary redundancy checks. They also demonstrate the competitiveness of the approach.

The paper is organized in seven sections. To effectively describe the proposed approach, we discuss existing ATPG-based design rewiring techniques and the problem of DEDC in Sections II and III, respectively. In Section IV, we present the new method and discuss its novelties. Complexity requirements are examined in Section V and experiments in Section VI validate the theory and motivate future work. Section VII contains the conclusions.

II. Previous Work

Existing ATPG-based design rewiring techniques optimize a netlist through a sequence of simple logic transformations [3]–[7], [9], [10], [15], [22], [17]. During each iteration, a target wire that violates a specification constraint(s) is first identified for removal. For example, the target wire may be on the critical path or it may have excessively high switching activity. Next, a logic transformation is performed to remove that wire. This transformation entails addition of some simple redundant logic, such as the introduction of a new gate with inputs existing lines in the circuit or an additional fan-in to an existing gate. The effect of this extra redundant logic is to make the target wire redundant so it can be removed.

In summary, RAR optimizes a design through an iterative sequence of the following operations.

- **Operation 1:** Identify target wire w_T to be removed.
- **Operation 2:** Compute new logic w_A that makes w_T redundant.
- **Operation 3:** Check if w_A is redundant.
- **Operation 4:** If w_A is redundant, delete w_T and other newly generated redundancies.

Example 1, taken from [10], outlines these steps for a single wire removal.

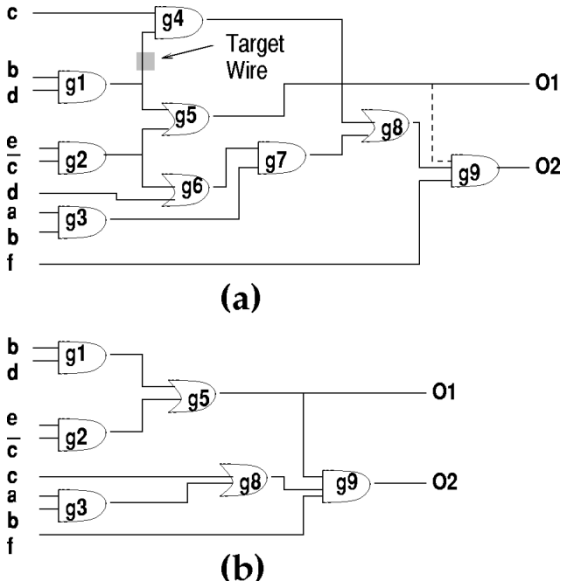

Fig. 1. Optimization through rewiring.

Example 1: Consider the circuit in Fig. 1(a) where wire $g5 \rightarrow g9$, indicated by a dotted line, is not part of the original netlist and assume that wire $w_T = g1 \rightarrow g4$, named target wire hereafter, needs to be removed (Operation 1). In Operation 2, new redundant connection $w_A = g5 \rightarrow g9$ is computed, shown as a dotted line in Fig. 1(a). Under the presense of w_A, wire w_T becomes redundant and new redundancies are introduced such as wire $g_6 \rightarrow g_7$. During Operation 3, w_A is added and redundancies w_T and $g_6 \rightarrow g_7$ are removed as well as logic that no longer has an influence on the primary outputs such as gates g_6 and g_4, wire $g_4 \rightarrow g_8$, etc. The new optimized circuit is shown in Fig. 1(b) where wire w_T has been removed and the gate count has been reduced.

Although a few different types of target logic have been considered in the literature, the vast majority of existing techniques are designed around the removal of a single wire (w_T). RAR techniques usually differ in the ATPG engine used to identify the structure and the location of the corrections.

In [3]–[6], [9], and [10], the target logic and the added logic are modeled as stuck-at faults and corrections are identified using fault dominating conditions [13] and sets of *mandatory assignments (MA)* [14]. Given a fault, MAs are logic values on lines respected by *all* input vectors that test this fault. For example, every test vector for $g_1 \rightarrow g_4$ stuck-at 1 in Fig. 1(a) necessitates a logic 0 on line $g_7 \rightarrow g_8$ and a logic 1 on lines c and f. MAs can provide sufficient (but not necessary) conditions for redundancy checking of a fault f; if the MAs of f are inconsistent then f is redundant [13]. The work by Kunz *et al.* [15] uses *logic implications* [14] to add logic which is known to be *a priori* redundant. References [22] and [17] represent the circuit as a set of logic clauses and different theoretical results allow perform redundancy identification, addition, and removal.

III. Design Error Diagnosis and Correction

Logic *design errors* occur during the design cycle of a VLSI chip due to the human factor or bugs in CAD tools [1]. These errors are functional mismatches between the specification and

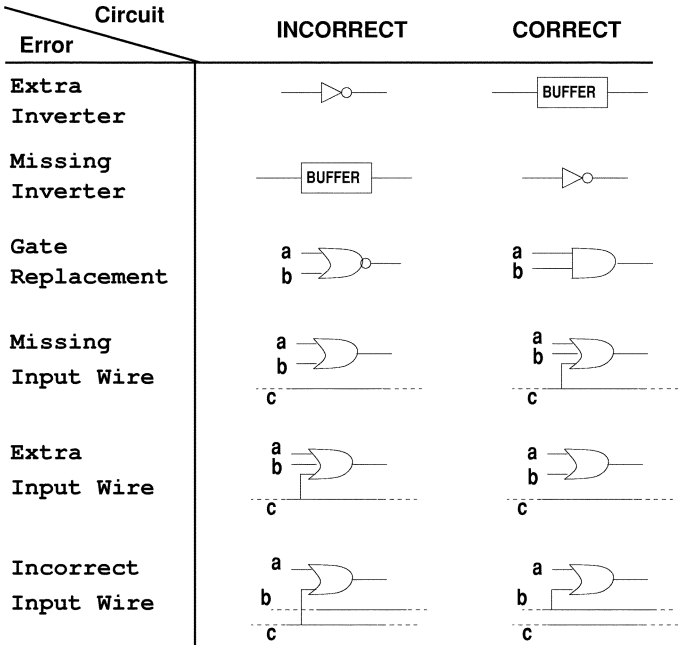

Fig. 2. Common design error types.

the gate level description. Most literature in the field uses a design error (correction) model, i.e., a small predetermined set of ten possible error types, proposed by Abadir *et al.* [1]. A list of common design error types, taken from the model of [1], is shown in Fig. 2. These errors are related to the work presented here. The gate types in Fig. 2 are indicative. Similar errors can occur on other gate types as well.

DEDC is the problem where given an erroneous design, a specification, and a design error model, we need to identify lines in the design that are potential sources of error (diagnosis) and suggest appropriate modifications on these lines from the design error model that rectify them (correction) [1]. DEDC is a well-studied problem with a significant amount of published work. In theory, test generation and design verification for DEDC are inherently difficult problems since the solution space grows exponentially with the number of injected errors [23]

$$\text{error space} = (\#\text{ of circuit lines})^{(\#\text{ of errors})}. \qquad (1)$$

This difficulty stems from the fact that the error location(s) is not known. On the other hand, it has been theoretically proven by Abadir *et al.* [1] and experimentally confirmed in [23] that a complete test set for stuck-at faults for the erroneous design detects the majority of design errors in Fig. 2 and it has a good chance to detect the remaining ones. For this reason, most DEDC techniques simulate test vectors for stuck-at faults and random test vectors to diagnose and correct a design. Provided vectors with erroneous responses, these methods can be exhaustive on the solution space yet remain efficient especially for single errors where the solution space grows linearly to the number of circuit lines [see (1)].

Intuitively, simulation-based DEDC methods obtain a solution by *intersecting* the solution space offered by each vector, as shown in Fig. 3. This solution space consists of the actual and *equivalent* corrections, that is, alternate ways to synthesize a

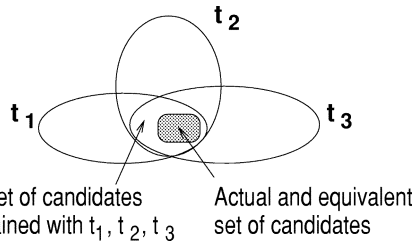

Fig. 3. Resolution of simulation-based DEDC.

function and rectify the design. A compendium of different simulation-based and symbolic-based DEDC approaches is found in [23].

In Section IV, we use the DEDC algorithm for single design errors from [23] which is based on fault simulation techniques. The *input* to the algorithm is an erroneous netlist, its specification, and a set of input test vectors. Some vectors have failing output responses and some do not. The algorithm *outputs all* applicable corrections that rectify the design for the given test vector set using a design error model which is a simple extension of the one presented in [1]. For example, with respect to the example in Fig. 1(a), removing target wire $w_T = g_1 \rightarrow g_4$ and running the DEDC algorithm described in [23], it returns equivalent correction $g_5 \rightarrow g_9$ as well as the original error. The existence of equivalent corrections is of paramount importance for the potential of the design rewiring method presented next.

IV. Design Rewiring Using ATPG

A closer view of the two problems described in Sections II and III shows that the process of design rewiring can be viewed from a DEDC perspective as follows. To eliminate w_T we can remove it and artificially introduce a "design error." Since the location of the error is known, ATPG [11], [13], [20], [14] can derive input test-vectors that detect it. These vectors can feed a simulation-based DEDC algorithm which is exhaustive on the correction space to get all "corrections" that rectify the design. Finally, an appropriate correction from the list of *equivalent* corrections is selected and ATPG-based verification of the final design is performed.

In the context of design optimization, an appropriate correction is one that satisfies the current optimization constraint(s). Since the algorithm operates on a (structural) gate level representation of the design, technology libraries are available to compute optimization tradeoffs and tune the process flow toward an optimum level of performance.

Here, simulation-based DEDC is used to efficiently compute all possible corrections. However, since it bases its results on a subset of the complete test vector space, these corrections rectify the design only for this set of vectors and not necessarily for the complete input test vector space [23]. To alleviate this problem, in Section IV-A we propose an ATPG-based [14] technique to guarantee the correctness of the final design. In Section V, we examine its complexity and we show that it equals that of existing ATPG-based methods.

In a sense, the design rewiring procedure described above (remove/add logic) works in the opposite direction from existing techniques (add/remove logic). However, as we discuss in Section IV-C, when it operates in this new direction it can take full

advantage of the internal don't cares of the design and provide a more systematic platform for ATPG-based logic resynthesis.

A. Method Overview

In the remaining section, for the sake of simplicity, we discuss the implementation of the method in terms of a simple logic transformation where we remove a target wire ("design error") and attempt to rectify the design by adding some other wire ("correction"). The method can be tailored to perform arbitrary amounts of logic resynthesis during error introduction and during correction.

ADDR performs the following four steps.

- **Step 1:** introduce design error by removing target logic.
- **Step 2:** derive test vectors for this design error.
- **Step 3:** use a DEDC algorithm to search for corrections.
- **Step 4:** verify the correctness of the final design.

The first step artificially introduces a design error by removing target wire w_T. In this discussion, we assume that a nonredundant target wire w_T is an input to gate G_T and G'_T is that gate when w_T is removed. We also assume that G_A is a gate where we can add an input wire w_A, resulting in gate G'_A. Next, a two-input multiplexer is added in the circuit with gates G_T and G'_T connected at its inputs and select line S. To derive vectors that detect the design error for the second step we run ATPG for S stuck-at 1.[1] Since, by hypothesis, w_T is nonredundant, the ATPG process is guaranteed to return with a set of vectors. Each of these vectors will also distinguish between the old (correct) and new (erroneous) circuit since, by construction, the output of the multiplexer implements the function at G_T when $S = v$ and the function at G'_T when $S = \overline{v}$ ($v = 0$ or 1).

In the third step, simulation-based DEDC returns all equivalent corrections. The input to DEDC is the correct and erroneous circuit along with the test vectors from ATPG (Step 2) and a small number of random test vectors. The output of DEDC is a list of all (actual and equivalent) corrections that rectify the design for the test vectors used. Let wire w_A which is a missing input wire to gate G_A be an equivalent correction proposed by the algorithm.

Finally, we need to verify the correctness of the final design when w_A is added in the circuit. To perform this verification process in Step 4, we attach a second two-input multiplexer to the fan-outs of G_A and G'_A with the same select line S so that when $S = v$ we select the old circuit and when $S = \overline{v}$ we select the new circuit. As a special case, if both the error and correction are on the same gate, that is, $G_T = G_A$, we simply let G_T and G'_A be the two inputs of the original multiplexer. It is clear that if ATPG for S stuck-at 1 returns with a nonempty test-vector set it also indicates that the new circuit (w_T removed, w_A added) is incorrect. However, if the test set is empty (i.e., S stuck-at 1 is redundant) then it guarantees the correctness of the new design.

B. Example

We give an example to illustrate the algorithm described in Section IV-A. With respect to the circuit in Fig. 4 assume that line c, an input to gate $G_T = g_1$, is the target wire, i.e., $w_T = c$ needs to be removed.

[1]An alternative approach that returns the same results runs ATPG for S stuck-at 0.

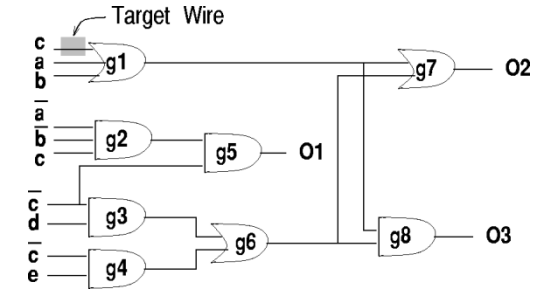

Fig. 4. Original circuit.

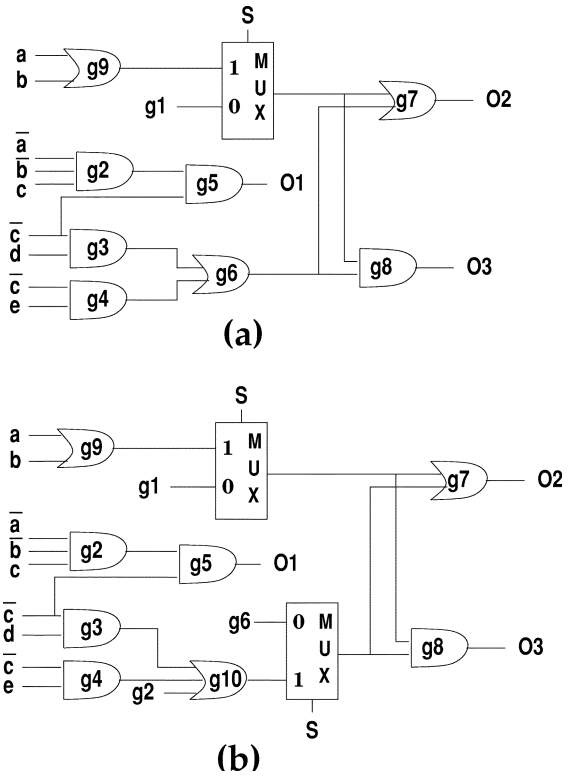

Fig. 5. (a) Test generation. (b) Circuit verification.

During the first two steps of the algorithm, shown in Fig. 5(a), gate $G'_T = g_9$ is introduced, that is, a gate similar to g_1 with w_T not present in its inputs. A multiplexer *MUX* with inputs the outputs of g_1 and g_9 and select line S is also introduced. For simplicity, in Fig. 5(a) we use g_1 to represent the circuitry that implements the respective Boolean function in Fig. 4. As explained in Section IV-A, an input test vector set $\mathcal{V}$ that detects the fault S stuck-at 1 is also a set of vectors with erroneous primary output responses when w_T is removed from the circuit in Fig. 4.

In the third step, DEDC is performed. The input to the DEDC algorithm is the original circuit (Fig. 4), the erroneous one (Fig. 4 with w_T removed), and vector set $\mathcal{V}$. The DEDC algorithm returns with the actual error (missing input wire c to g_1) and a set of equivalent corrections that includes missing input wire $w_A = g_2 \rightarrow g_6$.

To verify the final design, gate g_{10} is introduced with input lines g_3, g_4, and g_2, that is, the equivalent to gate g_6 in Fig. 4

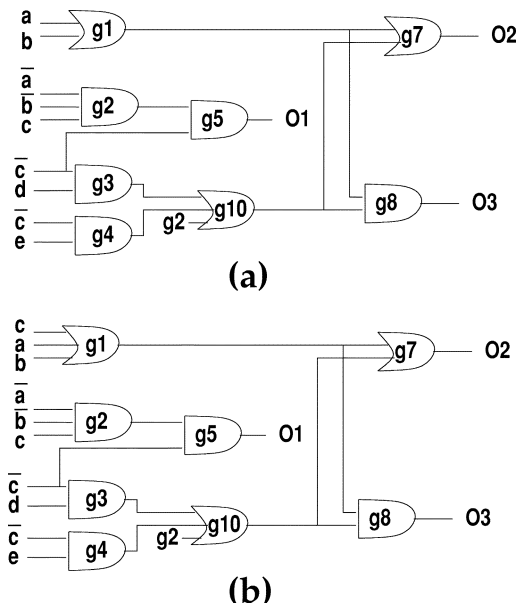

Fig. 6. (a) Final design. (b) An erroneous design.

with w_A added as an input. A second multiplexer with the same select line S as the first one and inputs g_6 and g_{10}, as shown in Fig. 5(b), is also attached in the circuit. An ATPG procedure for fault S stuck-at 1 indicates that the fault is redundant. Therefore, the circuits in Figs. 4 and 6(a) have the same logic functionality.

Observe that the solution returned by ADDR cannot be found by a conventional single wire RAR procedure. This is demonstrated in Fig. 6(b) where both w_A and w_T are present in the circuit. However, this circuit does not implement the same Boolean function at the primary outputs with the original one (Fig. 4) as input test vector $(a, b, c, d, e) = (0, 0, 1, 0, 0)$, for example, causes a failing response at O_3. Therefore, w_A is not redundant in the presense of the target wire w_T.

C. Discussion

At this point, it is of interest to discuss the characteristics of the method and compare it with existing techniques.

A significant advantage of the proposed method is found in its ease and flexibility in handling different types of target logic and corrections (Fig. 2) because it uses DEDC. For example, if a wire removal does not have a single correction, the target wire may be replaced with some other wire to introduce an error. As a result, the Boolean function at the primary outputs of the circuit may be altered differently and equivalent corrections may exist.

In general, the method allows one to arbitrarily resynthesize the function of a line(s) and correct this discrepancy somewhere else. It also makes the process of multiple logic transformations a straightforward extension of the four step process in Section IV-A, provided the use of an efficient multiple DEDC algorithm. Theoretical and experimental results, presented later in this paper, emphasize the importance and motivate the development of such algorithms.

Existing techniques identify corrections using mandatory assignments (MA) [14]. The more MAs available, the more corrections they return, but computing MAs is NP-hard and no method performs such an exhaustive computation. Instead, a subset of them is identified efficiently using structural, ATPG, and problem-specific observations [3]–[6], [9], [10]. By construction, these techniques perform a redundancy checking for *every* candidate correction. Since redundancy checking dominates the run-time, the work in [3]–[5], [9] aims to reduce the total number of unnecessary redundancy checks.

ADDR uses ATPG to return a few test vectors with erroneous output responses in Step 2 of the algorithm. ATPG is NP-complete and there exist tools that derive such test vectors efficiently. DEDC in Step 3 uses these vectors, as well as precomputed vectors for stuck at faults and random vectors, to return *all* possible corrections in *linear time*, according to (1). Finally, Step 4 verifies each correction in terms of a single redundancy checking on the common select line S.

In Section V, we show that, in theory, the complexity of Step 4 equals that of existing techniques. This complexity seems to be inherent to the problem of design rewiring. Experiments in Section VI suggest that, in practice, simulation-based DEDC screens most invalid candidates to help avoid unnecessary redundancy checks and improve performance.

V. Complexity Analysis

In this section we discuss complexity requirements and efficient implementations of the method presented in Section IV-A using ATPG. This study concludes with a new set of interesting results. During this presentation, we assume that any test pattern may occur at the primary inputs of the design, i.e., there are no *external don't care constraints*. In Section V-D, we relax this assumption and discuss its implications.

In this complexity analysis, we model the process of error and correction introduction with the injection of multiple (simultaneous) self-masking pattern faults. Let C be a circuit and let C' be the circuit after a number of logic (structural) transformations on a gate G of C. We define a *pattern fault* f in C' to be a combination of logic values on a set of circuit lines such that, if these logic values can be consistently justified, the logic value at the fan-out of G in C and C' are complementary. We also allow a pattern fault to be any set of pattern faults on possibly different gates in C by recursive application of the definition.

Observe that a pattern fault associates a set of *unique logic value conditions* on lines of the circuit such that the output of the resynthesized gate G becomes incorrect in C'. These conditions may be satisfied by a possibly nonempty set of input test vectors that *excite* the fault. Some of these vectors may also propagate the discrepancy at G to some primary output, that is, they *detect* the fault.

Let C be a design and C' be the design after the introduction of n pattern faults $f_1, f_2, \ldots, f_n$ on m different gates (lines) of C. We say that pattern fault $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ is *self-masked* if and only if C and C' are functionally equivalent. For brevity, in the remaining discussion, we use the term *fault* to refer to a pattern fault, unless otherwise stated. We also use the terms self-masking fault(s) and redundant fault(s) interchangeably.

Different logic transformation types can be modeled by a set of faults. Recall the various error (correction) types introduced in Fig. 2. A missing input wire error can be modeled by a fault

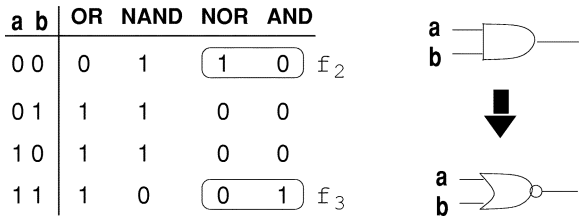

Fig. 7. Fault modeling of gate replacement error.

f_1 with set of excitation conditions $f_1 = \{a = 0, b = 0, c = 1\}$. These logic values give a logic 1 at the output of the gate in the original circuit C and a logic 0 in the new one C'. The reverse situation occurs for extra input wire error where the same set of conditions give a logic 0 in C and a logic 1 in C'.

Notice that this pattern fault formulation implies the stuck-at fault formulation for logic transformations adopted by RAR. Therefore, it comes as no surprise that similar conditions are presented for these two error types in [3]–[7], [9], [10]. Moreover, the presented formulation can map any piece of arbitrary logic resynthesis to a set of pattern fault(s) by enumerating all necessary excitation conditions on the error injected lines in C'. In fact, ATPG at Steps 2 and 4 of the design rewiring algorithm performs such an enumeration.

Unlike the above error types which can be modeled by a single fault, there are error types that require multiple faults (conditions) to completely justify all error effects. Consider the NOR to AND gate replacement error from Fig. 2, for instance. As the truth table for two-input gates in Fig. 7 reveals, there are two instances that this error is excited, each of which is modeled in terms of a distinct fault, f_2 and f_3. Other (nonnegated) gate type replacements are modeled similarly. An incorrect input wire also requires two faults. With respect to Fig. 2, these faults are $f_4 = \{a = 0, b = 1, c = 0\}$ and $f_5 = \{a = 0, b = 0, c = 1\}$.

With this formulation in mind, the problem of design rewiring can be stated as follows.

Definition 1: Design rewiring is the problem where given an (artificially introduced) error(s) modeled by fault $\mathcal{F}_E = \{f_1, f_2, \ldots, f_i\}$ at m lines of C we seek a correction(s) modeled by fault $\mathcal{F}_C = \{f_{i+1}, f_{i+2}, \ldots, f_n\}$ at p lines of the erroneous C ($p, m \geq 1$) so that fault $\mathcal{F} = \{f_1, f_2, \ldots, f_n\} = \mathcal{F}_E \cup \mathcal{F}_C$ in the new circuit C' is redundant.

As a side note, Definition 1 gives rise to a similar definition for brute-force DEDC with the exception that $\mathcal{F}_E$ and the set of m lines are not known, thus its inherent complexity during test generation and verification. Additionally, the introduction of fault $\mathcal{F}$ in the circuit may make more faults $f_{n+1}, f_{n+2}, \ldots, f_k$ redundant [8], [21], that is, $\mathcal{F} \cup \{f_{n+1}, f_{n+2}, \ldots, f_k\}$ remains redundant. Algorithms to identify such new redundancies, in favor of design optimization, have been developed in [4]–[6], [9], and [10] and apply to the presented work as well.

Under the presense of faults that model the error and the correction, the simulation of a test vector in C and C' may give different logic values at corresponding lines. To aid our presentation, we use Roth's nine-valued alphabet [18] with logic values taken from the set $\{0/0, 1/1, 0/1, 1/0, 0/X, X/0, 1/X, X/1, X/X\}$ to represent the logic value of a line in the *original/new* circuit C/C', respectively. Using Roth's alphabet, the redundancy requirement in Definition 1 implies that no 0/1 or 1/0 propagates to a primary output under the presense of $\mathcal{F}$.

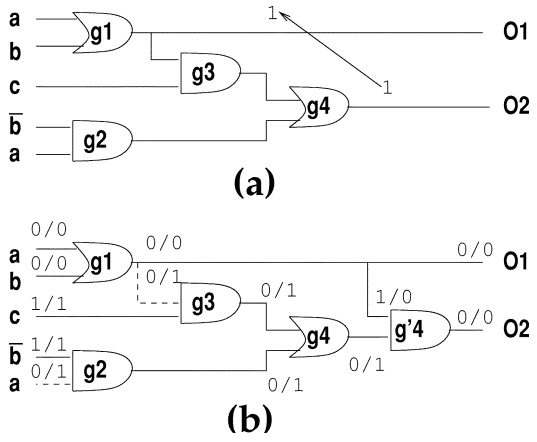

Fig. 8. Implication-based RAR.

The following examples illustrate the above concepts.

Example 2: The work by Kunz *et al.* [15] presents an RAR method which optimizes a circuit using *Boolean division* operations. In this example, borrowed from [15], we review the method and formulate it within the framework presented here.

With recursive learning [14], one finds that logic 0 on g_1 implies a logic 0 on g_4, that is, $g_1 = 0 \Rightarrow g_4 = 0$ in Fig. 8(a). This *logic implication* is equivalent to $g_4 = 1 \Rightarrow g_1 = 1$ by contraposition which allows [15] for g_4 to be replaced by $g_4' = g_4 g_1$ (ATPG-based Boolean division). This redundant transformation (correction), shown in Fig. 8(b), makes connections $g_1 \rightarrow g_3$ and $a \rightarrow g_2$ redundant (errors). Removing these connections in Fig. 8(b) leads to an optimized design with three gates.

We now translate this sequence of operations into the present operational framework. The addition of g_4' is equivalent to the injection of fault f_1 ($= \mathcal{F}_C$) with excitation conditions $\{g_4 = 1, g_1 = 0\}$ that cannot be simultaneously met in Fig. 8(a), thus, it is redundant. The two errors are represented with faults $f_2 = \{c = 1, g_1 = 0\}$ and $f_3 = \{a = 0, b = 0\}$ ($\mathcal{F}_E = \{f_2, f_3\}$), respectively.

Observe that fault $\mathcal{F} = \{f_1, f_2, f_3\}$ is redundant since no test vector propagates a 0/1 and/or a 1/0 value at a primary output(s) for any combinations of faults from $\mathcal{F}$ [21]. To see this, in Fig. 8(b) we attach the logic values on lines of C/C' when f_1 is excited and $c = 1$. The case when $c = 0$ is similar. To simplify the presentation, the dotted wires are pseudo-inputs with stable noncontrolling logic value 1. Notice that when f_1 is excited, faults f_2 and f_3 are excited. The reader can verify that the excitation of f_3 excites f_1 and the excitation of f_2 excites f_1. In all cases, the multiple faults are redundant.

Example 3: We re-examine the example in Section IV-B, redrawn in Fig. 9 for convenience. In that circuit, there are two faults involved, the error $c \rightarrow g_1 = \mathcal{F}_E = \{f_1\} = \{c = 1, a = 0, b = 0\}$ and the correction $g_2 \rightarrow g_{10} = \mathcal{F}_C = \{f_2\} = \{g_2 = 1, g_3 = 0, g_4 = 0\}$.

Fig. 9(a) contains the situation where both the error and the correction are present in the final circuit. Similar reasoning to

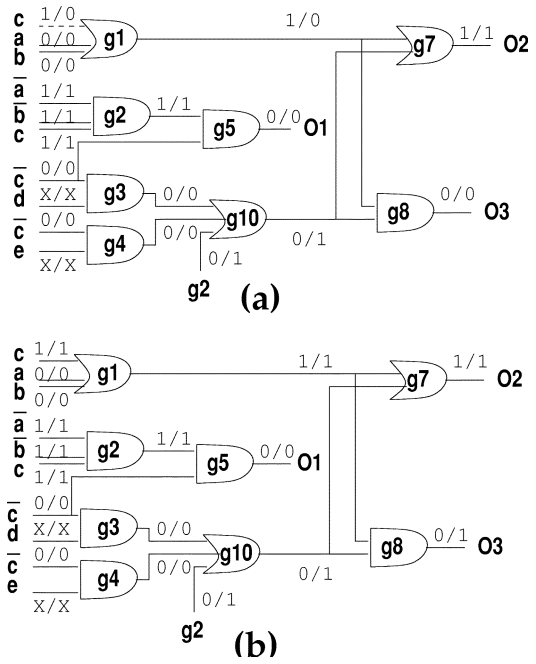

Fig. 9. Example of Section IV-B, revisited.

the one in Example 2 shows that fault $\mathcal{F} = \mathcal{F}_E \cup \mathcal{F}_C$ is redundant. Observe that meeting the excitation conditions of one fault excites the other. Section V-B shows that this is not a coincidence and it prompts toward design rewiring specific DEDC algorithms. On the other hand, fault $\mathcal{F}' = \{f_2\}$ is *not* redundant, as illustrated in Fig. 9(b).

A. ATPG and DEDC (Steps 2 and 3)

The focus is on the complexity requirements of ATPG (Step 2) and DEDC (Step 3) of the algorithm in Section IV-A. We perform this study in terms of the set of test vectors that detect faults $\mathcal{F}_E$ and $\mathcal{F}_C$ in Definition 1.

Consider a single execution of the proposed design rewiring algorithm. Design C is first corrupted with some error(s) modeled by fault $\mathcal{F}_E$. Let C^I denote the *intermediate circuit* after this error introduction operation, that is, C^I is functionally equivalent to C under the *presense* of fault effects from $\mathcal{F}_E$. Next, a correction(s) is applied on some lines of C^I to give circuit C' such that $C \equiv C'$. This correction(s) is modeled by fault $\mathcal{F}_C$.

Observe that C^I can be similarly defined as C' prior to the correction(s), that is, C^I is functionally equivalent to C' under the *absence* of fault effects from $\mathcal{F}_C$. This dual definition for C^I is due to the symmetric nature of DEDC: Any error/correction solution in C is a correction/error solution in C'. The locations and excitation conditions of the pattern faults $\mathcal{F'}_E/\mathcal{F'}_C$ associated with this correction/error in C' are in one-to-one correspondence to the ones in $\mathcal{F}_C/\mathcal{F}_E$ with complementary logic values at the respective gates.

Motivated by these observations, Theorem 1 classifies test vectors that detect $\mathcal{F}_E$ and $\mathcal{F}_C$. The proof of this theorem is a straightforward extension of the discussion above.

Theorem 1: Let faults $\mathcal{F}_E$ and $\mathcal{F}_C$ from Definition 1 and $\mathcal{F'}_E$ and $\mathcal{F'}_C$ as defined above. Test vector t detects some faults from $\mathcal{F}_E$ on a set of lines L^e in C^I if and only if t detects some faults from $\mathcal{F'}_E$ on a set of lines L^c in C^I.

Example 4: In Fig. 8, gate g_4' is added (correction) and wires $g_1 \rightarrow g_3$ and $a \rightarrow g_2$ are removed (errors). In Example 2, these logic transformations are modeled by faults f_1, f_2 and f_3, respectively. Assume that faults f_2 and f_3 are injected in the circuit of Fig. 8(a). Depending on the value of c, every vector with erroneous responses detects either: 1) f_2 and f_3 or 2) f_3. Theorem 1 suggests that these are also all the vectors that detect correction "replace g_4' by g_4" in Fig. 8(b), which is the case indeed.

Example 5: Consider the circuit under verification in Fig. 5(b) where $\mathcal{F}$ consists of faults f_1 = "missing input wire c to g_9" and f_2 = "extra input wire g_2 to g_{10}". According to [21], the redundancy of S stuck-at 1 fault is equivalent to the redundancy of: 1) f_1; 2) f_2; and 3) $f_1 \cup f_2$. Theorem 1 implies that any ATPG tool that attempts to prove redundancy of 1) will excite 2) to cancel the error effects of 1) and vice versa. The reader can verify this effect. Due to the containment property, the tool will not attempt 3). In other words, proving the redundancy of pattern fault $\mathcal{F}$ equals proving the redundancy of two single single stuck-at faults f_1 and f_2 independently.

Theorem 1 establishes a relation between the test vector(s) t that detect the error(s) and the ones that detect the correction(s) via the sets of pattern faults and their associated locations L^e and L^c. We view the merits of this theorem first for DEDC and then for ATPG.

In brute-force DEDC, the error location $\mathcal{L}^e$ is not known and no such test vector classification is possible, as discussed previously. DEDC for single errors remains efficient because all error effects originate from a single line and linear-time fault simulation algorithms are applicable [23]. On the other hand, if errors/corrections are present in multiple locations, the solution space explodes exponentially with the number of distinct error locations according to (1). When DEDC is used in design rewiring the case is different. Since the error location(s) is known, for every test vector t the set L^e can be computed and DEDC is presented with the additional information of Theorem 1. Although there is little to gain for the single error case, in light of this information, we believe that efficient *design rewiring specific DEDC* algorithms can be designed to tackle the multiple error/correction case.

Theorem 1 also suggests that ATPG should target every fault from $\mathcal{F}_E$ in the care set of the respective line(s) independently to aid DEDC resolution. Traditionally, ATPG is carried in two steps. The first step excites the fault and the second step propagates the fault effects to some primary output. Since all faults in $\mathcal{F}_E$ have unique excitation conditions, one can easily modify an ATPG engine to enumerate all required excitation conditions. However, this is not necessary and tradeoffs can be considered. We discuss some tradeoffs here and we conclude in Section VI.

Since the error effects of some faults from $\mathcal{F}_E$ may originate from a single line, one may run ATPG only on a subset of them to aid diagnosis. Next, a DEDC algorithm can return all corrections. The net effect is that some corrections may not

verify during simulation-based verification by DEDC (Step 3) or during Step 4 that perform such an exhaustive fault enumeration. If more time is spent in ATPG at Step 2, less time is expected to be spent in DEDC/verification and vice versa. In both cases, the set of corrections obtained is the same.

B. Multiple Fault Redundancy Checking (Step 4)

Step 4 of the algorithm verifies the correctness of the new design C' in terms of a redundancy checking for the stuck-at 1 fault on the common select line of all multiplexers. According to Definition 1, C' is structurally produced from C through a set of logic transformations represented by fault $\mathcal{F}$. As such, Step 4 checks the redundancy of underlying fault $\mathcal{F}$.

Proving the redundancy of multiple and simultaneous faults has been a well-examined problem of prominent importance due to its implications in logic testability [8], [21]. The following theorem, a simple restatement of the result by Smith [21], gives a necessary and sufficient condition for multiple fault undetectability.

Theorem 2: A fault $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ on m lines, $n \geq m$, in a circuit is redundant if and only if for each nonempty set $F_i \subseteq \mathcal{F}$ there exists nonempty set $F_j \subseteq \mathcal{F}$ such that $F_i \cup F_j$ is redundant.

As Smith's theorem [21] indicates, the complexity of redundancy checking for a set of n faults necessitates a computation of *exponential* (in n) size for modern ATPG tools as it requires enumeration and redundancy checking for every fault combination. Nevertheless, the presented fault-based formulation and the construction in Section IV-A allows us to capture nicely this complexity in the redundancy checking of a *single* fault.

Theorem 3 that follows formalizes this idea which, to the best of our knowledge, is the first result to allow efficient multiple fault redundancy checking. Since ATPG [11], [13], [20], [14] is very efficient when verifying single fault redundancies it also makes it a robust platform to implement the proposed design rewiring approach. Theorem 3 can also provide a proof that checking the redundancy of n faults is NP-complete.

Theorem 3: A fault $F = \{f_1, f_2, \ldots, f_n\}$ on m lines, $n \geq m$, in a circuit is redundant if and only if the stuck-at 1 fault on the common select line for the m multiplexers of the construction in Section IV-A is redundant.

C. Design Rewiring With Constraints

Consider the assumption at the beginning of the section that every test may occur at the primary inputs of the design. This assumption can be relaxed in favor of design optimization as follows.

Assume design C with r number of primary inputs and a structurally identical design C_C operating under a set of external don't care constraints. In other words, the complete input test vector set for C_C has strictly less than 2^r members. Given an error, Fig. 3 implies that an input test vector may reduce the solution (error location and/or correction) space for simulation-based DEDC but it never increases it. Therefore, for a fixed error, C_C is expected to have *at least* as many corrections as C.

Sets of external constraints can be taken into account by the presented design rewiring method if ATPG (Step 2 and 4) avoids generating input test patterns that belong in these sets or if DEDC ignores such test patterns when generating a solution. The discussion in the previous paragraph implies that ignoring test sets may increase the correction space in favor of optimization.

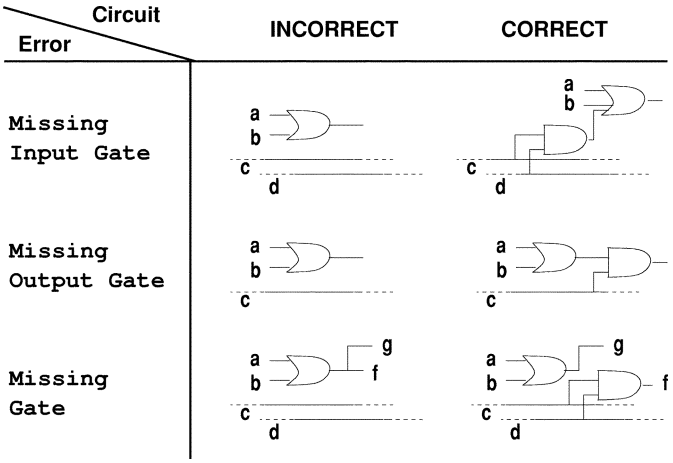

Fig. 10. Additional correction types.

VI. Experiments

We implemented the algorithm in Section IV-A in C and ran it on an Ultra 10 SUN workstation for ISCAS'85 benchmark circuits optimized for area using script.rugged in SIS [19]. The details of the ATPG and DEDC algorithms we employed can be found in [11], [14] and [23], respectively.

DEDC bases its results on a set of input test vectors comprised of the vectors returned by ATPG (Step 2), a small number of random vectors, and vectors for stuck-at faults [12]. Prior to execution, DEDC simulates 2000–3000 random test vectors to create a bit-list on each line of the circuit as in [23]. The ith entry of this list for line l contains the logic value of l when the ith vector is simulated. Intuitively, the logic values maintained in these bit-lists behave as an approximation of the Boolean function implemented at the respective line. We say that two lines have *similar* logic values if most of their respective bit-list entries are the same. Using this setup, we run two different experiments and report the average values of the results obtained.

In the first experiment, for every wire w_T in the circuit, we inject one error to eliminate it and we count the number of equivalent corrections. We consider three error types.

- *Type A:* remove w_T.
- *Type B:* replace w_T with an existing 75% similar wire.
- *Type C:* replace w_T with an existing 50% similar wire.

With respect to Figs. 2 and 10, the correction types DEDC uses are as follows:

- *Type 1:* gate replacement;
- *Type 2:* incorrect input wire;
- *Type 3:* extra input wire;
- *Type 4:* missing input wire;
- *Type 5:* missing input gate;
- *Type 6:* missing output gate and missing gate.

Corrections with two wires, such as missing input gate and missing gate, require quadratic time for DEDC. Heuristics to

TABLE I
PERFORMANCE CHARACTERISTICS

ckt name	# of lines	avg. # of corrections per type			% of correction types						CPU (sec)
		type A	type B	type C	type 1	type 2	type 3	type 4	type 5	type 6	
C432	412	7.4	2.7	2.6	0	57	0	15	6	22	0.2
C499	1249	8.2	3.0	2.1	0	26	0	4	39	31	0.3
C880	915	5.3	2.2	1.7	0	44	0	8	20	28	0.2
C1355	1238	8.1	2.2	1.7	0	23	0	6	41	30	0.3
C1908	859	7.6	3.8	3.6	1	25	0	7	30	37	0.6
C2670	1377	11.6	15.3	14.0	1	8	1	3	9	78	0.5
C3540	2282	18.7	4.0	3.6	1	42	0	11	34	12	0.6
C5315	3697	7.2	3.7	2.5	0	36	0	10	30	24	0.7
C6288	6319	12.1	21.7	16.1	1	15	0	2	8	74	0.8
C7552	5262	10.3	7.1	9.1	1	53	1	2	2	41	0.9

TABLE II
COMPARISON OF RESULTS AND OTHER STATISTICS

ckt name	# corrections		% hit-ratio		ADDR total # corrections	% same gate	% dom. gate	RAR # redund. check. per corr	% with pair corrections
	ADDR	RAR	ADDR	RAR					
C432	1204	1011	70	63	1989	75	42	18.2	0
C499	4989	886	68	52	12112	85	11	432.5	24
C880	2299	945	65	61	3891	90	26	72.8	19
C1355	5515	1022	69	54	7311	67	6	371.6	21
C1908	3174	643	65	56	6711	83	3	102.2	12
C2670	14922	2247	76	51	17311	61	49	47.3	62
C3540	8478	7801	68	62	16197	88	22	120.2	24
C5315	10665	3077	70	45	17833	72	6	110.0	17
C6288	18683	1615	52	23	35918	60	31	62.3	19
C7552	20349	12234	80	56	31766	67	22	58.8	41

speed the search process for such corrections are developed in [4], [5], and [25]. For wire related corrections, wires that do not create loops in the combinational circuitry are considered. We allow adding an inverter if it increases the potential to find a correction.

General information on the performance of the algorithm can be found in Table I. The first two columns contain circuit characteristics. The next three columns show the average number of equivalent corrections returned for each error type independently. These average values are a conservative estimate as we set a user-defined limit on the maximum number of missing input gate (Type 5) and missing gate (Type 6) corrections that we consider.

We observe that removal of w_T returns more corrections, on the average, compared to the other two error types. This may be explained because the number of conditions involved with the set of pattern faults for incorrect input wire is more than that for missing input wire, as explained in Section V; thus, it is harder to correct it. In the experiments, we also observed that there is little overlap between the sets of corrections returned for different error types on the same w_T. This confirms the flexibility of ADDR since the designer is presented with more opportunity to eliminate the target logic *and* correct it.

Columns 6–11 in Table I contain detailed information on the correction types used. It can be seen that certain types of corrections are more useful. The last column of the table contains the average run-time, in seconds, to find one equivalent correction. This number equals the CPU time for all four steps of the algorithm in Section IV-A. On the average, the time spent in ATPG and redundancy checking dominates the overall time which confirms the robustness of DEDC in design rewiring.

To demonstrate the potential of ADDR, it is of interest to compare its performance with the one of RAR. Table II contains information on the number of corrections returned by a recent RAR procedure [3] and by our method for wire removal error type (type A) and the same correction types (a subset of types $1 \ldots 6$). Compared to previous approaches, the work in [3] usually returns more alternatives because it considers adding logic not only at dominating gates but also at gates that have implied mandatory assignments.

Columns 2 and 3 in Table II show the number of corrections returned by ADDR and RAR [3] for the same set of error/correction type experiments. It is seen that the proposed method outperforms RAR as it returns more corrections. In fact, it returns all corrections since it uses a DEDC method exhaustive on the correction space. Moreover, columns 4 and 5 contain the percentage of target logic with alternative corrections (success hit-ratio) for the complete set of wire removal experiments. We observe that ADDR can find alternatives for target logic removal cases that RAR cannot in favor of design optimization. Our experiments also indicate that more than 99% of the corrections found by ADDR are redundant in presense of the error.

Due to the flexibility of DEDC to handle a wide variety of correction types, the total number of corrections returned by the method for error type A and correction types $1 \ldots 6$ is much larger (Column 6). This competitiveness is additionally justified if we consider the locations of the proposed corrections. Column 7 contains the percentage of corrections on the gate w_T

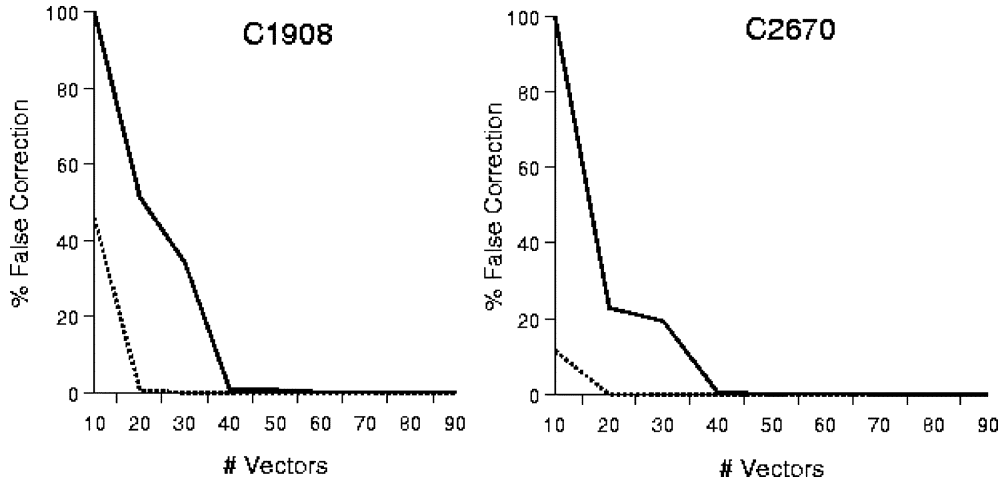

Fig. 11. Simulation-based verification.

drives. For the remaining ones, column 8 shows the percentage of corrections on a dominator of w_T. These numbers suggest that many corrections exist on nondominating gates.

To further demonstrate the effectiveness of simulation-based DEDC in design rewiring, Fig. 11 depicts the set of *false* corrections returned by DEDC for two benchmarks. Since simulation-based DEDC bases its results on a subset of the complete input test vector space, it is of interest to know the quality of these corrections for the complete input test vector space. This is because the fewer false corrections returned, the less time design rewiring spends in ATPG-based redundancy checking (Step 4), as pointed out in Section IV-C.

In that figure, a bold line indicates the percentage of false corrections returned when DEDC (Step 3) uses random vectors and a dotted one when stuck-at vectors [12] are included. The plots confirm results in [1] and [23] as a small number of vectors provides sufficient resolution to DEDC. As a result, most corrections qualify Step 4 and ADDR performs 1.1 redundancy checkings per nonfalse correction it finds. To appreciate this result, one needs compare it with the respective average for existing techniques [3], shown in column 9 in Table II. We conclude, that, in practice, ADDR performs far less redundancy checkings, an important computational saving. Fig. 11 also suggests that Step 2 of design rewiring may be occasionally omitted since vectors for stuck-at faults give sufficient resolution to DEDC.

In the second experiment, we randomly select and remove a target wire w_T to introduce an error and we try to correct it with a single correction. If no single correction exists, we are interested in having DEDC find two corrections that rectify it. The average values of the results are found in Table II.

Column 4 of this table shows the percentage of errors that can be corrected with a single correction, as explained earlier. For those errors that no single correction exists, the last column in Table II contains the ones that can be corrected with two corrections. We observe that a significant amount of errors cannot be corrected with a single correction but they can be corrected with a pair of corrections. Similar experiments in [4] confirm this result for a different suite of benchmark and industrial designs where a significant percentage of single wire related errors with no single alternatives have triple alternatives.

Since the success of design rewiring during optimization depends on its ability to eliminate target logic, it is evident that multiple corrections will increase the solution space and may return further gains. This suggests the development of efficient design rewiring specific multiple DEDC algorithms that will offer more alternatives to meet optimization goals, as discussed in Section V-B.

VII. Conclusion

We presented a new ATPG-based design rewiring methodology and discussed efficient implementation tradeoffs. This method injects an error to eliminate the target logic and uses a simulation-based design error diagnosis and correction algorithm to correct it. ATPG performs test generation and design verification. We also study the complexity requirements of this approach by reducing the process of error/correction injection to the process of injecting a set of multiple redundant pattern faults. This study arrives at a new set of interesting results and shows that ATPG-based design rewiring is efficient. Experiments confirm the theory and motivate future work in the field.

Acknowledgment

The authors would like to thank I. Ting, M. Amiri, and R. Chang for contributions to portions of the work described here.

References

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic verification via test generation," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 138–148, Jan. 1988.

[2] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.

[3] S. C. Chang and Z. Z. Wu, "Theory and extensions of single wire replacement," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 1159–1163, Sept. 2001.

[4] S. C. Chang, K. T. Cheng, N. S. Woo, and M. Marek-Sadowska, "Post-layout logic restructuring using alternative wires," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 587–596, June 1997.

[5] S. C. Chang and M. Marek-Sadowska, "Perturb and simplify: Multi-level Boolean network optimizer," in *Proc. Int. Conf. Computer-Aided Design*, 1994, pp. 2–5.

[6] S. C. Chang, J. C. Chuang, and Z. Z. Wu, "Synthesis for multiple input wires replacement of a gate for wiring consideration," in *Proc. Int. Conf. Computer-Aided Design*, 1999, pp. 115–118.

[7] M. Chatterjee, D. Pradham, and W. Kunz, "LOT: Logic optimization with testability – New transformations using recursive learning," in *Proc. Int. Conf. Computer-Aided Design*, 1995, pp. 115–118.
[8] R. Dandapani and S. M. Reddy, "On the design of logic networks with redundancy and testability considerations," *IEEE Trans. Comput.*, vol. C-23, Nov. 1974.
[9] J. A. Espejo, L. Entrena, E. San Millàn, and E. Olìas, "Functional extension of structural logic optimization techniques," in *Proc. Asian-South-Pacific Design Automation Conf.*, 2001, pp. 467–472.
[10] L. A. Entrena and K. T. Cheng, "Combinational and sequential logic optimization by redundancy addition and removal," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 909–916, July 1995.
[11] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Comput.*, vol. C-32, Dec. 1983.
[12] I. Hamzaoglu and J. H. Patel, "New techniques for deterministic test pattern generation," in *Proc. VLSI Test Symp.*, 1998, pp. 446–452.
[13] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," in *IEEE Design Automation Conf.*, 1987, pp. 502–508.
[14] W. Kunz and D. K. Pradhan, "Recursive learning: A new implication technique for efficient solutions to CAD problems-test, verification, and optimization," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1143–1158, Sept. 1994.
[15] W. Kunz, D. Stoffel, and P. R. Menon, "Logic optimization and equivalence checking by implication analysis," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 266–281, Mar. 1997.
[16] Y. M. Jiang, A. Krstic, K. T. Cheng, and M. Marek-Sadowska, "Post-layout logic restructuring for performance optimization," in *IEEE Design Automation Conf.*, 1997, pp. 662–665.
[17] B. Rohfleisch, A. Kolbl, and B. Wurth, "Reducing power dissipation after technology mapping by structural transformations," in *Proc. Design Automation Conf.*, 1996, pp. 789–794.
[18] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM J. Res. Development*, vol. 10, pp. 278–291, June 1966.
[19] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design*, 1992, pp. 328–333.
[20] M. H. Schulz and E. Auth, "Improved deterministic test pattern generation with applications to redundancy identification," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 811–816, July 1989.
[21] J. E. Smith, "On necessary and sufficient conditions for multiple fault undetectability," *IEEE Trans. Comput.*, vol. c-28, pp. 801–802, Oct. 1979.
[22] G. Stenz, B. M. Riess, B. Rohfleisch, and F. M. Johannes, "Performance optimization by interacting netlist transformations and placement," *IEEE Trans. Computer-Aided Design*, vol. 19, Mar. 2000.
[23] A. Veneris and I. N. Hajj, "Design error diagnosis and correction via test vector simulation," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 1803–1816, Dec. 1999.
[24] A. Veneris, M. S. Abadir, and I. Ting, "Design rewiring based on diagnosis techniques," in *Proc. Asian-South-Pacific Design Automation Conf.*, 2001, pp. 479–484.
[25] A. Veneris, M. Amiri, and I. Ting, "Design rewiring for power minimization," in *IEEE Int. Symp. Circuits Systems*, 2002.

Andreas Veneris (S'96-M'99) was born in Athens, Greece. He received the Diploma in computer engineering and informatics from the University of Patras, in 1991, the M.S. degree in computer science from the University of Southern California, Los Angeles, in 1992, and the Ph.D. degree in computer science from the University of Illinois, Urbana-Champaign, in 1998.

He is currently an Assistant Professor at the University of Toronto, cross-appointed with the Department of Electrical and Computer Engineering and the Department of Computer Science. His research interests include CAD for synthesis, diagnosis, and verification of digital circuits and combinatorics. He is coauthor of one book.

Dr. Veneris was a corecipient of a best paper award in ASP-DAC'01. He is a member of the ACM, AAAS, Technical Chamber of Greece, and the Planetary Society.

Magdy S. Abadir (SM'00) received the B.S. degree with honors in computer science from the University of Alexandria, Egypt, in 1978, the M.S. degree in computer science from the University of Saskatchewan, Saskatoon, Canada, in 1981, and the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, in 1985.

Currently he is with Motorola working as the Manager of the High Performance Tools and Methodology Group at the Advanced Systems and Platform Group in Austin, TX. Prior to that, he was the General Manager of Best IC Labs in Austin Texas. From 1986 to 1994, he worked at the Microelectronics and Computer Technology Corporation (MCC). He is also an adjunct faculty member of the Computer Engineering Department at the University of Texas, Austin. He has published over 100 technical journal and conference papers in the areas of microprocessor test and verification, test economics, expert systems, and design for test. He co-edited three books on the subject of test economics.

Dr. Abadir founded and chaired three workshops on microprocessor test and verification. He also co-chaired five workshops on the economics of design, test, and manufacturing.