Design Rewiring Using ATPG *

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

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

Mandana Amiri
University of Toronto
Dept. ECE
Toronto, ON M5S 3G4
amiri@eecg.toronto.edu

Abstract

Technology dependent logic optimization is usually carried through a sequence of design rewiring operations. In [18] a new design rewiring method is proposed that combines error diagnosis and correction techniques with ATPG. In this work, we examine its complexity and we arrive to a new set of results with interesting theoretical and practical applications. We also present experiments that confirm the competitiveness of the approach and motivate future work in the field.

1 Introduction

In logic optimization, the gate level implementation obtained from high-level synthesis tools is modified to meet different constraints such as minimizing the area and power, satisfying timing constraints, or improving the testability of the circuit. Recently, ATPG-based design rewiring techniques for technology dependent logic optimization [3] [4] [5] [6] [7] [11] [16] have gained increasing popularity.

In general, existing ATPG-based techniques optimize a design through an iterative sequence of *Redundancy Addition/Removal (RAR)* operations. At each step of this sequence, a target wire is first identified that violates some optimization constraint(s). Next, some redundant logic is added in the design to make the target wire redundant so it can be removed. Finally, these methods delete any newly generated redundancies to further optimize the design.

*This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under Contract #227044–00.

For example, consider the circuit in Fig. 1(a) [5] where dotted wire $g5 \rightarrow g9$ is not part of the original netlist. Assume that wire $w_T = g1 \rightarrow g4$, named target wire hereafter, needs to be removed. To remove it, these techniques identify new redundant connection such as $w_A = g5 \rightarrow g9$ that makes w_T redundant. Subsequently, they add w_A and delete w_T as well as other newly generated redundancies, such as $g_6 \rightarrow g_7$, to obtain an optimized circuit shown in Fig. 1(b).

In [18], a new ATPG/Diagnosis-based Design Rewiring (ADDR) design rewiring operational framework is presented. This new methodology combines existing *Design Error Diagnosis and Correction (DEDC)* techniques [1] [18] with advances in ATPG [8] [10]. It can be shown [18] that this methodology has a number of significant characteristics that add and complement to existing techniques.

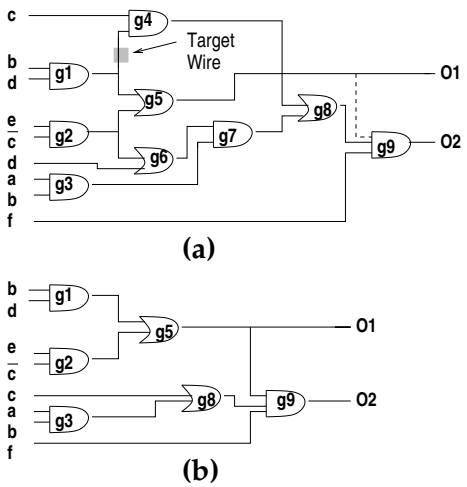

Figure 1: Optimization through rewiring

In this work we analyze the complexity requirements of the method by Veneris et al. [18]. In detail, we show that the unique features of the proposed approach

come at no additional complexity cost when compared to existing techniques. To prove this, in Section 3 we reduce the problem of design rewiring to the problem of *multiple and simultaneous self-masking pattern fault* [15] 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 is no harder than that for a single pattern fault, an interesting result that stands on its own. Therefore, recent advances in ATPG [8] [10] provide computationally efficient implementations for the method. Finally, we present experimental data that confirm the effectiveness of simulation-based DEDC as it helps design rewiring avoid unnecessary redundancy checks. The same experiments demonstrate the competitiveness of the approach and motivate future work in the field.

The paper is organized in five sections. In Section 2 we briefly review DEDC and the design rewiring method in [18]. A complexity analysis for this method is presented in Section 3. Experiments are found in Section 4 and Section 5 concludes this work.

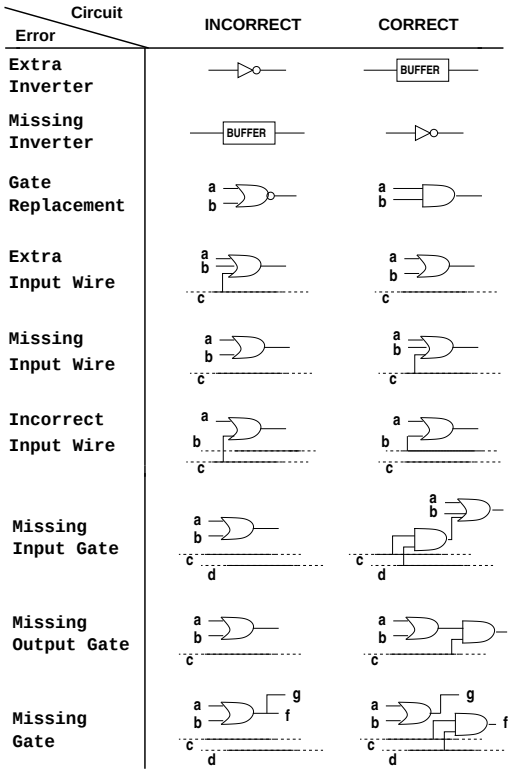

Figure 2: Design error types

2 Background

2.1 DEDC

Logic *design errors* are functional mismatches between the specification and the gate–level description [1] [17]. Most literature uses a design error (correction) model, *i.e.* a small predetermined set of possible error types, proposed by Abadir et al. [1]. A list of common design error types from this model is shown in Fig. 2. These errors are related to the present work. Gate types in Fig. 2 are indicative, similar errors can occur on other gate types as well.

In simulation-based DEDC, given an erroneous design, a specification, a design error model and a set of input test vectors, we need identify lines in the design that are potential sources of error (*diagnosis*) and suggest appropriate modifications from the design error model used that rectify it (*correction*) [1] [17]. Note that the set of corrections returned by a DEDC algorithm may contain some *equivalent* corrections. For example, if we introduce a design error by removing $w_T = g_1 \rightarrow g_4$ in Fig. 1, DEDC returns corrections $g_1 \rightarrow g_4$ (actual) and $g_5 \rightarrow g_9$ (equivalent).

Test vector generation and verification in DEDC are inherently difficult problems [1] [17] because the error location is not known. On the other hand, it has been theoretically proven by Abadir et al. [1] and experimentally confirmed in [17], that a complete test set for stuck-at faults as well as a small set of random vectors, detects the majority of single errors and it has a good chance to detect the remaining ones. For example, the first four errors in Fig. 2 are guaranteed to be detected with such a test.

Most DEDC techniques simulate test sets for stuck-at faults and random test vectors to diagnose and correct a design. Provided vectors with erroneous responses, these methods are very efficient, especially for single errors where their complexity is *linear* to the number of circuit lines [17]. Intuitively, these methods obtain a solution by *intersecting* the solution space offered by each vector, as shown in Fig. 3. In this work, we use a simulation-based DEDC algorithm [17] for single errors which is exhaustive on the solution space. The input to the algorithm is an erroneous netlist, its specification and a set of input test vectors. The output of the algorithm is a list of *all* applicable corrections.

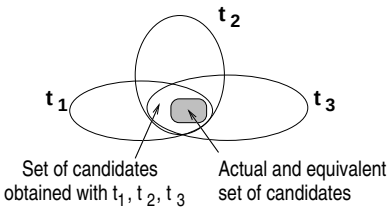

Figure 3: Resolution of simulation-based DEDC

2.2 ADDR

To eliminate w_T, ADDR [18] removes the wire artificially to introduce a "design error" and it uses DEDC to get all equivalent corrections that rectify the design. In general, this new view to design rewiring 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 process, provided the use of an efficient multiple DEDC algorithm. Multiple transformations have been computationally intensive for existing techniques [6] but they are important as they may increase the solution space in favor of optimization [4] [5]. Theoretical and experimental results, presented later in this paper, emphasize the importance and motivate the development of such algorithms.

In detail, ADDR performs the following four steps to eliminate target wire w_T:

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

In the following example [18], we review the implementation details in terms of a wire removal.

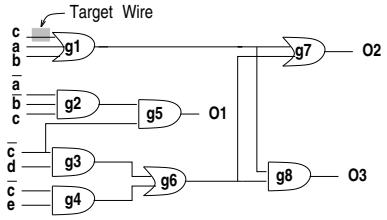

Figure 4: Original Circuit

Example 1: With respect to the circuit in Fig. 4 assume that line c, input to gate $G_T = g_1$, is the target wire that needs to be removed. 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 without w_T in its support. A multiplexer MUX with inputs being 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 indicate the circuitry that implements the respective boolean function in Fig. 4. Clearly, any input test vector set V that detects fault S stuck-at 1 contains test vectors that give erroneous primary output responses when w_T is removed from the circuit in Fig. 4 (Step 2). We derive such a set V with the use of ATPG [8] [10].

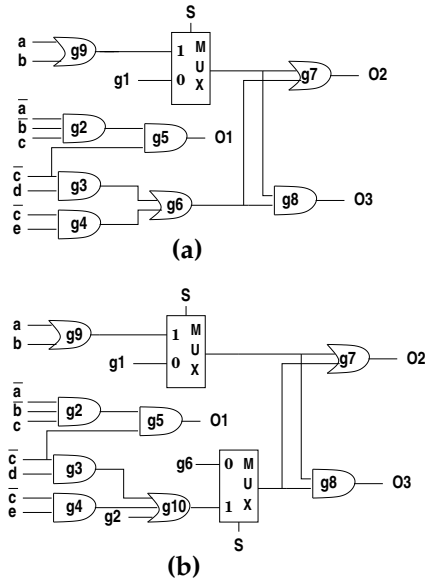

Figure 5: (a) Test Generation (b) Circuit Verification

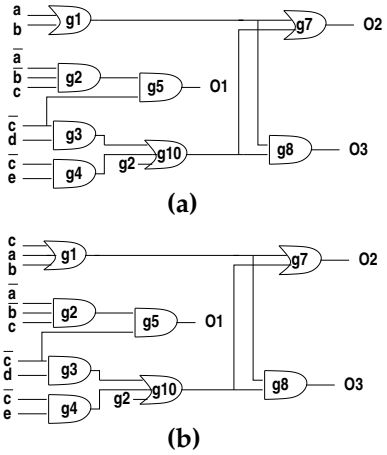

Figure 6: (a) Final Design (b) An Erroneous Design

During the third step of the algorithm, DEDC [17] 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 V. The DEDC algorithm returns with the actual error (missing input wire c to g_1) and a set of equivalent corrections that contains a missing input wire $w_A = g_2$ as an input to g_6.

To verify the correctness of the final design (step 4), 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 with w_A added as an input. A second multiplexer is attached with the same select line S as the first one and inputs g_6 and g_{10}, as shown in Fig. 5(b). An ATPG procedure for fault S stuck-at 1 verifies that the fault is redundant and the correction qualifies. Therefore, the circuits in Fig. 4 and Fig. 6(a) implement the same function at the primary outputs.

Observe that the solution returned by the proposed method cannot be found by RAR. In Fig. 6(b) both w_A and w_T are present but this circuit does not have the same functionality as the original one (Fig. 4) because 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 presense of the target wire w_T.

It is seen, that ADDR uses ATPG to return a few test vectors with erroneous output responses in Step 2 of the algorithm. DEDC in Step 3 uses these vectors, as well as pre-computed vectors for stuck at faults and random vectors, to return *all* possible corrections in *linear time*. Finally, Step 4 verifies these corrections in terms of single redundancy checking on the common select line S.

In Section 3 we show that the complexity of Step 4 equals that of existing techniques. This complexity seems to be inherent to the problem of rewiring. Experiments in section 4 suggest that simulation-based DEDC helps design rewiring avoid the vast amount of unnecessary redundancy checks to improve performance.

3 Complexity Analysis

In this Section we discuss complexity requirements and efficient implementations of the method in [18] 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 subsection 3.3 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 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 non-empty 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 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 [7].

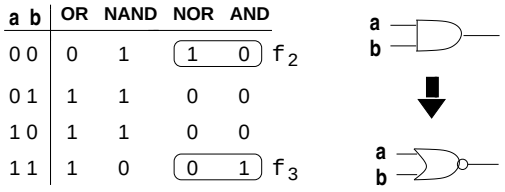

Figure 7: Fault modeling of a gate replacement error

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 (non-negated) 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 $f_5 = \{a = 0, b = 0, c = 1\}$.

It is seen, that 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 Step 2 and 4 of the design rewiring algorithm performs such an enumeration. 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 sidenote, the introduction of fault $\mathcal{F}$ in the circuit may make more faults $f_{n+1}, f_{n+2}, \ldots, f_k$ redundant [15], 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] [5] [6] [7] and apply to the presented work as well.

Under the presense of $\mathcal{F}$, the simulation of a test vector in C and C' may give different logic values at corresponding lines. To aid the presentation, we use Roth's 9-valued logic value alphabet [13] $\{0/0, 1/1, 0/1, 1/0, 0/X, X/0, 1/X, X/1, X/X\}$ 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 shows to a primary output under the presense of $\mathcal{F}$.

The following examples illustrate the above concepts.

Example 2: The work by Kunz et al. [11] presents an RAR method which optimizes a circuit using *Boolean division* operations. In this example, borrowed from [11], we review the method and formulate it within the framework presented here.

With recursive learning [10], 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 and allows [11] for $g_4' = g_4 g_1$ to be added at the output of g_4 as shown

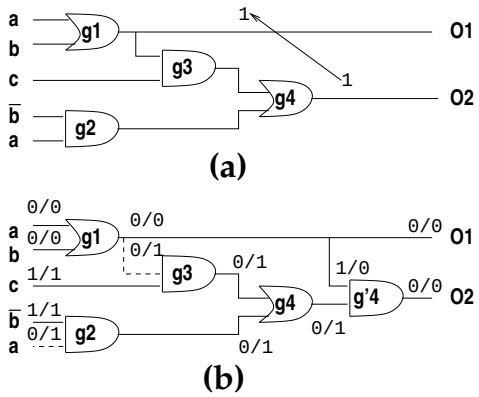

Figure 8: Implication-based RAR

in Fig. 8(b) (ATPG-based Boolean division). Adding redundant gate g_4' (correction) 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 3 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 $\mathcal{F}$ [15]. 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 non-controlling 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 Example 1, redrawn in Fig. 9 for convenience. There are two faults in that circuit, $c \rightarrow g_1 = \mathcal{F}_E = \{f_1\} = \{c = 1, a = 0, b = 0\}$ and $g_2 \rightarrow g_{10} = \mathcal{F}_C = \{f_2\} = \{g_2 = 1, g_3 = 0, g_4 = 0\}$.

Fig. 9(a) contains the situation where both the error and the correction are present in the final circuit. Similar reasoning to 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. Subsection 3.1 shows that this is not a coincidence and it prompts towards design rewiring specific DEDC algorithms. On the other hand, fault $\mathcal{F}' = \{f_2\}$ is *not*

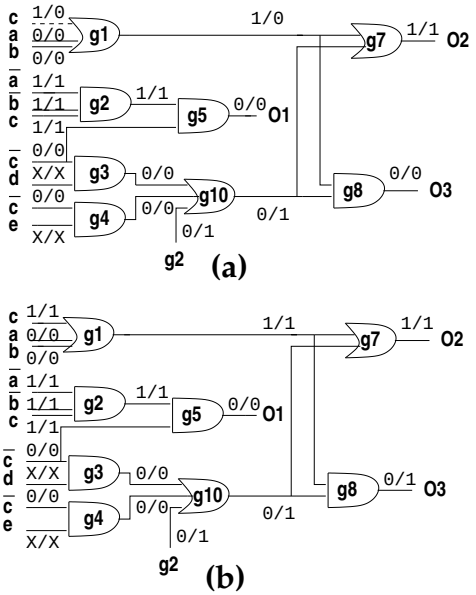

Figure 9: Example of subsection 2.2, revisited

redundant, as illustrated in Fig. 9(b).

3.1 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 [18]. 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, 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 straightforward from 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 *(i)* f_2 and f_3, or *(ii)* f_3. Theorem 1 suggests that these are also all the vectors that detect logic transformation "remove 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 [15], proving the redundancy of S stuck-at 1 fault is equivalent to proving the redundancy of *(i)* f_1, *(ii)* f_2 and *(iii)* $f_1 \cup f_2$. Theorem 1 implies that any ATPG tool that attempts to prove redundancy of *(i)* will excite *(ii)* to cancel the error effects of *(i)* and vice versa. The reader can verify this effect. Due to the containment property, the tool will not attempt *(iii)*. Therefore, in practice, 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. DEDC for single errors remains efficient (*i.e.,* linear [17]) because all error effects originate from a single line. If multiple errors/corrections are present, then the solution space explodes exponentially with the number of error locations. On the other hand, in design rewiring 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 trade-offs can be considered. We discuss some trade-offs here and we conclude in Section 4.

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 performs 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.

3.2 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 generated from the original design C by introducing an error(s) and a correction(s). These logic transformations can be uniquely represented by fault $\mathcal{F}$. As such, Step 4 is equivalent to checking the joint redundancy of the underlying faults.

Proving the redundancy of multiple and simultaneous faults has been a well examined problem of prominent importance due to its implications in logic testability [15]. The following theorem, a simple restatement of the result by Smith [15], 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 non-empty set $F_j \subseteq \mathcal{F}$ such that $F_i \cup F_j$ is redundant.

As Smith's Theorem 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 subsection 2.2 allows 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 that allows for efficient multiple fault redundancy checking using ATPG. Since ATPG 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 subsection 2.2 is redundant.

3.3 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 let 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 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.

4 Experiments

We implemented and run the algorithm in subsection 2.2 on an Ultra 10 SUN workstation for ISCAS'85 circuits optimized for area using SIS (`script.rugged`) [14]. We use the ATPG and DEDC engines from [12] and [17], respectively.

DEDC uses the vectors returned by ATPG (Step 2), a small number of random vectors and vectors for stuck-at faults [9]. Prior to execution, DEDC simulates 2,000-3,000 random test vectors to create an indexed bit-list of logic values on each line of the circuit as in [17]. 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 Fig. 2, correction types 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

For wire related corrections, we consider wires that do not create loops in the combinational circuitry. We also allow adding an inverter to an existing wire, as in [4] [5], if this increases the potential to find a correction.

General information on the performance of the algorithm can be found in Table 1. 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 conditions involved with the pattern faults for incorrect input wire are more than that for missing input wire, as explained in Section 3, thus, it is harder to correct them. 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 the proposed approach since the designer is presented with more opportunity to eliminate the target logic *and* correct it.

Table 1: Performance Characteristics

ckt name	# of lines	avg. # of corrections per type			CPU (sec)
		type A	type B	type C	
C432	412	7.4	2.7	2.6	0.2
C499	1249	8.2	3.0	2.1	0.4
C880	915	5.3	2.2	1.7	0.2
C1355	1238	8.1	2.2	1.7	0.3
C1908	859	7.6	3.8	3.6	0.4
C2670	1377	11.6	15.3	14.0	0.8
C3540	2282	18.7	4.0	3.6	0.6
C5315	3697	7.2	3.7	2.5	0.8
C6288	6319	12.1	21.7	16.1	0.8
C7552	5262	10.3	7.1	9.1	1.0

The last column of the table contains the average runtime, in seconds, to find one equivalent correction. This number equals the CPU time for all four steps of the algorithm in subsection 2.2. On the average, the time spent on ATPG (Steps 2 and 4) dominate the time spent for DEDC. This confirms the robustness of DEDC for the problem and the efficiency of the approach.

To demonstrate the potential of ADDR, it is of interest to compare its performance with the one of RAR. Table 2 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...6). Compared to previous approaches, the work in [3] 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 2 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 existing techniques as it returns more corrections. In fact, it computes all corrections since it uses a linear-time DEDC algorithm which is exhaustive on the solution space.

Columns 4 and 5 contain the percentage of target logic with alternative corrections (success hit-ratio) for all 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

Table 2: Comparison results and other statistics

ckt	# corrections		% hit-ratio		ADDR total	% same	% dom.	RAR # redund.	% with pair
name	ADDR	RAR	ADDR	RAR	# corrections	gate	gate	check. per corr	corrections
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

types 1..6 is much larger (Column 6). This is justified if we consider the locations of the proposed corrections. Column 7 contains the percentage of corrections on the gate w_T drives and column 8 shows the percentage of corrections on a dominator of w_T. These numbers suggest that corrections exist on non-dominating gates.

To further demonstrate the effectiveness of simulation-based DEDC in design rewiring, Fig. 10 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 lesser time design rewiring spends in ATPG-based redundancy checking (Step 4), as pointed out in subsection 3.1.

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 [9] are included. The plots confirm results in [1] [17] as a small number of vectors provides sufficient resolution to DEDC. As a result, most corrections qualify Step 4 and, on the average, ADDR performs 1.1 redundancy checkings per non-false 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 2. We conclude, that, in practice, ADDR performs far less redundancy checkings, an important computational saving. Fig. 10 also suggests that Step 2 of design rewiring may be occasionally omited 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 try to correct it with a single correction. If no single correction exists, DEDC tries to find two corrections to rectify it. The average values of the results are found in Table 2.

Column 5 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 2 contains the ones that can be corrected with two corrections. We observe that a significant amount of errors can be corrected only with two corrections. Similar experiments in [4] confirm this result for a different suite of benchmark and industrial designs. In detail, it is shown that 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 subsection 3.1.

5 Conclusions

We studied the characteristics and complexity requirements of the ATPG-based design rewiring methodology in [18]. To perform this, we reduce the process of design rewiring to the process of multiple self-masking pattern faults injection. The study arrives to a new set of interesting theoretical and practical results. Experiments confirm the theory and exhibit the competitiveness of the approach. They also motivate future research in the field.

Acknowledgments

We thank Prof. S. C. Chang and Z. Z. Wu who kindly provided us with the package that implements [3].

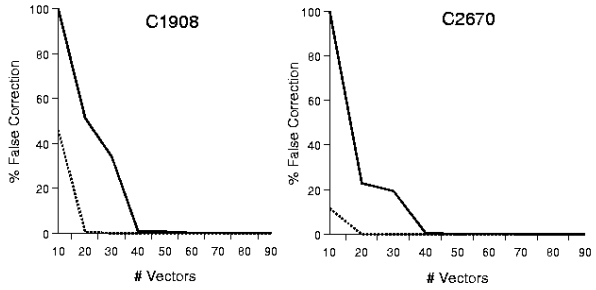

Figure 10: Simulation-based verification

References

[1] M. S. Abadir, J. Ferguson and T. E. Kirkland, "Logic Verification Via Test Generation," in *IEEE Trans. on Computer-Aided Design*, vol. 7, pp. 138-148, January 1988.

[2] R. E. Bryant, "Graph–based algorithms for Boolean function manipulation," in *IEEE Trans. on Computers, vol. C–35, no. 8, pp. 677-691,* 1986.

[3] S. C. Chang and Z. Z. Wu "Theory and Extensions of Single Wire Replacement," in *IEEE Trans. on Computer-Aided Design, vol. 20, no. 9 pp. 1159-1163,* 2001.

[4] S. C. Chang, K. T. Cheng, N. S. Woo and M. Marek-Sadowska, "Postlayout Logic Restructuring Using Alternative Wires," in *IEEE Trans. on Computer-Aided Design, vol. 16, no. 6 pp. 587-596,* 1997.

[5] S. C. Chang and M. Marek-Sadowska, "Perturb and Simplify: Multi-Level Boolean Network Optimizer," in *Proc. Int'l Conference on Computer-Aided Design, pp. 2-5,* 1994.

[6] J. A. Espejo, L. Entrena, E. San Millàn, and E. Olìas, "Functional extension of structural logic optimization techniques," in *Proc. of Asian-South-Pacific Design Automation Conference, pp. 467-472,* 2001.

[7] L. A. Entrena, and K. T. Cheng, "Combinational and Sequential Logic Optimization by Redundancy Addition and Removal," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol.14, no.7, pp. 909-916,* July 1995.

[8] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," in *IEEE Trans. on Computers, vol. C-32, no. 12,* December 1983.

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

[10] W. Kunz and D. K. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems–Test, Verification, and Optimization," in *IEEE Trans. on Computer-Aided Design, vol. 13, no. 9, pp. 1143-1158* September 1994.

[11] W. Kunz, D. Stoffel and P. R. Menon, "Logic Optimization and Equivalence Checking by Implication Analysis," in *IEEE Trans. on Computer-Aided Design, vol. 16, no. 3, pp. 266-281,* March 1997.

[12] Mentor Graphics, "FastScan: Suite of ATPG Tools," available from *www.mentor.com/dft/fastscan_ds.pdf*, 2001.

[13] J. P. Roth, "Diagnosis of automata failures: A calculus & a method," *IBM Journal of Research Development, vol. 10, pp. 278-291,* June 1966.

[14] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. of Int'l Conference on Computer Design, pp. 328-333,* 1992.

[15] J. E. Smith, "On Necessary and Sufficient Conditions for Multiple Fault Undetectability," in *IEEE Trans. on Computers, vol. c-28, no. 10, pp. 801-802,* October 1979.

[16] G. Stenz, B. M. Riess, B. Rohfleisch and F. M. Johannes, "Performance Optimization by Interacting Netlist Transformations and Placement," in *IEEE Trans. on Computer-Aided Design, vol. 19, no. 3,* March 2000.

[17] 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.

[18] A. Veneris, M. S. Abadir and I. Ting, "Design Rewiring based on Diagnosis Techniques," in *Proc. of Asian-South-Pacific Design Automation Conference, pp. 479-484,* 2001.