Automating Logic Rectification by Approximate SPFDs

Yu-Shen Yang
Dept. of ECE
University of Toronto
Toronto, Canada
yangy@eecg.utronto.ca

Subarna Sinha
Synopsys Inc.
Mountain View, United States
Subarna.Sinha@synopsys.com

Andreas Veneris
Dept. of ECE
University of Toronto
Toronto, Canada
veneris@eecg.utronto.ca

Robert K. Brayton
Dept. of EECS
University of California
Berkeley, United States
brayton@eecs.berkeley.edu

ABSTRACT

In the digital VLSI cycle, a netlist is often modified to correct design errors, perform small specification changes or implement incremental rewiring-based optimization operations. Most existing automated logic rectification tools use a small set of predefined logic transformations when they perform such modifications. This paper first shows that a small set of predefined transformations may not allow rectification to exploit the full potential of the design. Then, it proposes an automated simulation-based methodology to "approximate" Sets of Pairs of Functions to be Distinguished (SPFDs) and avoid the memory/time explosion problem. This representation is used by a SAT-based algorithm that devises appropriate logic transformations to fix a design. The SAT method is later complemented by a greedy one that improves on runtime performance. An extensive suite of experiments documents the added potential of the proposed rectification methodology.

1. Introduction

In the lifetime of the digital VLSI cycle, a synthesized design is often readjusted to achieve different goals. *Logic rectification* is required in Design Error Diagnosis and Correction (DEDC) [4][9][17], a procedure that follows verification and debugs an erroneous design to comply with its specification. In DEDC, diagnosis returns a set of candidate error locations while correction applies simple logic transformations on those locations to correct the design. It is also important in some post-synthesis rewiring optimization techniques [16][18]. These methods modify an already optimized logic implementation (netlist) in an iterative manner to refine it for different constraints such as power, delay, area etc. Logic rectification is also crucial when applying Engineering Changes (EC) [11]. In a typical VLSI design process, specifications may change even at a late stage of the design cycle. Given a correctly designed netlist and its new set of specifications, in EC we are interested in applying a minimal set of modifications so that the design complies to its new set of specifications.

Modern approaches fix a design at some given netlist line(s). This line(s) has been identified using diagnosis [4][9][17][18] and formal synthesis [4][11] techniques. The permissible set of transformations on this line comes usually from a *dictionary model*, that is, a small set of predetermined logic transformations such as a single gate type change and/or a single wire addition/deletion. It is evident that the success of dictionary-based logic rectification depends on the ability of the underlying dictionary to accommodate its needs.

Clearly, logic rectification can be viewed as a simpler instance of the general logic synthesis problem. Despite this fact, there are unique reasons for the development of dedicated automated rectification tools. Since existing synthesis tools tend to find a minimal representation of the requested function [13][14], they may modify the design significantly, jeopardizing part of the engineering effort invested in it. Also, a full-blown synthesis step may prove to be a resource intensive procedure for a design that needs just a few structural local netlist changes. Dedicated automated tools that restructure the design minimally are desired to preserve the engineering effort and expedite rectification.

In this paper, we first show that dictionary-based rectification may not capture the complete solution space offered by the don't care space of a design. As a result, it may miss opportunities to achieve the specified rewiring, debugging, etc goals. Next, we propose an automated rectification methodology for combinational circuits that does not use a predetermined dictionary model. This methodology defines *approximate SPFDs* where the results of test vector simulation are used to represent and manipulate SPFDs. SPFDs [19] are a relatively new representation of Boolean functions that provides additional degrees of flexibility during synthesis [5][16][19]. Using simulation to approximate them allows one to remain memory efficient while enjoying their benefits. A SAT-based algorithm using approximate SPFDs is described to automate correction with more complex and effective local logic transformations. This algorithm is later complemented by a greedy one that may sacrifice on optimality to improve performance.

Extensive experiments confirm the theoretical results and show that approximate SPFDs provide an effective alternative to dictionary-based rectification sometimes by orders of magnitude. They return modifications where dictionary-based restructuring fails increasing the impact of debugging, rewiring, EC, etc tools. This encourages further researchs in SPFDs as a mean to aid existing methodologies [4][9][16][17][18].

The paper is structured as follows. Section 2 provides the motivation, Section 3 discusses SPFDs and Section 4 defines approximate SPFDs. Section 5 builds upon this new concept with an optimal SAT-based resynthesis algorithm and a greedy one. Section 6 presents experiments while the last Section concludes the work.

2. Motivation

Most logic rectification approaches use transformations from a predetermined dictionary to change a design when correcting it, optimizing it, etc. For instance, in DEDC, some EC instances and diagnosis-based rewiring, the dictionary model of [1] is widely used. This model contains 11 predetermined types of possible logic transformations that involve addition and deletion of single wires and single gates. For example, the "missing inverter" transformation inserts an inverter while "missing gate" adds a 2-input *i.e.*, `AND`, `OR`, `NAND` and `NOR` gate to the circuit. Equivalently, "missing/extra wire" adds/deletes an existing wire in/from the netlist. Other rewiring techniques [3][10] also use a finite number of predefined transformation types similar to the ones in [1].

A predetermined dictionary model, although effective for some instances, may not be adequate when complex transformations are required such as the addition/deletion of several gates and wires. To study the effectiveness of dictionary-based rectification and to motivate the present work, we perform the following experiment. For circuits in the ISCAS'85 suite of benchmarks we introduce an error. A "simple" error involves a change of gate type for a single gate followed by the addition or deletion of some wires in the support of that gate. A "complex" error applies more transformations such as the deletion and addition of many gates and wires in the fan-in cone of a single gate.

In this study, the effectiveness of the general purpose dictionary model in [1] is measured against that of the error equation. The *error equation* [4] uses formal methods to answer with certainty whether there exists a modification that corrects the design at a specified circuit location. The error equation does not return the actual correction as it merely reports whether some amount of resynthesis *may* or *may not* correct that design at the particular location. It achieves this by forming the miter circuitry between the primary outputs of the golden model and the design, introducing a new variable at the candidate location and solving a system of Boolean equations using formal techniques [2]. If the system has a solution for the new variable, this solution also specifies the range of the new function. Otherwise, if it has no solution, no transformation on that location can modify the design to correct it.

Table 1 contains average results from 10 single error experiments for each circuit. Circuits with the suffix "c" are injected with a single complex error while single simple errors are introduced in the remaining circuits, which have the suffix "s". To identify candidate locations for modification we use a path-trace simulation-based diagnosis method [9] that guarantees to return all such single candidate locations. Since this method uses a subset of the complete input test space, it may return a few locations that cannot be corrected but it will never miss a solution, that is, it will return all locations where some form of resynthesis can be applied to correct the design [9][18]. The second and sixth columns of Table 1 contain the average number of error locations returned by path-trace. The following two columns show the percentage of error locations that can be fixed according to the error equation and according to an exhaustive dictionary-based rectification method [9].

It can be seen that, on the average, the dictionary model in [1] fails for as much as half of the cases with simple errors. For example, in circuit c499, the error equation claims that some modification on five locations (76% of 7.1 locations) can rectify the design whereas the dictionary model is successful in only two cases. As shown, the success of the dictionary model diminishes further when more complex resynthesis is required. This is because complex modifications perturb the functionality of the design in ways that simple dictionary-driven transformations may not be able to address. Such modifications, though, are common in today's intricate design environment where errors or changes in the Register-Transfer Level (RTL) necessitates complex local changes in the netlist [11]. Automated logic rectification tools that can address those problems effectively are therefore desirable to increase the impact of the underlying debugging, rewiring, EC, etc engines.

3. Sets of Pairs of Functions to be Distinguished

SPFDs [19] provide a powerful formalism to express the implementation flexibility of circuit nodes during logic synthesis. An SPFD

$$R = \{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \cdots, (g_{na}, g_{nb})\} \quad (1)$$

denotes a set of pairs of functions that must be distinguished i.e., for each pair $(g_{ia}, g_{ib}) \in R$, the minterms in g_{ia} must produce a different value from the minterms in g_{ib} at the output of the node (wire) associated with R. R can be represented as a graph $G = (V, E)$ [16], where

$$\begin{aligned} V &= \{m_k \mid m_k \in g_{ij}, 1 \le i \le n, j = \{a, b\}\} \\ E &= \{(m_i, m_j) \mid \{(m_i \in g_{pa}) \text{ and } (m_j \in g_{pb})\} \\ &\quad \text{or } \{(m_i \in g_{pb}) \text{ and } (m_j \in g_{pa})\}, 1 \le p \le n\} \end{aligned} \quad (2)$$

Edges $e \in E$ are referred to as SPFD edges. Figure 1 shows the SPFDs at the input and output of a 2-input OR gate. It shows that input a can distinguish (vectors in terms of ab) 00 and 01 from 10 and 11, and input b can distinguish 00 and 10 from 01 and 11. The output c can distinguish 01, 10 and 11 from 00. The SPFD of a node/wire can be derived in a multitude of ways depending on its application during logic synthesis. For instance, SPFDs can be computed in a compatible fashion (similar to the compatible don't care computation [13]) from the primary outputs to the primary inputs [16]. In rewiring applications, the SPFD of a wire (n_a, n_b) can denote the minimum set of edges in the SPFD of n_b that can only be distinguished by n_a (but none of the remaining fanins of n_b) [16]. In all these methods, it is necessary to ensure that the SPFD of a node is a subset of the union of the SPFDs of its fanins. Thus,

$$\cup_{i=1}^{m} R_i \supseteq R_o, \quad (3)$$

where node n_o has m fanins $\{n_1, \cdots, n_m\}$, R_i denotes the SPFD of the ith fanin n_i and R_o denotes the SPFD of n_o . The above equation indicates that the "information" at a node has to be a subset of all the information at its fanins.

A function f is said to satisfy an SPFD $R = \{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \cdots, (g_{na}, g_{nb})\}$, if for each $(g_{ia}, g_{ib}) \in R$, $f(g_{ia}) \neq f(g_{ib})$. In graph-theoretic terms, f has to be a valid coloring of the SPFD graph of R, i.e. any two nodes connected by an edge must be colored differently. In this paper, we use the automated approach by Cong et. al. [5] to synthesize a two-level AND-OR network for the function f of some given node so that it satisfies the given SPFD R. Thus, the set of minterms that belong to the onset of the node are derived from R (note that no two minterms that must be distinguished in R can belong to the onset) and imaged onto the local fanin space of the node. This image function gives the function f that satisfies R. The minterms that are not represented in R can be used as don't cares to simplify f.

Table 1: Quantifying logic transformations

ckt name	error loc.	error equat.	dict. model	ckt name	error loc.	error equat.	dict. model
c432_s	9.8	75%	44%	c2670_s	9.2	99%	11%
c499_s	7.1	76%	40%	c5135_s	6.4	100%	25%
c880_s	3.8	67%	38%	c3540_c	3.0	100%	6%
c1355_s	5.3	100%	19%	c5315_c	6.4	97%	16%
c1908_s	18.0	84%	23%	c7552_c	20.6	64%	20%

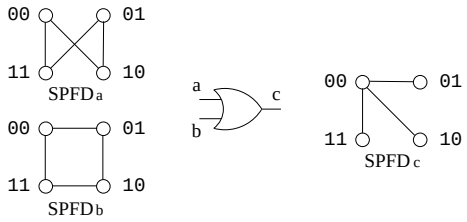

Figure 1: SPFDs for OR gate: $SPFD_c \subset SPFD_a \cup SPFD_b$

4. Approximating SPFDs

SPFD computations represented by BDDs [2] may suffer from memory explosion problems for large circuits [16]. A recently proposed SPFD computation with simulation and SAT [20] is memory efficient at representing SPFDs but it may become computationally intensive because every edge in the SPFD of a node/wire is computed.

One way to reduce the complexity of representing and computing SPFDs is to *approximate* the SPFDs. An approximated SPFD represents a subset of the edges of the original SPFD. This is done to alleviate the memory and runtime problems of formal approaches. A non-trivial problem is figuring out a good subset of edges to pick when approximating the SPFD. We show that logic rectification operations, where only a small portion of the circuit is being changed, are well served by approximate SPFDs.

To see this, it is instructive to think of rectification as a pair of "error/correction" operations. This is indeed the case for DEDC [9][17], EC [11] and, more recently, rewiring, that was also expressed as a debugging process [18]. The actual resynthesis (correction) consists of a set of existing netlist wires that need to be added as fanins to the node. Once this set is found, according to the theory of approximate SPFDs developed next, the automated algorithm in [5][19] can be used to resynthesize the design at that location and provide a broad range of powerful modifications not available in predefined dictionary models.

The set of edges represented by the approximate SPFD can be chosen depending on the "error" in the circuit, which is also a metric for the "amount" of rectification needed. Different errors at the same node would result in a different set of edges being chosen to approximate the SPFD. The circuit containing the error is simulated using a set of primary input vectors $\mathcal{V}$ chosen so that the error can be observed at the primary outputs for some of the vectors. A set of SPFD edges that specifies the minterm pairs that need to be distinguished in $\mathcal{V} \times \mathcal{V}$ in order to correct the error forms the approximate SPFD for the erroneous node.

DEFINITION 1. *The* **global SPFD (*gSPFD*)** *R_i of a node n_i specifies that the primary input minterms in the onset of n_i have to be distinguished from the primary input minterms in the offset of n_i.*

Previous work uses either the *gSPFDs* or a compatible subset of *gSPFDs* to drive logic optimizations. A common feature of both these types of SPFDs is that they analyze the entire set of primary input minterms to decide what the node needs to distinguish. This may lead

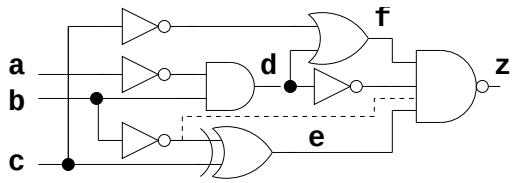

Figure 2: Circuit for Examples 1 and 2

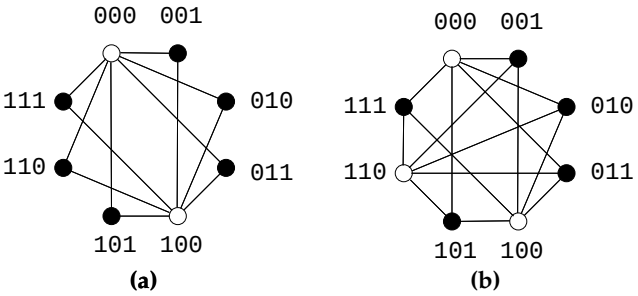

Figure 3: (a) Original circuit SPFD$_z$ (b) Modified circuit SPFD$_{z_{mod}}$

Table 2: Truth table for circuit in Example 1

a	b	c	d	e	f	z	z_{mod}
0	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1
0	1	0	1	0	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	0
1	0	1	0	0	0	1	1
1	1	0	0	0	1	1	0
1	1	1	0	1	0	1	1

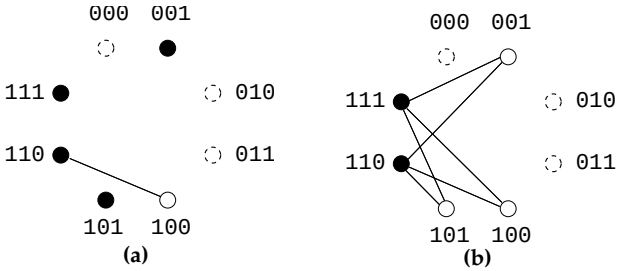

Figure 4: (a) *aSPFD* of z_{mod} (b) *aSPFD* of b in modified circuit

to prohibitive amounts of memory to store or enumerate the SPFDs. To reduce this complexity, we allow for an SPFD representation as follows:

DEFINITION 2. *Given a subset $\mathcal{V}$ of the primary input minterms V, the* **approximate SPFD (*aSPFD*)** *$R_i^{appx}(\mathcal{V})$ w.r.t $\mathcal{V}$ of a node n_i specifies the minterm pairs in $\mathcal{V} \times \mathcal{V}$ that n_i has to distinguish. R_i^{appx} contains no information about the minterms in $V - \mathcal{V}$.*

In other words, the *aSPFD* of a node considers what needs to be distinguished only for a subset of the primary input vectors $\mathcal{V}$. Since it stores less information than *gSPFDs*, it is inherently less expensive to represent, manipulate and compute.

Next, we describe a procedure to compute *aSPFD* for the circuit nodes using test set $\mathcal{V}$. In practice, $\mathcal{V}$ contains 2,000 tests for the ISCAS'85 circuits and includes vectors that fail (produce a difference at one of the outputs) and ones that do not. In the experiments, $\mathcal{V}$ contains vectors with high stuck-at fault coverage but other methods, such as simulation and verification, to obtain $\mathcal{V}$ can be used [6]. Assume that the original circuit (specification) is denoted as $\mathcal{C}^c$ and the erroneous circuit $\mathcal{C}^e$. Let n_{err} denote the erroneous node in $\mathcal{C}^e$, i.e. the node that needs to be corrected to make $\mathcal{C}^e$ equivalent to $\mathcal{C}^c$. The *aSPFD* of n_{err} and the remaining nodes are computed differently. This is due to the fact that the *aSPFD* of n_{err} should capture information about the error that needs to be fixed and the *aSPFDs* of the remaining nodes represent what they can distinguish. The *aSPFD* for n_{err} is computed as follows:

1. Simulate the correct circuit $\mathcal{C}^c$ and the erroneous circuit $\mathcal{C}^e$ using the $\mathcal{V}$ set of vectors.
2. Let $\mathcal{V}^c$ denote the vectors for which all outputs of $\mathcal{C}^c$ are equal to their respective outputs of $\mathcal{C}^e$. Let $on(\mathcal{V}^c)$ denote the vectors in $\mathcal{V}^c$ for which n_{err} is equal to 1 in $\mathcal{C}^e$ and $off(\mathcal{V}^c)$ denote the remaining vectors (for these vectors n_{err} is equal to 0 in $\mathcal{C}^e$).
3. Let $\mathcal{V}^e$ denote the vectors for which an output of $\mathcal{C}^c$ is not equal to the corresponding output of $\mathcal{C}^e$. Let $on(\mathcal{V}^e)$ denote the vectors of $\mathcal{V}^e$ for which n_{err} is equal to 1 in $\mathcal{C}^e$ and $off(\mathcal{V}^e)$ denote the remaining vectors (for these vectors n_{err} is equal to 0 in $\mathcal{C}^e$).
4. The approximate SPFD of n_{err} states that the minterms in $on(\mathcal{V}^c)$ have to be distinguished from the minterms in $on(\mathcal{V}^e)$ and the ones in $off(\mathcal{V}^e)$ have to be distinguished from the ones in $off(\mathcal{V}^c)$.

Notably, to generate the *aSPFD* for the erroneous node, both the correct and the erroneous vectors need to be simulated. This is because it is necessary to have both kinds of primary input vectors: ones for which the error is observed at the outputs and ones for which no error is observed. This ensures that the *aSPFD* of n_{err} contains information on what needs to be corrected and what does not. Let $TFO(n_{err})$ denote the transitive fanout of n_{err}. For each node $n_k \notin TFO(n_{err})$, the *aSPFD* of n_k distinguishes the onset minterms of n_k in $\mathcal{V}$ from the offset minterms of n_k in $\mathcal{V}$. Thus, the *aSPFD* of each of these nodes specifies the primary input pairs that each node can (or has) to distinguish in $\mathcal{C}^c$ and in $\mathcal{C}^e$ for the vectors in $\mathcal{V} = \mathcal{V}^c \cup \mathcal{V}^e$. This is true because the SPFDs in the two circuits are identical for gates not in $TFO(n_{err})$. Since resynthesis cannot use nodes in $TFO(n_{err})$ as they create combinational feedback, it is not necessary to compute the *aSPFDs* for these nodes.

EXAMPLE 1. *Figure 2 depicts a sample circuit for the truth table in Table 2 where the dotted line is not a wire of the original design. The gSPFD of the output z in the sample circuit in terms of primary inputs is shown in Figure 3(a). The black (white) nodes indicate that z has logic value 1 (0) when the labeled input vector is applied. Assume the wire (e,z) is removed, e.g. $z_{mod} = NAND(\overline{d}, f)$. The last column "$z_{mod}$" of Table 2 shows the logic value of z after the modification.*

The gSPFD of the primary output of the modified circuit z_{mod} in terms of primary inputs is shown in Figure 3(b). Suppose the original and modified circuits are simulated with primary input vectors $\mathcal{V}$: 001, 100, 101, 110 *and* 111. *The discrepancy is observed for vector* 110. *In terms of the notation used in the aSPFD calculation procedure, $\mathcal{V}^c = \{001, 100, 101, 111\}$ and $\mathcal{V}^e = \{110\}$. Furthermore, $on(\mathcal{V}^c) = \{001, 101, 111\}$, $off(\mathcal{V}^c) = \{100\}$, $on(\mathcal{V}^e) = \phi$ and $off(\mathcal{V}^e) = \{110\}$. The aSPFD of z_{mod} specifies that the minterms in $on(\mathcal{V}^c)$ have to be distinguished from the minterms in $on(\mathcal{V}^e)$ and the minterms in $off(\mathcal{V}^c)$ have to be distinguished from the minterms in $off(\mathcal{V}^e)$. Since $on(\mathcal{V}^e) = \phi$, only the minterms in $off(\mathcal{V}^c)$ have to be distinguished from the minterms in $off(\mathcal{V}^e)$. Thus, the aSPFD of z_{mod} w.r.t to $\mathcal{V}$ is $\{110, 100\}$, as shown in Figure 4(a). It is approximate since it only contains a subset of the complete information about SPFDs that z_{mod} needs to distinguish and maintain correct functionality. For instance, to ensure that z_{mod} is equivalent to z, it will also be necessary to distinguish* 110 *from* 000. *However, this information is not contained in the aSPFD of z_{mod} derived using the vectors $\{001, 100, 101, 110, 111\}$. The* aSPFD *of b, which is in the fanin cone of z_{mod}, for the same set of primary input vectors is shown in Figure 4(b).*

The penalty when approximating SPFDs is that once resynthesis is done, the circuit has to undergo formal verification. This is because *aSPFDs* do not contain the complete information about what the nodes need to distinguish but only a subset of it that is contained in $\mathcal{V}$. In DEDC and EC where the specification acts as a "black box" and has no

structural similarity to the netlist, a full blown verification step [6][8] (*i.e.*, equivalence checking) is already mandatory since the error location is unknown [17]. In rewiring, an order of magnitude faster dedicated formal verification step is usually performed since the locations of both the error and the correction are known [3][10][18].

Experiments show that *aSPFDs* based on a set of vectors $\mathcal{V}$ with 2,000 tests provide a good approximation for the design functionality because the transformations pass formal verification in most (80-90%) cases. Intuitively, this is partly because the error effects are limited to a small region and the *aSPFD* construction at the erroneous node can be manipulated in ways that correct the error with a high degree of probability. In addition, the mapping from the primary input space to the local fanin space of the error location is typically many-to-one. Consequently, a primary input minterm pair (m_a, m_b) that is missing in the *aSPFD* may still be covered. This could happen if its image (y_{ad}, y_{be}) in the local fanin space is the same as the image of a minterm pair (m_d, m_e) present in the *aSPFD*. In other words, some of the primary input minterm pairs get distinguished for free. This is particularly likely to happen in scenarios where there aren't too many minterms pairs that need to be distinguished to correct the error at the first place.

5. Automating Rectification

The proposed rectification procedure seeks to re-implement location n_{err} using one or more additional fanins so that $\mathcal{C}^e$ implements the same function as the original circuit $\mathcal{C}^c$. This location n_{err} is suspect for correction in DEDC, rewiring, EC etc. The theory of approximate SPFDs developed in the previous section can be used to find these additional fanins as well as the new function desired at n_{err}.

Algorithm 1 shows the pseudo-code to find a set of fanins for n_i. The basic idea is to compute the *aSPFD* of n_{err} (w.r.t $\mathcal{V}$), $R_{err}^{appx}(\mathcal{V})$. Then, for each $e_i \in R_{err}^{appx} - \cup_{i=1}^{k} R_i^{appx}$, where $e_i \in E$, the set of nodes that are not in $TFO(n_{err})$ and whose *aSPFDs* w.r.t $\mathcal{V}$ contain e_i are inserted in $Cover(e_i)$. Finally, a minimal set S of nodes from *Cover* is selected such that at least one node in $Cover(e_i)$, for each $e_i \in R_{err}^{appx} - \cup_{i=1}^{k} R_i^{appx}$, is contained in S. A new function is implemented at n_{err} using the nodes in S as additional fanins according to the methodology described by Cong et al. in [5]. If the new function does not pass verification, a new set of nodes is selected from *Cover* and the process is repeated.

LEMMA 1. *The aSPFD of a node $n_k \in \mathcal{C}^e$, where $n_k \notin TFO(n_{err})$, is a subset of the gSPFD of the corresponding node $\tilde{n_k}$ in $\mathcal{C}^c$.*

Proof: Suppose, towards a contradiction, it is not true. Then, there exists an edge $e = (m_1, m_2)$ that belongs to the *aSPFD* of n_k in $\mathcal{C}^e$ and does not belong to the *gSPFD* of $\tilde{n_k}$ in $\mathcal{C}^c$. Thus, m_1 and m_2 both belong to either the onset or the offset of $\tilde{n_k}$ in $\mathcal{C}^c$. Assume, they both belong to the onset. Since $n_k \in \mathcal{C}^e$ is identical to the corresponding node $\tilde{n_k} \in \mathcal{C}^c$, m_1 and m_2 have to produce the same values at the output of n_k as at the output of $\tilde{n_k}$. Thus, m_1 and m_2 both belong to the onset of n_k. Hence by construction, (m_1, m_2) cannot belong to the *aSPFD* of n_k. ■

LEMMA 2. *Each edge in the aSPFD of a node $n_k \in \mathcal{C}^e$, where $n_k \notin TFO(n_{err})$, is contained in the aSPFDs of one or more of its fanins.*

Proof: Assume it is not true. Then, there exists at least one minterm pair $(m_1, m_2) \in R_k^{appx}$ not belonging to the *aSPFDs* of the fanins in n_k. By Lemma 1, the *aSPFD* of n_k in $\mathcal{C}^e$ is a subset of the *gSPFD* of the corresponding node $\tilde{n_k}$ in $\mathcal{C}^c$. Equation 3 states that each edge in the *gSPFD* of $\tilde{n_k}$ has to be contained in the *gSPFDs* of one or more of its fanins. Since n_k and its fanins implement the same function in $\mathcal{C}^c$ and $\mathcal{C}^e$, the *gSPFD* of $\tilde{n_k}$ and its fanins in $\mathcal{C}^c$ are identical to the *gSPFD* of the corresponding nodes in $\mathcal{C}^e$. Thus, (m_1, m_2) is contained in the *gSPFD* of n_k and at least one of its fanins in $\mathcal{C}^e$. Hence, by construction, (m_1, m_2) has to belong to the *aSPFDs* of one of the fanins of n_k. ■

EXAMPLE 2. *Example 1 illustrates that the aSPFD of z_{mod} is contained in the aSPFD of b. If b is used as an additional fanin of z_{mod} (dotted wire in Figure 2) a new function $f = NAND(\overline{b}, \overline{d}, f)$ at z_{mod} produces an equivalent circuit to the original. Using this new function, the XOR gate can be removed reducing the original circuit gate count from eight to seven gates.*

Algorithm 1 Correction Using SPFD

```
1: C^c := Original Circuit (or Specification)
2: C^e := Erroneous Circuit
3: procedure CORRECT_WITH_SPFD(C^e, C^c)
4:     Compute the aSPFD of n_err and n_k ∉ TFO(n_err) in C^e.
5:     Set Candidate := complement(TFO(n_err))
6:     Compute E = R_err^appx − ∪_{i=1}^k R_i^appx where {n_1, ..., n_k} are the
           fanins of n_err in C^e
7:     for all e_i ∈ E do
8:         Cover(e_i) := ∅
9:         for all n_k ∈ Candidate do
10:            Add n_k in Cover(e_i) if R_k^appx contains e_i
11:        end for
12:    end for
13:    Select a minimal set of nodes (if one exists) from Cover such
           that at least one node is in Cover(e_i) for each edge in E
           and re-implementing n_err with the the fanins of n_err and the
           nodes in Cover(e_i) makes C^e equivalent to C^c.
14: end procedure
```

Next, we describe two approaches that find the set of nodes that can be used as additional fanins to n_{err} for correcting the error. One is SAT-based and it finds the optimal answer in terms of number of fanins. The other is a greedy one that uses heuristics.

5.1 SAT-based Automated Rectification

We turn `Step 13` of the correction Algorithm 1 into an instance of a Boolean SAT problem [7][12][15]. Give the erroneous node n_{err} and the *aSPFDs* of n_{err} and the nodes that are not in its transitive fanout in $\mathcal{C}^e$, the SAT instance Φ_{err} is set up as follows: Each node n_i in $\mathcal{C}^e$ is associated with a variable v_i. Two types of clauses are added to Φ_{err}:

- *Covering Clauses*: Each edge e_j in the *aSPFD* of n_{err} is converted to a clause c_j. If the *aSPFD* of $n_k \notin TFO(n_{err})$ contains e_j, then c_j has the variable v_k.

- *Blocking Clauses*: For each node $n_k \notin TFO(n_{err})$, Lemma 2 states that the *aSPFD* of a node is a subset of the union of the *aSPFDs* of its fanins. Thus, it is not necessary to add a node to S once all its fanins are included in S. Thus, for each node $n_k \notin TFO(n_{err})$, the following clause is added: $(\bigcup_{i=1}^{m} \overline{v}_i + v_k^{new}) \cdot \bigcap_{i=1}^{m} (v_i + \overline{v}_k^{new}) \cdot (\overline{v}_k^{new} + \overline{v}_k)$ where $\{n_1, \cdots, n_m\}$ are the fanins of n_k and v_k^{new} is a new variable that becomes 1 when all v_i, $1 \le i \le m$ are 1. This new added variable can get an implied value but it cannot be assigned by the decision process of the solver itself.

Given a satisfying assignment for Φ_{err}, a node n_k is added to the set S, if $v_k = 1$. The covering clauses ensure that S can cover all the edges in the *aSPFD* of n_{err}. Blocking clauses reduce the possibility of the same set of edges being covered by multiple nodes in S. If the function derived by the satisfying assignment from the set $S = \{v_1, v_2, \cdots, v_n\}$ does not qualify formal verification, then $\{\overline{v_1} + \overline{v_2} + \ldots + \overline{v_n}\}$ is added as a blocking clause to Φ_{err} and the SAT solver is invoked again to find another candidate transformation. With respect to Examples 1 and 2, the covering clause is $\Phi_C = (b + e)$ and the blocking clauses are $\Phi_B = (\overline{a} + \overline{b} + d^{new}) \cdot (a + \overline{d}^{new}) \cdot (b + \overline{d}^{new}) \cdot (\overline{d}^{new} + \overline{d})$ while $\Phi_{err} = \Phi_C \cdot \Phi_B$ and d^{new} is the new variable introduced by the algorithm. Note, the solver will always come back with an answer (if it exists) as nothing constrains the number of new fan-in wires N to add during resynthesis.

In experiments, we use a pseudo-Boolean constraint SAT solver [7] to return an optimal solution, that is, a solution with the smallest number of new fan-in wires N. The use of a pseudo-Boolean solver is not mandatory and any DPLL-based SAT solver [12][15] can be used instead. A way to achieve this is to encode the adder/comparator circuitry from [17] in CNF and enumerate values of $N = 1, 2, \ldots$. This circuitry can be coded using a linear number of clauses w.r.t. *# circuit lines*. Once translated/added to Φ_{err}, it can enforce that no more than N variables can be set to a logic 1 simultaneously or Φ_{err} becomes unsatisfiable.

Table 3: *aSPFDs* comparison results for different types of single errors

ckt name	error loc.	error equat.	dict. [1] model	*aSPFD*	% improve	avg # wires	min # wires	time (sec) first corr.	avg # corr/loc.	gate count	% verified first	% verified all
c1355_s	5.3	100.0%	18.8%	81.3%	433	1.7	1.7	3.5	8.3	5.9	100.0%	46.2%
c1908_s	18.0	84.4%	13.2%	84.2%	640	1.4	1.4	18.9	8.1	4.2	90.2%	62.6%
c2670_s	9.2	97.8%	11.1%	82.2%	740	2.4	2.2	21.9	6.2	17.0	100.0%	74.7%
c3540_s	7.2	100.0%	27.8%	86.1%	310	1.1	1.1	9.3	4.5	3.7	100.0%	66.1%
c5315_s	6.4	100.0%	25.0%	100.0%	400	1.9	-	7.6	5.4	5.9	89.7%	76.6%
c7552_s	11.8	88.1%	19.2%	50.0%	260	1.7	-	25.7	3.1	4.9	88.9%	54.7%
c1355_m	2.7	100.0%	12.5%	100.0%	800	2.1	2.0	32.0	7.0	4.8	100.0%	52.1%
c1908_m	5.8	100.0%	3.4%	82.8%	2400	2.5	2.5	11.0	10.6	6.3	100.0%	68.8%
c2670_m	5.2	96.2%	4.0%	60.0%	1500	3.4	2.9	95.4	9.4	15.9	100.0%	59.8%
c3540_m	3.2	100.0%	25.0%	100.0%	400	1.6	1.6	54.2	6.1	4.1	84.9%	78.2%
c5315_m	9.6	93.8%	2.2%	100.0%	4500	2.9	-	46.8	5.7	7.4	100.0%	76.9%
c7552_m	8.8	100.0%	9.1%	90.9%	1000	1.9	-	39.2	6.9	6.3	100.0%	78.5%
c1355_c	3.7	100.0%	0.0%	72.7%	inf.	2.9	2.9	38.4	3.3	6.8	100.0%	40.0%
c1908_c	15.8	46.8%	40.5%	70.3%	173	1.4	1.3	19.0	7.2	7.1	100.0%	87.9%
c2670_c	12.4	98.4%	31.1%	62.3%	200	1.7	1.7	33.2	4.7	5.9	100.0%	76.2%
c3540_c	3.0	100.0%	6.7%	66.7%	1000	3.6	3.4	122.4	3.8	6.1	100.0%	32.5%
c5315_c	6.4	96.9%	16.1%	100.0%	620	2.7	-	20.0	9.1	8.4	100.0%	78.7%
c7552_c	20.6	64.1%	19.7%	50.0%	254	1.9	-	23.7	3.5	5.2	91.2%	42.7%
Average	8.6	92.6%	15.9%	80.0%	**504**	2.0	-	29.2	6.4	6.6	**96.0%**	**67.4 %**

Constraining the number N in this manner, any DPLL-based SAT solver can return with the answer. When it fails, we increase the value of the guess N and run the solver to find a solution with more wires.

5.2 A Greedy Approach

Though the SAT-based formulation can return the minimum set of fanins to resynthesize n_{err} and correct the circuit, experiments show that it may require excessive runtimes. To improve the runtime performance, the following greedy approach to search solutions is proposed:

1. Sort the edges E in the *aSPFD* of n_{err} according to the number of nodes whose *aSPFDs* contain each edge.
2. Find the edge e_{min} such that cardinality of $N_{e_{min}}$, which is the set of nodes whose *aSPFDs* contain e_{min}, is the smallest.
3. Add $S \leftarrow S \cup n_k$, where $n_k \in N_{e_{min}}$ and the *aSPFD* of n_k contains the largest set of edges in E.
4. $E \leftarrow E - \{e \mid e \in E \text{ and } e \in R_k^{appx}\}$. If $E \neq \varnothing$, go back to Step 1.

The solutions identified by the greedy approach may contain more wires than the minimum set. However, the experiments in the next section indicate that the greedy approach can achieve similar quality results in terms of the number of wires involved during resynthesis in a more computationally efficient manner.

6. Experiments

This section presents empirical results using the original benchmark ISCAS'85 circuits. In experiments, each circuit is simulated with a set $\mathcal{V}$ of 2,000 input test vectors with high stuck-at fault coverage. Experiments are collected on a Pentium 2.7GHz workstation with 1GB of RAM using the SAT solver from [7]. Motivated by the numbers of Table 1 in Section 2, we quantify the rectification potential of the *aSPFD*-based algorithms against that of the dictionary-model of [1] and that of the error equation [4].

Table 3 contains the results of experiments where we inject three different types of single errors in each circuit and we try to rectify it. The location and the type of injected errors are randomly selected. We assume that the location n_{err} to be corrected is provided by a fast linear-time diagnosis method [9]. All numbers in that table are averages over 20 experiments/circuit and times are in seconds. Simple errors (suffix "s") involve the addition/deletion of a single wire or a gate type replacement. This error type borrows exact transformations from [1]. Medium complexity errors on a gate, such as multiple wire additions/deletions followed by a gate type change, have a suffix "m". The final error type is the complex one from Section 2 (suffix "c") with multiple errors at the fan-in cone of a gate. This alters the functionality of the circuit the most and presents a challenging rectification task.

Columns 2…6 of Table 3 contain rectification comparison results with the error equation and the dictionary model. The second column has the number of locations returned by diagnosis [9][17] while the next column shows the percentage of locations the error equation claims a solution. The two columns that follow contain the percentage of error equation lines the dictionary and *aSPFDs* can resynthesize successfully. For example, in circuit *c*1908_*s*, the error equation claims that resynthesis can fix 15.4 from the 18.0 diagnosis locations. The dictionary is successful in only 2 locations (13.2% of 15.4) while *aSPFDs* resynthesizes 13 locations (84.2% of 15.4). It is seen, that *aSPFDs* outperforms the dictionary-model for as much as 45 times (*c*5315_*m*) with a success ratio that increases with error complexity.

The next five table columns present specifics about the algorithms of Section 5. Column 7 has the average number of additional wires returned by the greedy algorithm and column 8 has the minimum number of wires reported by the optimal SAT-based approach. We observe that the greedy heuristic performs well when compared to the optimal answer. Since the SAT-based approach may run into run-time problems as the number of new wires increases, it times out ("–") after 1,000 seconds if it does not return with a solution. Column 9 contains the average time for the greedy algorithm to return the first transformation that passes formal verification for a single location and the next column has its average number of solutions per location. Overall, it is observed that the greedy approach provides a fine balance between performance and optimality. Column 11 shows the average gate count for the two-level circuitry introduced by [5]. The gate count is small and the modifications are local to respect the existing engineering effort.

The final two columns of Table 3 contain the percentage of *aSPFD* transformations that pass formal verification. The first column considers only the very first correction, if any, returned by the greedy approach for each location. The last column contains averages over all corrections returned for a single location. Since there are locations that rectification may return thousands of modifications, the algorithm times out after a limit. We see that the vast majority of the very first corrections returned pass verification, a result that confirms [18] for simulation-based rectification using a dictionary. It also confirms the viability of *aSPFDs*.

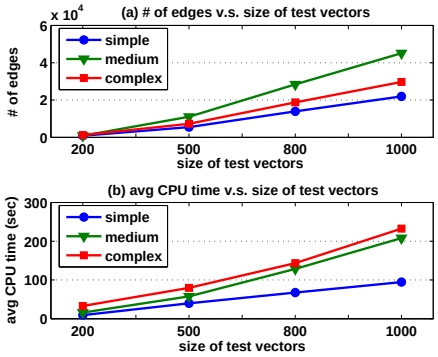

Figure 5: Resynthesis performance characteristics

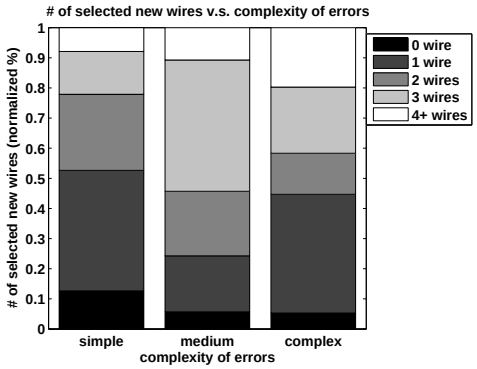

Figure 6: New wire selection effort

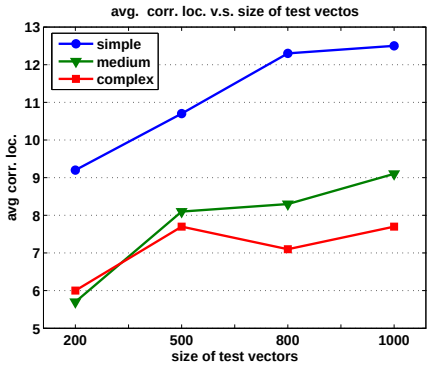

Figure 7: # corrections vs. size of $\mathcal{V}$

Because experiments are centered around rewiring, to formally verify the transformations we use the technique from [18] where a pair of multiplexers is attached at the error/correction location. This simplifies the problem of verification to that of the redundancy of multiple stuck-at faults. Experiments show that this method serves as an excellent platform to verify *aSPFD* produced transformations. For example, SAT-based equivalence checking step takes 43 seconds (on the average) to verify instances of *c*5315 whereas the construction in [18] reduces this time to less than 0.4 seconds.

The plot in Figure 5(a) depicts the number of graph *aSPFD* edges that increases as the number of test vectors $\mathcal{V}$ gets larger. This has an effect on the CNF size and the overall SAT solver performance. In the future, we plan to investigate techniques to reduce the CNF size and speed-up the SAT-based method. Figure 5(b) has the average time for the greedy method to find the first correction. It is seen that time increases as the number of vectors increases because the new vectors introduce additional constraints the greedy method needs to satisfy.

Figures 6 and 7 contain information about the actual transformations. The first graph shows the minimum number of new wires N required (along with the original gate support) for a transformation that passes verification. The number of new wires increases as the error gets more complex, a result that reinforces the need for automated resynthesis approaches such as the one presented here. The second graph shows the number of locations that have transformations that pass verification versus the size of $\mathcal{V}$. As explained in the last paragraph of Section 4, the more vectors available, the more minterms *aSPFD* will probably cover. As a result, the algorithm in [5] will have a higher probability to correct the design successfully, an observation confirmed by the plot.

7. Conclusion

We presented a new representation for SPFDs, namely *aSPFDs*, that uses test vector simulation to approximate their information and avoid any memory/time explosion. We also outlined algorithms that use this new representation to perform rectification for DEDC, EC and rewiring. Experiments show that *aSPFDs* provide a powerful and dynamic method to resynthesize a design to a new set of specifications where other methods fail. They also encourage further research in automated resynthesis tools using *aSPFDs*. We plan to extend the work to rectify circuits that require rectifications at multiple locations and sequential circuits.

8. References

[1] M. S. Abadir, J. Ferguson and T. E. Kirkland, "Logic Verification Via Test Generation", in *IEEE Trans. on CAD*, vol. 7, pp. 138–148, Jan. 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, M. Marek-Sadowska and K. T. Cheng, "Perturb and Simplify: Multi-level Boolean Network Optimizer," in *IEEE Trans. on Computer-Aided Design, vol. 15, no. 12, pp. 1494-1504*, Nov 1996.

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

[5] J. Cong, Y. Lin, and W. Long, "SPFD-Based Global Rewiring," in *Proc. ACM/SIGDA Int'l Symp. on FPGAs, pp. 77-84*, 2002.

[6] R. Drechsler, *Advanced Formal Verification*, Kluwer Academic Publishers, 2004.

[7] N. Eén and N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT," in *Journal on Satisfiability, Boolean Modeling and Computation, vol. 2, pp. 1-26*, March 2006.

[8] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 2000.

[9] S. Y. Huang and K. T. Cheng, *Formal Equivalence Checking and Design Debugging*, Kluwer Academic Publishers, 1998.

[10] 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.

[11] C.-C. Lin, K.-C. Chen and M. Marek-Sadowska, "Logic Synthesis for Engineering Change," in *Trans. on Computer-Aided Design, vol. 18, no. 3, pp. 282-292*, March 1999.

[12] M.H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of DAC, pp. 530-535*, June 2001.

[13] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting local don't cares for network optimization," in *Proc. of ICCAD, pp. 514-517*, 1991.

[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 ICCAD, pp. 328-333*, 1992.

[15] J. P. M.-Silva and K. A. Sakallah, "GRASP – A Search Algorithm for Propositional Satisfiability," in *IEEE Trans. on Computers, vol. 48, no. 5, pp. 506-521*, May 1999.

[16] S. Sinha, "*SPFDs: A New Approach to Flexibility in Logic Synthesis*," Ph.D. Thesis, University of California, Berkeley, May 2002.

[17] A. Smith, and A. Veneris, and M. F.Ali and A. Viglas, "Fault Diagnosis and Logic Debugging Using Boolean Satisfiability," in *IEEE Transactions in Computer-Aided Design, vol. 24, no. 10, pp. 1606-1621*, Oct. 2005.

[18] A. Veneris and M. S. Abadir, "Design Rewiring Using ATPG," in *Proc. IEEE Trans. on Computer-Aided Design, vol. 21, no. 12, pp. 1469-1479*, Dec. 2002.

[19] S. Yamashita, H. Sawada and A. Nagoya, "SPFD: A new Method to Express Functional Flexibility," in *IEEE Trans. on Computer-Aided Design, vol. 19, no. 8, pp. 840-849*, Aug. 2000.

[20] J. Zhang, S. Sinha, A. Mishchenko, R. K. Brayton and M. C-. Jeske, "Simulation and Satisfiability in Logic Synthesis," in *IWLS*, 2005.