Automating Logic Transformations with Approximate SPFDs

Yu-Shen Yang, *Student Member, IEEE,* Andreas Veneris, *Senior Member, IEEE,* Subarna Sinha, and Robert K. Brayton, *Fellow, IEEE*

***Abstract*—During the VLSI design process, a synthesized design is often required to be modified in order to accommodate different goals. To preserve the engineering effort already invested, designers seek small logic structural transformations to achieve these logic restructuring goals. This paper proposes a systematic methodology to devise such transformations automatically. It first presents a simulation-based formulation to approximate SPFDs and avoid the memory/time explosion issue inherent with the original representation. Then it uses this new data structure to devise the required transformations dynamically without the need of a static dictionary model. The methodology is applied to both combinational and sequential designs with transformations at a single or multiple locations. An extensive suite of experiments documents the benefits of the proposed methodology when compared to existing practices.**

***Index Terms*—Logic restructuring, VLSI, SPFD, correction, debug, engineering change, logic rewire, optimization**

I. Introduction

The consumer's demand for devices with increased functionality and performance continues to drive the growth of the semiconductor industry. In part, this growth can be attributed to the use of Computer Aided Design (CAD) tools that complement the designer experience to develop computer chips with the increasing size and complexity.

During the chip design cycle, small structural transformations in logic netlists are often required to accommodate different goals, For example, the designer needs to rectify designs that fail functional verification at locations identified by a debugging program [1], [2]. In the case of engineering changes (EC) [3], a logic netlist is modified to reflect specification changes at a higher level of abstraction. Logic transformations are also important during rewiring-based post-synthesis performance optimization [4], [5], where designs are optimized at particular internal locations to meet specification constraints.

Clearly, logic restructuring 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 tools to perform this task. A full-blown synthesis step may be a time-consuming and resource-intensive process for a design that needs only a few of structural changes. Furthermore, existing synthesis tools may significantly modify the structure of the design and jeopardize the engineering effort already invested in it [6], [7].

Most existing logic restructuring techniques modify the netlist by using permissible transformations from a *dictionary model* [8]. This model contains a set of simple modifications such as single gate and wire additions/removals. It is evident that the success of these methods directly depends on the ability of the underlying predetermined dictionary to accommodate the necessary netlist changes [9]. Previous work has also shown that transformations at a single location may not always be adequate [1], [3], [4], [10]–[12]. For instance, when applying an engineering change, a modification in the high-level design description can potentially be mapped into multiple locations in the netlist. Hence, it remains important to research automated incremental logic restructuring methodologies that can perform multiple transformations dynamically.

This work aims to develop a comprehensive methodology to automate the process of logic restructuring in combinational and sequential circuits [9], [13]. First, it proposes a novel model of Boolean representation, namely *Approximate Sets of Pairs of Functions to be Distinguished (a*SPFDs*)*. This allows one to perform the required logic transformations algorithmically and without the restriction of a static pre-defined dictionary model. In the past, Sets of Pairs of Functions to be Distinguished (SPFDs) have proved to provide additional degrees of flexibility during logic synthesis [14], [15]. However, computing SPFDs can be computationally expensive in terms of runtime and memory [16]. To address this problem, *a*SPFDs approximate the information contained in SPFDs using the results of test-vector simulation. Applications that utilize *a*SPFDs can remain memory and runtime efficient while taking advantage of most benefits of SPFDs. In addition, results from this paper show that when this new representation is used to model a sequential circuit, it circumvents the exponential state space explosion of the original formulation [17], as the encoding contains only a small portion of the complete state space, that is, the one exercised during simulation.

Using *a*SPFDs to perform logic restructuring using a SAT engine entails two stages. The first stage constructs the respective *a*SPFDs and identifies the function required at specific circuit line(s) such that the design complies to its specification. Next, using *a*SPFDs as a guideline, the algorithm searches for the necessary nets to construct the required function.

Y.-S. Yang is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, M5S 3G4, Canada (Email: yangy@eecg.utoronto.ca)

A. Veneris is with the Department of Electrical and Computer Engineering and with the Department of Computer Science, University of Toronto, Toronto, ON, M5S 3G4 (Email: veneris@eecg.utoronto.ca)

S. Sinha is with Synopsys, Inc. Mountain View, CA, 94043, USA (Email: subarna@synopsys.com)

R. K. Brayton is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, 94708, USA (Email: brayton@eecs.berkely.edu)

Two approaches are presented to perform this search. The first is an algorithm using Boolean Satisfiability (SAT) to identify the minimum number of new lines required for the desired transformation. Although optimal, the SAT-based approach may require excessive computation at times. The second algorithm, a greedy approach, is later presented to improve performance. One may think that it would sacrifice on optimality, but experiments show that in most cases it returns near optimal results, a favorable trade-off between performance and resolution.

Extensive experiments confirm the theoretical results and show that *a*SPFDs provide an effective alternative to dictionary-based transformations. The proposed technique returns modifications where dictionary-based restructuring fails, increasing the impact of tools, such as debugging, rewiring, EC, etc. Experiments also show the feasibility of using *a*SPFDs to restructure sequential designs and designs with multiple errors. For combinational circuits, the proposed approach can identify five times more valid transformations than a dictionary-based one. Nevertheless, since the method bases its results on a small sample of the input test vector space, verification must follow to confirm the validity of the proposed transformations. Overall, empirical results show that in both the combinational and sequential circuits, more than 90% of the first transformations returned by our approach passes formal validation.

The remainder of this paper is structured as follows. Section II summarizes previous work and gives the motivation for this paper. It also covers background material. Section III defines approximate SPFDs and the procedures to generate *a*SPFDs. Section IV presents the transformation algorithms utilizing *a*SPFDs. Applying transformations at multiple locations is discussed in Section V. Experimental results are given in Section VI, followed by conclusions in Section VII.

II. Background and Motivation

A. Previous work

Most research done on logic restructuring deals with combinational designs. In [18], the authors insert circuitry before and after the original design so that the functionality of the resulting network complies with the required specifications. The disadvantage of this approach is that the additional circuitry can dramatically change the performance of the design.

Redundancy addition and removal (RAR) [5], [19] is a post-synthesis logic optimization technique. It optimizes designs through the iterative addition and removal of redundant wires. All logic restructuring operations performed by RAR techniques are limited to single wire additions and removals. There is little success in trying to add and remove multiple wires simultaneously due to a large search space and complicated computation [20].

A commonly used solution to logic transformations in the concept of design error correction is to use the predetermined error dictionary model of Abadir et al. [8]. Most debugging methods utilize this model to correct the localized errors [21], [22]. This model contains a static set of error types that are similar to the modification performed by RAR. The dictionary model has been used for rectifying designs at multiple locations as well [21].

Sequential circuits can be hard to restructure due to the presence of memory elements. That is, the Boolean function of a net may depend on previous states of the design. This increases the complexity of the underlying analysis severely. Sequential circuits are commonly modelled in the *Iterative Logic Array (ILA)* representation. In this formulation, the design is unfolded over time to maintain its combinational functionality [17], [23]. The side-effect of the ILA representation is that the input vector space of the unrolled circuit grows exponentially in relation to the number of cycles the circuit has unrolled. This can become computationally expensive for some methods if the size of the underlying data structure is correlated with the number of primary inputs [17].

In [17], Sinha et al. define sequential SPFDs, which are later used to optimize the sequential state encoding. To avoid the input vector space explosion mentioned above, the method unrolls the circuit incrementally until no more useful information can be attained. The authors conclude that the size of the input space remains a major challenge for some circuits.

Along the lines of the work presented here for design debugging, the authors in [10]–[12] introduce the concept of Pairs of Bits to be Distinguished. We will discuss the similarities and differences of this technique with our approach in a later section.

B. Motivation

As mentioned earlier, most common approaches for logic transformation use a dictionary model similar to the one proposed in [8], which contains 11 predetermined types of possible logic transformations. For example, “missing/extra wire” adds/deletes an existing wire in/from the netlist.

A predetermined dictionary model, although effective at times, may not be adequate when complex transformations are required such as the addition/deletion of multiple gates and wires. To study the effectiveness of dictionary-based transformations, 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 a wire 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 dictionary model in [8] is measured against that of *Error equation* [2] which uses formal methods to answer with certainty whether there exists an appropriate modification at a specified circuit location. *Error equation* does not return the actual modification as it merely reports whether resynthesis *may* or *may not* correct that design at the particular location.

Table I contains average results from 10 single error experiments per 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

TABLE I
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%

path-trace simulation-based diagnosis method [22], [24] that guarantees to return all such single candidate locations. The second and sixth columns of Table I 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 *Error equation* and according to an exhaustive dictionary-based rectification method [22].

It can be seen that, on the average, the dictionary model in [8] fails for as much as half of the cases with simple errors. For example, in c499_s, *Error equation* claims that some modifications 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, are common in today's intricate design environment where errors or changes in the Register-Transfer Level (RTL) necessitate complex local changes in the netlist [3]. Automated logic transformation tools that can address those problems effectively are desirable to increase the impact of the underlying debugging, rewiring, EC, etc engines.

C. Sets of Pairs of Functions to be Distinguished

Sets of Pairs of Function to be Distinguished (SPFD) are first proposed by Yamashita et al. [25]. It is a representation that provides a powerful formalism to express the functional flexibility of a design to allow synthesis/optimization on it [17], [26]–[28].

Formally, an SPFD

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

denotes a set of pairs of functions that must be *distinguished*, i.e., for each pair $(g_{ia}, g_{ib}) \in R$, the minterm g_{ia} must produce a different value from the minterm g_{ib} at the output of the node (wire) associated with R. An SPFD can be represented as a graph $G = (V, E)$ [27], where

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

Figure 1 depicts the graph representation of the SPFD, $R = \{(ab, \overline{a}b),\ (\overline{a}\overline{b}, a\overline{b})\}$. The graph contains four vertices

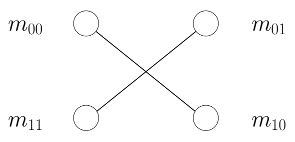


Fig. 1. The graphical representation of SPFD $R = \{(ab, \overline{a}b), (\overline{a}\overline{b}, a\overline{b})\}$

that represent minterms $\{00, 01, 10, 11\}$ in terms of $\{a, b\}$. Two edges are added for $(ab, \overline{a}b)$ and $(\overline{a}\overline{b}, a\overline{b})$. The edges are referred to as *SPFD edges*.

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 [6]) from the primary outputs to the primary inputs [27]. 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) [27]. 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,

$$\cup_{i=1}^{m} R_i \supseteq R_o, \tag{3}$$

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 . This equation implies that each minterm pair that needs to be distinguished by a node must be distinguished by one of 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. [26] to synthesize a two-level `AND-OR` network for the function f of a node so that it satisfies the given SPFD R. In summary, the set of minterms that belong to the onset of f at the node are derived from R and projected 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.

III. APPROXIMATING SPFDS

SPFDs are traditionally implemented with BDDs or with SAT. Like all BDD-based techniques, computing BDDs of some types of circuits (e.g., multipliers) may not be memory efficient [16]. The SAT-based approach alleviates the memory issue with BDDs, but it can be computationally intensive to obtain all the minterm pairs that need to be distinguished [16].

Intuitively, the runtime and memory overhead of the aforementioned approaches can be reduced if fewer minterms are captured by the formulation. Hence, this section presents a simulation-based approach to "approximate" SPFDs to reduce the information that needs to be processed. The main idea behind *a*SPFDs is that they only consider a subset of minterms that are important to the problem. Hence, an *a*SPFD is defined as follows:

Definition 1 *Let $\mathcal{M}$ consist of all primary input minterms and $\mathcal{M}'$ be a subset, where $\mathcal{M}' \subseteq \mathcal{M}$, the* **approximate SPFD (*a*SPFD)**, R_i^{appx}, *of a node n_i w.r.t $\mathcal{M}'$ specifies the minterm pairs in $\mathcal{M}' \times \mathcal{M}'$ that n_i has to distinguish. R_i^{appx} contains no information about the minterms in $\mathcal{M} - \mathcal{M}'$.*

In other words, the *a*SPFD of a node considers what needs to be distinguished only for a subset of the primary input minterms. Note that any techniques used to represent SPFDs, for example, minterm-based representation or BDD-based representation, can also apply to *a*SPFDs. It is true that, in some cases, functions with more minterms may actually result in a more compact BDD, or create a bigger cube. However, since the number of minterms considered is sometimes orders of magnitude less than the complete set (for instance, in our experiments, only 2000 out of 2^{41} minterms are used for c1355), there is a greater possibility that the representation structure of *a*SPFDs is smaller. Therefore, *a*SPFDs are inherently less expensive to represent, manipulate and compute.

To determine a good selection of minterms, logic restructuring can be effectively viewed as a pair of "error/correction" operations [4]. In this context, the required transformation simply corrects an erroneous netlist to a new specification. This is indeed the case in debugging, engineering change and more recently, it has been shown to hold for design rewiring as well [4]. From this point of view, it is constructive to see that test vectors used for diagnosis are a good means of determining minterms required to construct *a*SPFDs for logic restructuring. This is because test vectors can be thought of as the description of the erroneous behavior and minterms explored by test vectors are more critical than others.

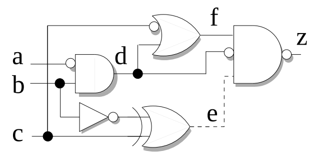
The next two subsections present the procedures to compute *a*SPFDs using a test set $\mathcal{V}$ and a SAT solver for nodes in combinational and sequential circuits, respectively.

A. Computing a*SPFDs for Combinational Circuits*

Consider two circuits, C and C', with the same number of the primary inputs and primary outputs. Let $\mathcal{V} = \{v_1, \cdots, v_q\}$ be a set of vectors, where each $v_i \in \mathcal{V}$ is a single vector. Let n_{err} be the node in C' where the correction is required, such that C' is functionally equivalent to C after restructuring. Node n_{err} can be identified using diagnosis [1], [2] or formal synthesis [3] techniques, and is referred to as a *transformation node* in the remaining discussion.

Let $f'_{n_{err}}$ denote the new function of n_{err}. The *a*SPFD of n_{err} should contain the pairs of primary input minterms that $f'_{n_{err}}$ needs to distinguish. To identify those pairs, the correct values of n_{err} under the test vectors $\mathcal{V}$ are first identified. Those values are what $f'_{n_{err}}$ should evaluate to under $\mathcal{V}$ after restructuring is implemented. Such a set of values is referred to as the *expected trace*, denoted as E_T. Finally, $on(n)(\mathit{off}(n))$ denotes the set of primary input minterms wherein the function of node n in the design evaluates to a logic value 1(0).

After the expected trace of n_{err} is calculated, the procedure uses the trace to construct the *a*SPFD of n_{err}. In practice, $\mathcal{V}$ includes vectors that detect errors ($\mathcal{V}^e$), i.e., discrepancies are observed at the primary outputs, as well as ones that do not

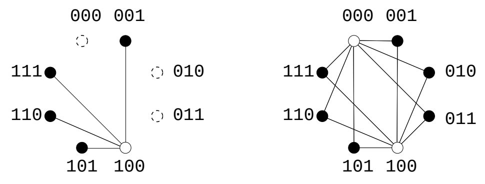


(a) Circuit

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

(b) Truth table

Fig. 2. The circuit for Examples 1 and 4



(a) *a*SPFD of z_{mod} (b) SPFD of z_{mod}

Fig. 3. The SPFD and *a*SPFD of z_{mod} in Figure 2

($\mathcal{V}^c$). Both types of vectors can provide useful information about the required transformation.

The procedure to compute the *a*SPFD of the transformation node in C' w.r.t. $\mathcal{V}$ is as follows:

1) Simulate C' with the input vector $\mathcal{V}$
2) Let $V^c(n_{err})/V^e(n_{err})$ denote the value of n_{err} when C' is simulated with $\mathcal{V}^c/\mathcal{V}^e$. Since n_{err} is the only location where transformation will be applied, the new function at n_{err} has to evaluate to the complemented value of $V^e(n_{err})$ in order to eliminate the discrepancy at the POs. Hence, the expect trace of n_{err}, denoted by $E_T^{n_{err}}$, is $\{\overline{V^e(n_{err})}, V^c(n_{err})\}$ for vectors $\{\mathcal{V}^e, \mathcal{V}^c\}$.
3) The *a*SPFD of n_{err} states that the minterms in $on(E_T^{n_{err}})$ have to be distinguished from the minterms in $\mathit{off}(E_T^{n_{err}})$, i.e., R_{err}^{appx} contains an edge for each pair $(a, b) \in \{on(E_T^{n_{err}}) \times \mathit{off}(E_T^{n_{err}})\}$.

Example 1 *Figure 2(a) depicts a sample circuit of which the truth table is shown in Figure 2(b). Assume the wire (e, z) (the dotted line) is removed, e.g., $z_{mod} = \mathsf{NAND}(\overline{d}, f)$. The value of z_{mod} is shown in the column eight of the truth table (Figure 2(b)).*

Suppose the design is simulated with test vectors $\mathcal{V}$ = {001, 100, 101, 110, 111}. The discrepancy is observed when the vector, 110, is applied. Let z_{mod} be the transformation node. As described in the Step 2 of the procedure, $V^e(z_{mod}) = \{0\}$ for $\mathcal{V}^e = \{110\}$, and $V^c(z_{mod}) = \{1, 0, 1, 1\}$ for $\mathcal{V}^c = \{001, 100, 101, 111\}$. Hence, the expected trace of z_{mod} consists of the complement of $V_e(z_{mod})$ and $V_c(z_{mod})$ as shown in the final column of Figure 2(b). Finally, the a*SPFD of z_{mod} w.r.t. $\mathcal{V}$ is generated according to E_T and contains four edges, between {100} and {001, 101, 110, 111}, as shown in Figure 3(a). The black (white) vertices indicate that z_{mod} has a logic value 1 (0) under the labelled minterm. The dotted nodes indicate that the labelled minterm is a don't care w.r.t. $\mathcal{V}$. For comparison, the SPFD of z_{mod} is shown in Figure 3(b). One can see that information included in the* a*SPFD of z_{mod} is much less than what the SPFD representation includes. The* a*SPFD of z_{mod} only contains a subset of the complete* *information about minterms that the function of z_{mod} needs to distinguish to maintain correct design functionality. The minterms that are not encountered during the simulation are considered as don't cares. For instance, since the vector 000 is not simulated, the* a*SPFD of z_{mod} does not contain an edge between 110 and 000, which is included in the SPFD of z.*

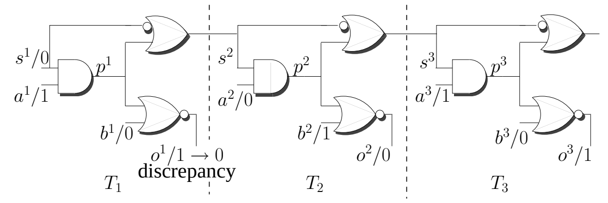
B. Computing a*SPFDs for Sequential Circuits*

Consider a sequential circuit, C', with primary input set $\mathcal{X}$, state input set $\mathcal{S}$, and primary output set $\mathcal{O}$. In this work, sequential circuits are modelled using their ILA representation. Let symbol T_i denote the i^{th} simulated timeframe and the superscript of a symbol refers to the cycle of the unfold circuit. For example, $\mathcal{X}^2$ represents the set of the primary inputs in the second timeframe (T_2). For sequential circuits, $\mathcal{V}$ represents a set of input vector sequences with k cycles.

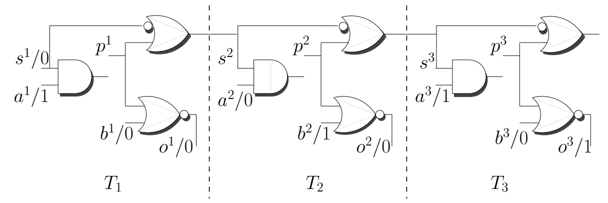
The procedure of generating *a*SPFDs presented in Section III-A cannot be used directly to generate *a*SPFDs for sequential circuits. In these circuits, the value of nets in the circuit at T_i for some input vector sequences is a function of the initial state input and the sequence of the primary input vectors up to and including cycle T_i, i.e., $f(\mathcal{S}^1, \mathcal{X}^1, \cdots, \mathcal{X}^i)$. This implies that the space of minterm pairs that considered by the *a*SPFD at n_{err} is different across timeframes. Each of these *a*SPFDs is equally important, and a valid transformation at n_{err} has to satisfy all of them. To simplify the complexity of the problem, we construct one *a*SPFD that integrates information stored in each individual *a*SPFDs. Such the *a*SPFD is generated over the input space $\{\mathcal{S} \cup \mathcal{X}\}$. Note that this approach might result in missing some sequential behavior, but it still offers more information when compared to the one that treats only the combinational circuitry of a sequential design.

To construct such an *a*SPFD, we need to determine the values of the state elements in each timeframe for the given set of input vectors. Then, a partially specified truth table, in terms of the primary input and the current states, of $f'_{n_{err}}$ can be generated. *a*SPFDs over the input space $\{\mathcal{S} \cup \mathcal{X}\}$ can be constructed according to the truth table. The complete procedure is summarized below:

1) Extract the expected trace E_T of n_{err} for an input vector sequence v. Given the expected output response ($\mathcal{Y}$) under v, a satisfiability instance, $\Phi = \prod_{i=0}^{k} \Phi^i_{C'}(v^i, \mathcal{Y}^i, n^i_{err})$, is constructed. Each $\Phi^i_{C'}$ represents a copy of C' at T_i, where n^i_{err} is disconnected from its fanins and treated as a primary input. The original primary inputs and the primary outputs of C'^i are constrained with v^i and $\mathcal{Y}^i$, respectively. The SAT solver assigns values to $\{n^0_{err}, \cdots, n^k_{err}\}$ to make C' comply with the expected responses. These values are the desired values of n_{err} for v.
2) Simulate C' with v at the primary inputs and E_T at n_{err} to determine state values in each timeframe. Those state values are what should be expected after the transformation is applied. Subsequently, a partial specified truth table (in terms of $\{\mathcal{X} \cup \mathcal{S}\}$) of $f'_{n_{err}}$ in C' can be constructed.



(a) Unrolled circuit



(b) Unrolled circuit with p's as primary inputs

Fig. 4. The circuit for Example 2

3) The *a*SPFD of n_{err} contains an edge for each minterm pair in $\{on(n_{err}) \times \mathit{off}(n_{err})\}$ according to the partially specified truth table.

Example 2 *Figure 4(a) depicts a sequential circuit unrolled for three cycles under the simulation of a single input vector sequence. Assume the correct response at o^1 should be* 0 *and net p is the transformation node. To determine the expected trace of p, p's are made into primary inputs, as shown in Figure 4(b). A SAT instance is constructed from the modified circuit with the input and output constraints. Given the instance to a SAT solver,* 110 *is returned as a valid expected trace for p. Next, simulating C' with the input vector and the expected value of p, $s^2 = 1$ and $s^3 = 1$ are obtained. Then, the partially specified truth table of p states that p evaluates to* 1 *under minterms (in terms of $\{a, b, s\}$) $\{100, 011\}$ and to* 0 *under $\{101\}$. Therefore, the* a*SPFD of p contains two edges: $(100, 101)$ and $(011, 101)$.*

A special case needs to be considered in Step 1. For any two timeframes, T_i and T_j, of the same test vector, if the values of the primary inputs and the states at these two timeframes are the same, the value of n^i_{err} must equal the value of n^j_{err}. Hence, additional clauses are added to Φ to ensure that the values of n_{err} are consistent when such conditions occur.

Example 3 *With respect to Example 2, another possible assignment to (p^1, p^2, p^3) is* 100*. However, in this case, the values of $\{a, b, s\}$ at T_1 and T_3 are both* 100*, while p^1 and p^3 have opposite values. Consequently, this is not a valid expected trace. To prevent this assignment returned by the SAT solver, the clauses*

$$(s^1 + s^3 + r) \cdot (\overline{s^1} + \overline{s^3} + r) \cdot (\overline{r} + \overline{p^1} + p^3) \cdot (\overline{r} + p^1 + \overline{p^3})$$

are added to the SAT instance. The new variable, r, equals 1*, if s^1 equals s^3. When that happens, the last two clauses ensure that p^1 and p^3 have the same value.*

C. Optimizing *a*SPFDs with Don't Cares

The procedure of *a*SPFDs generation described above does not take into account all external don't cares in the design. Identifying don't cares for n_{err} can further reduce the size of *a*SPFDs, since all SPFD edges connected to a don't care can be removed from the *a*SPFDs. Consequently, the constraints of qualified solutions for restructuring is relaxed.

There are two types of combinational don't cares: Satisfiability Don't Cares (SDCs) and Observability Don't Cares (ODCs). Since *a*SPFDs of nodes in designs are built over the minterms explored by test vectors, only ODCs need to be considered. ODCs are minterm conditions where the value of the node has no effect on the behavior of the design. Hence, ODCs of n_{err} can only be found under $\mathcal{V}_c$. Minterms encountered under the simulation of $\mathcal{V}_e$ cannot be ODCs, because, otherwise, no erroneous behavior can be observed at the primary outputs. ODCs can be easily identified by simulating the circuit with $\mathcal{V}_c$ and the complement of the original simulation value at n_{err}. If no discrepancy is observed at the primary outputs, the respective minterm is an ODC.

To obtain combinational don't cares for sequential designs, one can add the following procedures after Step 2 in Section III-B. Following Step 2, the expected trace E_T and the values of states $\hat{S}$ in each timeframe are available. One can obtain another expected trace E'_T by solving the SAT instance Φ again with additional constraints that (1) force $\hat{S}$ on all state variables (2) block E_T from the solution. If the solver returns an answer, say E'_T, we can compare E_T and E'_T to identify the timeframe T_i they have different value. Consequently, the minterm at T_i is a combinational don't care. This procedure can be repeated until no new expected trace can be found.

The procedure described above for obtaining ODCs in sequential circuits identifies expected traces with the same state transitions. To further explore equivalent states, one can obtain a trace with different state transitions. This can be done by adding additional constraints to block $\hat{S}$ assigned to state variables and solving the SAT instance again. However, these additional traces may assign conflict logic values to the transformation node for the same minterms.

Let E_{T1} and E_{T2} represent two expected traces of the same node for the same test vector sequence. Assume a conflict occurs for minterm m (in terms of the primary input and the current state) between the assignment to E_{T1} at cycle T_i and the assignment to E_{T2} at cycle T_j. In this instance, one of the two cases below is true:

- *Case 1:* The output responses and the next states at cycle T_i for E_{T1} and T_j for E_{T2} are the same. This implies that the value of the transformation node under m does not affect the behavior of the design. Hence, m is a combinational ODC.
- *Case 2:* The next states are different. This can happen when the circuit has multiple state transition paths of which the initial transitions have the same output responses. Since the proposed analysis is bounded by the length of the input vector sequences, it may not process enough cycles to differentiate these different paths. Hence, multiple assignments at the transformation node can be valid within the bounded cycle range and, consequently, cause conflicts. Since the algorithm does not have enough information to distinguish the correct assignment, minterm m in this case is considered to be a don't care as well. This issue can be resolved if longer vector sequences are used instead.

D. Validating *a*SPFDs

For combinational circuits, *a*SPFDs only explore the portion of the input space covered by the given input vectors. Similarly, for sequential circuits, *a*SPFDs only consider states that are reachable during simulation of the given input vector sequences. As a result, this new data structure requires less computation and memory resources, but these benefits come with the penalty that it has to undergo verification after restructuring to guarantee correctness. This is because the transformation is guaranteed to be valid only under the input space exercised by the given set of input test vectors. In some cases, such as rewiring, a full blown verification may not be required but a faster proof method can be used [4], [29], [30].

Furthermore, because of approximation, the accuracy of the transformation depends on the amount of information provided by the input vectors. With more vectors, *a*SPFDs become better representations of the original SPFDs. As the result, the chance that the modified circuits pass verification is higher. At the same time, the algorithm requires more resources to solve the problem. Hence, there is a trade-off between resolution and performance. Experiments show that this trade-off is a favorable one as, on average, in 90% of the cases, the first transformation returned also passes verification.

In the case where the transformation fails verification, it implies that the critical minterms are missed in the *a*SPFDs. Therefore, more input vectors need to be included. Instead of randomly generating more vectors, the counter-example returned by verification can be useful, as suggested in [10]. Intuitively, the counter-example excites the difference between the modified design and the golden model.

E. Discussion

Another recent work proposed by Chang et al. [10]–[12] presents a similar technique to the one proposed here in the context of design debugging. This technique was developed independently and first published at the same time with part of the work presented in this paper. It uses a set of input vectors and the SAT-based technique described in [1] to generate the signature of a node. This is the bit-vector of logic values that the function of the node should evaluate to in order to rectify the error. Then, it identifies all pairs of 1-0 bits of the signature and ensures that, for each pair, at least one of the fanins of the node has a value transition as well. Such a pair is referred to as a *Pair of Bits to be Distinguished (PBD)*. In a sense, due to the use of bit-values by both PBDs and *a*SPFDs, these two methodologies share in common in terms of implementation and operation. However, *a*SPFDs are defined upon the concept of SPFDs, a theory that represents all minterm pairs. The work in [10]–[12] is utilizing don't care and care sets, as they always use signatures and these can only represent a special type of subset of SPFD edges, namely a clique of the bipartite graph of the SPFD [27]. If we force ourselves to use only signatures to process the information, we can't generate the more general SPFD that might be there because a minterm is either in the care onset, the care offset, or don't care set. Due to this reason, a variety of techniques developed for SPFDs by other research groups in the past decade can be inherited with ease to operate on *a*SPFDs, a fact not true for PBDs. Finally, the methodologies to manipulate *a*SPFDs illustrated in this paper are unique and tailored to operate within this context. However, in our use of the SPFD concepts in the current paper, we also only use effectively don't cares and not the full power of SPFDs, so there is no inherent superior efficiency between our methods and the one in [10]–[12].

IV. Logic Transformations with *a*SPFDs

In this section, we show how to systematically perform logic transformations with *a*SPFDs. Recall, the goal is to re-implement the local network at n_{err} in a circuit C' such that C' implements the same function as the golden reference, C. The proposed restructuring procedure seeks the transformation at n_{err} using one or more additional fanins using *a*SPFDs.

The procedure is summarized in Algorithm 1. The basic idea is to find a set of nets such that every minterm pair of the *a*SPFD of the new transformation implemented at n_{err} is distinguished by at least one of the nets, implied by Equation 3. Hence, the procedure starts by constructing the *a*SPFD (including don't cares) of n_{err}, denoted by R_{err}^{appx}. To minimize the distortion that may be caused by restructuring, the original fanins are kept for restructuring. In other words, it is sufficient that the function of additional fanins only need to distinguish edges in R_{err}^{appx} that cannot be distinguished by any original fanins (line 6). Those undistinguished edges are referred to as *uncovered edges*. A function is said to *cover* an SPFD edge if it can distinguish the respective minterm pair. Let $TFO(n_{err})$ denote the transitive fanout cone of n_{err}. The function, SelectCover, is used to select a set of nodes ($Cover$) from nodes not in $TFO(n_{err})$ such that each uncovered edge is distinguished by at least one node in $Cover$ (line 8). SelectCover is further discussed in the next subsections. Finally, a new two-level `AND-OR` network is constructed at n_{err} using the nodes in $Cover$ as additional fanins as discussed in Section II-C.

Example 4 *Returning to Example 1, the only SPFD edge of the* a*SPFD of z_{mod} that is not covered by the fanins of z_{mod}, $\{f, d\}$ is (110, 100). This means that the additional fanins required for restructuring at z_{mod} must distinguish this edge. One can verify that the function of b can distinguish this minterm pair. Hence, b is used as the additional fanin for restructuring z_{mod}; the new function of z_{mod} is* `NAND`$(\overline{b}, \overline{d}, \overline{f})$*. With this new function, the XOR gate can be removed, which reduces the original gate count from eight to seven gates.*

Two approaches to find $Cover$ are presented in the following subsections: an optimal SAT-based approach that finds the minimal number of fanin wires and a greedy approach that exchanges optimality for performance.

Algorithm 1 Transformation using *a*SPFDs
```
 1: $C'$ := Erroneous circuit
 2: $\mathcal{V}$ := A set of input vectors
 3: $n_{err}$ := Transformation node
 4: procedure (Transformation_With_aSPFD)($C'$, $\mathcal{V}$. $n_{err}$)
 5:     Compute the aSPFD and don't cares of $n_{err}$
 6:     $E \leftarrow (m_i, m_j) \in R_{err}^{appx} | (m_i, m_j)$ cannot be
        distinguished by any fanin of $n_{err}$
 7:     Let $\mathcal{N} := \{n_k \mid n_k \in C'$ and $n_k \notin \{TFO(n_{err}) \cup n_{err}\}$
 8:     $Cover \leftarrow$ SelectCover($\mathcal{N}$)
 9:     Re-implementing $n_{err}$ with the original fanins and
        the nodes in $Cover$
10: end procedure
```

A. SAT-based Searching Algorithm

The search problem in Algorithm 1 (line 8) is formulated as an instance of Boolean satisfiability. Recall that the algorithm looks for a set of nodes outside $TFO(n_{err})$ such that those nodes can distinguish SPFD edges of R_{err}^{appx} that cannot be distinguished by any fanins of n_{err}.

The SAT instance Φ is formulated as follows. Each node n_k is associated with a variable w_k. Node n_k is added to the set $Cover$ if the value of w_k is 1. The instance contains two components: $\Phi(\mathcal{W}, \mathcal{P}) = \Phi_C(\mathcal{W}) \cdot \Phi_B(\mathcal{W}, \mathcal{P})$, where $\mathcal{W} = \{w_1, w_2, \cdots\}$ for each $n_k \notin \{TFO(n_{err}) \cup n_{err}\}$ and $\mathcal{P}$ is a set of new variables introduced.

The first component, $\Phi_C(\mathcal{W})$, contains one clause for each edge in E. The clause, c_i, for the edge, e_i, contains w_k if the function of n_k covers e_i. The clause is satisfied if one of the included candidate nodes is selected.

The second component, $\Phi_B(\mathcal{W}, \mathcal{P})$, defines the condition where a candidate node, n_k, should not be considered as a solution. As discussed in Section II-C, minterms that can be distinguished by a node must be able to be distinguished by one of its fanins. It implies that the function of the node does not cover more SPFD edges if all of its fanins are selected already. Hence, for each candidate node n_k, a new variable p_k is introduced; p_k is 1 if all of fanins of n_k are selected, and 0 otherwise. Consequently, w_k is assigned with 0 when p_k is 1, i.e., n_k is not considered for the solution.

With respect to Example 4, the SAT instance of the searching problem, $\Phi = \Phi_C \cdot \Phi_B$, is constructed as follows. Since there is only one SPFD edge (110, 100) in the *a*SPFD of z_{mod} that needs to be covered, Φ_C consists of one clause only, which indicates candidate nodes of which function can distinguish the minterm pair. In this case, the candidate nodes of edge (110, 100) are b and e. Therefore, the formulation of Φ_C is $(b + e)$. That is, this edge is covered if the SAT solver assigns 1 to either b or e.

Next, considering node e, of which fanins are b and c. As discussed earlier, if b and c are already selected as additional fanins to the new structure, e does not provide the ability to distinguish more minterm pairs and can be removed from the candidate list. This idea is formulated as, $\Phi_B = (\overline{b} + \overline{c} + p_e) \cdot (b + \overline{p}_e) \cdot (c + \overline{p}_e) \cdot (\overline{p}_e + \overline{e})$, where p_e is a new variable that has the value of 1 if both b and c are selected. When p_e equals 1, e is forced to be 0; that is, e cannot be selected.

In order to obtain the optimal solution, in experiments, we solve the SAT instance with a pseudo-Boolean constraint SAT solver [31] that returns a solution with the smallest number of nodes. The use of a pseudo-Boolean solver is not mandatory and any DPLL-based SAT solvers [32], [33] can be used instead. A way to achieve this is to encode the counter circuitry from [1] to count the number of selected nodes. Then, by enumerating values $N = 1, 2, \ldots$, the constraint enforces that no more than N variables can be set to a logic 1 simultaneously or Φ becomes unsatisfiable. Constraining the number N in this manner, any DPLL-based SAT solver can return the optimal answer.

B. Greedy Searching Algorithm

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

1) For each edge $e \in E$, let N_e be the set of nodes $n \notin \{TFO(n_{err}) \cup n_{err}\}$ that can distinguish the edge. Sort $e \in E$ in descending order by the cardinality of N_e.
2) Select the edge, e_{min}, with the smallest cardinality of $N_{e_{min}}$. It ensures that the edge that can be covered with the least number of candidates is targeted first.
3) Select n_k from $N_{e_{min}}$ such that n_k covers the largest set of edges in E, and add n_k to $Cover$
4) Remove edges that can be covered by n_k from E. If E is not empty, go back to Step 1 to select more nodes.

The solutions identified by the greedy approach may contain more wires than the minimum set. However, experiments indicate that the greedy approach can achieve similar quality results with the SAT-based approach in a more computationally efficient manner.

C. Methodology Analysis

In theory, the transformations returned using *a*SPFDs may not pass the verification since they are based on a small set of test vectors. Nevertheless, experiments show that the success rate is very high and more than 90% of the first fix returned by the method qualifies verification. This implies that a small amount of test vectors can provide sufficiently enough information to generate a qualified transformation. Here, we elaborate on the reasons why the proposed *a*SPFD-based representation has such a high success rate.

Definition 2 *The* **local minterms** *of a node n are minterms in terms of the immediate fanins of n. The* **local SPFD** *(*c*SPFD) of a node n specifies the local minterms in the onset of n that have to be distinguished from the local minterms in the offset of n.*

The local *a*SPFD of n, $R_n^{local} = (V', E')$, can be *translated* from $R_n^{appx} = (V, E)$ through the following steps. Let M_i be a primary input minterm and m_i be the corresponding local minterm of n. First, for every $M_i \in V$, m_i is added to V'. Then, for every edge $e = (M_i, M_j)$, where $e \in E$, an edge between (m_i, m_j) is added to E'.

The local network represented by SPFDs (or *a*SPFDs) is synthesized by identifying local minterms in the onset of the node [26]. Hence, given an SPFD, R, and an *a*SPFD, R^{appx}, of the same node, the transformations constructed based on R or R^{appx} are the same if the same local SPFD can be derived from R and R^{appx}. This can be achieved if the input vector set used to construct *a*SPFDs complies with the following lemma.

Lemma 1 *The local* a*SPFD of n, R_n^{local}, translated from R_n^{appx}, contains the same pairs of minterms to be distinguished as the* c*SPFD of n (*c*SPFD$_n$), if, for each possible local minterm of n that is not a don't care, one of its corresponded primary input minterms is included in the R_n^{appx}.*

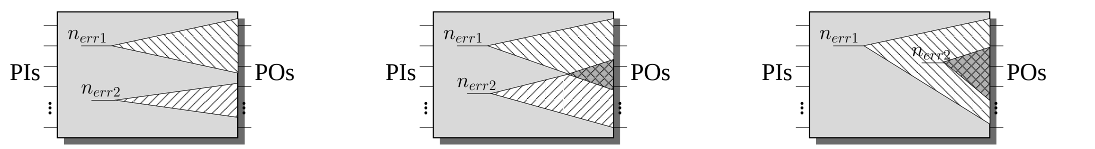
Proof: To show that R_n^{local} and *c*SPFD$_n$ contain the same pairs of minterms to be distinguished, it requires (a) that both R_n^{local} and *c*SPFD$_n$ contain the same number of vertices and (b) that every edge in *c*SPFD$_n$ is also in R_n^{local}. The first requirement can be derived from the assumption that R_n^{appx} contains at least one primary input minterm that can be mapped to each local minterms in R_n^{local}. This leads to the conclusion that both R_n^{local} and *c*SPFD$_n$ contain the same number of nodes. For the second requirement, because R_n^{appx} contains an edge for every pair of minterms that are assigned with opposite values, by construction, R_n^{local} also contains an edge for every pair (m_i, m_j), where $f_n(m_i) \neq f_n(m_j)$. As a result, edges in *c*SPFD$_n$ must be in R_n^{local} as well. ■

According to Lemma 1, if the set of primary input minterms exercised by the input vectors complies with the lemma, the *a*SPFD constructed by the proposed approach contains all of the necessary information about the required function at the transformation node. As a result, the transformation that is constructed based on the *a*SPFD can pass verification. Furthermore, if the transformation is constrained to have at most N fanins, one may conclude that 2^N test vectors are sufficient to perform the restructuring, if the selected test vectors comply to Lemma 1. Since N is usually much less than the number of the primary inputs, 2^N test vectors are a small portion of the complete vector space.

V. Extension to Multiple Locations

All discussion this far deals with restructuring one location each time. However, as noted in the introduction, in practice, there are several situations where restructuring at multiple locations is necessary. In this section, we show how to apply the proposed methodology to perform multiple transformations in a combinational design. This concept can be easily extended to sequential circuits as well.

Given two combinational designs C and C' and, without loss of generality, assume transformations need be applied on two locations, n_{err1} and n_{err2}, in C' simultaneously such that C' becomes functionally equivalent to C. Depending on the locations of n_{err1} and n_{err2}, as shown in Figure 5, a different approach can be followed.



(a) Non-overlap between $TFO(n_{err1})$ and $TFO(n_{err2})$ (b) Overlap between $TFO(n_{err1})$ and $TFO(n_{err2})$ (c) $TFO(n_{err1})$ includes $TFO(n_{err2})$

Fig. 5. Relation of two transform locations

The first case (Figure 5(a)) depicts the situation where the transitive fanout cones of n_{err1} and n_{err2} are exclusive to each other. This implies that these two locations can be restructured independently, because changes at one location do not affect the other one. In this case, the proposed methodology discussed in Section III-A can be applied directly at each one location at a time.

In the second case shown in Figure 5(b), there are common POs in the transitive fanout cones of the two nodes. The discrepancy observed at those POs may require fixes on both nodes to be resolved. Here, an approach similar to the one described for sequential designs in Section III-B can be used as summarized below:

1) Extract the expected trace E_{T1} for n_{err1} and E_{T2} for n_{err2}. This can be done by formulating a Boolean satisfiability instance from the circuit C' where n_{err1} and n_{err2} are marked as primary inputs.
2) Obtain combinational don't cares of n_{err1} by solving all possible expected traces on n_{err1} while the value of n_{err2} is as specified in E_{T2}. Then, as in Section III-C, if there are conflict assignments between expected traces, the corresponding minterms are don't cares.
3) Construct a partially specified truth table of n_{err1} from the expected traces. The table specifies the function that can resolve all erroneous observation at POs when it is implemented at n_{err1} and n_{err2} is assigned with E_{T2}.
4) Generate the *a*SPFD of n_{err1} from the truth table and construct a qualified transformation.
5) Apply the transformation and repeat Step 2 – 4 for n_{err2}.

The procedure described above for two locations can be easily generalized when restructuring occurs in three or more places in a design. Overall, the procedure restructures one targeted node in each iteration from Step 2 to Step 4. This is necessary in order to obtain the correct combinational don't cares for the targeted node, since they can be different depending on the transformations applied at other nodes. Hence, after each transformation, the expected trace of the next target node is re-calculated to take the effects from the previous transformations into account.

Finally, the third case of the relation of two transformation locations is shown in Figure 5(c), where n_{err2} is inside $TFO(n_{err1})$. In this case, the same procedure described for the second case can be applied. Although it may seem that n_{err1} needs to be restructured before n_{err2}, it is not necessary. This is because aSPFDs are constructed based on the expected traces extracted in Step 1 of the above procedure. Those values show the behavior of the transformation nodes after restructuring. Therefore, even if n_{err2} is restructured first, the minterm pairs that the original fanins of n_{err2} can distinguish after restructuring can be determined without constructing the transformation at n_{err1}. Consequently, it is possible to restructure locations in any order.

VI. Experiments

Empirical results of the proposed methodology are presented in this section. ISCAS'85 benchmarks are used for experiments on combinational transformations, while ISCAS'89 benchmarks are used for sequential cases. The diagnosis algorithm from [1] is used to identify the restructuring locations and Minisat [34] is the underlying SAT solver. The restructuring potential of the *a*SPFD-based algorithms is compared with that of the dictionary-model of [8] and both methodologies are contrasted against the complete results of *Error equation* [2]. Experiments are conducted on a Core 2 Duo 2.4GHz processor with 2GB of memory while runtime is reported in seconds.

A. Experiment setup

In our experimental setup, three different complexities of modifications are injected in the original benchmark. Experiments involve correcting designs with those changes to evaluate the performance of the proposed methodology. The locations and the types of modifications are randomly selected. Simple complexity modifications (suffix "s") involve the addition/deletion of a single wire or a gate type replacement. Moderate modifications (suffix "m") on a gate include addition/deletion multiple fanins and a gate type change. The final complexity modifications, complex (suffix "c"), inject multiple simple complexity modifications on a gate and in the fanin cone of the gate.

For each of the above types, five testcases are generated from each benchmark. The proposed algorithm is set to find 10 transformations different to the original, if they exist, for each location identified by the diagnosis algorithm. Functional verification is carried out at the end to check the validity of the transformations.

B. Performance of the Methodology

Table II summarizes the experimental results for a single transformation in combinational circuits. In this experiment, circuits are simulated with 2000 input vectors with high stuck-at fault coverage. The first column of Table II lists the benchmarks and the types of the modification inserted as described in Section VI-A. Columns two and three show the numbers of five testcases per benchmark that the dictionary model and the proposed approach can find at least one solution

TABLE II
COMBINATIONAL LOGIC TRANSFORMATION RESULTS FOR VARIOUS COMPLEXITIES OF MODIFICATIONS

ckt. name	Correctablility (Per testcase)		Correctablility (Per location)				avg # wires	min # wires	avg # corr/loc.	avg. time (sec)	% verified	
	dict. model [8]	*a*SPFD	error loc.	error equat.	dict. model	*a*SPFD					first	all
c1355_s	4	5	5.3	100%	19%	81%	1.7	1.7	8.3	3.5	100%	46%
c1908_s	3	5	18.0	84%	13%	84%	1.4	1.4	8.1	18.9	90%	62%
c3540_s	3	5	7.2	100%	28%	86%	1.1	1.1	4.5	9.3	100%	66%
c7552_s	4	5	11.8	88%	19%	50%	1.7	–	3.1	25.7	88%	54%
c1355_m	2	5	2.7	100%	13%	100%	2.1	2.0	7.0	32.0	100%	52%
c1908_m	1	4	5.8	100%	3%	83%	2.5	2.5	5.6	11.0	100%	68%
c3540_m	3	5	3.2	100%	25%	100%	1.6	1.6	6.1	54.2	84%	78%
c7552_m	3	5	8.8	100%	9%	91%	1.9	–	6.9	39.2	100%	79%
c1355_c	0	3	3.7	96%	0%	73%	2.9	2.9	3.3	38.4	100%	40%
c1908_c	4	4	15.8	47%	41%	70%	1.4	1.3	7.2	19.0	100%	88%
c3540_c	1	5	3.0	100%	7%	67%	3.6	3.4	3.8	122.4	100%	33%
c7552_c	3	5	20.6	64%	20%	50%	1.9	–	3.5	23.7	91%	43%
Average	2.5	4.7	8.8	90%	16%	78%	2.0	–	5.6	33.1	96%	59%

at any location, respectively. From the result, one can see that the proposed approach is able to restructure most cases, while the ability of the dictionary model approach to rectify designs drops as the complexity of the modifications increases. This implies the advantage of the proposed approach when deals with problems, such as engineer changes, where required transformations are more complex than the addition/removal of a single net.

The number of locations that the proposed approach can restructure is compared with the result of *Error equation* and the dictionary model approach as shown in columns four – seven. The fourth column has the average number of locations returned by the diagnosis program for the five testcases. The percentage of those locations where *Error equation* claims an existence of a solution is shown in column five. The next two columns show the percentage of locations (out of those in column four) that the dictionary-approach and the proposed *a*SPFD approach can successfully find a valid solution. A valid solution is the one that the restructured circuit passes verification. Taking `c1908_s` as an example, there are 18 locations returned by the diagnosis program. *Error equation* claims that resynthesis can fix 15 out of those locations. The dictionary approach successfully identifies two locations (13% of 15) while the *a*SPFD approach can restructure 13 locations (84% of 15). In this case, the proposed approach has, on average, five times improvement over the dictionary one. Overall, the proposed methodology outperforms the dictionary approach in all cases and achieves greater improvement when the modification is complicated.

Columns 8 – 13 present the quality of the transformations in terms of the wires involved as well as some algorithm performance metrics. Note, except column nine which reports the result of the SAT-based search approach, the remaining columns report the result of the greedy search approach. Column eight has the average number of additional wires returned by the greedy algorithm and column nine has the minimum number of wires selected by the optimal SAT-based searching algorithm. The average is computed over all valid transformations of all transformation nodes in five test cases. As shown in the table, the greedy heuristic performs well comparing to the optimal solutions. Because the SAT-based approach may run into runtime problems as the number of new wires increases, it times out (”-”) after 1000 seconds if it does not return with a solution. The experimental results show that the number of additional wires overall is less than four. This suggests that the transformations only alter the design with small changes, which is important in logic rewiring, debugging or when applying engineering changes. Furthermore, we also observe that when the algorithm selects more than five wires as additional fanins to the transformation node, there is a higher probability that the transformations fail verification. As discussed in Section IV-C, the transformation is guaranteed to pass verification if each local minterm that is not a don’t care can be mapped from one of the primary input minterms included in the *a*SPFD of the transformation node (Lemma 1). As the number of selected wires increases, the chance that some critical location minterms are missed increases as well. Consequently, those transformations are not valid solutions.

As mentioned earlier, for each location, the algorithm is set to find at most 10 transformations if they exist. Column 10 shows the average number of transformations identified for each location. One can see that for all cases, more than one transformation can be identified. This is a desirable characteristic since engineers can have more options to select the best fit for the application. Column 11 contains the average runtime to find 10 transformations using the greedy heuristics. Note that the runtime does not include the procedure of identifying transformation nodes, since any diagnosis or verification technique can be used to identify those locations. The final two columns show the average percentage of transformations that pass verification. The first column only considers the first transformation identified, while the second column has this percentage for 10 transformations identified. One can observe that the vast majority of first-returned transformations pass verification, a fact that confirms the viability of *a*SPFDs.

Similar experiments are conducted for sequential designs as well. The vector set for sequential circuits contains 500 input vector sequences with a length of 10 cycles. To verify the correctness of transformations, a bounded sequential equivalent checking [35] is used. It verifies the resulting design against the reference within a finite number of cycles, which is set to 10 cycles in our experiment.

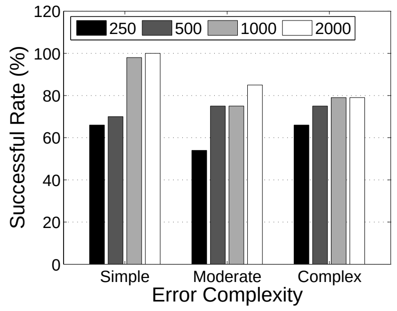
TABLE III
SEQUENTIAL LOGIC TRANSFORMATION RESULTS FOR VARIOUS COMPLEXITIES OF MODIFICATIONS

ckt. name	error loc.	error equat.	*a*SPFD	avg # wires	avg # corr/loc.	avg. time (sec)	% verified first	% verified overall	% unique
s510_s	2.4	100%	75%	0.3	1.8	384	100%	92%	100%
s953_s	1.8	100%	33%	1.0	3.3	223	100%	37%	70%
s1238_s	1.6	100%	38%	1.1	5.0	781	100%	100%	55%
s1488_s	2.8	86%	43%	1.7	5.0	258	83%	46%	68%
s510_m	2.0	100%	90%	0.3	4.2	68	100%	38%	99%
s953_m	1.6	63%	40%	1.2	1.2	105	100%	100%	100%
s1238_m	2.6	85%	72%	2.2	4.3	218	100%	76%	47%
s1488_m	3.4	100%	0%	–	–	83	–	–	–
s510_c	1.6	100%	38%	0.5	1.5	166	100%	92%	100%
s953_c	2.2	73%	0%	–	–	122	–	–	–
s1238_c	1.2	100%	14%	0	–	328	100%	–	100%
s1488_c	1.8	71%	30%	1.7	1.5	98	33%	27%	100%
Average	2.1	90%	39%	1.0	3.1	236	92%	68%	82%

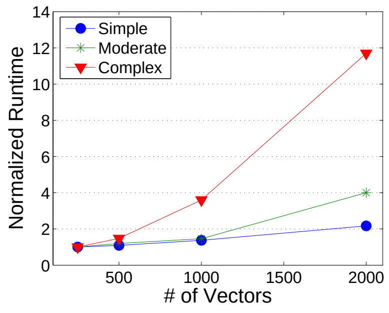
The results for sequential designs are summarized in Table III. Benchmarks used are listed in column one. Column two presents the average number of locations for transformation reported by the diagnosis program while the percentage of those locations that are claimed to be correctable by *Error equation* are recorded in column three. Note that *Error equation* in [2] is developed for combinational circuits. Hence, here we convert the sequential circuits into a combinational one by treating the states as pseudo-input/output. In this way, the number of locations reported by *Error equation* is the lower bound of the locations that are correctable, since it constrains the states to be equivalent after the restructuring as well.

The percentage of locations that the proposed methodology finds a valid transformation is reported in column four. Overall, our approach can restructure 39% of the locations. The reason why the algorithm fails to correct some of the locations is because the input vectors do not provide enough information to generate a good *a*SPFD. This occurs when the algorithm characterizes a minterm as a don't care when this minterm is not exercised by the input vectors. Consequently, the resulting transformation does not distinguish all necessary minterm pairs that are required to correct the design.

Column 5 – 7 report the average number of additional wires used in the transformations, the average number of transformations per location, and the average runtime to find 10 transformations, respectively. Note that, for some locations, the transformation only needs to be resynthesized with the existing fanin nets without any additional wires. This is the reason why cases, such as `s510_s`, use less than one additional wire on average. The next two columns show the average percentage of cases where the first transformation passes verification and the average percentage of 10 transformations that passe verification. Similar to the combinational circuits, there is a high percentage of the first transformation that passes verification if the proposed methodology can find any. This indicates that the *a*SPFD is a good metric to prune out invalid solutions. Finally, those transformations are checked whether they can be identified by restructuring the sequential designs as if they are pure combinational ones. This check is carried out by performing combinational equivalence checking between the transformed circuit and the reference. If two designs are not combinational equivalent, it means that the transformation changes the state assignments as well. Overall, 82% of the valid transformations are uniquely identified by the sequential approach and they cannot be found by a combinational logic restructuring method.



(a) Solution success rate

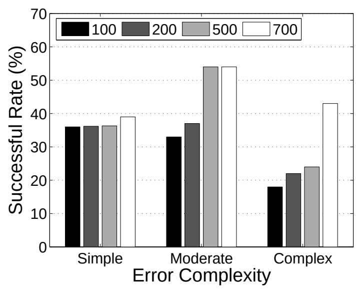


(b) Runtime profiling

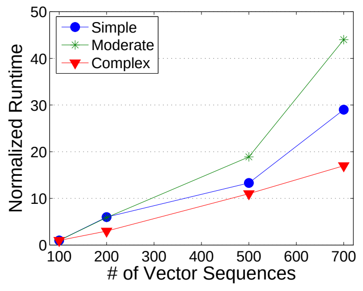
Fig. 6. Performance of restructuring with various numbers of vectors for combinational designs

C. Impact of Test Vectors

In the second set of the experiments, we first investigate the performance of the restructuring when various numbers of test vectors are used. For combinational circuits, four sizes are used: 250, 500, 1000, and 2000 test vectors. Results are depicted in Figure 6. In details, Figure 6(a) shows the percentage of the locations where the proposed algorithm can identify a valid transformation. As shown, the success rate is increased as the size of input vectors increases for each



(a) Solution success rate



(b) Runtime profiling

Fig. 7. Performance of restructuring with various numbers of vectors for sequential designs

error complexity group. This is expected since more vectors provide more information for *a*SPFDs. The chance that the algorithm incorrectly characterizes a minterm as a don't care is also reduced.

Although using a larger vector set can improve the success rate of the restructuring, it comes with the penalty that more computational resources are required to tackle the problem. The average runtime is plotted in Figure 6(b) and normalized by comparing it to the runtime of the case with 250 vectors. Each line represents one error complexity type. Taking `Complex` as an example, the runtime is 12 times longer when the vector size is increased from 250 to 2000. Note that there is a significant increase when the size of the vector set increases from 1000 to 2000. If we look back Figure 6(a), one may see that the success rate of cases when 1000 vectors are used is close to the rate of those with 2000 vectors. This suggests that, for those testcases, 1000 input vectors can be a good size to have a balance between the resolution of solutions and the runtime performance.

The same set of analysis is applied to sequential designs as well. For sequential cases, the vector sizes are set to 100, 200, 500, and 700. All have a length of 10 cycles. The success rate and the normalized runtime are shown in Figure 7(a) and Figure 7(b), respectively. One can see that the behavior observed earlier for the combinational cases is also observed here. The success rate of the restructuring decreases as the size of the vector decreases. Among different error complexities, the benchmarks with complex errors are affected most. This is because a complex error can be excited in various ways and requires more vectors to fully characterize the erroneous behavior. As a result, the algorithm needs more vectors to construct an accurate transformation.

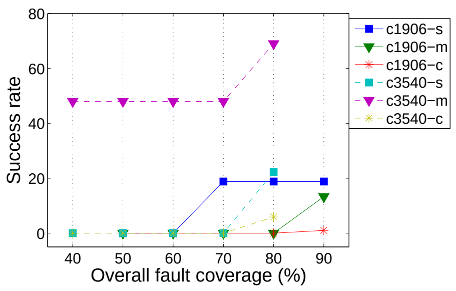


Fig. 8. Performance of restructuring with test vector sets that have various fault coverages

Lastly, the runtime plotted in Figure 7(b) also shows a significantly reduction with the decrease of the size of vectors. One may notice that the runtime for complex modification cases increases slower with the number vectors, compared with the runtime for simple modification cases. Note that, for the same base benchmark, the locations where the modifications are inserted for each complexity are different. Hence, it is not necessary that the runtime growth rate of the complex modification cases has to increase faster than the runtime growth rate of the simple modification cases. The key point here is that there is a great runtime increase as more test vectors are used for restructuring.

When the number of test vectors varies, the fault coverage of the test vectors changes as well. To study whether the performance of the algorithm is sensitive to the fault coverage of the given test vector set, an experiment that uses vector sets that have different fault coverage metrics is conducted. Two benchmarks, `c1908` and `c3540`, are used. A set of 100 test vectors with five different fault coverage metrics are generated for each benchmark. The fault coverage is calculated for the whole design because in our experiments there are several transformation nodes randomly distributed within the design. For `c1908`, the fault coverage is 50% – 90%; for `c3540`, the fault coverage is 40% – 80%. The success rate of each case is plotted in Figure 8. We observe that, when vector sets with low coverage are used, the algorithm may not be able to find a valid solution for many cases. However, when the coverage metric is increased, there is a higher success rate of finding a valid transformation.

D. *Transforming Multiple Locations*

In the last set of experiments, we apply the algorithm to perform simultaneous restructuring at multiple locations. For the purpose of demonstration, we only perform experiments for two and three locations with simple types of errors. In this experiment, simulation-based verification is performed after a transformation at each transformation node is constructed. It uses 100 random vectors to check any discrepancies at the primary outputs that are only in the fanout cone of the transformation node. Another transformation is sought if verification fails; at most 10 transformations are checked before the algorithm returns back to the previous transformation node. The algorithm stops when a valid solution is identified or the

TABLE IV
LOGIC TRANSFORMATION AT TWO AND THREE LOCATIONS

ckt. name	2-location					3-location				
	error loc.	succ. rate	avg # wires	avg # trans. fail	avg. time (sec)	error loc.	succ. rate	avg # wires	avg # trans. fail	avg. time (sec)
c880	9.2	39%	1.1	0	329	10.0	32%	1.6	0	666
c1355	10.0	66%	1.2	0	649	10.0	68%	1.2	0.06	1226
c1908	10.0	36%	1.0	0	852	10.0	16%	1.4	0	1136
c2670	8.6	37%	1.1	0	1596	9.6	32%	1.3	0	2608
c3540	7.4	70%	0.9	0.03	2626	9.2	48%	1.1	0	2757
Average	9.0	50%	1.1	0.006	1210	9.8	39%	1.3	0.01	1679

failing limit has been reached. The experimental results are summarized in Table IV.

Columns 2–6 in Table IV summarize the results when restructuring is applied at two locations, while columns 7–11 contain results when three locations are restructured simultaneously. Columns two and seven contain the average number of locations returned by the diagnosis program for the five experiments. Note that because there are multiple errors, the diagnosis program may return numerous candidate solution tuples [21]. As such, in the experiments, we only randomly pick at most 10 location tuples as the candidates for restructuring.

Columns three and eight have the percentage of the selected location tuples that our algorithm successfully identifies a valid transformation tuple. The average number of additional wires required to construct the transformations are shown in columns four and nine. For example, in the case of 2-location restructuring for `c1335`, our algorithm is able to identify corrections for 7 out of 10 tuples returned by the diagnosis tool. The average number of additional wires used to construct the corrections is only 1.2. Our empirical observation has been that when the number of additional wires required increases, there is a high chance that the transformation will not pass verification. Columns five and ten show the average number of transformation tuples that fail verification before a valid one is found.

As shown in the table, in all cases valid solutions are usually identified at the beginning of the search process. Taking `c1355` as an example, in both cases the first transformation tuple determined by the proposed technique is a valid solution. This result is consistent with the observation in the single-transformation experiments described earlier. Finally, the average runtime is recorded in columns six and eleven. As expected, the runtime increases as the number of restructured locations increases. In summary, it is seen that, for both cases, our technique corrects, on average, 50% and 39% of location tuples with only less than two additional wires for each constructed transformation. The result confirms that the proposed methodology has the ability to restructure multiple locations efficiently and minimally as well.

VII. CONCLUSION

In this work, a simulation-based procedure for a new representation of SPFDs, namely *a*SPFDs, is first presented. The *a*SPFD is an approximation of the original SPFD as it only contains information that is explored by the simulation vectors. As a result, this technique can alleviate the memory/runtime issues that may encountered by formal approaches. In addition, this work proposes an *a*SPFD-based logic restructuring algorithm with SAT for both combinational and sequential designs. This technique can be used for a wide range of applications, such as logic optimization, debugging and when applying engineer changes. Experiments demonstrate that *a*SPFDs provide a powerful and dynamic method to restructure a logic design to a new set of specification. It is able to restructure designs at a location where other methods fail. Further empirical analysis confirms that the resolution of the transformation depends on the number of vectors used. A higher success rate can be achieved if more input vectors are provided but, at the same time, more memory/computation resources may be required.

The work and experiments of this paper promote further research in *a*SPFDs as a means to logic restructuring. This may include improving the *a*SPFD construction for sequential designs to increase the success rate of viable solutions. Another application involves changing of timing-elements in sequential designs, for instance, addition/removal of sets of states for retiming. This promises a new set of possibilities in algorithmic restructuring for sequential designs.

REFERENCES

[1] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.

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

[3] C. C. Lin, K. C. Chen, and M. Marek-Sadowska, "Logic synthesis for engineering change," *IEEE Trans. on CAD*, vol. 18, no. 3, pp. 282–292, March 1999.

[4] A. Veneris and M. S. Abadir, "Design rewiring using ATPG," *IEEE Trans. on CAD*, vol. 21, no. 12, pp. 1469–1479, Dec. 2002.

[5] L. Entrena and K. T. Cheng, "Combinational and sequential logic optimization by redundancy addition and removal," *IEEE Trans. on CAD*, vol. 14, no. 7, pp. 909–916, July 1995.

[6] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting local don't cares for network optimization," in *Proc. of Int'l Conf. on CAD*, 1991, pp. 514–517.

[7] 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 Conf. on CAD*, 1992, pp. 328–333.

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

[9] Y.-S. Yang, S. Sinha, A. Veneris, and R. K. Brayton, "Automating logic rectification by approximate SPFDs," in *Proc. of ASP Design Automation Conf.*, Jan. 2007, pp. 402–407.

[10] K. H. Chang, I. L. Markov, and V. Bertacco, "Fixing design errors with counterexamples and resynthesis," in *Proc. of ASP Design Automation Conf.*, Jan. 2007, pp. 944–949.

[11] ——, “Fixing design errors with counterexamples and resynthesis,” *IEEE Trans. on CAD*, vol. 27, no. 1, pp. 184 – 188, Jan. 2008.

[12] K. H. Chang, I. L. Markov, and V. Vertacco, *Functional Design Errors in Digital Circuits: Diagnosis, Correction and Repair*. Springer, 2009.

[13] Y. S. Yang, S. Sinha, A. Veneris, R. K. Brayton, and D. Smith, “Sequential logic rectifications with approximate SPFDs,” in *Proc. of Design, Automation and Test in Europe*, April 2009, pp. 1698 – 1703.

[14] J. Zhang, S. Sinha, A. Mishchenko, R. K. Brayton, and M. C. Jeske, “Simulation and satisfiability in logic synthesis,” in *Int’l Workshop on Logic Synth.*, 2005.

[15] S. Yamashita, H. Sawada, and A. Nagoya, “SPFD: A new method to express functional flexibility,” *IEEE Trans. on CAD*, vol. 19, no. 8, pp. 840–849, Aug. 2000.

[16] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. K. Brayton, and M. Chrzanowska-Jeske, “Using simulation and satisfiability to compute flexibilities in Boolean networks,” *IEEE Trans. on CAD*, vol. 25, no. 5, pp. 743–755, May 2006.

[17] S. Sinha, A. Kuehlmann, and R. K. Brayton, “Sequential SPFDs,” in *Proc. of Int’l Conf. on CAD*, 2001, pp. 84–90.

[18] Y. Watanabe and R. K. Brayton, “Incremental synthesis for engineering changes,” in *Int’l Conf. on Comp. Design*, 1991, pp. 40–43.

[19] S. C. Chang, M. Marek-Sadowska, and K. T. Cheng, “Perturb and simplify: Multi-level Boolean network optimizer,” *IEEE Trans. on CAD*, vol. 15, no. 12, pp. 1494–1504, Dec. 1996.

[20] S.-C. Chang and D. I. Cheng, “Efficient Boolean division and substitution using redundancy addition and removing,” *IEEE Trans. on CAD*, vol. 18, no. 8, Aug. 1999.

[21] A. Veneris and I. N. Hajj, “Design error diagnosis and correction via test vector simulation,” *IEEE Trans. on CAD*, vol. 18, no. 12, pp. 1803–1816, Dec. 1999.

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

[23] M. Fujita, “Methods for automatic design error correction in sequential circuit,” in *Design Automation Conf.*, 1993, pp. 76–80.

[24] A. Veneris, M. S. Abadir, and I. Ting, “Design rewiring based on diagnosis techniques,” in *Proc. of ASP Design Automation Conf.*, 2001, pp. 479–484.

[25] S. Yamashita, H.Sawada, and A. Nagoya, “A new method to express functional permissibilities for LUT based FPGAs and its applications,” in *Proc. of Int’l Conf. on CAD*, Nov. 1996, pp. 254–261.

[26] J. Cong, J. Y. Lin, and W. Long, “SPFD-based global rewiring,” in *Int’l Symposium on FPGAs*, Feb. 2002, pp. 77–84.

[27] S. Sinha and R. K. Brayton, “Implementation and use of SPFDs in optimizing Boolean networks,” in *Proc. of Int’l Conf. on CAD*, Nov. 1998, pp. 103–110.

[28] S. Almukhaizim, Y. Makris, Y.-S. Yang, and A. Veneris, “On the minimization of potential transient errors and SER in logic circuits using SPFD,” in *iolts*, July 2008, pp. 123 – 128.

[29] W. Kunz, D. Stoffel, and P. R. Menon, “Logic optimization and equivalence checking by implication analysis,” *IEEE Trans. on CAD*, vol. 16, no. 3, pp. 266–281, March 1997.

[30] J. H. Jiang and R. K. Brayton, “On the verification of sequential equivalence,” *IEEE Trans. on CAD*, vol. 22, no. 6, pp. 686 – 697, June 2003.

[31] N. E*é*n and N. S*ö*rensson, “Translating pseudo-boolean constraints into SAT,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, March 2006.

[32] J. P. Marques-Silva and K. A. Sakallah, “GRASP: a new search algorithm for satisfiability,” *IEEE Trans. on Comp.*, vol. 48, no. 5, pp. 506–521, May 1999.

[33] M. W. Moskewicz, C. F. Madigan, and S. Malik, “Chaff: Engineering an efficient SAT solver,” in *Design Automation Conf.*, 2001, pp. 530–535.

[34] N. E*é*n and N. Sörensson, “An extensible SAT-solver,” in *SAT*, 2003.

[35] S. Safarpour, G. Fey, A. Veneris, and R. Drechsler, “Utilizing don’t care states in SAT-based bounded sequential problems,” in *Great Lakes VLSI Symp.*, 2005.

PLACE PHOTO HERE

Yu-Shen Yang recieved the B.A.Sc, M.A.Sc and Ph.D degrees in computer engineering from University of Toronto in 2002, 2004, and 2010, respectively. He is currently a senior software engineer at Vennsa Technologies, Toronto, Ontario, Canada, where he is in charge of research and development. His research interestes include logic design debugging and correction, logic resynthesis and silicon debug.

PLACE PHOTO HERE

Andreas Veneris received a Diploma in Computer Engineering and Informatics from the University of Patras in 1991, an M.S. degree in Computer Science from the University of Southern California, Los Angeles in 1992 and a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1998. In 1998 he was a visiting faculty at the University of Illinois until 1999 when he joined the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto where today he is an Associate Professor. His research interests include CAD for debugging, verication, synthesis and test of digital circuits/systems, and combinatorics. He has received several teaching awards and a best paper award. He is the author of one book and he holds three patents.

He is a member of ACM, IEEE, AAAS, Technical Chamber of Greece, Professionals Engineers of Ontario and The Planetary Society.

PLACE PHOTO HERE

Subarna Sinha Subarnarekha Sinha received her Bachelors degree from the Department of Electronics and Electrical Communication Engineering, Indian Institute of Technology, Kharagpur and her Masters and Ph.D from the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley in 1996, 1998 and 2002, respectively.

She is currently at Synopsys, Inc. Her research interests cover various aspects of electronic design automation ranging from logic synthesis to mask synthesis.

Dr Sinha was the recipient of the Synopsys Inventor Award in 2009 and the Institute silver medal for outstanding academic performance in 1996.

PLACE PHOTO HERE

Robert K. Brayton Robert Brayton received the BSEE degree from Iowa State University in 1956 and the Ph.D. degree in mathematics from MIT in 1961. He was a member of the Mathematical Sciences Department of the IBM T. J. Watson Research Center until he joined the EECS Department at Berkeley in 1987. He held the Edgar L. and Harold H. Buttner Endowed Chair and retired as the Cadence Distinguished Professor of Electrical Engineering at Berkeley. He is currently a professor in the graduate school at Berkeley.

He is a member of the US National Academy of Engineering and has received the following awards: IEEE Emanuel R. Piore (2006), ACM Kanallakis (2006), European DAA Lifetime Achievement (2006), EDAC/CEDA Phil Kaufman (2007), D.O. Pederson best paper in Trans. CAD (2008), ACM/IEEE A. Richard Newton Technical Impact in EDA (2009), and Iowa State University Distinguished Alumnus (2010).

He has authored over 450 technical papers, and 10 books in the areas of the analysis of nonlinear networks, simulation and optimization of electrical circuits, logic synthesis, and formal design verification.