Chapter 1
Automated Logic Restructuring with *a*SPFDs

1.1 Chapter Overview

This chapter presents a comprehensive methodology to automate logic restructuring in combinational and sequential circuits. This technique algorithmically constructs the required transformation by utilizing a functional flexibility representation called *Set of Pairs of Function to be Distinguished* (SPFD). SPFDs can express more functional flexibility than the traditional don't cares and have proved to provide additional degrees of flexibility during logic synthesis [21, 27].

Computing SPFDs may suffer from memory or runtime problems [16]. Therefore, a simulation-based approach to approximate SPFDs is presented to alleviate those issues. The result is called *Approximate SPFDs* (*a*SPFDs). *a*SPFDs approximate the information contained in SPFDs using the results of test-vector simulation. With the use of *a*SPFDs as a guideline, the algorithm searches for the necessary nets to construct the required function. Experimental results indicate that the proposed methodology can successfully restructure locations where a previous approach that uses a dictionary model [1] as the underlying transformation template fails.

The remainder of this chapter is structured as follows. Section 1.2 discusses the motivation of restructuring logic-level design. Section 1.3 summarizes previous work in logic restructuring, as well as the basic concept of SPFDs. Section 1.4 defines *a*SPFDs and the procedures used to generate *a*SPFDs. Section 1.5 presents the transformation algorithms utilizing *a*SPFDs. Experimental results are given in Section 1.6, followed by the conclusion in Section 1.7.

1.2 Introduction

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 [23, 6]. In the case of engineering changes (EC) [13], 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 [24, 10] where designs are optimized at particular internal locations to meet specification constraints.

Although these problems can be resolved by another round of full logic synthesis, directly modifying logic netlists is usually preferable in order to preserve any engineering effort that has been invested. Hence, logic restructuring has significant merits when compared to re-synthesis. Today, most of these incremental logic changes are implemented manually. The engineer examines the netlist to determine what changes need to be made and how they can affect the remainder of the design.

One simple ad-hoc logic restructuring technique modifies the netlist by using permissible transformations from a *dictionary model* [1], which contains a set of simple modifications, such as single wire additions or removals. This technique is mostly adapted for design error correction [6, 18] and has been used in design rewiring as well [24]. A predetermined dictionary model, although effective at times, may not be adequate when complex transformations are required. It has been shown that a dictionary-model based design error correction tool can only successfully rectify 10-30% of cases [28]. Complex modifications perturb the functionality of the design in ways that simple dictionary-driven transformations may not be able to address. Therefore, automated logic transformation tools that can address these problems effectively are desirable.

The work presented in this chapter aims to develop a comprehensive methodology to automate logic restructuring in combinational and sequential circuits. It first presents a simulation-based technique to approximate SPFDs, or simply *a*SPFDs. Using aSPFDs can keep the process memory and runtime efficient while taking advantage of most of the benefits of SPFDs. Then, an *a*SPFD-based logic restructuring methodology is presented. It uses *a*SPFDs as a guideline to algorithmically restructure the functionality of an internal node in a design. Two searching algorithms, an SAT-based algorithm and a greedy algorithm, are proposed to find nets required for restructuring the transformation. The SAT-based algorithm selects minimal numbers of wires, while the greedy algorithm returns sub-optimal results with a shorter runtime.

Extensive experiments confirm the theory of the proposed technique and show that *a*SPFDs provide an effective alternative to dictionary-based transformations. It returns modifications where dictionary-based restructuring fails, increasing the impact of tools for debugging, rewiring, EC, etc. For combinational circuits, the proposed approach can identify five times more valid transformations than a dictionary-based one. Experiments also demonstrate the feasibility of using *a*SPFDs to restructure sequential designs. Although this method bases its results on a small sample of the input test vector space, empirical results show that more than 90% of the first solution returned by the method passes formal validation.

1.3 Background

This section reviews previous work on logic restructuring and summarizes the concept of SPFDs.

1.3.1 Prior Work on Logic Restructuring

Most research done on logic restructuring deals with combinational designs. In [26], the authors insert circuitry before and after the original design so that the functionality of the resulting network complies with the required specifications. The main disadvantage of this approach is that the additional circuitry may be too large and can dramatically change the performance of the design.

Redundancy addition and removal (RAR) [10, 5] 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 [4].

In [24], the authors view logic optimization from a logic debugging angle. It introduces a design error into the design, identifies locations for correction with a debug algorithm, and rectifies those locations with a dictionary model [1]. This method has been shown to exploit the complete solution space, and offers great flexibility in optimizing a design and achieving larger performance gains. The technique presented in this chapter adapts the same viewpoint to logic restructuring.

Two recent approaches [14, 3] are similar to the one presented in this chapter. They construct the truth table of the new function at the location that requires restructuring and synthesize the new function based on the table. However, both approaches provide few descriptions on the application on sequential designs.

1.3.2 Sets of Pairs of Functions to be Distinguished

Sets of Pairs of Function to be Distinguished (SPFD) is a representation that provides a powerful formalism to express the functional flexibility of nodes in a multilevel circuit. The concept of SPFD was first proposed by Yamashita et al. [27] for applications in FPGA synthesis, and has been used in many applications for logic synthesis and optimization [7, 21, 22].

Formally, an SPFD

$$R = \{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \cdots, (g_{na}, g_{nb})\} \tag{1.1}$$

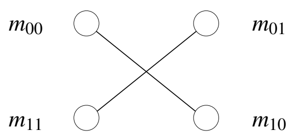


Fig. 1.1 The graphical representation of SPFD $R = \{(ab, \overline{a}b), (\overline{a}\overline{b}, a\overline{b})\}$

denotes a set of pairs of functions that must be *distinguished*. That is, for each pair $(g_{ia}, g_{ib}) \in R$, the output of the node (wire) associated with R must have different values between a minterm of g_{ia} and a minterm of g_{ib}.

In [21], an SPFD is represented as a graph, $G = (V, E)$, where

$$\begin{aligned} V &= \{m_k \mid m_k \in g_{ij}, 1 \leq i \leq n, j = \{a,b\}\} \\ E &= \{(m_i, m_j) \mid \{(m_i \in g_{pa}) \text{ and } (m_j \in g_{pb})\} \\ &\quad \text{or } \{(m_i \in g_{pb}) \text{ and } (m_j \in g_{pa})\}, \\ &\quad 1 \leq p \leq n\} \end{aligned} \tag{1.2}$$

This graphical representation makes it possible to visualize SPFDs and can explain the concept of SPFDs more intuitively. Figure 1.1 depicts the graph representation of the SPFD, $R = \{(ab, \overline{a}b), (\overline{a}\overline{b}, a\overline{b})\}$. The graph contains four vertices 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 edge is referred to an *SPFD edge*.

SPFDs of a node or wire can be derived in a multitude of ways, depending on their application during logic synthesis. For instance, SPFDs can be computed from the primary outputs in reverse topology order [21, 27]. An SPFD of a node represents the minterm pairs in which the function of the node must evaluate to different values. In rewiring applications, the SPFD of a wire, (η_a, η_b), can denote the minimum set of edges in the SPFD of η_b that can only be distinguished by η_a (but none of the remaining fanins of η_b) [21]. In all these methods, the SPFD of a node complies with Property 1.1, which indicates that the ability of a node to distinguish minterm pairs cannot be better than the ability of all of its fanins. This property is the key to performing logic restructuring with SPFDs in this work.

Property 1.1. Given a node η_k whose fanins are $\{\eta_1, \eta_2, \cdots, \eta_n\}$, the SPFD of η_k is the subset of the union of the SPFDs of its fanin nodes [20].

Finally, a function of a node can be synthesized from its SPFD into a two-level `AND-OR` network with an automated approach by Cong et al. [7].

1.4 Approximating SPFDs

SPFDs are traditionally implemented with BDDs or with SAT. However, each approach has its own disadvantage. Computing BDDs of some types of circuits (e.g., multipliers) may not be memory efficient [2]. 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. Although *a*SPFDs are based on a small set of the complete input space, experiments show that *a*SPFDs include enough information to construct valid transformations.

To determine a good selection of minterms, logic restructuring can be effectively viewed as a pair of "error/correction" operations [24]. In this context, the required transformation simply corrects an erroneous netlist to a new specification. 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. Since an *a*SPFD of a node stores less information than its respected SPFD, it is inherently less expensive to represent, manipulate and compute.

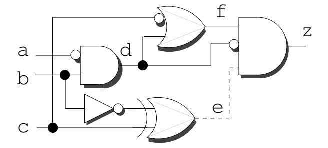
Due to the loss of information, the transformation is guaranteed to be valid only under the input space exercised by the given set of input test vectors only. The transformations may fail to correct designs respected to the complete input space. Hence, the design has to undergo verification after restructuring to guarantee its correctness. However, in some cases, such as in rewiring, a full blown verification may not be required, but a faster proof method can be used [12, 24, 11].

The next two subsections present the procedures used to compute *a*SPFDs using a test vector set for nodes in combinational and sequential circuits, respectively.

1.4.1 Computing a*SPFDs for Combinational Circuits*

Consider two circuits, C_e and C_c, with the same number of the primary inputs and primary outputs. Let $\mathscr{V} = \{v_1, \cdots, v_q\}$ be a set of vectors. For combinational circuits, each $v_i \in \mathscr{V}$ is a single vector, while for sequential circuits, each v_i is a sequence of input vectors. Let η_{err} be the node in C_e where the correction is required, such that C_e is functionally equivalent to C_c after restructuring. Node η_{err} can be identified using diagnosis [6, 23] or formal synthesis [13] techniques, and is referred to as a *transformation node* in the remaining discussion.

Let $f'_{\eta_{err}}$ denote the new function of η_{err}. As discussed earlier, the *a*SPFD of η_{err} should contain the pairs of primary input minterms that $f'_{\eta_{err}}$ needs to distinguish. To identify those pairs, the correct values of η_{err} under the test vectors $\mathscr{V}$ are first identified. Those values are what $f'_{\eta_{err}}$ should evaluate for $\mathscr{V}$ after restructuring is implemented. Such a set of values is referred to as *expected trace*, denoted as E_T. Finally, $on(n)(off(n))$ denotes the set of minterms that n is equal to 1(0).

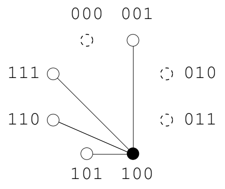


(a) Circuit

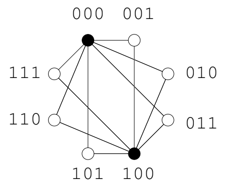
a	b	c	d	e	f	z	z_{mod}	E_T
0	0	0	0	1	1	1	1	-
0	0	1	0	0	0	0	0	0
0	1	0	1	0	1	0	0	-
0	1	1	1	1	1	0	0	-
1	0	0	0	1	1	1	1	1
1	0	1	0	0	0	0	0	0
1	1	0	0	0	1	0	1	0
1	1	1	0	1	0	0	0	0

(b) Truth table

Fig. 1.2 The circuit for Examples 1.1 and 1.5



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



(b) SPFD of z_{mod}

Fig. 1.3 SPFD and *a*SPFDs of z_{mod} in Figure 1.2

After the expected trace of η_{err} is calculated, the procedure uses the trace to construct *a*SPFDs of η_{err}. In practice, $\mathscr{V}$ includes vectors that detect errors ($\mathscr{V}_e$), as well as ones that do not ($\mathscr{V}_c$). Both types of vectors can provide useful information about the required transformation.

The procedure to generate the *a*SPFD of the transformation node in C_e w.r.t. $\mathscr{V} = \{\mathscr{V}_e \cup \mathscr{V}_c\}$ is as follows. First, C_e is simulated with the input vector $\mathscr{V}$. Let $V_c(\eta_{err})$ and $V_e(\eta_{err})$ denote the value of η_{err} when C_e is simulated with $\mathscr{V}_c$ and $\mathscr{V}_e$. To rectify the design, $f'_{\eta_{err}}$ has to evaluate to the complemented values of $V_e(\eta_{err})$. That is, the expected trace of η_{err}, denoted by $E_T^{\eta_{err}}$, is $\{\overline{V_e(\eta_{err})}, V_c(\eta_{err})\}$ for vectors $\{\mathscr{V}_e, \mathscr{V}_c\}$. Finally, The *a*SPFD of η_{err} states that minterms in $on(E_T^{\eta_{err}})$ have to be distinguished from minterms in $off(E_T^{\eta_{err}})$.

Example 1.1. Figure 1.2(a) depicts a circuit; its truth table is shown in Figure 1.2(b). Let the wire $e \longrightarrow z$ (the dotted line) be the target to be removed. After the removal of $e \longrightarrow z$, an erroneous circuit is created where the new z, labelled z_{mod}, becomes $\texttt{NAND}(\overline{d}, f)$. The value of z_{mod} is shown in the eighth column of the truth table.

Suppose the design is simulated with test vectors $\mathscr{V} = \{001, 100, 101, 110, 111\}$. The discrepancy is observed when the vector 110 is applied. Therefore, $\mathscr{V}_e = \{110\}$ and $\mathscr{V}_c = \{001, 100, 101, 111\}$. Let z_{mod} be the transformation node. $V_e(z_{mod}) = \{1\}$ and $V_c(z_{mod}) = \{0, 1, 0, 0\}$. Hence, the expected trace of z_{mod} consists of the complemented values of $V_e(z_{mod})$ and $V_c(z_{mod})$, as shown in the final column of Figure 1.2(b). Finally, the *a*SPFD of z_{mod} w.r.t. $\mathscr{V}$ is generated according to E_T and contains four edges, as shown in Figure 1.3(a). The dotted vertices indicate that the labelled minterm is a don't care w.r.t. $\mathscr{V}$. For comparison, the SPFD of z_{mod} is shown in Figure 1.3(b). One can see that information included in *a*SPFD of z_{mod} is much less than what the SPFD representation includes. The minterms that are not encountered during the simulation are considered don't cares in *a*SPFDs. For instance, the minterm pair, (110, 000), does not need to be distinguished in the *a*SPFD of z_{mod} because the vector 000 is not simulated.

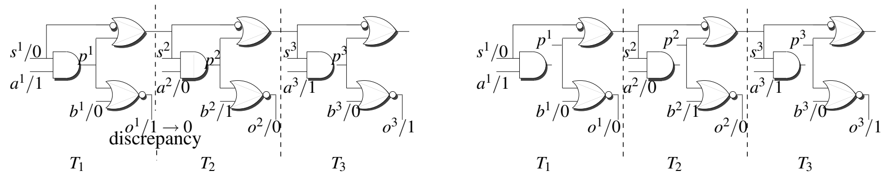
1.4.2 *Computing* a*SPFDs for Sequential Circuits*

The procedure of building *a*SPFDs of nodes in sequential circuits is more complicated than the procedure for combinational circuits due to the state elements. Each node in the circuit not only depends on the present value of the primary inputs, but also on values applied to them in previous timeframes. This time-dependency characteristic of sequential designs prohibits the application of the procedure of generating *a*SPFDs presented in Section 1.4.1 directly to sequential designs. First of all, the expected trace of the transformation node, η_{err}, is not simply the complemented values of the node under the erroneous vector sequences. Because it is not known in which timeframe the error condition is excited, complementing values in all timeframes risks the introduction of more errors. Moreover, when modeling sequential designs in the ILA representation, 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. Hence, the input space of *a*SPFD of a node is different in each timeframe.

To simplify the complexity of the problem, the proposed procedure constructs one *a*SPFD over the input space $\{\mathscr{S} \cup \mathscr{X}\}$ that integrates information stored in the *a*SPFD in each timeframe. It first determines the values of the state elements in each timeframe for the given set of input vectors that should take place after the transformation. Then, a partially specified truth table of the new function at η_{err}, in terms of the primary input and the current states, can be generated. The *a*SPFD of η_{err} over the input space $\{\mathscr{S} \cup \mathscr{X}\}$ is constructed based on the truth table. The complete procedure is summarized below:

Step 1. Extract the expected trace E_T of η_{err} for an input vector sequence v. Given the expected output response ($\mathscr{Y}$) under v, a satisfiability instance, $\Phi = \prod_{i=0}^{k} \Phi_{C_e}^{i}(v^i, \mathscr{Y}^i, \eta_{err}^{i})$, is constructed. Each $\Phi_{C_e}^{i}$ represents a copy of C_e at timeframe i, where η_{err}^{i} is disconnected from its fanins and treated as a primary input. The original primary inputs and the primary outputs of C_e^i are constrained with v^i and $\mathscr{Y}^i$, respectively. The SAT solver assigns values to $\{\eta_{err}^{0}, \cdots, \eta_{err}^{k}\}$ to make C_e comply with the expected responses. These values are the desired value of η_{err} for v.

Step 2. Simulate C_e with v at the primary inputs and E_T at η_{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 $\{\mathscr{X} \cup \mathscr{S}\}$) of $f'_{\eta_{err}}$ in C_e can be constructed.



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

Fig. 1.4 The circuit for Example 1.2

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

Example 1.2. Figure 1.4(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 1.4(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_e 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 η^i_{err} must equal the value of η^j_{err}. Hence, additional clauses are added to Φ to ensure that the values of η_{err} are consistent when such conditions occur.

Example 1.3. With respect to Example 1.2, 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.

1.4.3 *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 η_{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 η_{err} can only be found under $\mathscr{V}_c$. Minterms encountered under the simulation of $\mathscr{V}_e$ 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 $\mathscr{V}_c$ and complement of the original simulation value at η_{err}. If no discrepancy is observed at the primary outputs, the respected minterm is an ODC.

Similarly, combinational ODCs of node η in a sequential design are assignments to the primary inputs and current states such that a value change at η is not observed at the primary outputs or at the next states. However, combinational ODCs of sequential designs may be found in erroneous vector sequences. This is because the sequential design behaves correctly until the error is excited. Having this in mind, the following procedures can be added after Step 2 in Section 1.4.2 to obtain combinational ODCs.

Step 2a. Let E_{T1} denote the expected trace obtained in Step 2 in Section 1.4.2 and $\hat{S}$ denote the values of states in each timeframe. Another expected trace E_{T2} can be obtained by solving the SAT instance Φ again with additional constraints that (a) force $\hat{S}$ on all state variables, and (b) block E_{T1} from being selected as a solution again. Consequently, the new expected trace consists of different values at the transformation node such that the same state transition is maintained.

Step 2b. Let E_{T2} be the second expected trace, and T_i be the timeframe where E_{T1} and E_{T2} have different values. It can be concluded that the minterm at T_i is a combinational don't care, since, at T_i, the values of the primary outputs and the next states remain the same, regardless of the value of η.

Step 2c. Repeat this procedure until no new expected trace can be found.

Example 1.4. In Example 1.2, an expected trace, $E_T = 110$, has been obtained, and the state value, $\{s^1, s^2, s^3\}$, is $\{0,1,1\}$. To obtain another expected trace at p, additional clauses, $(\overline{s^1})(s^2)(s^3)$, are added to ensure states $\{s^1, s^2, s^3\}$ to have the value $\{0,1,1\}$. Another clause, $(\overline{p^1} + \overline{p^2} + p^3)$, is added to prevent the SAT solver to assign $\{1,1,0\}$ to $\{p^1, p^2, p^3\}$ again. In this example, another expected trace of p, $E_{T2} = 010$, can be obtained. The values of E_T and E_{T2} are different at T_1, which implies that the minterm 100 in terms of $\{a,b,s\}$ is a don't care. Hence, the *a*SPFD of p can be reduced to contain only one edge, (011, 101).

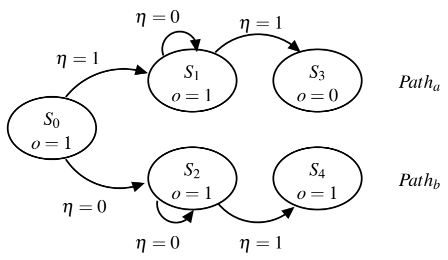


Fig. 1.5 State transition

1.4.3.1 Conflicts in Multiple Expected Traces

In the case of sequential circuits, there can exist multiple expected traces for the given input sequences and the expected output responses. The procedure described earlier 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 with the same initial transitions. Figure 1.5 shows an example of this scenario. Let η be the transformation node. The graph depicts a state transition diagram for a single-output design. The state transition depends on the value of η; the value of the output is indicated inside the state. Assume a test vector makes the design start at S_0. It takes at least three cycles to differentiate the transition $Path_a$ and $Path_b$, since the value of the primary output is not changed until the design is in S_4. 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. In the example, if the circuit is only unrolled for two cycles, both paths ($S_0 \rightarrow S_1$ and $S_0 \rightarrow S_2$) would seem to be the same from the observation of the primary outputs. It implies that η can have either logic 0 and logic 1 in S_0. Since the algorithm does not have enough information to distinguish the correct

Algorithm 1 Transformation using *a*SPFDs

```
 1: $C_e$ := Erroneous circuit
 2: $\mathscr{V}$ := A set of input vectors
 3: $\eta_{err}$ := Transformation node
 4: TRANSFORMATION_WITH_ASPFD($C_e$, $\mathscr{V}$. $\eta_{err}$) {
 5:     Compute aSPFDs of $\eta_{err}$
 6:     $E \leftarrow (m_i, m_j) \in R^{appx}_{\eta_{err}} | (m_i, m_j)$ cannot be distinguished by any fanin of $\eta_{err}$
 7:     Let $\mathscr{N}$ := $\{\eta_k \mid \eta_k \in C_e$ and $\eta_k \notin \{TFO(\eta_{err}) \cup \eta_{err}\}$
 8:     $Cover \leftarrow$SELECTCOVER($\mathscr{N}$)
 9:     Re-implementing $\eta_{err}$ with the original fanins and the nodes in Cover
10: }
```

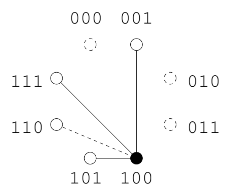
assignment, minterm m in this case is considered to be a don't care as well. This issue can be resolved if vector sequences that are long enough are used instead.

1.5 Logic Transformations with *a*SPFDs

In this section, the procedure to systematically perform logic restructuring with *a*SPFDs is presented. The proposed restructuring procedure uses *a*SPFDs to seek transformations at the transformation node, η_{err}. The transformations are constructed with one or more additional fanins.

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 η_{err} is distinguished by at least one of the nets, as stated in Property 1.1. Hence, the procedure starts by constructing the *a*SPFD of η_{err}, $R^{appx}_{\eta_{err}}$. To minimize the distortion that may be caused by the rectification, the original fanins are kept for restructuring. In other words, it is sufficient that *a*SPFDs of additional fanins only need to distinguish edges in $R^{appx}_{\eta_{err}}$ 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 respected minterm pair. Let $TFO(\eta_{err})$ denote the transitive fanout of n_{err}. The function SELECTCOVER is used to select a set of nodes (*Cover*) from nodes not in $TFO(\eta_{err})$ such that each uncovered edge is distinguished by at least one node in *Cover* (line 8). The function SELECTCOVER is further discussed in the next subsections. Finally, a new two-level `AND-OR` network is constructed at η_{err} using the nodes in *Cover* as additional fanins as discussed in Section 1.3.2.

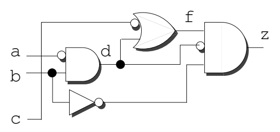
Example 1.5. Returning to Example 1.1, the *a*SPFD of z_{mod} is shown in Figure 1.6(a) and the partial truth table of remaining nodes are shown in Figure 1.6(b). Figure 1.6(a) shows that the edge (110, 100) (the dotted line) is the only SPFD edge that is not distinguished by the fanin of z_{mod}, $\{f, d\}$. Hence, the additional fanins required for restructuring at z_{mod} must distinguish this edge. According to the truth table, this edge can be distinguished by b. As a result, b is used as the additional fanin for restructuring z_{mod}. Since the minterm 100 is the only minterm in the onset



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

a	b	c	d	e	f
0	0	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	0	0	1
1	1	1	0	1	0

(b) Partial Truth Table



(c) Modified circuit

Fig. 1.6 *a*SPFD of z_{mod} and the partially specified truth table of nodes in Figure 1.2 and modified circuit

of new z_{mod} w.r.t. $\mathscr{V}$. It implies $b = 0$, $d = 0$ and $f = 1$. Therefore, the new function of z_{mod} is NAND($\overline{b}, \overline{d}, \overline{f}$), as shown in Figure 1.6(c).

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

1.5.1 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 $TFO(\eta_{err})$ such that those nodes can distinguish SPFD edges of $R^{appx}_{\eta_{err}}$ that cannot be distinguished by any fanins of η_{err}.

Construction of the SAT instance is fairly straightforward. Each uncovered SPFD edge in the *a*SPFD of η_{err} has a list of nodes that can distinguish the edge. The SAT solver selects a node from each list such that at least one node in the list of each uncovered SPFD edge is selected. The set, *Cover*, consists of these selected nodes. The formulation of the SAT instance Φ is as follows. Each node η_k is associated with a variable w_k. Node η_k is added to the set *Cover* if w_k is assigned a logic value 1. The instance contains two components: $\Phi_C(\mathscr{W})$ and $\Phi_B(\mathscr{W}, \mathscr{P})$, where $\mathscr{W} = \{w_1, w_2, \cdots\}$ is the set of variables that are associated with nodes in the circuit and $\mathscr{P}$ is a set of new variables introduced.

- *Covering clauses* ($\Phi_C(\mathscr{W})$): A covering clause lists the candidate nodes for an uncovered edge. A satisfied covering clause indicates that the associated edge is covered. One covering clause, c_j, is constructed for each uncovered edge e_j in the *a*SPFD of η_{err}. Let $\mathscr{D}_j$ be the candidate nodes which can cover edge e_j. Clause c_j contains w_k if η_k in $\mathscr{D}_j$ covers e_j; that is, $c_j = \bigvee_{\eta_k \in \mathscr{D}_j} w_k$. Hence, this clause is satisfied if one of the included candidate nodes is selected.
- *Blocking clauses* ($\Phi_B(\mathscr{W}, \mathscr{P})$): Blocking clauses define the condition where a candidate node η_k should not be considered as a solution. They help to prune the solution space and prevent spending time on unnecessary searches. For each node $\eta_k \notin \{TFO(\eta_{err}) \cup \eta_{err}\}$, according to Property 1.1, η_k does not distinguish

more edges if all of its fanins are selected already. Hence, for each candidate node η_k, a new variable, p_k, is introduced; p_k is assigned a logic value 1 if all of the fanins of η_k are selected, and 0 otherwise. Consequently, w_k is assigned a logic value 0 (i.e., η_k is not considered for the solution) when p_k has a logic value 1. The blocking clause for node $\eta_k = f(\eta_1, \eta_2, \cdots, \eta_m)$, where η_i, $1 \leq i \leq m$, is a fanin of η_k, is as follows: $(\bigvee_{i=1}^{m} \overline{w_i} + p_k) \cdot \bigwedge_{i=1}^{m} (w_i + \overline{p_k}) \cdot (\overline{p_k} + \overline{w_k})$.

Given a satisfying assignment for Φ, a node η_k is added to the set *Cover* if $w_k = 1$. The covering clauses ensure that *Cover* can cover all the edges in the *a*SPFD of η_{err}. Blocking clauses reduce the possibility of the same set of edges being covered by multiple nodes in *Cover*. If the function derived by the satisfying assignment from the set $Cover = \{w_1, w_2, \cdots, w_n\}$ fails formal verification, then $(\overline{w_1} + \overline{w_2} + \cdots + \overline{w_n})$ is added as an additional blocking clause to Φ and the SAT solver is invoked again to find another solution.

Note that in the above formulation because there are no constraints on the number of nodes that should be selected to cover the edges, the solution returned by the solver may not be optimal. In order to obtain the optimal solution, in experiments, SAT instances are solved with a pseudo-Boolean constraint SAT solver [8] 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 [15, 17] can be used instead. One way to achieve this is to encode the counter circuitry from [23] 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 value 1 simultaneously or Φ becomes unsatisfiable. Constraining the number N in this manner, any DPLL-based SAT solver can find the minimum size of *Cover*.

1.5.2 Greedy Searching Algorithm

Although the SAT-based formulation can return the minimum set of fanins to re-synthesize η_{err}, experiments show that, at times, it may require excessive runtime. To improve the runtime performance, a greedy approach to search solutions is proposed:

Step 1. Let E be the set of SPFD edges in the *a*SPFD of η_{err} that needs to be covered. For each edge $e \in E$, let N_e be the set of nodes $\eta \notin \{TFO(\eta_{err}) \cup \eta_{err}\}$ which can distinguish the edge. Sort $e \in E$ in descending order by the cardinality of N_e.

Step 2. Select the edge, e_{min}, with the smallest cardinality of $N_{e_{min}}$. This step ensures that the edge that can be covered with the least number of candidates is targeted first.

Step 3. Select η_k from $N_{e_{min}}$ such that η_k covers the largest set of edges in E and add η_k to *Cover*

Step 4. Remove edges that can be covered by η_k from E. If E is not empty, go back to Step 1 to select more nodes.

Table 1.1 Characteristics of benchmarks

Combinational				Sequential			
Circ.	# PI	# FF	# Gates	Circ.	# PI	# FF	# Gates
c1355	41	0	621	s510	19	6	256
c1908	33	0	940	s713	35	19	482
c2670	157	0	1416	s953	16	29	476
c3540	50	0	174	s1196	14	18	588
c5315	178	0	2610	s1238	14	18	567
c7552	207	0	3829	s1488	8	6	697

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

1.6 Experimental Results

The proposed logic restructuring methodology using *a*SPFDs is evaluated in this section. ISCAS'85 and ISCAS'89 benchmarks are used. The diagnosis algorithm from [23] is used to identify the restructuring locations and Minisat [9] is the underlying SAT solver. The restructuring potential of the *a*SPFD-based algorithms is compared with that of a logic correction tool from [25] which uses the dictionary-model of [1]. Both methodologies are compared against the results of a formal method, called *error equation* [6]. This method answers with certainty whether there exists a modification that corrects design at a location. Experiments are conducted on a Core 2 Duo 2.4GHz processor with 4GB of memory while the runtime is reported in seconds.

Table 1.1 summarizes the characteristics of benchmarks used in this experiment. Combinational benchmarks are listed in the first four columns, while sequential benchmarks are shown in the last four columns. The table includes the number of primary inputs, the number of flip-flops and the total number of gates in each column, respectively.

In this work, performance of the proposed methodology is evaluated with the ability to correct errors in logic netlists. Three different complexities of modifications are injected in the original benchmark. The locations and the types of modifications are randomly selected. Simple complexity modifications (suffix "s") involve the addition or deletion of a single wire, replacement of a fanin with another node and a gate-type replacement. Moderate modifications (suffix "m") on a gate include multiple aforementioned changes on a single gate. The final type of modification complexity, complex (suffix "c"), injects multiple simple complexity modifications on a gate and those in the fanout-free fanin cone of the gate.

Table 1.2 Combinational logic transformation results for various complexities of modifications

Circ.	Error loc.	Error equat.	Dict. model	*a*SPFD	Avg time (sec)	Avg # wires (greedy)	Min # wires (SAT)	Avg # corr/loc.	% verified First	% verified All
c1355_s	5.3	100%	19%	81%	3.5	1.7	1.7	8.3	100%	46%
c1908_s	18.0	84%	13%	84%	18.9	1.4	1.4	8.1	90%	62%
c2670_s	9.2	98%	11%	82%	21.9	2.4	2.2	6.2	100%	75%
c3540_s	7.2	100%	28%	86%	9.3	1.1	1.1	4.5	100%	66%
c5315_s	6.4	100%	25%	100%	7.6	1.9	–	5.4	89%	77%
c7552_s	11.8	88%	19%	50%	25.7	1.7	–	3.1	88%	54%
c1355_m	2.7	100%	13%	100%	32.0	2.1	2.0	7.0	100%	52%
c1908_m	5.8	100%	3%	83%	11.0	2.5	2.5	5.6	100%	68%
c2670_m	5.2	96%	4%	60%	95.4	3.4	2.9	9.4	100%	60%
c3540_m	3.2	100%	25%	100%	54.2	1.6	1.6	6.1	84%	78%
c5315_m	9.6	94%	2%	100%	46.7	2.9	–	5.7	100%	77%
c7552_m	8.8	100%	9%	91%	39.2	1.9	–	6.9	100%	79%
c1355_c	3.7	96%	0%	73%	38.4	2.9	2.9	3.3	100%	40%
c1908_c	15.8	47%	41%	70%	19.0	1.4	1.3	7.2	100%	88%
c2670_c	12.4	98%	31%	62%	33.2	1.7	1.7	4.7	100%	76%
c3540_c	3.0	100%	7%	67%	122.4	3.6	3.4	3.8	100%	33%
c5315_c	6.4	97%	16%	100%	20.0	2.7	–	9.1	100%	79%
c7552_c	20.6	64%	20%	50%	23.7	1.9	–	3.5	91%	43%
Average	8.6	93%	16%	80%	29.2	2.0	–	6.4	96%	67%

For each of the above types, five testcases are generated from each benchmark. The proposed algorithm is set to find, at most, 10 transformations for each location identified first by the diagnosis algorithm. Functional verification is carried out at the end to check whether the 10 transformations are valid solutions.

1.6.1 Logic Restructuring of Combinational Designs

The first set of experiments evaluates the proposed methodology for a single location in combinational circuits. Experimental results are summarized in Table 1.2. In this experiment, circuits are simulated with a set of 1000 input vectors that consists of a set of vectors with high stuck-at fault coverage and random-generated vectors.

The first column lists the benchmarks and the types of modifications inserted as described earlier. The second column has the average number of locations returned by the diagnosis program for the five experiments. The percentage of those locations where the error equation approach proves the existence of a solution is shown in the third column. The next two columns show the percentage of locations (out of those in the second column) for which the dictionary-approach and the proposed *a*SPFD approach can successfully find a valid solution. A valid solution is one in which the restructured circuit passes verification. The sixth column contains the average runtime, including the runtime of verification, to find all 10 transformations using greedy heuristics.

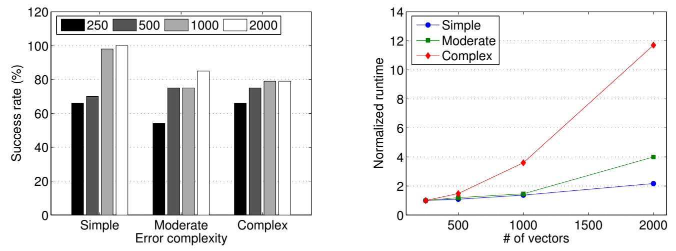
Taking `c1908_s` as an example, there are, on average, 18 locations returned by the diagnosis program. The error equation check returns that 15 (84% of 18) out of those locations can be fixed by re-synthesizing the function of the location. The dictionary approach successfully identifies two locations (13% of 15) while the *a*SPFD approach can restructure 13 locations (84% of 15). This shows that the proposed approach is seven times more effective than the dictionary approach. Overall, the proposed methodology outperforms the dictionary approach in all cases and achieves greater improvement when the modification is complicated.

The quality of the transformations, in terms of the wires involved as well as some algorithm performance metrics, are summarized in column 7 – 11 of Table 1.2. Here, only cases where a valid solution is identified by the proposed algorithm are considered. The seventh and the eighth columns list the average number of additional wires returned by the greedy algorithm and by the SAT-based searching algorithm, respectively. As shown in the table, the greedy heuristic performs well compared to the SAT-based approach. Because the SAT-based approach may run into runtime problems as the number of new wires increases, it times out ("-") after 300 seconds if it does not return with a solution.

As mentioned earlier, the algorithm is set to find, at most, 10 transformations for each location. The ninth column shows the average number of transformations identified for each location. It shows 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. The final two columns show the percentage of transformations that pass verification. The first column of these two columns only considers the first identified transformation, while the second column has the percentage of all 10 transformations that pass verification. One can observe that the vast majority of first-returned transformations pass verification.

Next, the performance of restructuring with various numbers of test vectors is investigated. Four sizes are used: 250, 500, 1000 and 2000 test vectors. The results are depicted in Figure 1.7. Figure 1.7(a) shows the percentage of the locations where the proposed algorithm can identify a valid transformation. As shown, the success rate increases as the size of input vectors increases for each 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 1.7(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. Since the success rate of cases when 1000 vectors are used is close to the success rate of those with 2000 vectors (Figure 1.7(a)), 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.



(a) Solution success rate (b) Runtime profiling

Fig. 1.7 Performance of restructuring with variable vector sizes for combinational designs

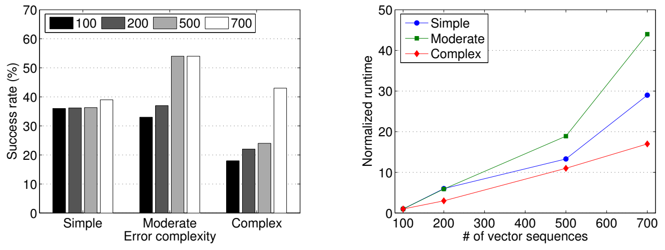
Table 1.3 Sequential logic transformation results for various complexities of modifications

Circ.	Error loc.	Error equat.	*a*SPFD	Avg. time (sec)	Avg # wires	Avg # corr/loc.	% verified First	% verified All	% unique
s510_s	2.4	100%	75%	384	0.3	1.8	100%	92%	100%
s713_s	5.0	72%	0%	325	–	–	–	–	–
s953_s	1.8	100%	33%	223	1.0	3.3	100%	37%	70%
s1196_s	1.8	100%	56%	237	2.0	5.0	83%	92%	64%
s1238_s	1.6	100%	38%	781	1.1	5.0	100%	100%	55%
s1488_s	2.8	86%	43%	258	1.7	5.0	83%	46%	68%
s510_m	2.0	100%	90%	68	0.3	4.2	100%	38%	99%
s713_m	2.8	43%	36%	689	0.6	1.4	100%	41%	60%
s953_m	1.6	63%	40%	105	1.2	1.2	100%	100%	100%
s1196_m	1.2	83%	66%	27	1.8	2.6	100%	72%	83%
s1238_m	2.6	85%	72%	218	2.2	4.3	100%	76%	47%
s1488_m	3.4	100%	0%	83	–	–	–	–	–
s510_c	1.6	100%	38%	166	0.5	1.5	100%	92%	100%
s713_c	3.4	71%	47%	1124	1.0	1.0	100%	100%	75%
s953_c	2.2	73%	0%	122	–	–	–	–	–
s1196_c	2.0	50%	20%	588	0.5	2.3	50%	32%	100%
s1238_c	1.2	100%	14%	328	0	–	100%	–	100%
s1488_c	1.8	71%	30%	98	1.7	1.5	33%	27%	100%
Average	2.1	90%	39%	236	1.0	3.1	92%	68%	82%

1.6.2 Logic Restructuring of Sequential Designs

The second set of the experiments evaluates the performance of logic restructuring with *a*SPFDs in sequential designs. The vector set for sequential circuits contains 500 random input vector sequences with a length of 10 cycles. To verify the correctness of transformations, a bounded sequential equivalent checker [19] is used. This tool verifies the resulting design against the reference within a finite number of cycles, which is set to 20 cycles in our experiment.

The performance of the proposed methodology is recorded in Table 1.3. The benchmarks and the type of the modification inserted are listed in the first column. The second column presents the average number of locations for transformations



(a) Solution success rate (b) Runtime profiling

Fig. 1.8 Performance of restructuring with variable vector sizes for sequential designs

reported by the diagnosis program while the percentage of these locations that are proven to be correctable by error equation are recorded in the third column. The percentage of locations in which the proposed methodology finds a valid transformation is reported in the fourth column, followed by runtime.

Note that the error equation approach in [6] is developed for combinational circuits. Hence, here the sequential circuits are converted into combinational ones by treating the states as pseudo primary inputs/outputs. In this way, the number of locations reported by the error equation approach is the lower bound of the locations that are correctable, since it constrains that the design after restructuring has to be combinationally functional equivalent to the reference design. This constraint discards any solutions that utilize equivalent states. Overall, the proposed approach can restructure 39% of the locations. The reason why the algorithm fails to correct some of the locations is because the input vector sequences 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 vector sequences, or when conflict values are required for this minterm, as discussed in Section 1.4.3. Consequently, the resulting transformation does not distinguish all the necessary minterm pairs that are required to correct the design.

The sixth and the seventh columns report the average number of additional wires used in the transformations and the average number of transformations per location, respectively. Note, because the transformation at some locations only needs to be re-synthesized with the existing fanin nets without any additional wires, cases such as `s510_s` use less than one additional wire on average. The next two columns show the percentage of cases where the first transformation passes verification, and the percentage of 10 transformations that pass verification. Similar to the combinational circuits, there is a high percentage of the first transformations that passes verification if the proposed methodology returns a valid transformation. This indicates that *a*SPFD is a good metric to prune out invalid solutions.

As the result shown in the last column, the valid solutions are further checked for whether or not they are unique to the sequential *a*SPFD-based algorithm. That is, the modified design is not combinationally functional equivalent to the reference design; otherwise, such restructuring can be identified by the combinational approach.

If two designs are not combinationally equivalent, it means that the transformation changes the state assignments as well. Consequently, these transformations will be pruned out by the combinational approach. Overall, 82% of the valid transformations are uniquely identified by the sequential approach. restructuring method.

Finally, the impact of test vector sizes on the performance of the presented methodology is studied. Here, the number of test vector sequences is set to 100, 200, 500 and 700. These test sequences have a length of 10 cycles and are randomly generated. The success rate and the normalized runtime are shown in Figure 1.8(a) and Figure 1.8(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 number of the test sequences decreases. Among the 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 test sequences to fully characterize the erroneous behavior. As a result, the algorithm needs more vector sequences to construct an accurate transformation. Moreover, Figure 1.8(b) shows a significant reduction of the runtime with the decrease of the number of vector sequences.

1.7 Summary

In this chapter, a simulation-based procedure to approximate SPFDs, namely *a*SPFDs, is first presented. An *a*SPFD is an approximation of the original SPFD, as it only contains information that is explored by the simulation vectors. Next, an *a*SPFD-based logic restructuring algorithm for both combinational and sequential designs is presented. This technique can be used for a wide range of applications, such as logic optimization, debugging and applying engineer changes. Experiments demonstrate that *a*SPFDs provide a powerful approach to restructuring a logic design to a new set of specifications. This approach is able to construct required logic transformations algorithmically and restructure designs at a location where other methods fail.

References

1. Abadir, M.S., Ferguson, J., Kirkland, T.E.: Logic verification via test generation. IEEE Trans. on CAD **7**, 138–148 (1988)
2. Bryant, R.E.: On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. IEEE Trans. on Comp. **40**(2), 205–213 (1991)
3. Chang, K.H., Markov, I.L., Bertacco, V.: Fixing design errors with counterexamples and resynthesis. IEEE Trans. on CAD **27**(1), 184 – 188 (2008)
4. Chang, S.C., Cheng, D.I.: Efficient Boolean division and substitution using redundancy addition and removing. IEEE Trans. on CAD **18**(8) (1999)

5. Chang, S.C., Marek-Sadowska, M., Cheng, K.T.: Perturb and simplify: Multi-level Boolean network optimizer. IEEE Trans. on CAD **15**(12), 1494–1504 (1996)
6. Chung, P.Y., Hajj, I.N.: Diagnosis and correction of multiple design errors in digital circuits. IEEE Trans. on VLSI Systems **5**(2), 233–237 (1997)
7. Cong, J., Lin, J.Y., Long, W.: SPFD-based global rewiring. In: Int'l Symposium on FPGAs, pp. 77–84 (2002)
8. E*é*n, N., S*ö*rensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2**, 1–26 (2006)
9. E*é*n, N., Sörensson, N.: An extensible SAT-solver. In: SAT, pp. 502–518 (2003). Http://dblp.uni-trier.de/db/conf/sat/sat2003.html#EenS03
10. Entrena, L., Cheng, K.T.: Combinational and sequential logic optimization by redundancy addition and removal. IEEE Trans. on CAD **14**(7), 909–916 (1995)
11. Jiang, J.H., Brayton, R.K.: On the verification of sequential equivalence. IEEE Trans. on CAD **22**(6), 686 – 697 (2003)
12. Kunz, W., Stoffel, D., Menon, P.R.: Logic optimization and equivalence checking by implication analysis. IEEE Trans. on CAD **16**(3), 266–281 (1997)
13. Lin, C.C., Chen, K.C., Marek-Sadowska, M.: Logic synthesis for engineering change. IEEE Trans. on CAD **18**(3), 282–292 (1999)
14. Ling, A.C., Brown, S.D., Zhu, J., Saparpour, S.: Towards automated ECOs in FPGAs. In: Int'l symposium on field programmable gate arrays, pp. 3–12 (2009)
15. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a new search algorithm for satisfiability. IEEE Trans. on Comp. **48**(5), 506–521 (1999)
16. Mishchenko, A., Chatterjee, S., Brayton, R.K., E*é*n, N.: Improvements to combinational equivalence checking. In: Proc. of Int'l Conf. on CAD, pp. 836 – 843 (2006)
17. Moskewicz, M.W., Madigan, C.F., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Design Automation Conf., pp. 530–535 (2001)
18. Nayak, D., Walker, D.M.H.: Simulation-based design error diagnosis and correction in combinational digital circuits. In: VLSI Test Symp., pp. 70–78 (1999)
19. Safarpour, S., Fey, G., Veneris, A., Drechsler, R.: Utilizing don't care states in SAT-based bounded sequential problems. In: Great Lakes VLSI Symp. (2005)
20. Sinha, S.: SPFDs: A new approach to flexibility in logic synthesis. Ph.D. thesis, University of California, Berkeley (2002)
21. Sinha, S., Brayton, R.K.: Implementation and use of SPFDs in optimizing Boolean networks. In: Proc. of Int'l Conf. on CAD, pp. 103–110 (1998)
22. Sinha, S., Kuehlmann, A., Brayton, R.K.: Sequential SPFDs. In: Proc. of Int'l Conf. on CAD, pp. 84–90 (2001)
23. Smith, A., Veneris, A., Ali, M.F., Viglas, A.: Fault diagnosis and logic debugging using Boolean satisfiability. IEEE Trans. on CAD **24**(10), 1606–1621 (2005)
24. Veneris, A., Abadir, M.S.: Design rewiring using ATPG. IEEE Trans. on CAD **21**(12), 1469–1479 (2002)
25. Veneris, A., Hajj, I.N.: Design error diagnosis and correction via test vector simulation. IEEE Trans. on CAD **18**(12), 1803–1816 (1999)
26. Watanabe, Y., Brayton, R.K.: Incremental synthesis for engineering changes. In: Int'l Conf. on Comp. Design, pp. 40–43 (1991)
27. Yamashita, S., H.Sawada, Nagoya, A.: A new method to express functional permissibilities for LUT based FPGAs and its applications. In: Proc. of Int'l Conf. on CAD, pp. 254–261 (1996)
28. Yang, Y.S., Sinha, S., Veneris, A., Brayton, R.K.: Automating logic rectification by approximate SPFDs. In: Proc. of ASP Design Automation Conf., pp. 402–407 (2007)