Sequential Logic Rectifications with Approximate SPFDs

Yu-Shen Yang[1], Subarna Sinha[2], Andreas Veneris[1], Robert K. Brayton[3], and Duncan Smith[4]

[1]Dept. of ECE, University of Toronto, Toronto, Canada. {yangy, veneris}@eecg.utoronto.ca
[2]Synopsys Inc., Mountain View, United States, Subarna.Sinha@synopsys.com
[3]Dept. of EECS, University of California, Berkeley, United States, brayton@eecs.berkeley.edu
[4]Vennsa Inc., Toronto, Canada, duncan.smith@vennsa.com

ABSTRACT

In the digital VLSI cycle, logic transformations are often required to modify the design to meet different synthesis and optimization goals. Logic transformations on sequential circuits are hard to perform due to the vast underlying solution space. This paper proposes an SPFD-based sequential logic transformation methodology to tackle the problem with no sacrifice on performance. It first presents an efficient approach to construct approximate SPFDs (aSPFDs) for sequential circuits. Then, it demonstrates an algorithm using aSPFDs to perform the desirable sequential logic transformations using both combinational and sequential don't cares. Experimental results show the effectiveness and robustness of the approach.

1. Introduction

In the process of designing VLSI chips, small logic modifications are often needed to re-adjust a netlist for different goals. In design debugging [1] for example, the designer tries to rectify a circuit that failed verification. Once the source of the error(s) is localized, simple logic transformations are implemented on those locations to correct it. Another example is when implementing engineering changes; small specification changes have to be reflected at late stages of the design cycle for a design that has already been synthesized and optimized. In such cases, one seeks to apply a minimal set of modifications to achieve the changes [2]. Rewiring techniques for post-synthesis optimization also rely on logic transformations to improve design performance [3, 4].

In all the cases discussed above, the logic transformations usually involve small changes on a few design nets. A full-blown synthesis step can be a resource intensive process and an unnecessary one, since existing synthesis tools may modify the design significantly, jeopardizing some of the engineering effort invested in it [5]. As such, dedicated automated tools that restructure the design minimally are desired to implement such logic transformations.

An automated logic restructuring methodology for combinational circuits is proposed in [6]. The method proposes a novel approach to construct Sets of Pairs of Functions to be Distinguished (SPFDs) using simulation. SPFDs were introduced by Yamashita et al. [7] and they provide additional degrees of flexibility during synthesis [7–9]. However, they may require prohibitive amounts of memory to manage when used for logic restructuring [9]. The work in [6] alleviates this by using simulation to approximate SPFDs when modifying combinational circuits. Results show that *approximate SPFDs*, or simply *aSPFDs*, provide a memory and run-time efficient alternative.

In this paper, an efficient aSPFD-based sequential logic transformation methodology is proposed. It applies a small set of modifications to the combinational part of sequential circuits to match a desired behavior. The advantage of the proposed methodology is that it avoids exploring the complete state space, which is usually one of the issues when dealing with sequential circuits [9]. The aSPFD allows us to explore only the portion of the state space that is important to carry out the transformation thus alleviating the state space explosion issue.

Nevertheless, generating aSPFDs of nodes in sequential circuits can still be computationally intensive. It may be necessary to translate aSPFDs between various input spaces due to temporal effects from different cycles in the sequential circuit. The proposed methodology presents an approach to construct aSPFDs without the need for state-space expansion. It also takes into account both combinational and sequential don't cares to attain further flexibilities in constructing the transformation. Extensive experiments in this paper demonstrate that aSPFDs can be used to restructure a sequential design with a small set of changes.

The remainder of the paper is organized as follows. Section 2 further motivates the problem, presents notational conventions and gives background information on SPFDs and aSPFDs. The procedure to construct aSPFDs on sequential circuits is discussed in Section 3. The proposed sequential logic restructuring methodology utilizing aSPFDs is presented in Section 4. Experimental results are given in Section 5 followed by the conclusion in Section 6.

2. Background

2.1 Notation

In this work, we consider a sequential circuit with primary input set $\mathcal{X} = (x_1, \cdots, x_m)$, state input set $\mathcal{S} = (s_1, \cdots, s_p)$, and primary output set $O = (o_1, \cdots, o_n)$. Throughout this paper, the superscript of a symbol refers to the cycle of the unrolled circuit. For example, $\mathcal{X}^2$ represents the set of the primary input in the second cycle. x_1^2 refers to the primary input x_1 in the second cycle. We also let symbol T_i denote the i-th simulated cycle. An input vector sequence with k cycles is denoted as $\mathcal{V} = (v^1, \cdots, v^k)$. Values for each vector represent logic values on the primary input only. We assume that initial state values are all at logic 0. However, the method can work for any set of initial state values.

2.2 Motivation and Prior Work

Sequential circuits can be hard to re-structure due to the presence of memory elements. Given a simulation input vector sequence, values of a node may depend on the states of the circuit from previous cycles. This can increase the complexity of the underlying analysis severely. There are two common practices to reduce the sequential temporal effects so that techniques that work on combinational circuits can be also applied onto sequential ones.

The first approach is reducing the sequential behavior of a circuit to a single-copy version of a combinational one by considering the outputs (inputs) of the registers as pseudo-primary inputs (outputs). The issue is that this approach can lead to dramatically sub-optional results because sequential don't cares such as unreachable states or equivalent state are not taken into account [10].

Another common approach is modeling the circuit using the *Iterative Logic Array* (ILA) representation. In this case, one can "unroll" the design over time to maintain the state transition information [10, 11]. The side-effect is that the input vector space of the unrolled circuit grows exponentially, i.e. $O(2^k)$, where k is the number of times the circuit is unrolled. This can become computationally expensive for some methods such as SPFDs since the size of the representing data structures depends on the number of primary inputs.

Previous work on restructuring sequential circuits include [10] and [11]. Sinha et. al. [10] define sequential SPFDs and use them to optimize the sequential state encoding. To avoid the input vector space explosion

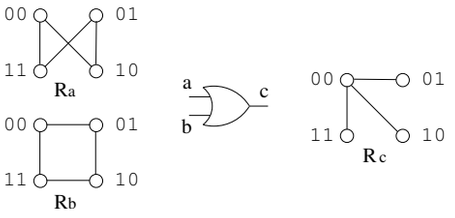


Figure 1: SPFDs for `OR` gate: $R_c \subset \{R_a \cup R_b\}$

described earlier, the method unrolls the circuit incrementally until no more useful information can be extracted. The authors conclude that the size of the input space remains a major challenge for some circuits. A formal technique to re-design a sequential circuit is proposed in [11]. It uses Binary Decision Diagrams (BDDs) [12] to represent the Boolean relation of nodes and the state transitions. Since no experiments are presented, the scalability of the method remains unclear.

2.3 SPFD and aSPFD

SPFDs express the functional flexibility of a design [7]. The SPFD of a node defines pairs of minterms that have to be distinguished by the function of the node. A function distinguishes two minterms if it evaluates to different values under the minterms. The SPFD can be represented as a graph [13], where vertices are minterms and edges connect minterms that need to be distinguished. The graph edges are also referred as the SPFD edges. For example, Figure 1 shows SPFDs of an `OR` gate. R_c, the SPFD of the node c, consists of four vertices (minterms in terms of ab). Since `OR` gates evaluate to 0 only if both inputs are 0, R_c contains edges between 00 and $\{01, 10, 11\}$, Similarly, R_a (R_b) is the SPFD of the node a (b).

Let the SPFD of node l be represented as R_l. SPFDs contain all the information that is necessary to synthesize the fanout network. For each node in a circuit, the SPFD of the node must be a subset of the union of the SPFDs of its fanins. Thus,

$$\cup_{i=1}^{m} R_{l_i} \supseteq R_{l_o}, \tag{1}$$

where node l_o has m fanins $\{l_1, \cdots, l_m\}$. Eq. 1 indicates that the information at a node has to be a subset of all the information at its fanins. For example, in Figure 1, R_c is indeed a subset of $R_a \cup R_b$.

Common approaches of computing SPFDs use BDDs and SAT. SPFD computations with BDDs [12] may need prohibitive amounts of memory for some circuits [9]. Computing SPFDs with SAT is memory efficient, but it can be computationally expensive since every edge in the SPFD of a node needs to be computed.

In [6], SPFDs are approximated using simulation to reduce the memory requirements of their representation and also ease operations on them. The same work also demonstrates the applicability of approximate SPFDs (or aSPFDs) to compute appropriate logic transformations for combinational circuits. Essentially, an aSPFD represents a subset of edges of the original SPFD. The set of edges represented in an aSPFD can be suitably chosen according to the specific synthesis task at hand. Since logic transformations usually involve only a small portion of the circuit, aSPFDs with properly selected SPFD edges can contain enough information to carry out logic re-structuring accurately. After building an aSPFD, the work in [6] searches for a set of support nodes to implement the logic transformation at the netlist location of interest such that Eq. 1 is satisfied. Experiments in [6] demonstrate that using a set of 1000-2000 input vectors to choose the edges is effective for logic transformations in benchmark circuits.

3. Approximating Sequential SPFDs

The act of applying logic transformations is treated as a pair of "error/correction" operations in [6]. We follow this terminology here to ease the presentation. An "error" means a functional discrepancy between the desired specification and the current design where we seek a "correction" (*i.e.,* the transformation itself) that rectifies it. We assume the location of the error, that is, the location of where the transformation should be applied, is known to our algorithm. We refer to such a location as the *transformation node* in the remaining paper.

To transform the current design into the desired specification, a new function that eliminates the observed discrepancy needs to be implemented at the transformation node. It is achieve by deriving an aSPFD at the node such that functions satisfying the aSPFD (i.e. a function that distinguishes all the SPFD edges in the aSPFD) can rectify the discrepancy in the circuit. Let R_{trans} be the new aSPFD. Devising the transformation becomes a task of finding a set of nets that satisfies Eq. 1 for R_{trans}. Then, those nets can serve as the new fan-ins to the transformation node. A fan-in network that satisfies R_{trans} can be synthesized into a two-level `AND-OR` network by the procedure described in [8]. This becomes the new function at the transformation node. Therefore, to successfully apply the transformation, generating R_{trans} that contains the necessary information of the required transformation is a crucial step.

3.1 Constructing Sequential aSPFDs

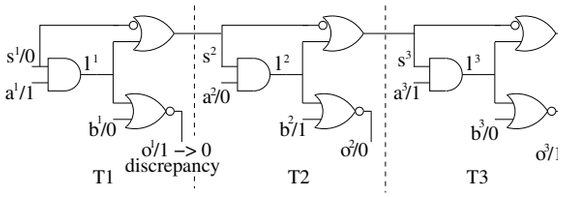
The technique proposed in [6] cannot be used directly to generate aSPFDs for sequential circuits. In a sequential circuit, the value of nets in the circuit at T_i for some input vector sequences is a function of the initial state input and the sequence of the primary input vectors up to and including 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 needs to be distinguished by the aSPFD at the transformation node is different across different cycles. For example, in Figure 2(a), the value of l in T_2 is a function of s^1, a^1 and a^2. Hence, the R_l in T_2 has to distinguish minterms in the space of $\{s^1, a^1, a^2\}$.

The aSPFDs for the same net over different cycles are needed cumulatively to determine the logic transformation that has to be implemented at the erroneous node and they are all equally important. A valid transformation has to satisfy all those aSPFDs. Intuitively, the complexity of logic transformation can be greatly reduced if one "global" aSPFD that integrates all information stored in each individual aSPFD can be constructed. This makes it necessary to convert the aSPFDs into a unified input space. One way to achieve this goal is to translate all aSPFDs into a domain that is the superset of the complete input space, such as the one consisting of the initial state and the primary input from T_1 to T_{last}. Although feasible, this approach may not be computationally desirable as it requires the use of formal techniques (BDDs or SAT) to perform the image computations [9, 13].

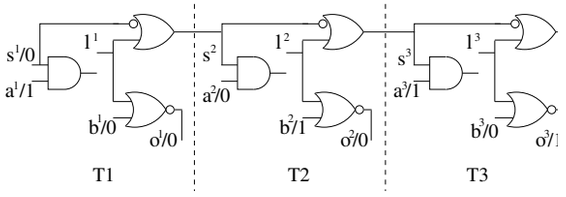
Instead of translating aSPFDs into the largest input space, our approach generates aSPFDs over the input space $\{\mathcal{S} \cup \mathcal{X}\}$. Although this might results in treating some of the sequential behavior as combinational behavior, it still offers more flexibility when compared to treating a sequential circuit as a pure combinational one. We determine the values of the state in each cycle for the given input vector sequences and generate a partially specified truth table of the new function that the transformation should obey. The truth table specifies the logic values for the new function under the simulated input vector sequences where minterm variables are taken from the set $\{\mathcal{S} \cup \mathcal{X}\}$. The complete procedure of this step is further discussed in Section 3.2. Once the truth table is determined, the aSPFDs of the node can be determined by identifying its onset and offset sets. Let $On(l)$ ($Off(l)$) denote the onset (offset) of the node l. Then, R_l necessitates that minterms in $On(l)$ have to be distinguished from minterms in $Off(l)$. That is, R_l contains an edge for each pair $(a,b) \in \{On(l) \times Off(l)\}$

Recall that the aSPFD is an approximation of the SPFD. For a combinational circuit, it only explores the portion of the input space covered by the given input vectors. Similarly, aSPFDs for sequential circuits only consider states that are reachable during simulation of the given input vector sequences. Hence, in our approach, only reachable states are considered. By simulating input vector sequences, unreachable states are pruned out by construction reducing the state space analyzed.

EXAMPLE 1. *Figure 2(a) depicts a sequential circuit unrolled for three cycles under simulation of a single input vector sequence. Assume*



(a) Unrolled circuit



(b) Unrolled circuit with l as a primary input

Figure 2: The circuit for Examples 1 and 2

the correct response at o^1 should be 0 *and net l is the transformation node. One can verify that if the values of l are* 110 *at cycles $T_1 \cdots T_3$, the discrepancy at o can be resolved, i.e. $o^1 = 0$. This gives the truth table of l (in terms of abs) stating that l evaluates to* 1 *under minterms* $\{100, 011\}$ *and to* 0 *under* $\{101\}$. *We conclude that* $On(l) = \{100, 011\}$ *and* $Off(l) = \{101\}$. *The aSPFD of l contains two edges:* $(100, 101)$ *and* $(011, 101)$.

3.2 Generating the Truth Table

In this section the procedure to generate the partial truth table of the new function at the transformation node is described. Note, the truth table contains only the values of the function under the minterms explored during simulation of the input vector sequences. This partial truth table is used to compute the value of the new function.

DEFINITION 1. *Given an input vector sequence, $\mathcal{V}$, the* ***expected trace*** *of a node l is a sequence of values the function of l should evaluate for $\mathcal{V}$ after the transformation is implemented.*

In other words, an expected trace defines the value of internal states and the function of the transformation. It can be extracted by formulating the circuit into a Boolean SAT problem and solving the problem with SAT solvers. The formulation is similar to the one presented in [1]. In summary, given a circuit, the input vectors $\mathcal{V}$ and the output responses $\mathcal{Y}$, the solver tries to find an assignment to all the nodes in the circuit such that the values of the output of the circuit match $\mathcal{Y}$ under $\mathcal{V}$. $\mathcal{V}$ and $\mathcal{Y}$ serve as the constraints to the SAT instance. An all-solution SAT solver can return all possible assignments that satisfy the instance.

The procedure GENERATETRUTHTABLE in Algorithm 1 gives the pseudo-code to calculate the expected trace and the truth table of transformation node l. The basic idea is to make l a new primary input (line 1.6) and let the SAT solver determine the value at l in each cycle such that the SAT instance is satisfied.

LEMMA 1. *Let C be a sequential circuit and $l \in C$ denotes a logic transformation node candidate. If the values of the primary input $\mathcal{X}^i$ equal the values of $\mathcal{X}^j$ for any two cycles, (T_i, T_j), and the values of the state $\mathcal{S}^i$ equal the values of $\mathcal{S}^j$, the value of l^i and l^j are the same.*

Proof: Nodes in a sequential circuit can be expressed as a many-to-one function in terms of the primary input $(\mathcal{X})$ and the state elements $(\mathcal{S})$, $f : \{\{0,1\}^{|\mathcal{X}|+|\mathcal{S}|} \rightarrow \{0,1\}\}$. Suppose, towards a contradiction, the lemma is not true. Then, there exists an input assignment $\hat{I} \in \{0,1\}^{|\mathcal{X}|+|\mathcal{S}|}$ such that $f_l(\hat{I}) = 0$ at T_i and $f_l(\hat{I}) = 1$ at T_j. Since the logic transformations discussed here do not add/remove a state input, the input space is not changed after the transformation. Such one-to-many mappings of f_l are incorrect and the values of l^i and l^j have to be the same. □

Algorithm 1 Computing the truth table of the transformation node

```
1: C :=  the unrolled sequential circuit
2: l :=  the transformation node
3: V :=  the input vector sequence
4: Y :=  the expected output response
5: procedure GENERATETRUTHTABLE(C, l, V, Y)
6:     Make l as a primary input in C
7:     k ← the length of V
8:     φ ← CONVERTCIRCUITTOCNF(C, V, Y)
9:     for all (i, j) | 0 < i, j < k, i ≠ j do
10:        if v^i == v^j then
11:            add constraints to φ that force l^i = l^j if the state input
                   have the same values at T_i and T_j.
12:        end if
13:    end for
14:    {min, val} ← SOLVEWITHSATSOLVER (φ)
15: end procedure
```

In order to comply with Lemma 1, additional constraints are added to the SAT instance. For every pair of (T_i, T_j) when the primary input vectors are the same, a checker that ensures l^i and l^j have the same value when the states at these two cycles are the same is added to the SAT instance (line 1.11). If the SAT solver can find a valid assignment satisfying the resulting SAT instance, the values assigned to l is the desired expected trace that can be used to construct the aSPFD.

For the given input sequences and the expected output responses, there can exist multiple expected traces that satisfy the SAT instance. Those traces may cause conflicts in the new truth table at the transformation node, i.e. different logic values are assigned for the same minterm. In the following discussion, we explain the cases under which such a conflict can arise. Let $\hat{E_1}$ and $\hat{E_2}$ represent two expected traces returned by the SAT solver. Assume a conflict occurs for minterm M between the assignment to $\hat{E_1}$ at cycle T_i and the assignment to $\hat{E_2}$ at cycle T_j. Then, one of the two cases below is true.

- The output responses and the states at cycle $T_i + 1$ for $\hat{E_1}$ and $T_j + 1$ for $\hat{E_2}$ are the same. This implies that the value of the transformation node under M does not affect the output response or the next state. Hence, M is a *combinational don't care*. Identifying these don't cares is desirable as it can reduce the number of minterm pairs required to be distinguished in aSPFDs.

- The output responses and/or the next states are different. This can happen when the circuit has multiple state transition paths for the same initial transitions. Since our analysis is bounded by the length of the input vector sequences, it may not process enough cycles to differentiate those different paths. As the result, multiple assignments are valid within the bounded cycle range and they can cause conflicting assignments to the transformation node. Since the algorithm does not have enough information to distinguish the correct assignment, we consider the minterm M in this case to be a don't care as well.

Figure 3 shows an example of a scenario where the second case outlined above can arise. Let l be the transformation node. The graph defines a state transition diagram for a single-output design that depends on the value of l. The condition for the state transition is labeled on the edge and the value of the output is indicated below the state. Assume the design starts at S_0. It takes at least three cycles to differentiate the transition $Path_a$ and $Path_b$. If the circuit is only unrolled for two cycles, both paths will seem the same. Consequently, l can be 0 or 1 in S_0.

As evident from the discussion above, it will be necessary to consider all the expected traces returned by the SAT solver to generate the truth

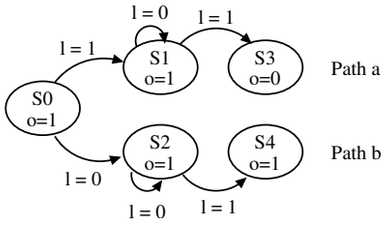


Figure 3: State transition

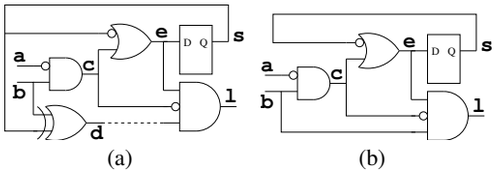


Figure 4: (a) Original design C_{orig}. (b) Corrected design after removing the wire (d,l)

Table 1: Test vector sequences

Vector	v_1	v_2	v_3
Pattern (a,b)	10,01,00	11,00,10	01,00,11

Table 2: Truth table of l

minterms	000	001	010	100	110	111
l	0	0	1	0	1	1

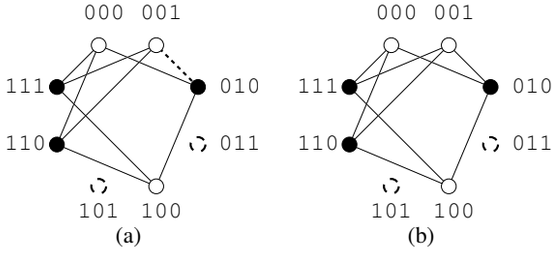


Figure 5: (a) Required $aSPFD$ for l in Figure 4(a). (b) aSPFD for b

table for the transformation. To perform this task effectively, the line 14 in Algorithm 1 has to be extended with the following steps:

1. Obtain all solutions to the SAT instance with the SAT solver.
2. Extract minterms in the $\{\mathcal{S} \cup \mathcal{X}\}$ space and expected traces from the solutions.
3. Update the truth table of the transformation node. Minterms at cycles where values of expected traces are conflicted are considered as don't cares.

EXAMPLE 2. *Returning to Example 1, here we show how to generate the expected trace of l. The first step is to make l into a primary input as shown in Figure 2(b) and then construct a SAT instance representing the design. Values shown next to the primary output are the respective correct output response. Because the set of primary input (a,b) has the value $(1,0)$ at T_1 and T_3, we add constraints to the SAT instance such that if s^1 and s^3 have the same value, lines l^1 and l^3 must have the same value as well. The value* 110 *is returned by the SAT solver as the expected trace of l.*

One can verify that without the additional constraints, the assignment 100 *to (l^1,l^2,l^3) can also satisfy the given output response. However, in this case, abs at T_1 and T_3 are both* 100*, but l^1 and l^3 have opposite values. This is not a valid expected trace since it can cause a conflicting assignment in the truth table of l. It should be noted that don't cares can only arise due to conflicts across different expected traces.*

4. Logic Changes with Sequential aSPFDs

In this section, we present an aSPFD-based technique to re-structure the combinational circuitry of sequential designs. It uses input vector sequences to generate aSPFDs (as described in the previous section) that guide synthesis for the resulting transformation.

We explain the complete sequential transformation procedure by presenting an example. Given a sequential circuit, C_{orig}, as shown in Figure 4(a), assume the goal is to remove wire (d,l) (the dotted line). Let C_{mod} denote the circuit where wire (d,l) is removed. Three input vector sequences that detect the "error" in C_{mod} are listed in Table 1, each of which comprises of three cycles [3]. The initial value of the state s is set to 0. Assume that l is the transformation node. The procedure to perform an appropriate logic transformation at l is as follows:

1. Call GENERATETRUTHTABLE in Algorithm 1 to obtain the truth table of the new function at l. The table is shown in Table 2.
2. Construct the aSPFD of the new function at l using the truth table. According to the table, $Off(l) = \{000,001,100\}$ and $On(l) = \{010,110,111\}$. R_l contains six minterms and those in $On(l)$ have to be distinguished from minterms in $Off(l)$ (Figure 5(a)). The black (white) nodes indicate that l has a logic value $1(0)$ when the labeled minterm is applied. The dotted nodes refer to don't cares or minterms that are not explored by the input vectors.
3. Construct the aSPFDs of remaining nets in the design via simulation. The circuit is simulated with minterms and the values of l defined in the truth table for one cycle. This allows us to obtain the truth table of the nets after the transformation. Then, the onset/offset of each remaining net can be determined and used to construct its aSPFD.
4. Remove edges of R_l that are included in the aSPFDs of fanins of l. According to Eq. 1, the remaining edges are missing information for l in C_{mod}. In our example only the edge $(001,010)$ (the dotted line in the figure) is not a part of aSPFDs of fanins of l.
5. Search for additional wires such that Eq. 1 is fulfilled. The greedy-based procedure similar to the one in [6] is used. The algorithm iteratively selects a wire whose aSPFD contains most SPFD edges of R_l until all edges are covered. Here, the SPFD edge $(001,010)$ is contained in R_b (Figure 5(b)). Thus, b is qualified as the additional input to l. The resulting design is shown in Figure 4(b).

It should be noted that, just like in combinational aSPFDs [6], the penalty of using aSPFD to re-structure the design is that the resulting circuit needs to be verified. This is because an aSPFD does not contain all minterm pairs that it needs to distinguish but only those excited by the input vector sequence. Hence, using aSPFDs for logic transformations necessitates a verification step after the transformation. Since the structural changes are incremental, a full blown verification step may not be required and other faster proof methods can be used [3, 14].

Because the algorithm selects minterms based on the given input vector sequences, it may characterize a minterm as a don't care if it is not exercised by those sequences. In such a case, the proposed approach may fail to find a logic transformation that passes verification. Since the verification tool will return a counter-example, this can be added to the input vector sequences and the algorithm can be re-executed to find new transformation(s).

On the other hand, the input vector sequences may not provide all the information for the don't care space of the design. In this case, the algorithm will accidentally specify a well defined logic value for a

Table 3: Sequential logic transformation results for different complexities of modification

circuit	total	error.	succ	hit	first solution				multiple solution			
name	loc	equat.	loc.	rate (%)	avg # wires	avg sec	% verified	% unique	avg # sol.	avg # wire	% verified	% unique
s510_s	12	12	9	75	0.3	384	100	100	1.8	0.6	92	100
s713_s	25	18	0	0	-	325	-	-	-	-	-	-
s953_s	9	9	3	33	1	223	100	67	3.3	1.5	37	70
s1196_s	9	9	5	56	2	237	83	66	5	1.5	92	64
s1238_s	8	8	3	38	1.1	781	100	42	5	1.6	100	55
s1488_s	14	14	6	43	1.7	258	83	75	5	1.3	46	68
s510_m	10	10	9	90	0.3	68	100	100	4.2	2.0	38	99
s713_m	14	6	5	83.3	0.6	689	100	63	1.4	1.6	41	60
s953_m	8	5	2	40	1.2	105	100	100	1	1.2	100	100
s1196_m	6	5	4	80	1.8	27	100	83	2.6	2.5	72	83
s1238_m	13	11	8	72	2.2	218	100	38	4.3	2.5	76	47
s1488_m	17	17	0	0	-	83	-	-	-	-	-	-
s510_c	8	8	3	38	0.5	166	100	100	1.5	0.7	92	100
s713_c	17	12	8	67	1	1124	100	75	1	1	100	75
s953_c	11	8	0	0	-	122	-	-	-	-	-	-
s1196_c	10	5	2	40	0.5	588	50	100	1.2	0.7	32	100
s1238_c	7	7	1	14	0	328	100	100	-	-	-	-
s1488_c	14	10	3	30	1.7	98	33	100	1.5	2.6	27	100
Overall	206	168	64	38	1.0	319	90	80	2.7	1.5	67	76

minterm which is essentially a don't care. The following lemma shows that in those cases, there is no loss in the final resolution of the solution.

Our experimental results indicate that our methods derives transformations using sequential aSPFDs that most of them pass verification, thereby proving the efficacy of the proposed approach.

LEMMA 2. *Given two aSPFD, R_i and R_j. Let $R_i \subset R_j$. Any functions that satisfy R_j also satisfy R_i.*

Proof: Suppose, toward a contradiction, that this is not true. Then, there exists a function, F, that satisfies R_j, but it does not satisfy R_i. This implies that F fails to distinguish at least one minterm pair required by R_i. Because R_i is a subset of R_j, such minterm pair must exist in R_j as well. Thus, F cannot satisfy R_j. By contradiction, all functions that satisfy R_j have to satisfy R_i. □

In our approach, the aSPFDs may contain minterm pairs that are not required to be distinguished due to lack of information from the input vectors. According to Lemma 2, the transformations returned can still correct the design.

5. Experiments

This section presents the empirical results using ISCAS'89 benchmarks. The diagnosis algorithm from [1] is used to identify the location of the transformation node and zChaff [15] is the underlying SAT solver. Experiments are conducted on a Core 2 Duo 2.4GHz processor with 4GB of memory. Runtimes are reported in seconds.

In our experimental setup, three different complexities of modifications are injected in the original benchmark. Experiments involve correcting those changes to test the performance of our algorithm. The location 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 complexity modifications (suffix "m") on a gate include additions/deletions of multiple fanin wires and a gate type change. The final complexity modifications, complex (suffix "c"), inject multiple simple complexity modifications in the fanin cone of a gate.

In the first set of the experiment, 500 input vector sequences, each of which with a length of 10 cycles, are used. A bounded sequential equivalent checking [16] is used to verify the correctness of the transformation. It verifies the resulting design against the original design within a finite number of cycles. In our experiment, this bound is set to 10 cycles. If the transformation fails, the process is iterated for a maximum of 30 times. Results are summarized in Table 3.

The name of the original benchmark and the complexity of the modification injected are shown in Column 1. The overall result is summarized in the last row of the table. Numbers in Column 2-5 are the sum of results from 5 experiments/circuit, while those in the remaining columns are averages. The number of transformation locations returned by the diagnosis are reported in Column 2. The formal method in [17] is utilized to answer with certainty whether there exists a modification that corrects the design at those locations. In detail, that method does not return the actual transformation but indicates whether some amount of transformation may or may not correct the design at the particular location. These results are reported in Column 3 and they are used as the metric to measure the effectiveness of the proposed methodology.

The next column shows the total number of locations where a valid transformation is identified. A valid transformation is the one that passes verification. We also define the correction hit rate as the ratio of the number of locations with a valid transformation to the total number of locations. The hit rate is recorded in Column 5. Taking s510_s as an example, there are 12 locations can be corrected in the five experiments; our algorithm can find a valid transformation for nine of them. The hit rate in this case is 75% (9 out of 12).

From the table, one can see that our algorithm is able to correct 14 out of 18 test cases. For each individual case, the correction hit rate can go up to 83%. Overall, our algorithm successfully identifies transformations for 38% 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 aSPFD. As discussed in Section 4, 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.

Columns 6-9 summarize the results when a transformation that passes verification is found. Column 6 contains the average of the number of additional wires selected to construct the transformation. Overall the number of additional wires is limited to three, which suggests that the transformations only alter the design with small changes, a desirable characteristic. One may notice that the number is less than one in some cases. This is because there are cases where no additional wire is required to construct the transformation. The average runtime is recorded in Column 7. Column 8 shows the percentage of transformations that pass verification when the first valid transformation is returned by the algorithm. In most cases, the first transformation is a valid solution. This shows that aSPFDs is a good metric to prune out invalid solutions. Then, those transformations are checked whether they could be the solutions of the approach proposed in [6]. This is carried out by performing combinational equivalent checking between the transformed circuit and the reference. Note that transformations identified by [6] must main-

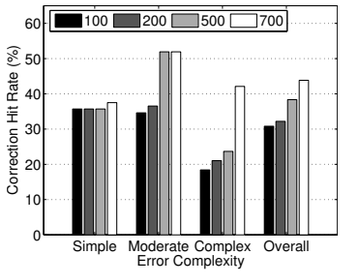


Figure 6: Solution hit rate

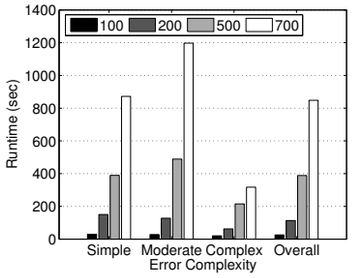


Figure 7: Runtime profiling

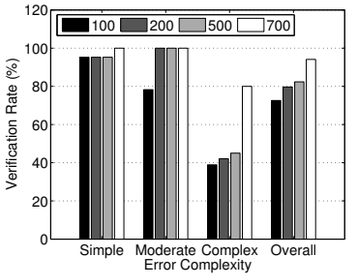


Figure 8: First hit rate

tain the state assignments, since the method treats registers as pseudo-primary inputs/outputs. If two designs are not combinational equivalent, it means that the transformation changes the state assignments. Therefore, the transformation will be pruned out by [6]. Overall, 80% of the valid transformations are uniquely identified by the proposed methodology as shown in Column 8.

Next, the algorithm is extended to find at most five transformations that pass verification. Column 10 contains the average number of valid transformations and the next column has the average number of additional wires selected for the new logic. In most cases, the algorithm is not able to find more transformations that pass verification, a fact that indicates the challenge of the problem. Finally, the percentage of transformations that pass verification and the percentage of transformations that are uniquely identified by the proposed methodology are shown in the last two columns.

In the final set of the experiments, we investigate the performance of the transformation with various sizes of the input vector sequences. We use four sizes: 100, 200, 500 and 700. All have a length of 10 cycles and the algorithm searches for one valid transformation. Figure 6 shows the correction hit rate for varies vector sizes. One can see that the hit rate is increased as the size of input vectors increases. This is expected since more vectors provide more information for the aSPFD and the chance that the algorithm incorrectly characterizes a minterm as a don't care is reduced. We also see that for all types of error complexity, the hit rate of the cases where 200 vectors are used is close to the rate of those with 500 vectors. This suggests that small number of input vectors is sufficient for an acceptable solution.

Although using a larger set of input vectors can improve the correction hit rate, the penalty is that the algorithm needs to utilize more computational resources to analyze the problem. The average runtime of each case is depicted in Figure 7. It shows a two-fold increase overall when the size of the vector set increases from 200 to 500 vectors. Finally, Figure 8 shows the percentage of transformation that passes verification. We see that cases with 100 input vectors have the lowest rate. This is because with fewer vectors the aSPFD contains fewer minterm pairs to be distinguished. As a result, there are more nets that can satisfy Eq. 1 to construct the transformation. However, most of them do not pass verification. Overall, with 200 or more input vectors, the algorithm is able to find a transformation that passes verification by the first few iterations.

6. Conclusion

A simulation-based aSPFD-driven algorithm for sequential logic transformation is presented. The algorithm utilizes aSPFDs to efficiently construct transformations. A procedure to construct aSPFDs for sequential circuits is also outlined in this work. It alleviates the input space explosion and computation issue by using simulation to select states that are important to restructuring construction. Experimental results demonstrate the performance of the proposed work. The algorithm is able to find transformations for up to 83% of the testcases. An analysis also shows that the correction hit rate can be improved if more input vectors are provided.

7. 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] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] M. H. Moshewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engierring an efficient SAT solver," in *Design Automation Conf.*, June 2001, pp. 530–535.

[16] S. Safarpour, G. Fey, A. Veneris, and R. Drechsler, "Utilizing don't care states in SAT-based bounded sequential problems," in *IEEE Great Lakes VLSI Symposium*, 2005.

[17] 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.