Multi-level Synthesis for Safe Replaceability

Carl Pixley
Motorola Inc., MD OE321
6501 Wm Cannon Drive West
Austin, TX 78735

Vigyan Singhal[*] **Adnan Aziz**[†] **Robert K. Brayton**
Dept. of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Abstract

We describe the condition that a sequential digital design is a safe replacement for an existing design without making any assumptions about a known initial state of the design or about its environment. We formulate a safe replacement condition which guarantees that if an original design is replaced by a new design, the interacting environment cannot detect the change by observing the input-output behavior of the new design; conversely, if a replacement design does not satisfy our condition an environment can potentially detect the replacement (in this sense the replacement is potentially unsafe). Our condition allows simplification of the state transition diagram of an original design. We use the safe replacement condition to derive a sequential resynthesis method for area reduction of gate-level designs. We have implemented our resynthesis algorithm and we report experimental results.

1 Introduction

We are concerned with the problem of sequential resynthesis for gate-level synchronous, sequential designs. We start with a given design and replace it with a modified design so that an environment around the original design cannot detect the replacement by observing the input-output behavior of the design. We want to make no assumptions about the environment. The state of a sequential design is captured the values of the latches in the design. We will not make any assumption about a known initial state of the sequential circuit. It is here that we differ from most previous research in sequential synthesis of circuits.

In many industrial-level designs many latches (or flip-flops) do not have a reset line. While it is well accepted that this statement is true in the data part of the designs, it is our experience that even in the control part many latches do not have a reset line. Avoiding routing reset lines yields significant gain in area, and is an important reason why latches may not have reset lines. Also, latches without reset lines cost less (in number of transistors required) that those with reset lines. While most designs described in a hardware description language may have a specified initial state, many gate-level designs do have latches without a reset line. Even if latches have a reset line, the reset line may be an output of derived logic (with possibly other latches in the transitive fan-in) or different latches may have different reset lines (thus, the initial state of the design is not uniquely determined). Also, for many designs it is true that the input/output behavior of the design, before the reset line is activated, is important.

We would like to replace the design with another without making any assumptions about the interacting environment and the state the design can power up in. We will assume that no latches have reset lines; designs where some of the latches have reset lines can easily be modeled by merely treating the reset line as another input.

Many researchers have been able to obtain and exploit sequential flexibility in gate-level designs by using the knowledge of the designated start state. Some of these include using don't care resulting from unreachable states [1], redundant latch removal [2], sequential redundancy removal [3] and equivalence net detection [4]. All these methods rely on the flexibility introduced because many states in the design are not reachable from the start state, and hence we are free to modify the behavior of any unreachable state arbitrarily. However, if latches do not have reset lines, *all* states are reachable, and methods which rely on unreachable states cannot provide any more flexibility than the regular combinational flexibility afforded by the network. In this paper, we use our replaceability notion to obtain area reductions without assuming a designated start state.

We describe our condition for safe replacement and synthesis techniques which does not assume reset lines. We describe how our condition differs from other notions used to describe sequential equivalence [5, 6]. We also discuss how other sequential resynthesis methods, like retiming/resynthesis [7] and synchronous relation minimization [8], which do not directly use the knowledge of a designated start state indirectly rely on the existence of an initial state. We will also show that the synthesis techniques in [9] make implicit assumptions about the environment of the design. For our safe replacement condition, it is surprising even though the design may power up in any state that we are able to obtain some area reductions beyond combinational resynthesis. Furthermore, it is not necessary to preserve the underlying state transition graph of the design (as in the state re-encoding problem).

In Section 2, we provide the basic definitions and terminology for this paper. Section 3 presents the previous work on sequential equivalence and motivates why we need a stronger equivalence condition. Sections 4 and 5 present "safe replaceability" and how we use this for multi-level sequential resynthesis. Finally, we conclude with some initial experimental experience.

2 Terminology and Background

We now make precise the notion of a finite state machine and our model for sequential hardware. We also define classical notions of equivalence for states in a machine, and for machines.

Definition 1 *A* deterministic Finite State Machine (DFSM) M *is a quintuple,* $(Q, I, O, \lambda, \delta)$*, where* Q *is the set of states,* I *is the set of input values,* O *is the set of output values,* λ *is the output function, and* δ *is the next state function. The output function* λ *is a completely-specified function with domain* $(Q \times I)$ *and range* O*. The next state function is a completely-specified function with domain* $(Q \times I)$ *and range* Q*.*

A hardware *design* D consists of a set of interconnected latches and gates, as illustrated in Figure 1. For the purposes of this paper, a design with n input wires, m output wires and t latches

[*] Research supported by NSF/DARPA Grant MIP-8719546 and a summer internship from Motorola, Inc.
[†] Research supported by SRC Grant 94-DC-324

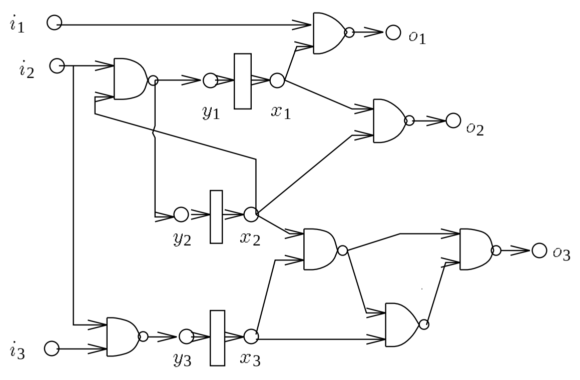

Figure 1: Gates + Latches = Sequential Network

is characterized by an associated DFSM with state space $Q_D = \{0,1\}^t$, input space $I = \{0,1\}^n$, and output space $O = \{0,1\}^m$; the next state and output functions are defined by the corresponding logic. I^* refers to the set of all finite input sequences.

We also use λ and δ to denote the output and next state functions on sequences of inputs. So, if $\pi = a_1 \cdot a_2 \cdot a_3 \cdots a_p \in I^p$ is a sequence of p inputs, these functions are recursively defined as $\lambda(s,\pi) = \lambda(s,a_1) \cdot \lambda(\delta(s,a_1),\pi')$ and $\delta(s,\pi) = \delta(\delta(s,a_1),\pi')$, where $\pi' = a_2 \cdot a_3 \cdots a_p$. Thus, the range-domain relationships are $\lambda : Q \times I^p \rightarrow O^p$ and $\delta : Q \times I^p \rightarrow Q$.

Two designs are said to be *compatible* if they have the same number of input and output wires. All notions of equivalence and replaceability developed in this paper are meaningful only for pairs of compatible designs. Henceforth, when talking about two different designs compatibility is assumed.

Definition 2 *Given a design D_0, and states $s_0 \in Q_{D_0}$ and $s_1 \in Q_{D_1}$, state s_0 is* equivalent *to state s_1 ($s_0 \sim s_1$) if for any sequence of inputs $\pi \in I^*$, $\lambda_{D_0}(s_0,\pi) = \lambda_{D_1}(s_1,\pi)$. It can be easily shown that if $s_0 \sim s_1$, then for any input sequence $\pi \in I^*$, $\delta_{D_0}(s_0,\pi) \sim \delta_{D_1}(s_1,\pi)$.*

The classical notion of equivalence between two DFSM's [10, page 23] is the following:

Definition 3 *Two DFSM's M_1 and M_2 are* equivalent *($M_1 \equiv M_2$) if for each state s in M_1 there is a state t in M_2 such that $s \sim t$, and for each state t in M_2 there is a state s in M_1 such that $s \sim t$.*

3 Previous Work

In this section we describe the few known notions of sequential equivalence for circuits which do not have reset lines, and argue why these might cause unsafe replacements in some cases.

3.1 Sequential Hardware Equivalence (SHE)

Here we will briefly review the work presented in [5] regarding equivalence between two gate-level hardware designs. When the design powers up, the state it powers up in cannot be predicted, and the desired input/output behavior is achieved from the design by driving a fixed initializing sequence of input vectors through the design after power-up.

Definition 4 *Given a design D_0, a sequence of inputs $\pi \in I^*$ is called a* initializing sequence *if for any pair of states $s_0, s_1 \in Q_{D_0}$, $\delta_{D_0}(s_0,\pi) \sim \delta_{D_0}(s_1,\pi)$. A design which has an initializing sequence is called* initializable.

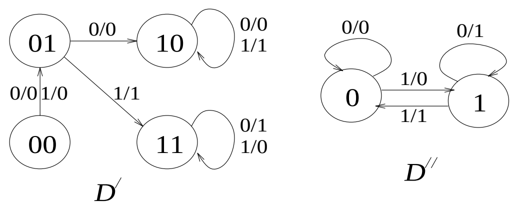

Figure 2: Designs which do not have any initializing sequences

Definition 5 *Given two designs D_0 and D_1, a state pair $(s_0,s_1) \in Q_{D_0} \times Q_{D_1}$ is* alignable *if there is a sequence of inputs $\pi \in I^*$ such that $\delta_{D_0}(\pi,s_0) \sim \delta_{D_1}(\pi,s_1)$. The sequence π is called an* aligning sequence.

The following definition defines the notion of sequential hardware equivalence.

Definition 6 *Designs D_0 and D_1 are* equivalent *($D_0 \approx D_1$) if all state pairs are alignable.*

Theorem 3.1 *$D_0 \approx D_1$ if and only if there is a $\underline{\text{single}}$ aligning sequence that aligns $\underline{\text{all}}$ state pairs in $Q_{D_0} \times Q_{D_1}$.*

Now we argue why the notion of SHE does not work for safe replacement of sequential designs.

From Theorem 3.1, two designs are considered equivalent if there exists a universal aligning sequence. This sequence is an initializing sequence for either design. However, in the design process, often the designers do not (or, can not [11]) know the initializing sequence for their designs. Even if they can determine such a sequence π for a design, it may not be possible for the environment to generate π. So, for a safe replacement we need to preserve $\underline{\text{all}}$ initializing sequences, and not just one. In that case the one used being used by the environment will be preserved.

The notion of SHE does not place any constraints on the outputs of the designs during the initialization phase. However, we claim that this condition is too weak for a safe replacement. *A priori*, we cannot assume that the external environment is not sensitive to the outputs during the initialization phase. This is especially important because there may be another interacting design whose initializing sequence may be driven by an output of design D_0. Thus affecting the outputs of D_0 during initialization may destroy that initializing sequence.

Finally, the notion of SHE does not work for designs which are not initializable (such a design is not even equivalent to itself because it does not have an aligning sequence with itself). For example, the design D' in Figure[1] 2 is not equivalent to itself because the state pair $(10, 11)$ is not alignable. However, we can imagine at least two classes of real designs which are not initializable. First, if the environment has some flexibility for the input/output behavior it can accept from the design, the design may have multiple steady-state behaviors (for example, design D' in Figure 2). In this example, the environment has a don't care condition so that the design is acceptable as long as it always toggles the input (state 11) or always outputs the input (state 10), after the initialization phase. For the second class, consider the design D'' in Figure 2. It can be seen that there is no initializing sequence for this design, and hence

[1]We frequently represent designs by state transition graphs (STG's). A t-bit binary-valued label on a state denotes that, in the design, the state is implemented by that assignment of the t latches. Notice that because a combinational function can be implemented in many different ways, the design-to-STG transformation is a many-to-one mapping.

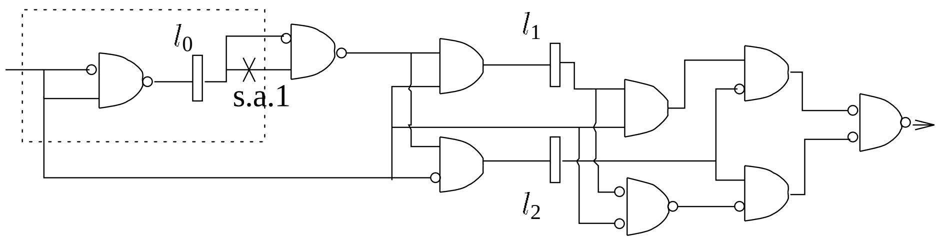

Figure 3: An irredundant stuck-at-fault for a circuit is redundant for a sub-circuit

this design is not initializable. However, once the design powers up, its state can be determined from its outputs, and based on the outputs the design can be driven to state 0. Thus, the behavior of this design can be controlled.

3.2 Redundancy Removal

Here we briefly describe the sequential equivalence condition used by Cheng [6] for resynthesis of circuits by removing redundant lines from the circuits. The basic idea is to check if the input/output "behavior" of the circuit is acceptable even after an internal line has been set to 0 or 1. If so, then the line can be replaced by a constant, and the circuit simplified.

Definition 7 *A fault is* sequentially redundant *if for any input sequence, any output line and any state of the faulty circuit D_1, the circuit D_1 produces 1 (0) whenever the original circuit D_0 produces 1 (0) from all states of D_0. If D_0 produces an unknown output U on some input sequence (i.e. 1 from at least one power-up state and 0 from at least another) then D_1 is allowed to produce either 0, 1 or U on that input sequence.*

If a fault is sequentially redundant, the circuit may be replaced by the faulty circuit, thereby simplifying the design. While the condition in Definition 7 makes sense for resynthesis of a single machine in isolation (because the new design is restricted to produce the same output as the original design if the output is deterministic), in hierarchical resynthesis, the condition can cause unacceptable (unsafe) replacements.

Consider the gate-level design shown in Figure 3. This circuit produces a 1 if and only if the inputs over the last two clock cycles are identical (the output on the first cycle is arbitrary). Suppose we select a window in the design (shown by the circuit inside the dotted rectangle) and resynthesize this sub-circuit. It is easily seen that the stuck-at-1 fault is sequentially redundant for this sub-circuit. However, for the faulty circuit, if the design of Figure 3 powers up in state $(l_0 = 0, l_1 = 0, l_2 = 0)$ and input sequence $1 \cdot 1$ is provided, it produces a 0 at the second clock cycle (the initial design outputs a 1). This motivates the need for a safe replacement condition— a condition so that the environment does not see any new input/output behavior after the replacement. Notice that if the entire design in Figure 3 were considered, the shown fault would no longer be redundant as per the condition in [6]. This points to the desirable compositionality property that we would like to see in a replacement condition— safe replacements for a sub-design should still be safe replacements when the sub-design is composed with another design.

The work in [9] is based on the replacement condition in [6] assuming that if the fault cannot be propagated to a primary output, then the fault is redundant. For the example which we just showed, the fault cannot be propagated to any single output of the circuit inside the window; so the fault might be considered redundant if the window is looked at in isolation. However, if the given environment is considered, the fault is not redundant since it can be propagated to the primary output of the entire design in Figure 3. Notice that in the above example, the fault can be propagated to *a set* of primary outputs but to no single primary output. We can also construct a similar example where the fault cannot be propagated to any set of primary outputs at a single time frame, but to a set of primary outputs for a sequence of time frames. Thus, it is necessary to guarantee that there can be no logic (combinational or sequential) feeding from the outputs of a design which can detect the fault.

In this paper, we consider the more general problem of modifying the internal nodes of a circuit arbitrarily so that the "behavior' of the modified circuit is acceptable according to our safe replacement criterion— we want to guarantee the safe replaceability without making *any* assumptions on the environment. Also, in Section 5.4, we show that setting the internal node input lines to constants and verifying the validity of the modified design is a special case; thus the exact solution for our proposed method covers the redundancy removal techniques.

3.3 Retiming and Resynthesis

Retiming and resynthesis [7] can be used to perform sequential optimization by alternating steps of moving of latches across combinational logic (retiming) and performing combinational resynthesis.

Retiming seems to be able to work if latches do not have reset lines. However, consider once again the circuit in Figure 3. Suppose latch l_0 is retimed across the fanout to two latches l_0' and l_0'', then the power-up state $(l_0' = 0, l_0'' = 1, l_1 = 0, l_2 = 0)$ produces a 0 on input sequence $1 \cdot 1$, whereas no power-up state in the original design exhibits this behavior. So the reason given in Section 3.2, for searching for a new safe replacement condition, applies here as well.

3.4 Synchronous Relations

Damiani and De Micheli [8] proposed using synchronous recurrence equations (or synchronous relations) to capture don't care information in sequential circuits. A synchronous relation expresses the flexibility for a sub-circuit in a sequential gate-level design as a Boolean relation on finite sequences of inputs and outputs. Although the synchronous relation does not depend on a designated start state, the start state has to be taken into account. For a design without reset lines, we can construct an example [12] where two designs satisfy the same synchronous relation but a state of one design may exhibit some behavior exhibited by no state of the other design.

4 Safe Replaceability

We want a condition for safe replacement which guarantees that if we replace an old design with a new one, it is impossible for any environment to detect that the replacement has been made.

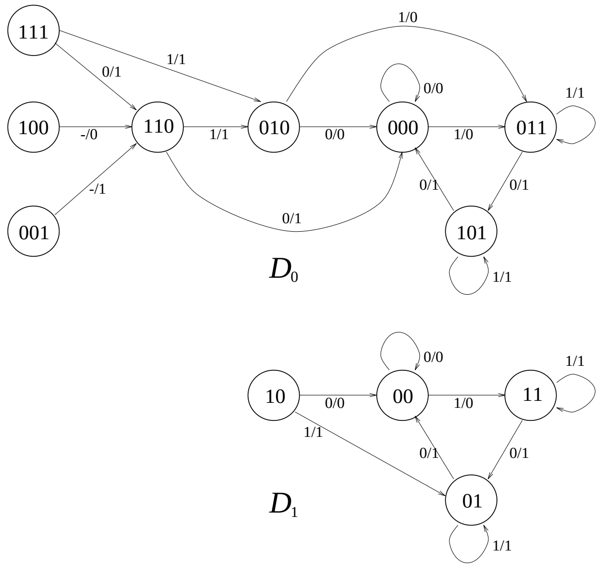

Figure 4: Example of a safe replacement

Conversely, we would like all replacements that cannot be detected by any environment to satisfy our condition. We assume that no latches have reset lines[2]. Since it cannot be predicted which state the design powers up in, we can safely assume that no matter which state the original design powers up in, the subsequent input/output behavior of the design is acceptable to the environment. Based on this observation, we give the following condition (the *safe replacement condition*):

Definition 8 *Design D_1 is a* safe replacement *for design D_0 (denoted by $D_1 \preceq D_0$) if given any state $s_1 \in Q_{D_1}$ and any finite input sequence $\pi \in I^*$, there exists* <u>*some*</u> *state $s_0 \in Q_{D_0}$ such that the output behavior $\lambda_{D_1}(s_1, \pi) = \lambda_{D_0}(s_0, \pi)$.*

We argue that the above condition provides maximum flexibility while guaranteeing that the replacement cannot be detected by the environment. First, if we make the above condition any weaker, then there exists an input sequence π and a state in the new design D_1 so that if the D_1 powers up in this state and sees the sequence π, it will produce a behavior which could not have been seen from any state in D_0. This violates our requirement that no environment should be able to detect the replacement. Secondly, since we have assumed that the power-up state of a design cannot be predicted, if $D_1 \preceq D_0$, then for every input sequence any power-up state of D_1 behaves like some power-up state of D_0. This implies that any behavior from any state of D_1 is acceptable. Thus our condition guarantees that replacing D_0 by D_1 cannot be detected by any environment.

Any design which has the same state transition graph as the original design trivially satisfies the safe replacement condition. As a non-trivial example consider designs D_0 and D_1 in Figure 4, where $D_1 \preceq D_0$. States 00, 11 and 01 in D_1 behave like states 000, 011 and 101, respectively, in D_0 for all input sequences. The remaining state 10 in D_1 behaves like state 010 for all input sequences starting with 0, and like state 101 for all input sequences starting with 1. Notice that state 10 in D_1 is not equivalent to any state in D_0; conversely, no state in D_1 is equivalent to state 001 in D_0. Definition 8 guarantees that there is no input/output behavior in D_1 which is not present in D_0. On the other hand, state 001 in D_0 outputs sequence $1 \cdot 1 \cdot 0$ on the input sequence $1 \cdot 1 \cdot 1$ whereas no state of D_1 can exhibit this behavior. However, we had claimed that no environment can detect if D_0 is replaced by D_1. This apparent paradox can be explained by the observation that since it is not true that every power-up state of D_0 exhibits this behavior, the environment of D_0 could not possibly depend on this behavior, and hence it cannot always expect the output sequence for $1 \cdot 1 \cdot 0$ for the input use $1 \cdot 1 \cdot 1$ each time the design powers up.

It is easy to see that the safe replacement condition does not suffer from any problems with the previous notions that we discussed in Section 3. Most importantly, it satisfies the compositionality property— if $D_1 \preceq D_0$, then $D_1 \otimes C \preceq D_0 \otimes C$, where $\otimes$ denotes composition of two designs.

Definition 9 *Given a design D, a set of states $S \subseteq Q_D$ is a* closed set *if for any input $a \in I$, any state $s \in S$: $\delta_D(s, a) \in S$.*

Definition 10 *A* terminal strongly connected component (tSCC) *of a design D is a closed set of states $S \subseteq Q_D$ such that for every pair of states $s_0, s_1 \in S$: there exists an input sequence $\pi \in I^*$ such that $\delta_D(s_0, \pi) = s_1$,*

We have shown other properties of safe replacement in [13]:

- The relation $\preceq$ is transitive and reflexive, but not symmetric. (The replaced design has fewer or same input/output behaviors as the original design).
- A replacement design can have fewer or more latches than the original design (in Figure 4, D_0 has 3 latches whereas D_1 has 2).
- Unlike sequential hardware equivalence [5], safe replaceability also applies to a design which does not have any initializing sequence.
- If $D_1 \preceq D_0$, every initializing sequence for D_0 is an initializing sequence for D_1 as well.
- If $D_1 \preceq D_0$, then every tSCC in D_1 must be equivalent (by Definition 3) to a tSCC in D_0.

5 Sequential Resynthesis

We want to exploit the flexibility provided by the safe replacement condition in Definition 8 to optimize synchronous sequential circuits. Unfortunately, Definition 8 does not directly provide a closed form expression to express all the flexibility for safe replacement.

A sequential gate-level design can be viewed as a connection between a purely combinational part and a set of latches (Figure 1). The inputs to the combinational part are the real primary outputs of the design $\vec{i}$ plus the wires from the latches, or the present state vector, denoted by $\vec{x}$. The outputs of the combinational part are the real primary outputs of the design $\vec{o}$ plus the wires to the latches, or the next state vector, denoted by $\vec{y}$. We want to optimize this combinational part while maintaining the safe replacement condition. If we can express the flexibility in Definition 8 by a Boolean relation in $(\vec{i}, \vec{x}, \vec{o}, \vec{y})$, we can use known techniques [14] for minimizing multi-level networks given a Boolean relation, in terms of the inputs and outputs of the network.

Unfortunately, the flexibility allowed by the safe replacement condition cannot be represented by a Boolean relation between the domain space $(\vec{i}, \vec{x})$ and the range space $(\vec{o}, \vec{y})$.

[2]If some latches have a reset line, they can be modeled by a latch without a reset line if we treat the reset line as another primary input; see [12] for details.

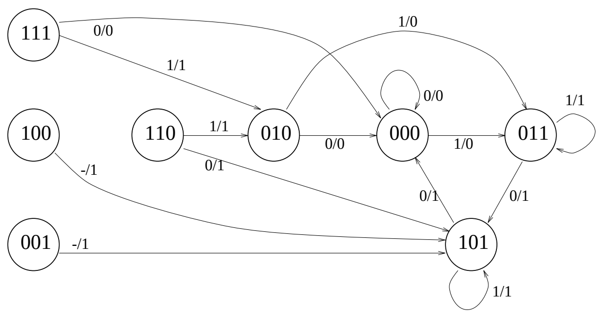

Figure 5: Design D_2 (a safe replacement for D_0)

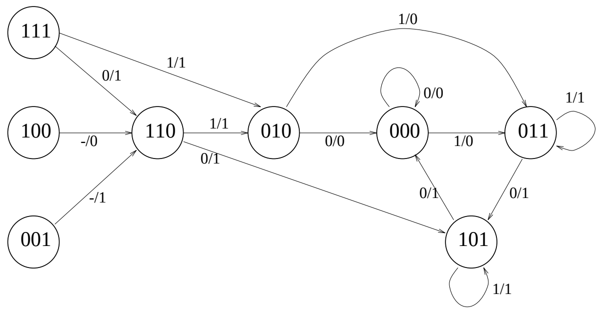

Figure 6: Design D_3 (an unsafe replacement for D_0)

Consider the design D_2 in Figure 5 which is a safe replacement of the design D_0 of Figure 4. The two designs differ on their mappings of the following 6 points in $(\vec{i}, \vec{x})$: $(0, 111), (0, 100), (1, 100, 1), (0, 001), (1, 001), (0, 110)$. One property of Boolean relations is that the flexibility for each point in the domain space is independent of other points [15]. So, if the flexibility for safe replacement could be expressed by a Boolean relation, then every design corresponding to a flexibility choice for each of these 6 domain points would be a valid replacement (there are 2^6 such designs). In particular, design D_3 in Figure 6, which behaves like D_2 on point $(0, 110)$ and like D_0 on the other points, would be a safe replacement. However, this is not so because if design D_3 powers up in state 111 and is given the input sequence $0 \cdot 0 \cdot 0$ it produces the output sequence $1 \cdot 1 \cdot 1$, whereas there is no state in D_0 which exhibits this behavior. Thus the flexibility for safe replaceable designs with the same number of latches cannot be expressed as a Boolean relation in $(\vec{i}, \vec{x}) \times (\vec{o}, \vec{y})$. One way to represent such flexibility would be through Multiple Boolean Relations [15], which are arbitrary sets of Boolean relations.

5.1 Sufficient Condition for a Safe Replacement

As we argued in the last section, the complete flexibility for safe replacement can be expressed by a multiple Boolean relation. However, because of the intractably large solution space of multiple Boolean relations, there are no known general techniques to use multiple Boolean relations for logic synthesis. We now provide a sufficient (but not necessary) condition for safe replacement, from which we will obtain a Boolean relation in $(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ to express partial flexibility for safe replacement.

Proposition 5.1 *Given designs D_0 and D_1 such that for every state $s_1 \in Q_{D_1}$ and input $a \in I$, there exists a state $s_0 \in Q_{D_0}$ such that $\lambda_{D_1}(s_1, a) = \lambda_{D_0}(s_0, a)$ and $\delta_{D_1}(s_1, a) \sim \delta_{D_0}(s_0, a)$. Then $D_1 \preceq D_0$.*

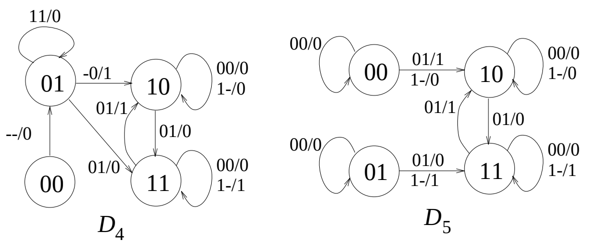

Figure 7: Example of a safe replacement

Proof: Choose any state $s_1 \in Q_{D_1}$, and any input sequence $\pi = a_0 \cdot a_1 \cdots a_p \in I^*$. Now, there exists a state $s_0 \in Q_{D_0}$ such that $\lambda_{D_1}(s_1, a_0) = \lambda_{D_0}(s_0, a_0)$ and $\delta_{D_1}(s_1, a_0) \sim \delta_{D_0}(s_0, a_0)$. Thus $\lambda_{D_1}(s_1, \pi) = \lambda_{D_0}(s_0, \pi)$, and hence $D_1 \preceq D_0$. ■

The above result is not a necessary condition for safe replacement. For example, consider the designs D_4 and D_5 in Figure 7. It can be seen that $D_5 \preceq D_4$. However, in design D_5 state 01 goes to state 01 on input 00, and no state in design D_4 is state equivalent to state 01 on design D_5 (as discussed in Section 4, there may be states in the replacement design which are not equivalent to any state in the original design). On the other hand, designs D_0 and D_1 of Figure 4 which satisfy the sufficient condition of Proposition 5.1, and thus $D_1 \preceq D_0$.

5.2 Flexibility for Resynthesis

We will use the sufficient condition in Proposition 5.1 to derive a method to extract and use flexibility for sequential resynthesis.

First, note that a necessary condition for safe replacement is that every tSCC in the new design must be equivalent to a tSCC in the old design. A tSCC can be thought of as defining a steady state behavior of the design. So at the state-transition level we cannot alter the behavior of the states in some tSCC of the original design. For our synthesis method we will choose a set of states which is closed under all inputs, called the *core*:

Definition 11 *Given a design D, a set of states $S \subseteq Q_D$ is called a* core *of the design if it is a closed set under all inputs, i.e. for any input a and any state $s \in S$: $\delta_D(s, a) \in S$.*

We will preserve the behavior of the core during resynthesis, and in order to make resynthesis tractable, we will preserve the encodings of the states in the core. Note that since the core is closed under all inputs it contains a tSCC of the original design and thus satisfies the necessary condition discussed above.

Proposition 5.1 indicates that each state reachable from some state in D_1 is equivalent to some state in D_0. The set of these states will serve as the core of the old design, which we will reproduce in the new design. We require that each of the remaining states in the new design satisfy the following Boolean relation $\mathcal{P}(\vec{i}, \vec{o}, \vec{y})$. In the following, $core(\vec{x}) = 1$ if and only if state $\vec{x}$ lies in the chosen core.

$$\mathcal{P}(\vec{i}, \vec{o}, \vec{y}) = \exists \vec{x}_0[(\lambda_{D_0}(\vec{x}_0, \vec{i}) = \vec{o}) \wedge (\delta_{D_0}(\vec{x}_0, \vec{i}) = \vec{y})] \wedge \textit{core}(\vec{y}) \quad (1)$$

Intuitively, a triple $(\vec{i}, \vec{o}, \vec{y})$ satisfies the relation $\mathcal{P}$ iff there is a state $\vec{x}_0$ in the given design which transitions to state $\vec{y}$ and outputs $\vec{o}$ on input $\vec{i}$, and $\vec{y}$ lies inside the core. It is easy to see that if the core of the old design is preserved and the relation $\mathcal{P}$ holds for the remaining states of the new design, the sufficiency condition of Proposition 5.1 is satisfied.

We can now form a Boolean relation $\mathcal{R}(\vec{i},\vec{x},\vec{o},\vec{y})$ which expresses the flexibility for the entire network including states inside the core:

$$\begin{aligned}\mathcal{R}(\vec{i},\vec{x},\vec{o},\vec{y}) = \\ &[\,\mathit{core}(\vec{x}) \wedge (\vec{y}=\delta_{D_0}(\vec{x},\vec{i})) \wedge (\vec{o}=\lambda_{D_0}(\vec{x},\vec{i}))] \vee \\ &[\neg(\mathit{core}(\vec{x})) \wedge \mathcal{P}(\vec{i},\vec{o},\vec{y})] \qquad (2)\end{aligned}$$

Intuitively, the relation $\mathcal{R}$ ensures that the states inside the core have the same behavior (next state and output) as the original design, while the states outside the core are free to choose any behavior which satisfies the relation $\mathcal{P}$.

We recall from Section 5 that the total flexibility for the network cannot be expressed by a Boolean relation; we are able to get a relation here because our condition in Proposition 5.1 is only a sufficient condition.

5.3 Choice of Core

For a given design there may be many choices for the core which satisfy Definition 11. Some natural choices are:

- The set of all states Q_{D_0}.

- Any onion ring [16] of the design. Onion rings $A_1, A_2, \ldots$ are defined recursively:

$$\begin{aligned} A_1 &= Q_{D_0}, \\ A_{k+1} &= \{y \mid \exists i \in I, x \in A_k : \delta_{D_0}(x,i)=y\} \qquad (3)\end{aligned}$$

  For design D_4 in Figure 7, $A_1 = \{00,01,10,11\}, A_2 = A_3 = \cdots = A_\infty = \{01,10,11\}$. A_∞ is also called the outer envelope of a design. The outer envelope is computable by a fixed-point computation starting from Q_{D_0}.

- Any tSCC of the design. Design D_4 has only one tSCC: $\{10,11\}$.

Any of the above choices satisfies Definition 11 and can be used in equation 2. The ideal choice is a small core to give us a large number of states outside the core because we have flexibility only in modifying the behavior of states outside the core. The smallest tSCC is the smallest set which qualifies as a core.

However, we are restricted by the requirement that the starting design must itself satisfy the Boolean relation $\mathcal{R}$. The only known method for minimizing multi-level networks under flexibility expressed by a Boolean relation [14] requires this restriction, as we shall see later in Section 5.4.

For example, if we choose the tSCC of design D_0 in Figure 4 (states 000, 011 and 101) as the core, the design D_0 does not satisfy relation $\mathcal{R}$ in expression 2 because the state 111 (a non-core state) does not jump to a state inside the core on input 1 and hence does not satisfy $\mathcal{P}$. Choices of core that guarantee that the given design satisfies $\mathcal{R}$ are Q_{D_0} (the set of all states) and the second onion ring A_2 (the set of states reachable in one step from Q_{D_0}). The former does not give any flexibility because all states are in the new design; so we make the latter choice. While it might seem that we lose much flexibility by having to choose a much larger core than the smallest possible, our experiments in Section 6 indicate that choosing A_2 gives us most of the flexibility for most of the examples.

5.4 Multi-level Synthesis

Previous sections referred to a *multi-level* combinational logic network that computes the next states and outputs of a sequential design. More precisely, the *Boolean network* $\mathcal{N}$ associated with a sequential circuit is a directed acyclic graph such that for each node in $\mathcal{N}$, there is a Boolean variable u_i and an associated Boolean function f_i such that $u_i = f_i$. The *support* of f_i is the set of variables corresponding to the immediate fan-ins of the node. The primary input variables $\{i_1, i_2, \ldots, i_n, x_1, x_2, \ldots, x_t\}$ correspond to the inputs and present states of the sequential circuit; the primary output variables $\{o_1, o_2, \ldots, o_m, y_1, y_2, \ldots, y_t\}$ correspond to the outputs and next states. *Intermediate nodes* are those which do not correspond to inputs or outputs. The network computes a function from the input space B^{n+t} to the output space B^{m+t} derived by composing the functions at the intermediate nodes.

Since hardware designs typically arise by composing small modules, it is very natural for circuits to have a multi-level structure. The nodes of the corresponding Boolean network represent the logical functionality of the modules. It has been observed that the area of the hardware implementation of a design is strongly correlated to the total number of literals in the *factored form* [17] representation of the functions at the logic nodes. Thus minimizing the function (with respect to the literal count) at the node constitutes a powerful synthesis technique.

At any intermediate node of a network there is a local function $f_i : B^r \rightarrow B$, where r is the cardinality of the support. *Node simplification* is the process of optimizing a Boolean network by using don't cares in conjunction with a two level minimizer [18] to optimize the functions at the nodes. These don't cares arise in several ways:

- Because of the structure of the network, only a certain subset of B^r may be generated by assignments to the inputs. This gives rise to *satisfiability don't care* (SDC) points for f_i [17].

- For certain input assignments, the values taken by the primary outputs of $\mathcal{N}$ may be independent of the function computed by a node; these are *observability don't care* points (ODC) for that node [19].

- For certain input assignments the functionality of the node can be changed without destroying safe replaceability; this flexibility leads to the *replaceability don't cares* points (RDC).

The ODC at a node u can be computed by considering the output α_u of the node to be another input; thus the primary outputs of $\mathcal{N}$ are expressed in terms of the primary inputs and α_u. The ODC is the set of points in B^r where no primary output of $\mathcal{N}$ depends on α_u.

Let $\mathcal{R}(\vec{i},\vec{x},\vec{o},\vec{y})$ be a Boolean relation expressing all the flexibility in the choice of combinational logic for a sequential circuit. Cerny and Marin [20] demonstrate a close relationship between optimizing a Boolean network with respect to a given Boolean relation, and computing observability don't care sets. The starting network $\mathcal{N}$ must satisfy the relation $\mathcal{R}$. The relation can be viewed as a single node with inputs $\vec{i},\vec{x},\vec{o},\vec{y}$; this node is referred to as the *observability node*. Composing this node with the network as shown in Figure 8 yields an *observability network* $\mathcal{N}'$. It is shown in [20, 14] that all the don't cares that can be used to optimize the nodes in $\mathcal{N}$ are derivable from the ODC of the node in the network $\mathcal{N}'$.

In our scenario, the relation $\mathcal{R}(\vec{i},\vec{x},\vec{o},\vec{y})$ is given by equation 2 in Section 5.2, with $core(\vec{x})$ being the set A_2 as defined in equation 3 in Section 5.3. As discussed in section 5.3, for this choice of *core*, the combinational logic network associated with the initial design is

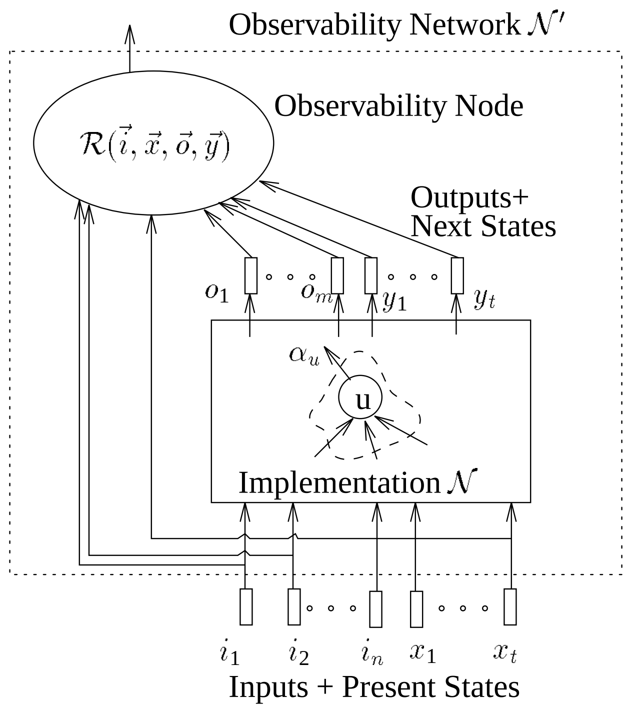

Figure 8: ODC's from the Observability Network yield flexibility under Observability Relation for nodes in Implementation Network.

an implementation for $\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$. Using $\mathcal{R}$ as the observability node guarantees that, for any other internal node, the ODC derived from the observability network contains the RDC for the original network.

We use Binary Decision Diagrams (BDD's) to represent the design, flexibility relation, and core. There is a variable associated with each primary input and each primary output; for each latch there is a present state and next state variable. Let u be a node in the design for which the ODC is to computed. We add a new BDD variable α_u corresponding to the output of u, and generate BDD's for the next state and output functions in terms of the primary inputs, latch outputs, and α_u. These are composed with the flexibility relation to obtain the BDD for the function computed by the observability network. Let $f(\vec{i}, \vec{x}, \alpha_u)$ be the output of the observability network; then the ODC for node u is given by $f(\vec{i}, \vec{x}, \alpha_u = 1)f(\vec{i}, \vec{x}, \alpha_u = 0) + \overline{f}(\vec{i}, \vec{x}, \alpha_u = 1)\overline{f}(\vec{i}, \vec{x}, \alpha_u = 0)$. This is in terms of primary inputs; we then project this set into the space comprised of the fan-ins of the node (as in [19]). These are used in conjunction with a subset of the satisfiability don't care set to optimize the function at u.

We are computing all inputs under which the outputs are independent of the function computed at the node. Thus we will detect cases where an internal line (which is an input to a node) can be set to a constant while maintaining compatibility with relation $\mathcal{R}$. This means that we automatically remove redundant faults (which satisfy $\mathcal{R}$) from the circuit. Note that we are able to detect these redundancies because we are computing the flexibility of each node just before minimizing it; if we had obtained compatible flexibility (as in [14]) for all nodes before minimizing them simultaneously we would not be able to claim this. The price we pay is in time: we need to simplify nodes on an individual basis, and if the node is simplified, potentially all the BDDs for functions that the node fans out to must be recomputed.

6 Experiments

We have implemented the above method for sequential resynthesis presented in this paper in the SIS sequential synthesis system [21]. We used BDD's to represent all sets and functions and to perform all the set manipulations implicitly. We have performed experiments on the ISCAS89 benchmark suite (from `s344` to `s1494`). Since some ISCAS benchmarks are very minor variations of each other and give almost identical results, we chose the largest example (for example, `s1288` represents `s1288` and `s1196`; `s444` represents `s444`, `s400` and `s382`, etc.) for each class.

We report our experiments in Table 1. First observe that, for many examples, the size of the core (the second onion ring A_2) is close to the size of the tSCC (each of the examples has exactly one tSCC) when compared to the total number of states (2^L, where L is the number of latches).

Since we are minimizing the network one node at a time, very small nodes are unlikely to yield much optimizations. The node sizes of the benchmark circuits were very small; we executed the SIS commands `sweep; eliminate 10` to partially collapse the network. For some benchmarks, a totally collapsed network had a smaller literal count than a partially collapsed one; for these circuits, we started with the totally collapsed network (using command `sweep; collapse`).

The table reports the reduction in the total number of literals of the network. We have partitioned the literal reduction into reductions due to satisfiability don't cares (SDC), observability don't cares (ODC) and replaceability don't cares (RDC) separately. We minimized each internal nodes three times: using SDC, using (SDC + ODC), using (SDC + ODC + RDC). The literal reductions under the ODC column are reductions in addition to those under the SDC column; those under the RDC column are reductions in to those under the SDC + ODC columns. Note that the literal savings under the RDC column, for example, does not indicate the savings due to the safe replaceability don't cares; it is actually the difference in the savings using (SDC + ODC + RDC) and (SDC + ODC). We have observed that a lot of the literal saving is common to all three methods: SDC, ODC and RDC. So a 0 in the RDC column might indicate that most of the flexibility due to RDC is already captured by SDC and ODC. However, the CPU times reported in the table under the three columns represent the total time taken for each method separately.

It is interesting to compare the RDC statistics to the ODC statistics because while ODC provides observability don't cares for an internal node in terms of the primary inputs, RDC provides safe replaceability don't cares in addition to the observability don't care points. Then we project these don't cares to the immediate fan-ins of the node using the same techniques used for projecting ODC points [19]. We observe that for some examples, ODC gives no literal reductions beyond SDC alone but RDC is able to obtain additional, though modest, reductions. These reductions are of the order of 5% on these circuits. On other circuits RDC gives no additional improvements over SDC and ODC. For these examples, most replaceability don't care points may be overlapping with satisfiability and observability don't care points. Except for 2 examples, the CPU time taken by RDC flexibility is of the same order as the time taken to utilize the SDC and ODC flexibility.

ISCAS89 benchmarks may not constitute a good source of examples to judge the potential of our approach. This is because for many of the circuits, collapsing the logic to two levels before applying our minimization yields a smaller circuit, thus negating the whole basis of doing multi-level logic minimizations. Furthermore,

Circuit	I	O	L	#states			#literals		Savings (in #literals)			Time (in seconds)		
				tSCC	core	2^L	start	end	SDC	ODC	RDC	SDC	ODC	RDC
s349	9	11	15	1487	23232	32768	173	157	9	7	0	0.57	1.40	123.89
s386	7	7	6	13	13	64	205	138	53	0	14	0.83	1.03	1.83
s444	3	6	21	8864	23740	2097152	236	171	50	15	0	0.76	1.56	4.71
s510	19	7	6	47	61	64	307	255	52	0	0	1.97	2.35	3.17
s526	3	6	21	8868	401460	2097152	323	233	79	0	11	6.32	6.61	10.80
s713	35	23	19	1544	6663	524288	285	timeout after 10000 sec						
s832	18	19	5	25	25	32	470	351	110	0	9	3.04	4.18	12.60
s953	16	23	29	504	504	536870912	700	597	78	20	5	3.85	17.14	37.56
s1238	14	14	18	2615	2652	262144	882	636	98	148	0	14.74	176.26	1075.49
s1494	8	19	6	48	48	64	896	542	353	0	1	26.81	34.97	58.17

Table 1: Experimental results. I, O and L denote the number of inputs, outputs and latches, respectively. The savings reported used ODC are in addition to those under SDC; those reported under RDC are in addition to those reported under (SDC+ODC). The sum of these three columns is the difference between the starting and ending literal count.

the average node size is too small to hope for any significant reduction over that given by the SDC. Hence the merit of our approach may be better judged on the basis of real multi-level designs.

Memory explosion is a common problem with BDD's. in the problem size. For our experiments we noted that the ability to finish the examples (and the time taken) was most influenced by the number of input wires to the design. The number of latches and the number of output wires are also a factor, though to a lesser degree. Since large designs routinely arise as set of interacting components, it is natural to decompose the design into smaller components and synthesize them independently. Intuitively, the components being smaller, will be less complex, and hence easier to synthesize. We expect that our methods will be most useful in resynthesizing an arbitrary portion of a design without worrying about any interaction with its environment.

7 Conclusions and Future Work

We have provided a notion of safe replaceability ($\preceq$) that is independent of initial states of a design and the intended environment of a design. We have used this notion to provide a method for sequential resynthesis towards an area reduction of gate-level designs; our notion does not require us to preserve the state transition behavior of the design. We have implemented our algorithm using BDD's. and we expect to be able to test our algorithm on industrial-level designs.

The synthesis method presented in this paper is just one way of exploiting the flexibility allowed by the safe replacement condition. We are looking at other methods for sequential synthesis for this notion of sequential equivalence.

We would also like to be able to handle arbitrarily large netlists for our resynthesis methods. For this we are working on partitioning algorithms which decompose a network into modules so that the number of wires running across the modules is minimized. Fewer interconnection wires gives us two advantages: we can handle larger number of latches before we run into a BDD explosion problem, and we can expect fewer states in the core (and hence more flexibility) if we have fewer controlling input lines running to the modules. Furthermore, reducing the number of output lines restricts the observable output sequences which leads to larger observable don't cares.

References

[1] B. Lin, H. J. Touati, and A. R. Newton, "Don't Care Minimization of Multi-level Sequential Logic Networks," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 414–417, Nov. 1990.

[2] C. Berthet, O. Coudert, and J. C. Madre, "New Ideas on Symbolic Manipulation of Finite State Machines," in *Proc. Intl. Conf. on Computer Design*, Oct. 1990.

[3] H. Cho, G. D. Hachtel, and F. Somenzi, "Redundancy Identification and Removal Based on Implicit State Enumeration," in *Proc. Intl. Conf. on Computer Design*, pp. 77–80, Oct. 1991.

[4] G. Berry and H. J. Touati, "Optimized Controller Synthesis Using Esterel," in *Workshop Notes of Intl. Workshop on Logic Synthesis*, (Tahoe City, CA), May 1993.

[5] C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 1469–1494, Dec. 1992.

[6] K.-T. Cheng, "Redundancy Removal for Sequential Circuits Without Reset States," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 13–24, Jan. 1993.

[7] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimization of Sequential Networks with Combinational Techniques," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 74–84, Jan. 1991.

[8] M. Damiani and G. De Micheli, "Synthesis and Optimization of Synchronous Logic Circuits from Recurrence Equations," in *Proc. European Conf. on Design Automation*, pp. 226–231, Mar. 1992.

[9] L. Entrena and K.-T. Cheng, "Sequential Logic Optimization by Redundancy Addition and Removal," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 310–315, Nov. 1993.

[10] J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines*. Intl. Series in Applied Mathematics, Englewood Cliffs, N.J.: Prentice-Hall, 1966.

[11] R. Rudell, Synopsys, Inc.. Personal communication, Mar. 1994.

[12] C. Pixley, V. Singhal, A. Aziz, and R. K. Brayton, "Multi-level Synthesis for Safe Replaceability," Tech. Rep. UCB/ERL M94/31, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Apr. 1994.

[13] V. Singhal and C. Pixley, "The Verification Problem for Safe Replaceability," in *Proc. of the Conf. on Computer-Aided Verification* (D. L. Dill, ed.), vol. 818 of *Lecture Notes in Computer Science*, pp. 311–323, Springer-Verlag, June 1994.

[14] H. Savoj and R. K. Brayton, "Observability Relations and Observability Don't Cares," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 518–521, Nov. 1991.

[15] E. M. Sentovich, V. Singhal, and R. K. Brayton, "Multiple Boolean Relations," in *Workshop Notes of the Intl. Workshop on Logic Synthesis*, (Tahoe City, CA), May 1993.

[16] S.-W. Jeong, *Binary Decision Diagrams and their Applications to Implicit Enumeration Techniques in Logic Synthesis*. PhD thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309, 1992.

[17] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE*, vol. 78, pp. 264–300, Feb. 1990.

[18] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[19] H. Savoj, R. K. Brayton, and H. Touati, "Extracting Local Don't Cares for Network Optimization," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 514–517, Nov. 1991.

[20] E. Cerny and M. A. Marin, "An Approach to Unified Methodology of Combinational Switching Circuits," *IEEE Trans. Computers*, vol. 27, no. 8, 1977.

[21] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. Intl. Conf. on Computer Design*, pp. 328–333, Oct. 1992.