LOGIC OPTIMIZATION OF FSM NETWORKS USING INPUT DON'T CARE SEQUENCES

by

Huey-Yih Wang and Robert K. Brayton

Memorandum No. UCB/ERL M95/42

8 June 1995

LOGIC OPTIMIZATION OF FSM NETWORKS USING INPUT DON'T CARE SEQUENCES

by

Huey-Yih Wang and Robert K. Brayton

Memorandum No. UCB/ERL M95/42

8 June 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Logic Optimization of FSM Networks Using Input Don't Care Sequences *

Huey-Yih Wang Robert K. Brayton
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Abstract

Current approaches to compute and exploit the flexibility of a component in an FSM network are all at the symbolic level [30, 40, 42, 41]. Exploitation of this flexibility relies on state minimizers for incompletely specified FSM's (ISFSM's) or pseudo non-deterministic FSM's (PNDFSM's) [42]. However, state-of-the-art state minimizers cannot handle large ISFSM's or PNDFSM's [18, 20, 43, 13, 21]. In addition, these exploitation techniques are at the symbolic level, not directly at the net-list logic level. In this paper, we present an approach to exploit exact or approximate input don't care sequences directly at the logic level. We demonstrate that many sequential logic optimization techniques can be employed to exploit input don't care sequences. As a result, computation and exploitation of input don't care sequences in larger FSM networks can be made efficient and effective. Finally, we give preliminary results on some artificially constructed FSM networks. Preliminary results indicate that our approach is effective in reducing the size of a component of an FSM network.

*This project was supported by NSF under contract number MIP-8719546.

1 Introduction

As digital system design complexity increases, hierarchical specification becomes vital. For example, hardware description languages, such as Verilog or VHDL, are typically used to specify industrial designs. Once the design is verified, logic synthesis tools are used to optimize the circuit implementation with respect to some objective. The objective can be minimum area, minimum delay, maximum testability, minimum power consumption, or any combination of these. An underlying model for a hierarchical specification in the synthesis and verification community is a network of interacting finite state machines (FSM's). In this paper, synchronous FSM networks with known initial states are considered. A severe limitation of current synthesis tools for sequential circuits is that only a single FSM is considered at a time, e.g., SIS [34].

Theoretically, we can collapse an FSM network into a single FSM. However, this is not preferred, because of the following reasons. (1) This single FSM may be too big to be handled by synthesis tools, e.g., state encoding programs. (2) Some components in the network may be non-deterministic FSM's which are not synthesizable, e.g., an abstract description of the environment. (3) The hierarchy specified by designers may contain important information which is useful for an efficient implementation. (4) Some modules may already synthesized well and should not be touched. With hierarchical specification, each component is likely specified in a reasonable size. Therefore, another approach to synthesizing an FSM network is to synthesize one component at a time. Due to interaction with other components, the controllability and observability of a component are reduced, so the flexibility for implementing this component increases. By exploiting this flexibility, the quality of the implementation may be improved. Therefore, a key to logic optimization in a hierarchical specification is to consider the interaction between components.

The flexibility in the context of an isolated combinational circuits can be expressed by don't cares, and for an individual component in a hierarchically specified combinational circuit, a Boolean relation [4] (observability relation [7, 31]) or symbolic relation [26] is required to express all its flexibility. Similarly, exploitation of flexibility is important for sequential circuits. Several approaches have been proposed. For example, in [27], unreachable or equivalent states are used in the optimization of an isolated sequential circuit. Damiani and De Micheli [14] introduced synchronous relations to deal with the logic optimization of sequential circuits with pipelined latches. In their approach, a circuit implementation is given as the starting point.

In the case of an individual component in an FSM network, there are several approaches. The first approach [42] used a pseudo non-deterministic FSM (PNDFSM), namely an E-machine, to express all flexibility. Later, Lin [24] proposed a different construction method for the E-machine, but subset construction [29] is required in the general case. The exploitation of E-machine usually is done by state minimization of PNDFSM's [43, 13, 21].

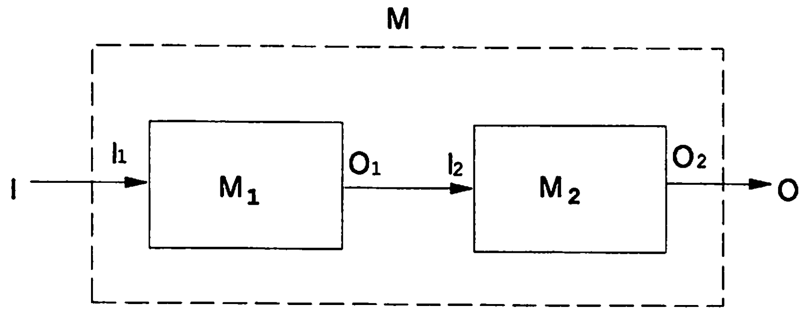

Figure 1: $\mathcal{N}_1$: A cascade circuit of two FSM's.

Another approach (which is an approximate one) is based on the notion of don't care sequences [30]. There are two kinds; input and output don't care sequences. Consider the cascade machine in Figure 1, where M_1 is the driving machine and M_2 the driven machine. Unger [38] observed that M_2, when driven by M_1, may possess more unspecified transitions than as an isolated machine, and proposed a method to approximate and exploit a subset of this information. Devadas [15] proposed a different but similar procedure. Kim and Newborn [22] proposed an elegant complete solution. For a two-way-communication network of FSM's, $\mathcal{N}_2$, as shown in Figure 2, Wang and Brayton [40] gave an exact computation, and demonstrated that state minimization for incompletely specified FSM's (ISFSM's) [18, 20] can be used to exploit input don't care sequences in general FSM Networks.

On the other hand, the flexibility in implementing M_1 when cascaded with M_2 is called *output don't sequences*. Devadas [15] proposed a method to exploit *sequential output don't cares*, and later Rho *et al.* [30] generalized Devadas' procedure to

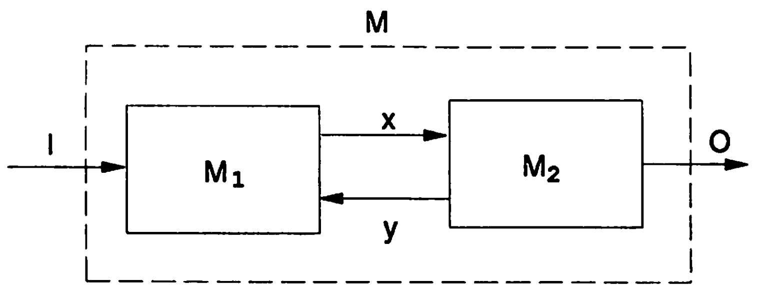

Figure 2: $\mathcal{N}_2$: A two-way-communication network of FSM's.

compute *fixed-length output don't care sequences.* Another approach based on FSM equivalence checking for approximating the set of output don't care sequences was proposed in [41].

The above algorithms for computing the flexibility of an individual component in an FSM network are all based on the manipulation of transition relations of FSM's, i.e., symbolic information is manipulated. Currently, exploitation of this flexibility hinges on state minimizers for ISFSM's [18, 20] or PNDFSM's [43, 13, 21]. Afterwards, state encoding and sequential optimization techniques are applied to the state-minimized machine. Presently, no existing state minimizers can handle large ISFSM's or PNDFSM's [18, 20, 43, 13, 21]. The problem of state minimization of ISFSM's is NP-hard. The computation of exact input don't care sequences in FSM networks can be efficiently done; however, the exploitation of them using state minimization is difficult [40]. To circumvent this, approximations are required to trade off between quality and efficiency [30, 40]. As a result, much flexibility may be lost.

Furthermore, in contrast to logic optimization techniques in sequential circuits, these algorithms do not use a circuit implementation as the starting point; the exploitation is not performed at the logic level. In terms of efficiency, effectiveness, and the size of circuits, optimization techniques for sequential logic circuits (net-lists) are in a more mature stage than symbolic methods, since most are able to produce acceptable results in larger circuit designs. However, manipulating symbolic information is indispensable for computing the flexibility of a component in an FSM network.

With this motivation, we propose an algorithm which takes a circuit implementation as the starting point and computes the flexibility at the symbolic level, but exploitation is directly at the logic level. We concentrate on input don't care sequences since they can be computed more efficiently and thus are more promising for large FSM networks. We give an overview and discuss the difficulties in exploiting them. Then, we propose an approach based on [22, 40], and demonstrate that input don't care sequences can be exploited using existing sequential optimization techniques. Our algorithm does not require a subset construction [29] as in the Kim and Newborn's procedure [22]. Finally, we give preliminary results on some artificially constructed FSM networks.

2 Preliminaries

2.1 Finite Automaton

A deterministic finite automaton (DFA), $\mathcal{A}$, is a quintuple $(K, \Sigma, \delta, q_0, F)$ where K is a finite set of states, Σ an alphabet, $q_0 \in K$ the initial state, $F \subseteq K$ the set of final states, and δ the transition function, $\delta : K \times \Sigma \rightarrow K$. A non-deterministic finite automaton (NFA), $\mathcal{A}$, is a quintuple $(K, \Sigma, \delta, q_0, F)$ where δ, the transition relation, is a finite subset of $K \times \Sigma^* \times K$, and Σ^* the set of all strings obtained by concatenating zero or more symbols from Σ. An input string is accepted by $\mathcal{A}$ if it ends up in one of final states of $\mathcal{A}$. The language accepted by $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, is the set of strings it accepts.

2.2 Finite State Machine

A finite state machine (FSM), M, is a six-tuple $(I, O, Q, \delta, \lambda, q_0)$, where I is a finite input alphabet, O a finite output alphabet, Q a finite set of states, δ the transition function, λ the output function, and q_0 the initial state. A machine is of *Moore* type if λ does not depend on the inputs, and *Mealy* otherwise. An FSM can be represented by a state transition graph (STG). A machine in which transitions under all input symbols from every state are defined is a *completely specified machine*; in other words, both δ and λ are complete functions. Otherwise, a machine is *incompletely specified.*

A *cascade* of FSM's M_1 and M_2, denoted $M_1 \rightarrow M_2$, is shown in Figure 1. M_1 is called the *driving machine*, M_2 the *driven machine*.

2.3 Sequential Testing and Redundancy

The single stuck-at fault model assumes that a single fault existing at a given wire in the circuit causes that wire to be permanently at a high voltage level (stuck-at-1), or a low-voltage level (stuck-at-0). Let M be a logic implementation of an FSM. A *test* for a fault f of M is a sequence of input vectors that, when applied to machine M with fault f starting from the reset state, causes output values different from those of the fault-free machine. If a fault f is untestable, it is *redundant*.

2.4 Set Computation and Operators

Let B designate the set $\{0,1\}$.

Definition 1 *Let E be a set and $S \subseteq E$. The* **characteristic function** *of S is the function $\chi_S : E \rightarrow B$ defined by $\chi_S(x) = 1$ if $x \in S$, and $\chi_S(x) = 0$, otherwise.*

Definition 2 *Let $f : B^n \rightarrow B$ be a Boolean function, and $x = \{x_1, ..., x_k\}$ a subset of the input variables. The* **existential quantification** *(***smoothing***) of f by x, with f_a denoting the* **cofactor** *of f by literal a is defined as :*

$$\begin{aligned} \exists_{x_i} f &= f_{x_i} + f_{\overline{x_i}} \\ \exists_x f &= \exists_{x_1} ... \exists_{x_k} f \,. \end{aligned}$$

Definition 3 *Let $f : B^n \rightarrow B^m$ be a Boolean function, $S_1 \subseteq B^n$ and $S_2 \subseteq B^m$. The* **image** *of S_1 by f is $f(S_1) = \{y \in B^m | y = f(x), x \in S_1\}$. $f(B^n)$ is the* **range** *of f. The* **inverse image** *of S_2 by f is $f^{-1}(S_2) = \{x \in B^n | f(x) = y, y \in S_2\}$.*

Definition 4 *Let $f : B^n \rightarrow B$ be a Boolean function, only depending on a subset of variables $y = \{y_1, ..., y_k\}$. Let $x = \{x_1, ..., x_k\}$ be another subset of variables, describing another subspace of B^n of the same dimension. The* **substitution** *of variables y by variables x in f is the function of x obtained by substituting x_i for y_i in f:*

$$(\theta_{y,x} f)(y) = f(x) \text{ if } x_i = y_i \text{ for all } 1 \leq i \leq k.$$

Definition 5 *Let $f : B^n \rightarrow B^m$ be a Boolean function. The relation (characteristic relation) associated with f, $F : B^n \times B^m \rightarrow B$, is defined as $F(x,y) = \{(x,y) \in B^n \times B^m | y = f(x)\}$. Equivalently, in terms of Boolean operations :*

$$F(x,y) = \prod_{1 \leq i \leq m} (y_i \equiv f_i(x)) \,.$$

Reduced ordered binary decision diagrams (BDD's) [5] are well suited to represent the characteristic functions of subsets of a set, and efficient algorithms [2, 5] exist to manipulate them to perform all standard Boolean operations. As a result, the above set operations can be done efficiently.

2.5 Multiple-Valued Functions

Let $X_1, X_2, \cdots X_n$ be multiple-valued variables ranging over sets $P_1, P_2, \cdots, P_n$ respectively, where $P_i = \{0, ..., p_i - 1\}$, and p_i are positive integers. A multiple-valued function f is a mapping

$$f : P_1 \times P_2 \times ... \times P_n \rightarrow B \,.$$

Let S_i be a subset of P_i, and $X_i^{S_i}$ represent the characteristic function

$$X_i^{S_i} = \begin{cases} 0 & \text{if } X_i \notin S_i \,. \\ 1 & \text{if } X_i \in S_i \,. \end{cases}$$

$X_i^{S_i}$ is called a *literal* of the variable X_i. If $|S_i| = 1$, this literal is a minterm of X_i. A *product term* or a *cube* is a Boolean product (AND) of literals. A *sum-of-products* is a Boolean sum (OR) of product terms. An *implicant* of a function f is a

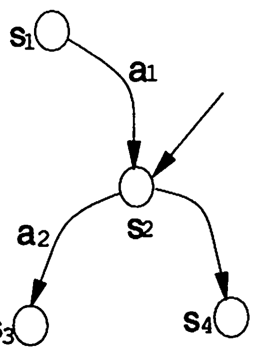

Figure 3: A part of the STG of M_2.

product term which does not contain any minterm in the OFF-set ($f^{-1}(0)$) of the function. A *prime implicant* of f is an implicant not contained in any other implicant of f.

Let a symbolic variable s assume values from $S = \{s_0, \ldots, s_{m-1}\}$. It can be represented by a multiple-valued variable, X, restricted to $P = \{0, \ldots, m-1\}$, where each symbolic value of s maps onto a unique integer in P.

We can use multiple-valued decision diagrams (MDD's) [35] to manipulate multiple-valued functions just like BDD's for Boolean functions. Furthermore, similar operations, such as existential, and universal quantification, and substitution, etc., are well defined in the MDD framework [35]. In the sequel, we just use the term BDD to interchangeably refer to characteristic functions of multiple-valued variables.

2.6 Implicit State Reachability Computation

The reachable states can be computed efficiently using implicit state enumeration techniques introduced by Coudert *et al.* [11]. These techniques are widely used in FSM verification [11, 12, 37], and in design verification [6, 36]. This approach is based on representing a set of states by a characteristic function which can be manipulated effectively using BDD's. In the following, we represent a finite state machine implicitly by a characteristic function using BDD's.

Definition 6 *The* **transition relation** *of a finite state machine $M = (I, O, Q, \delta, \lambda, q_0)$ is a function $T : I \times Q \times Q \times O \rightarrow B$ such that $T(i, p, n, o) = 1$ if and only if state n can be reached in one state transition from state p and produce output o when input i is applied.*

3 Previous Work

First we discuss input don't care sequences. Consider the cascade machine $M_1 \rightarrow M_2$ in Figure 1. Part of an STG of M_2 is shown in Figure 3. Consider the transitions from s_1 to s_2 and from s_2 to s_3. When M_2 does not interact with other machines, $(s_1 s_2 s_3)$ is a possible sequence of transitions. However, when M_2 is driven by M_1, this sequence may not happen. Thus, we can regard the input string $(a_1 a_2)$ causing these transitions as a don't care sequence starting at s_1. We cannot determine whether $(s_1 s_2 s_3)$ is a possible sequence by looking at M_2 in isolation, but such information is useful; we may get a smaller number of states for M_2 in the cascade $M_1 \rightarrow M_2$, than when M_2 is in isolation.

Unger [38] observed that when one FSM is driven by another, there are more *input-incompletely-specified don't cares (unspecified transitions)* than for an isolated machine, and proposed a method to approximate and exploit a subset of this information, i.e., limited length of input don't care sequences are considered.

3.1 K-N Procedure

Kim and Newborn [22] proposed an elegant approach which solves the problem of computing input don't care sequences for a driven machine in a cascade. The procedure is :

1. Construct an NFA $\mathcal{A}'$ to accept the language produced by machine M_1. This can be achieved by removing the input part in the STG of M_1, and assigning every state of M_1 as a final state. For a state s, if there are output symbols not

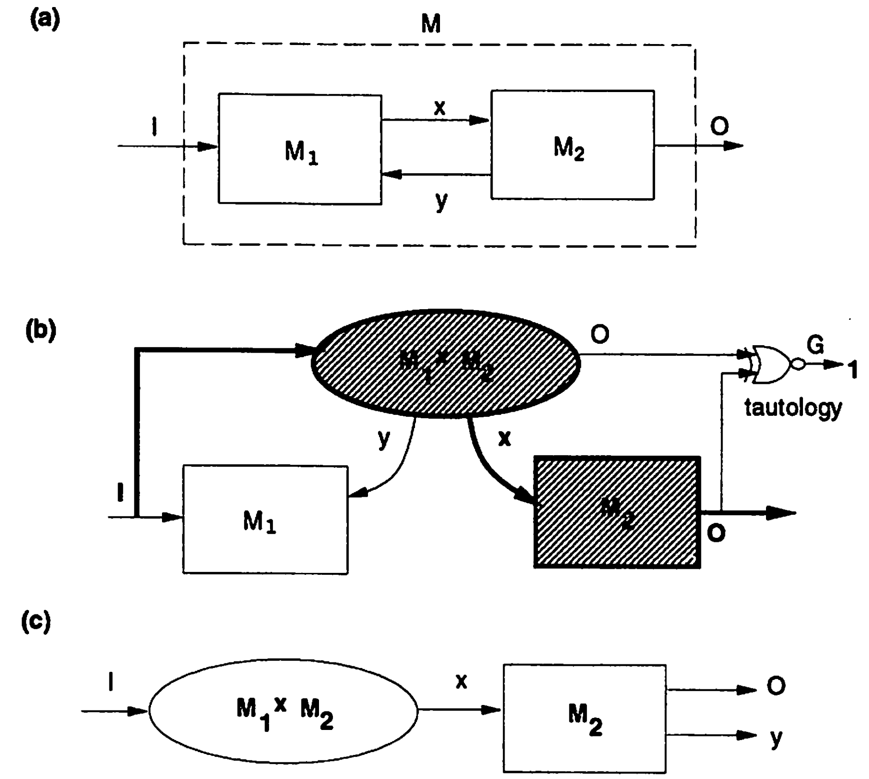

Figure 4: (a). A two-way-communication circuit $\mathcal{N}_2$. (b). $\mathcal{N}_2'$: An equivalent one-way-communication FSM network to $\mathcal{N}_2$. (c). An equivalent one-way-communication circuit for computing input don't care sequences of M_2.

emitted from it, a transition is inserted from s to the *dead state* d with those symbols. The dead state d is the only non-accepting state. Thus $\mathcal{A}'$ is completely specified but non-deterministic.

2. Convert $\mathcal{A}'$ to a minimized completely specified DFA $\mathcal{A}$. This can be done by using the subset construction [29] and then state minimization for DFA [19]. Note that efficient ($n \log n$) state minimization for completely specified machines can be used, since the subset construction produces a completely specified deterministic machine. (Actually, state minimization is not necessary here but desirable since this is a $n \log n$ operation.)

3. A modified machine M_2' is constructed as follows : construct $M_2 \times \mathcal{A}$ and delete any transition to a state that contains the dead state d in its subset. M_2' is deterministic but possibly incompletely specified. Thus state minimizers for ISFSM's must be used to minimize M_2'.

The key idea is that sequences not produced by M_1 are the input don't care sequences for M_2, and these are converted into unspecified transitions of a modified machine M_2'. The K-N procedure indeed captures *all* input don't care sequences for M_2. It can be seen that methods in [38, 15] are a subset of this, they are explicit and length limited, whereas the K-N procedure is implicit and not length limited.

3.2 Input Don't Care Sequences in FSM Networks with Arbitrary Topologies

Intuitively, computation of input don't care sequences for a component in an FSM network of arbitrary topology is much more complicated than for a cascade circuit. Nevertheless, it is not theoretically harder.

Wang and Brayton [40] demonstrated that the problem of computing and exploiting input don't care sequences for a component in an FSM networks with an arbitrary topology can be reduced to one for a cascade circuit. They derive an **abstract driving machine** in the computation of input don't care sequences in an FSM network. The pictorial explanation is shown in Figure 4. For example, the abstract driving machine to M_2 in Figure 1 is M_1, while the abstract driving machine to M_2 in Figure 4(a) is $M_1 \times M_2$. The abstract driving machine for a component in an FSM network is the composite machine of all components in this network, i.e., the network itself. However, if a component M_2 is in a one-way communication with other components as in Figure 1, its abstract driving machine will reduce to M_1. Then steps 1 and 2 of the K-N procedure can be used to compute the exact input don't care sequences. The correctness of the exploitation of input don't care sequences was proved in [40]. Therefore, with the notion of the abstract driving machine, the K-N procedure works in general FSM networks. In addition, in [40] an efficient implementation of the K-N procedure using BDD's was proposed.

An abstract driving machine itself may be a non-deterministic FSM which can be a collection of permissible FSM's; however, this does not affect the computation and exploitation of input don't care sequences in the K-N procedure. Consequently, we may start with a network of machines some of which are non-deterministic (e.g., the environment may be one of the machines).

4 Practical Issues of the K-N Procedure

Unfortunately, the worst case complexity for the transformation from an NFA to a DFA (i.e., from $\mathcal{A}'$ to $\mathcal{A}$) is exponential in the number of states [29]. Further, even if $\mathcal{A}$ can be built in a reasonable time, the resultant product machine $M_2{}'$ may have a large number of states before state minimization. Therefore, there are two purposes for approximations of input don't care sequences. (1) Control the possible state explosion in the subset construction. (2) The resultant modified machine $M_2{}'$ should be small enough for state minimizers.

Consider the cascade machine $M_1 \rightarrow M_2$ in Figure 1. Note that M_1 may be the abstract driving machine for M_2. Let output sequences produced by M_1 be $\mathcal{L}(M_1^o)$, a regular language over alphabet I_2. For computing and exploiting only a subset of input don't care sequences, any regular language $\mathcal{L}'$ such that

$$\mathcal{L}(M_1^o) \subseteq \mathcal{L}' \subseteq I_2{}^* \qquad (1)$$

gives rise to a feasible subset $\overline{\mathcal{L}'}$ of input don't care sequences. Approximation methods in [30, 40] can be used.

How much approximation is needed hinges on the ability of state minimizers, since approximation needs to be performed so that the state minimization of $M_2{}'$ can be completed. Thus even if the exact input don't care sequences can be efficiently computed in an FSM network, after approximation it may turn out that very limited information can be actually exploited. This becomes a problem when we consider optimization of large FSM networks.

Moreover, even if a circuit implementation is given as the starting point, the K-N procedure will completely ignore it. Further, state minimality is only a heuristic and does not imply that the resultant logic circuit after state encoding is minimized. In fact, it is just regarded as a good starting point for state encoding. In this sense, state minimization techniques are 'distant' to optimality at the logic level. In comparison, optimization techniques in sequential circuits which work directly at the logic level, are more mature in terms of their efficiency and effectiveness; hence the size of circuits they can handle is larger. Moreover, they work much closer to the optimality at the logic level. However, computing input don't care sequences requires symbolic manipulation. In the next section, we propose an approach to exploit input don't care sequences directly at the logic level.

5 Exploitation of Input Don't Care Sequences

5.1 Logic Optimization of the Driven Machine in a Cascade Circuit

Consider a cascade circuit $M \equiv M_1 \rightarrow M_2$ as shown in Figure 5(a), where M_1 is the driving machine, and M_2 the driven machine. M_1 and M_2 are logic implementations. Our goal is to optimize M_2 while the behavior of M_1 is kept unchanged.

We consider three sets of logic optimization techniques.

1. **Don't-care-based approach.** This is the traditional approach [3, 1], and widely used in logic synthesis. This set of techniques includes *kernel extraction*, *re-substitution*, *elimination*, and *node simplification* [3]. These techniques normally can get a large improvement from a given initial circuit [3, 33, 34]. Besides the node simplification method, another powerful approach to exploit don't cares is Muroga's *transduction methods* [28]. Output values not generated by M_1 are external don't cares to M_2. These can be exploited using node simplification technique in [32, 27] to further the improvement.

2. **Sequential ATPG-based techniques.** This is a greedy method and needs a good starting point, so the first set of techniques may be employed first. There are many existing efficient and effective techniques based on *sequential ATPG*

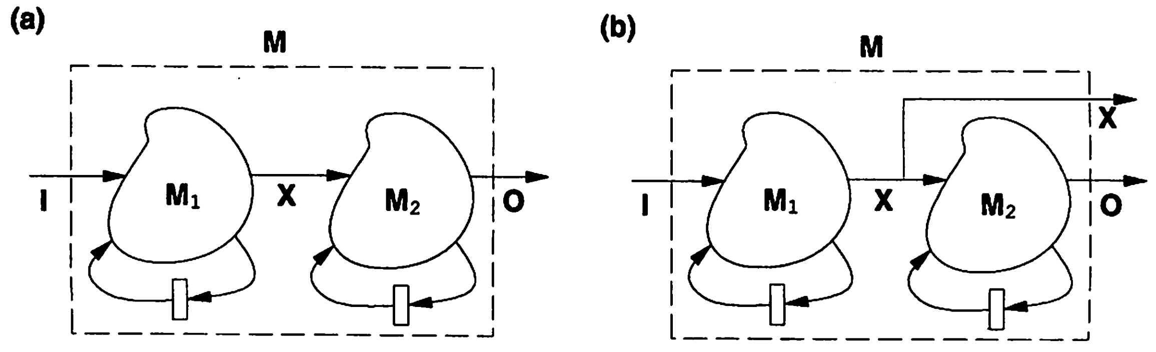

Figure 5: (a). $M \equiv M_1 \rightarrow M_2$, where I is input and O output. (b). M, where I is input, and X, O outputs.

to improve the quality of circuits. For example, techniques in [9, 10] are based on *redundancy removal*. Entrena and Cheng [17] proposed an approach based on *redundancy addition and removal*, and demonstrated encouraging results. Their method cleverly adds some redundancies in the Boolean network so that more redundancies can be removed later. This idea is similar to transduction methods in [28] but for sequential circuits. Later, techniques proposed in [8, 23] for combinational circuits are extensions of these ideas.

We require that the behavior of M_1 be kept unchanged and X the only communicating variable between M_1 and M_2. That is, we are only concerned about the logic optimization of M_2. Therefore, this set of techniques needs to be modified to optimize M_2 only. For example, a simple modification to the redundancy removal method is to set X as observable outputs. This guarantees that the behavior of M_1 is the same as before. That no internal nodes in M_1 are allowed to connect to M_2 guarantees that X is the only communicating variable. With this setting, we can perform redundancy removal on M and then disassemble M_1 from M afterwards. This results in an optimized circuit of M_2 using *redundancy removal*. This is illustrated in Figure 5(b). Note that M_1 need not be deterministic. If it is non-deterministic, it can be input determinized by adding additional inputs controlling the non-determinism. The resulting network be fed into SIS.

3. **Re-encoding and re-synthesis.** After a few iterations of the above two optimization techniques, M_2 may have been simplified to a reasonable size for re-encoding, e.g., the number of states may have been reduced. We may then be able to re-encode and re-synthesize M_2. There are good encoding algorithms for both two-level and multi-level logic implementations [39, 25, 16] when the circuits are reasonably small. Although state encoding does not guarantee more improvement than previously optimized results, it is likely to be a new good starting point for performing re-synthesis using the above two techniques.

From the K-N procedure [22], the flexibility of implementing M_2 comes from input don't care sequences. In fact, input don't care sequences are exploited by the above techniques in different ways. In the next subsection, this is discussed in more detail.

5.2 External Don't Cares and Sequential Redundancies vs. Input Don't Care Sequences

Let output sequences generated by M_1 be $\mathcal{L}(M_1^o)$. Based on the K-N procedure, the flexibility of M_2 when cascaded by M_1 is due to output sequences not generated by M_1, i.e., $\overline{\mathcal{L}(M_1^o)}$. In the following, we investigate the relationship between this flexibility and logic optimization techniques as described in Section 5.1.

We consider two exploitation techniques. Node simplification can exploit external don't cares both effectively and efficiently [32]. Output values not generated by M_1 are external don't cares to M_2. Therefore, node simplification only exploits partial flexibility; nevertheless, when combined with other optimization techniques, such as kernel extraction, elimination etc., we can efficiently get a good starting point for sequential ATPG-based techniques. Let the transition relation of M_1 be $T_1(i, p_1, n_1, x)$, the output values not generated by M_1 are

$$EDC(x) = \overline{\exists_{i,p_1,n_1} T_1(i, p_1, n_1, x)}\,. \qquad (2)$$

Output sequences not generated by M_1 are input don't care sequences to M_2. We prove that they are precisely what is exploited by sequential ATPG-based techniques. Consider the cascade machine in Figure 5(a). We assume that M_1 is deterministic.

Lemma 5.1 *For a stuck-at fault f in M_2, if there is a test sequence from I, then there is a test sequence $S \in \mathcal{L}(M_1^o)$ from X.*

Proof M_1 is a logic implementation, so the behavior is deterministic. Therefore, for an input sequence in I, M_1 produces a unique sequence $S \in \mathcal{L}(M_1^o)$ in X. ■

Lemma 5.2 *For a stuck-at fault f in M_2, if there is a test sequence $S \in \mathcal{L}(M_1^o)$ from X, then there exists a test sequence from I.*

Proof Since M_1 is deterministic, for an output sequence $S \in \mathcal{L}(M_1^o)$, there must exist an input sequence which drives M_1 to produce S. ■

Theorem 5.3 *Let $\mathcal{A}$ be a finite automaton which accepts $\mathcal{L}(M_1^o)$, i.e., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(M_1^o)$. A stuck-at fault f in M_2 is redundant with respect to input sequences $\mathcal{L}(\mathcal{A})$ if and only if it is redundant in $M \equiv M_1 \rightarrow M_2$ as shown in Figure 5(a).*

Proof Directly from Lemma 5.1 and 5.2. ■

Theorem 5.3 implies that sequential redundancies in M_2 when cascaded by M_1 are because there is no test sequence $S \in \mathcal{L}(M_1^o)$ from X. A stuck-at fault f in M_2, may have a test sequence S from X, but if $S \notin \mathcal{L}(M_1^o)$, f becomes untestable, and thus redundant. That is, with limited input sequences, there are likely to be more sequential redundancies in M_2. This demonstrates that sequential ATPG-based techniques in Section 5.1 can directly exploit the flexibility of M_2 coming from input don't care sequences.

Theorem 5.4 *Let M_1' and M_1 both generate the same set of output sequences, i.e., $\mathcal{L}(M_1'^o) = \mathcal{L}(M_1^o)$, and f be a stuck-at fault in M_2. Then f is redundant in $M_1 \rightarrow M_2$ if and only if it is redundant in $M_1' \rightarrow M_2$.*

Proof Directly from Theorem 5.3. ■

Theorem 5.4 implies that any sequential circuit M_1' with its set of output sequences equivalent to $\mathcal{L}(M_1^o)$, can be used to replace M_1 as the driving machine to M_2. This means that we have freedom to select such a machine M_1' that can expedite sequential ATPG-based algorithms, e.g., construction of BDD's etc.

5.3 Logic Optimization of an FSM with Input Don't Care Sequences

Theorems 5.3 and 5.4 lead to a method to optimize a machine M with input don't care sequences which, say, are not accepted by $\mathcal{A}$. Figure 6(a) shows conceptually the specified behavior of M with input sequences $\mathcal{L}(\mathcal{A})$. When an input sequence S is accepted by $\mathcal{A}$, there is a corresponding output sequence. If S is not accepted by $\mathcal{A}$, there is no output.

In practice, $\mathcal{L}(\mathcal{A})$ must be produced by another FSM (deterministic or non-deterministic) such that it can be the set of input sequences to M. Therefore, by the K-N procedure, we can assume that the only non-accepting state of $\mathcal{A}$ is the dead state d, and any transitions to d correspond to the unspecified behavior. Therefore, to exploit this flexibility, we can construct a deterministic FSM D whose set of output sequences is $\mathcal{L}(\mathcal{A})$ as the driving machine. This is shown in Figure 6(b).

There are many construction methods from $\mathcal{A}$ to such a deterministic FSM D. Automaton $\mathcal{A}$ can be deterministic or non-deterministic. We give one simple construction method.

- Case 1: $\mathcal{A}$ is deterministic. The construction is as follows. In automaton $\mathcal{A}$, the dead state d is removed, and any transitions edges to d are deleted. The remaining states in $\mathcal{A}$ are final states. For each transition edge out of a state s, the output o is set equal to input i. For any unspecified input in state s, we arbitrarily assign it to any one of the specified transitions from s with the corresponding output. The resultant FSM D is completely specified and deterministic, and its set of output sequences is equivalent to $\mathcal{L}(\mathcal{A})$.

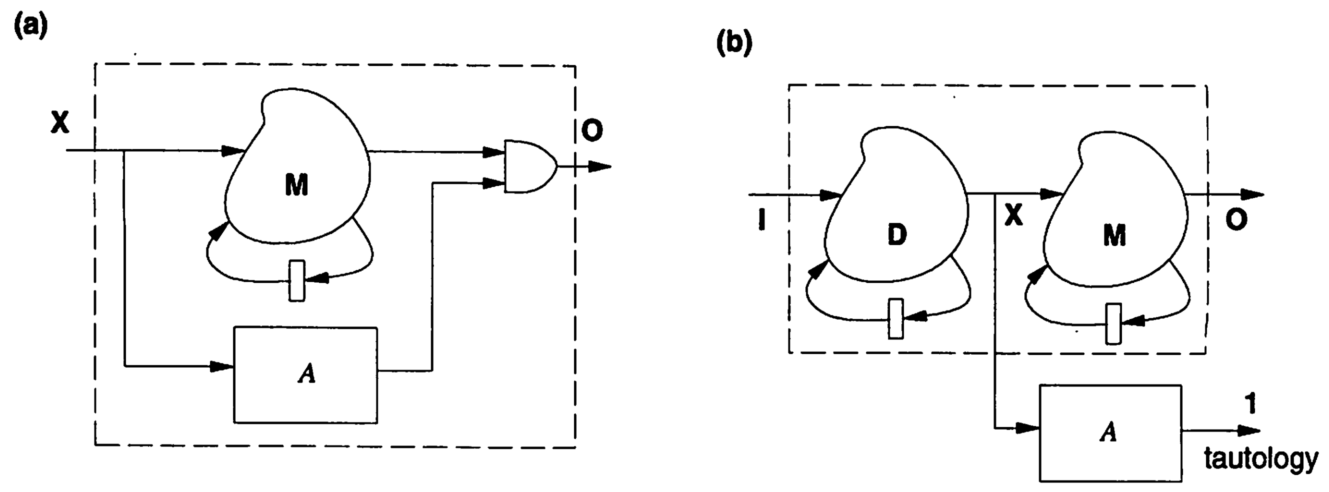

Figure 6: (a). The specified behavior of M with restricted input sequences $\mathcal{L}(\mathcal{A})$. (b). Construction of a driving machine D to M. The set of output sequences of D is equivalent to $\mathcal{L}(\mathcal{A})$.

- Case 2: $\mathcal{A}$ is non-deterministic. In automaton $\mathcal{A}$, the dead state d is removed, and any transitions edges to d are deleted. The remaining states in $\mathcal{A}$ are final states. Let the maximum number of transitions from any state in $\mathcal{A}$ be L, and $2^k \geq L$. We choose k Boolean variables as new inputs to machine D. For each transition edge out of a state s, its output value is set to be its old input value, and then a distinct value from B^k is assigned to be the new input value. Afterwards, for any unspecified value in B^k, we arbitrarily assign it to any one of the specified transitions from s with the corresponding output. The resultant FSM D is completely specified and deterministic, and its set of output sequences is equivalent to $\mathcal{L}(\mathcal{A})$. This is a form of "input determinization".

Based on the discussion in Section 5.2, we can assign arbitrary state encoding to FSM D, and then have a logic implementation of FSM D. Subsequently, the methods in Section 5.1 can be employed to optimize M.

5.4 Logic Optimization of FSM Networks

Based on the discussions in Sections 5.1, 5.2 and 5.3, we propose an approach for logic optimization of a component in a general FSM network. Given a logic implementation of a component M_2 in an FSM network, our procedure works as follows.

1. Construct the abstract driving machine M_1, same in [40]. It may be non-deterministic.
2. Construct an NFA $\mathcal{A}'$ to accept the language produced by machine M_1, as in the first step of the K-N procedure.
3. As described in Section 5.3, construct a deterministic FSM D whose set of output sequences is equivalent to $\mathcal{L}(\mathcal{A}')$. Then derive a logic implementation of FSM D.
4. Use the various optimization techniques in Section 5.1 to optimize M_2.

Note that if M_1 is deterministic, it can be handled by sequential optimization, hence we may use it directly in step 3.

Our approach can be regarded as a generalization of the K-N procedure. The advantages are:

- No subset construction is needed.
- Input don't care sequences are exploited using existing state-of-the-art sequential optimization techniques. Most can deal with larger sequential circuits and produce good results. In comparison, state minimizers for ISFSM's can only handle much smaller circuits.
- Circuit implementation objectives, such as area, timing, power etc., can be considered during the exploitation of input don't care sequences. In comparison, it is much harder to optimize these objectives at the symbolic level.

circuit	I	X	O	S_1	S_2	M_2 initial literals	K-N procedure				Our procedure opt + red		
							SM		enc + opt				
							S_2'	cpu	final lits	cpu	S_2'	final lits	cpu
ex1-s510	9	19	7	20	47	248	7	0.2	95	8.9	12	**37**	20.4
ex7-dk16	2	2	3	8	27	348	15	0.1	**63**	3.7	16	75	19.0
s820-s510	18	19	7	25	47	248	8	0.3	39	2.4	16	**34**	29.2
s832-s510	18	19	7	25	47	248	4	0.1	**14**	1.0	5	15	17.3
bbsse-keyb	7	7	2	16	19	314	18	1.3	193	53.2	18	**170**	40.3
keyb-dk16	7	2	3	19	27	348	19	0.1	120	11.7	20	**94**	41.4
s510-keyb	19	7	2	47	19	314	15	7.5	178	41.5	16	**93**	107.4
sand-ex1	11	9	19	32	20	280	8	83.8	239	59.3	9	**66**	552.5
bbsse-planet	7	7	19	16	48	617	-	spaceout	-	-	44	**454**	169.6
planet-s510	7	19	7	48	47	248	-	timeout	-	-	35	**165**	534.7
s510-planet	19	7	19	47	48	617	-	timeout	-	-	45	**441**	438.0
sand-styr	11	9	10	32	30	596	-	timeout	-	-	27	**375**	405.6

Table 1: Experimental results of one-way-communcation circuits.

M_1 (M_2):	driving machine (driven machine).
I, O, X:	number of PI's, PO's, interacting signals of $M_1 \rightarrow M_2$, respectively.
S_1 (S_2):	number of states of M_1 (M_2), respectively.
M_2 **initial literals:**	number of literals (in factored form) of the initial M_2.
final lits:	number of literals (in factored form) of M_2 after optimization.
S_2':	number of states of M_2 after exploiting input don't care sequences.
SM:	result for STAMINA.
enc + opt:	encoded by JEDI and then optimizd by running `script.rugged` twice.
opt + red:	optimized by runnig (`script.rugged` + `red_removal`) twice.
cpu:	CPU time in seconds on a DEC 3000/500 AXP with 160MB memory.
timeout:	set to 20,000 seconds of CPU time.

- If M_2 is exploited using the K-N procedure and then state-encoded, our approach can still be applied.

In addition, based on Equation (1), exact input don't care sequences can be approximated. Many approximation methods for dealing with large FSM networks have been proposed in [40]. There are many other approximation methods, e.g., hiding some state variables from $\mathcal{A}'$, and grouping states of $\mathcal{A}'$ etc. With our approach and powerful state-of-the-art sequential optimization techniques, less approximation is required, i.e., more input don't care sequences can be exploited.

6 Experimental Results

We present preliminary results on small networks. Due to the lack of FSM network benchmark examples, most of the examples here are obtained by connecting FSM's from MCNC benchmarks. These FSM's are completely specified and state-minimal.

Table 1 shows experimental results for some cascade circuits consisting of two FSM's. The circuit topology of these examples is shown in Figure 5(a). We employ both the K-N procedure and our procedure to optimize M_2 and then compare their results. The logic optimizer used is SIS [34], and its standard optimization procedure is called `script.rugged` [33] which includes *kernel extraction*, *re-substitution*, *elimination* and *node simplification*. In this experiment, we use unreachable states as don't cares which are exploited in node simplification. The initial circuit of M_2 is obtained by running `script.rugged` once. For the K-N procedure, we use the *bounded subset construction* in [40]; the bound on the number of states is set to 64. The state minimizer used here is STAMINA [18]. Afterwards, the state-minimized machine is encoded using JEDI [25], and then optimized by running `script.rugged` twice.

Our procedure takes the given circuit implementation of M_2 as the starting point. External don't cares, i.e., output values not generated by M_1, are extracted and exploited in `script.rugged`. This corresponds to the first set of optimization techniques in Section 5.1. We then use the construction in Figure 5(b), and apply the `red_removal` command in SIS to

circuit	M_2					
	after K-N procedure opt + red			after (opt + red) re-encoding + (opt + red)		
	initial lits	final lits	cpu	initial lits	final lits	cpu
ex1-s510	95	74	13.2	37	**26**	9.0
ex7-dk16	63	63	4.0	75	**58**	4.7
s820-s510	39	33	15.8	34	33	17.5
s832-s510	14	**12**	7.1	15	14	7.5
bbsse-keyb	193	152	48.9	170	**124**	20.9
keyb-dk16	120	101	34.9	94	**75**	27.6
s510-keyb	178	108	62.5	93	**70**	14.9
sand-ex1	239	113	323.0	66	**56**	111.2
bbsse-planet *	-	-	-	454	**396**	1243.1
planet-s510	-	-	-	**165**	186	512.2
s510-planet *	-	-	-	441	**377**	1154.9
sand-styr *	-	-	-	375	**312**	1826.3

Table 2: Experimental results for re-encoding and re-synthesis.

M_2:	driven machine.
initial lits:	initial number of literals (in factored form) of M_2.
final lits:	final number of literals (in factored form) of M_2 after optimization.
opt + red:	optimized by runnig (`script.rugged` + `red_removal`) twice.
re-encoding + (opt + red):	encoded using JEDI and then optimizd by running **opt + red**.
cpu:	CPU time in seconds on a DEC 3000/500 AXP with 160MB memory.
	* `full_simplify` in `script.rugged` is limited to 500 seconds.

remove sequential redundancies. This corresponds to the second set of optimization techniques in Section 5.1. The results shown in Table 1 are obtained by running these two sets of optimization techniques twice.

Our procedure achieves better results except for examples `ex7-dk16` and `s832-s510`. For the third set of examples (`bbsse-planet`, `planet-s510`, `s510-planet` and `sand-styr`), STAMINA cannot efficiently exploit input don't care sequences computed by the K-N procedure. As shown in Table 1, not only the factored literal count is reduced, but also the number of states is reduced. Most of CPU time for our procedure is spent either in node simplification or in removing sequential redundancies.

We also conducted the following experiments: (1) Apply our procedure on the results obtained by the K-N procedure. (2) Perform re-encoding and re-synthesis on the results obtained by our procedure. We compare these results in Table 2. For the first experiment, improved results are obtained, but half are still inferior to the results obtained by our procedure alone (see Table 1). For the second experiment, re-encoding and re-synthesis produce the best results except for examples `s832-s510` and `planet-s510`.

In our experiments, only *redundancy removal* is used, and we expect that better results can be achieved if *redundancy addition and removal* in [17] is employed. Our preliminary results indicate that our approach together with the notion of abstract driving machines [40] is promising for computing and exploiting input don't care sequences in general FSM networks.

7 Conclusion

We presented a novel approach to exploit exact or approximate input don't care sequences for a component in an FSM network directly at the logic level. This approach is based on the K-N procedure [22] and the notion of the abstract driving machine [40]. With our approach, many existing sequential logic optimization techniques can be directly applied to exploit input don't care sequences. Logic optimization of large FSM networks can then be achieved. Our preliminary results look promising but more FSM networks must be experimented on.

References

[1] K. Bartlett, R. K. Brayton, G. D. Hachtel, C. R. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level Logic Minimization Using Implicit Don't Cares. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 723–740, June 1988.

[2] K. L. Brace, R. E. Bryant, and R. L. Rudell. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

[3] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: Multiple-Level Logic Optimization System. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 1062–1081, November 1987.

[4] R. K. Brayton and F. Somenzi. Boolean Relations and the Incomplete Specification of Logic Networks. In *VLSI'89*, August 1989.

[5] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *27th ACM/IEEE Design Automation Conference*, pages 46–51, Orlando, June 1990.

[7] E. Cerny and M. A. Marin. An Approach to Unified Methodology of Combinational Switching Circuits. In *IEEE Transactions on Computers*, pages 745–756, August 1977.

[8] S.-C. Chang and M. Marek-Sadowska. Perturb and Simplify : Multi-level Boolean Network Optimizer. In *IEEE International Conference on Computer-Aided Design*, pages 2–5, November 1994.

[9] K.-T. Cheng. On Removing Redundancy in Sequential Circuits. In *28th ACM/IEEE Design Automation Conference*, pages 164–169, June 1991.

[10] H. Cho, G. D. Hachtel, and F. Somenzi. Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 935–945, July 1993.

[11] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[12] O. Coudert and J.C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *IEEE International Conference on Computer-Aided Design*, pages 126–129, November 1990.

[13] M. Damiani. Nondeterministic Finite State Machines and Sequential Don't Cares. In *The European Design and Test Conference*, pages 192–198, February 1994.

[14] M. Damiani and G. De Micheli. Recurrence Equations and the Optimization of Synchronous Circuits. In *28th ACM/IEEE Design Automation Conference*, pages 556–561, June 1992.

[15] S. Devadas. Optimizing Interacting Finite State Machines Using Sequential Don't Cares. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 1473–1484, December 1991.

[16] X. Du, G. D. Hachtel, B. Lin, and A. R. Newton. MUSE : A Multilevel Symbolic Encoding Algorithm for State Assignment. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 28–38, January 1991.

[17] L. Entrena and K.-T. Cheng. Sequential Logic Optimization By Redundancy Addition and Removal. In *IEEE International Conference on Computer-Aided Design*, pages 310–315, November 1993.

[18] G. D. Hachtel, J. K. Rho, F. Somenzi, and R. Jacoby. Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines. In *The European Conference on Design Automation*, 1991.

[19] J. E. Hopcroft. An nlog(n) Algorithm for Minimizing the States in a Finite Automaton. In *The Theory of Machines and Computation, ed. Z. Kohavi*, 1971.

[20] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. A Fully Implicit Algorithm for Exact State Minimization. In *31st ACM/IEEE Design Automation Conference*, pages 683–690, June 1994.

[21] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Minimization of Non-Deterministic FSM's. In *International Workshop on Logic Synthesis*, May 1995.

[22] J. Kim and M. M. Newborn. The Simplification of Sequential Machines With Input Restrictions. In *IEEE Transactions on Computers*, pages 1440–1443, December 1972.

[23] W. Kunz and P. R. Menon. Multi-Level Optimization by Implication Analysis. In *IEEE International Conference on Computer-Aided Design*, pages 6–13, November 1994.

[24] B. Lin, G. de Jong, and Kolks T. Modeling and Optimization of Hierarchical Synchronous Circuits. In *The European Desgin and Test Conference*, pages 144–149, Paris, March 1995.

[25] B. Lin and A. R. Newton. A Generalized Approach to the Constrained Cubical Imbedding Problem. In *International Conference on Computer Design*, October 1989.

[26] B. Lin and F. Somenzi. Minimization of Symbolic Relations. In *IEEE International Conference on Computer-Aided Design*, pages 88–91, November 1990.

[27] B. Lin, H. Touati, and A. R. Newton. Don't Care Minimization of Multi-Level Sequential Logic Networks. In *IEEE International Conference on Computer-Aided Design*, pages 414–417, November 1990.

[28] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction Method - Design of Logic Networks Bases on Permissible Functions. In *IEEE Transactions on Computers*, October 1989.

[29] M. Rabin and D. Scott. Finite Automata and Their Decision Problems. In *IBM Journal of Research and Development*, pages 114–125, 1959.

[30] J. K. Rho, G. D. Hachtel, and F. Somenzi. Don't Care Sequences and the Optimization of Interacting Finite State Machines. In *IEEE International Conference on Computer-Aided Design*, pages 418–421, November 1991.

[31] H. Savoj and R. K. Brayton. Observability Relations and Observability Don't Cares. In *IEEE International Conference on Computer-Aided Design*, pages 518–521, November 1991.

[32] H. Savoj, R. K. Brayton, and H. Touati. Extracting Local Don't Cares for Network Optimization. In *IEEE International Conference on Computer-Aided Design*, pages 514–517, November 1991.

[33] H. Savoj, H.-Y. Wang, and R. K. Brayton. Improved Scripts in MIS-II for Logic Minimization of Combinational Circuits. In *International Workshop on Logic Synthesis*, May 1991.

[34] E. M. Sentovich, K. J. Singh, L. Lavagno, R. Moon, C. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report Memorandum UCB/ERL M92/41, University of California, Berkeley, May 1992.

[35] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for Discrete Function Manipulation. In *IEEE International Conference on Computer-Aided Design*, pages 92–95, November 1990.

[36] H. Touati, R. K. Brayton, and R. Kurshan. Testing Language Containment for ω-Automata using BDD's. In *Proceedings of ACM/SIGDA International Workshop on Formal Method s in VLSI Designs*, Miami, January 1991.

[37] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *IEEE International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[38] S. H. Unger. *Asynchronous Sequential Switching Circuits*. John Wiley, 1969.

[39] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State Assignment for Optimal Two-level Logic Implementations. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 905–924, September 1990.

[40] H.-Y. Wang and R. K. Brayton. Input Don't Care Sequences in FSM Networks. In *IEEE International Conference on Computer-Aided Design*, pages 321–328, November 1993.

[41] H.-Y. Wang and R. K. Brayton. Permissible Observability Relations in FSM Networks. In *31st ACM/IEEE Design Automation Conference*, pages 677–683, June 1994.

[42] Y. Watanabe and R. K. Brayton. The Maximum Set of Permissible Behaviors for FSM Networks. In *IEEE International Conference on Computer-Aided Design*, pages 316–320, November 1993.

[43] Y. Watanabe and R. K. Brayton. State Minimization of Pseudo Non-Deterministic FSM's. In *The European Design and Test Conference*, pages 184–191, February 1994.