Multi-level Logic Optimization of FSM Networks *

Huey-Yih Wang Robert K. Brayton
Department of EECS, University of California, 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 [23, 30, 33, 31]. Conventionally, exploitation of this flexibility relies on state minimizers for incompletely specified FSM's (ISFSM's) or pseudo non-deterministic FSM's (PNDFSM's) [33]. However, state-of-the-art state minimizers cannot handle large ISFSM's or PNDFSM's [12, 14, 34, 8, 15]. In addition, these exploitation techniques are at the symbolic level, not directly at the net-list logic level. We present a general approach to exploit exact or approximate flexibility directly at the net-list logic level, and we demonstrate that many sequential logic optimization techniques can be applied in exploitation. Moreover, we propose a new procedure for input don't care sequences. As a result, both 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 can be effective in reducing the size of a component of an FSM network.

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 [27].

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 be 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 [3] (observability relation [24]) is required to express all its flexibility. Similarly, exploitation of flexibility is important for sequential circuits. Several approaches have been proposed. For example, in [20], unreachable or equivalent states are used in the optimization of an isolated sequential circuit. In this 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 [33] used a pseudo non-deterministic FSM (PNDFSM), called the E-machine, to express all flexibility. Later, [17, 1] proposed different construction methods for the E-machine, but subset construction [22] is required in the general case. The exploitation of the E-machine usually is done by state minimization of PNDFSM's [34, 8, 15].

Another approach (which is an approximate one) is based on the notion of don't care sequences [23]. There are two kinds; input and output don't care sequences. Consider the cascade machine in Figure 1(a), where M_1 is the driving machine and M_2 the driven machine. Kim and Newborn [16] proposed an elegant complete solution. For a two-way-communication network of FSM's, $\mathcal{N}_2$, as shown in Figure 1(b), Wang and Brayton [30] gave an efficient computation, and demonstrated that state minimization for incompletely specified FSM's (ISFSM's) [12, 14] 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 [9] proposed a method to exploit *sequential output don't cares*, and later Rho *et al.* [23] generalized Devadas' procedure to 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 [31].

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 [12, 14] or PNDFSM's [34, 8, 15]. Afterwards, state encoding and sequential optimization techniques are applied to the state-minimized machine. Presently, no existing state minimizer can handle large ISFSM's or PNDFSM's [12, 14, 34, 8, 15]. For example, the computation of input don't care sequences in FSM networks can be efficiently done; however, the exploitation of them using state minimization is difficult [30] since the problem of exact state minimization of ISFSM's is NP-hard. To circumvent this, approximations are required to trade off between quality and efficiency [23, 30]. As a result, much flexibility may be lost.

Furthermore, in contrast to net-list logic optimization techniques for sequential circuits, these algorithms do not use a circuit implementation as the starting point; the exploitation is not performed at the net-list logic level. In terms of efficiency, effectiveness, and the size of circuits, optimization techniques for sequential net-list logic circuits 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 a general approach which takes a circuit implementation as the starting point and computes the flexibility at the symbolic level, but exploitation is directly at the net-list logic level. In addition, we discuss the difficulties in previous approaches [16, 30], and then we propose a new procedure which makes both computation and exploitation of input don't care sequences more efficient and effective. This procedure does not require a subset construction [22] as in the Kim and Newborn's procedure [16]. As a result, this procedure look promising for larger FSM networks. Finally, we give preliminary results on some artificially constructed

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

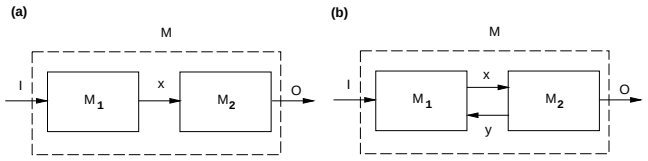

Figure 1: (a) $\mathcal{N}_1$: A cascade circuit of two FSM's. (b) $\mathcal{N}_2$: A two-way-communication network of FSM's.

FSM networks.

2 Preliminaries

2.1 Finite Automata and Finite State Machines

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.

A finite state machine (FSM), M, is a quintuple (I, O, Q, T, q_0) where I is a finite input alphabet, O a finite output alphabet, Q a finite set of states, T, the transition relation, a finite subset of $I \times Q \times Q \times O$, and q_0 the initial state. An FSM can be represented by a state transition graph (STG). An FSM is *input-complete* if for all input symbols from every state, the transitions are defined; otherwise, it is *input-incomplete*. A *deterministic* FSM (DFSM) is an FSM in which for all transitions $(i, p, n, o) \in T$, (i, p) is associated with a unique (n, o). Otherwise, an FSM is called a *non-deterministic* FSM (NDFSM). A *pseudo non-deterministic* FSM (PNDFSM) is an input-complete FSM and in which for all transitions $(i, p, n, o) \in T$, (i, p, o) is associated with a unique next state n. A *completely specified* FSM (CSFSM) is an FSM which is input-complete and deterministic. An *incompletely specified* FSM (ISFSM) is either an NDFSM or an input-incomplete FSM, and in which for all transitions $(i, p, n, o) \in T$, (i, p) is associated with a unique next state n. A CSFSM is of *Moore* type if the output value does not depend on inputs, and *Mealy* otherwise. A *cascade* of FSM's M_1 and M_2, denoted $M_1 \rightarrow M_2$, is shown in Figure 1(a). M_1 is called the *driving machine*, M_2 the *driven machine*.

The single stuck-at fault model assumes that a single fault 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 a CSFSM. A sequence of input vectors is a *test* for a fault f of M if it causes output values different from those of the fault-free machine when it is applied to machine M with the single fault f starting from the reset state. If a fault f has no test, it is *redundant*.

2.2 Set Computation and Operators

We denote B designate as the set $\{0, 1\}$. 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 1 *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 :*

$$\exists_{x_i} f = f_{x_i} + f_{\overline{x_i}} , \qquad \exists_x f = \exists_{x_1} ... \exists_{x_k} f .$$

Definition 2 *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 :*

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

Reduced ordered binary decision diagrams (BDD's) [4] are well suited to represent the characteristic functions of subsets of a set, and efficient algorithms exist to manipulate them to perform all standard Boolean operations. As a result, the above set operations can be done efficiently.

The reachable states can be computed efficiently using implicit state enumeration techniques introduced by Coudert *et al.* [7]. 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 3 *The* **transition relation** *of a finite state machine $M = (I, O, Q, T, q_0)$ 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.*

The compatible projection operator is defined in [19] and can be manipulated efficiently using BDD's.

Definition 4 *Let $y_1 \prec \cdots \prec y_n$ be an ordering of Boolean variables. The distance between two vertices $\alpha, \beta \in B^n$ is defined as [7, 28]* $d(\alpha, \beta) = \sum_i^n |\alpha_i - \beta_i| 2^{n-i}$.

Using the above distance metric, a total ordering of all the vertices of a Boolean space relative to some reference vertex α can be defined; $order(x) = d(\alpha, x)$.

Definition 5 *Given $\alpha \in B^n$, $\mathcal{C} \subseteq B^n$, the* **closest interpretation** *of α in $\mathcal{C}$ for a given variable ordering is defined as [19]*

$$\mathcal{P}(\alpha, \mathcal{C}) = argmin_{x \in \mathcal{C}} d(\alpha, x) .$$

The closest interpretation $\mathcal{P}$, relative to a reference vertex α, is unique for a given variable ordering.

Definition 6 *For a relation, $\mathcal{R} \subseteq B^r \times B^n$, and $\alpha \in B^n$, the closest interpretation of α relative to $\mathcal{R}$ (called* **compatible projection** *in [19]) is :*

$$\perp(\alpha, \mathcal{R}) = \{(x, y) | (x, y) \in \mathcal{R}, y = \mathcal{P}(\alpha, \mathcal{R}_x)\} .$$

Conceptually, the $\perp$ operator selects a unique minterm y for each minterm x defined in the relation $\mathcal{R}$. Thus, $\perp(\alpha, \mathcal{R})$ results in the characteristic function of a function defined on the domain $\exists_y \mathcal{R}(x, y)$; $\perp(\alpha, \mathcal{R}) : \exists_y \mathcal{R}(x, y) \times B^n \rightarrow B$.

3 Logic Optimization of FSM Networks : General Approach

3.1 Permissible Behaviors

Current approaches for synthesizing a component in an FSM network have two steps : (1) computing the flexibility (i.e., a collection of permissible implementations), and (2) finding a permissible implementation, with respect to some optimization objective, using the flexibility. There are many studies [16, 9, 23, 30, 33, 34, 8, 31, 15, 17, 1] in computing and exploiting the flexibility. A key idea of these approaches is to implicitly express a collection of permissible implementations of a component in an FSM network using some variants of FSM's. A permissible implementation is called a *permissible behavior*. Watanabe and Brayton [33] demonstrated that an E-machine (a PNDFSM) can express the whole set of permissible behaviors of a component in an FSM network and then proposed a method to compute it. Later, [17, 1] proposed different methods for computing the E-machine. The fact that the set of permissible behaviors due to input don't care sequences can be expressed using an ISFSM was demonstrated in [30] That is, a PNDFSM or an ISFSM can be used to implicitly express a collection of permissible behaviors. They are defined as follows and are consistent with the definitions of [33, 30].

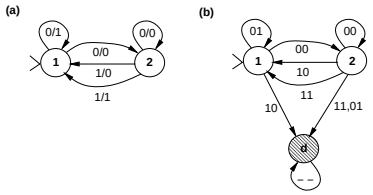

Figure 2: (a) A PNDFSM M_1. (b) $\mathcal{A}^{M_1}$.

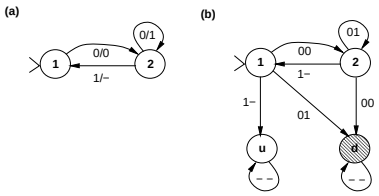

Figure 3: (a) An ISFSM M_2. (b) $\mathcal{A}^{M_2}$.

Definition 7 *The* **defined behavior** *of an FSM $M = (I, O, Q, T, q_0)$, a set of sequences $\subseteq (I \times O)^*$, is the language of a finite automaton $\mathcal{D}^M = (K, \Sigma, \delta, q_0, F)$, where $K = Q \cup \{d\}$, $\Sigma = I \times O$, $F = Q$, and $\delta = \{(p, (i, o), n) \mid (i, p, n, o) \in T\} \cup \{(p, (i, o), d) \mid (i, p, o)$ are unspecified in $T\} \cup \{(d, (i, o), d) \mid (i, o) \in I \times O\}$. The defined behavior of an FSM M is denoted as $\mathcal{L}(\mathcal{D}^M)$. In general, $\mathcal{D}^M$ may be an NFA.*

Definition 8 *The set of permissible behaviors expressed by a PNDFSM (or a CSFSM) $M = (I, O, Q, T, q_0)$, a set of sequences $\subseteq (I \times O)^*$, is the language of a finite automaton, $\mathcal{A}^M$, where $\mathcal{A}^M = \mathcal{D}^M$. That is, the set of permissible behaviors expressed by a PNDFSM M is equivalent to its defined behavior. Note that $\mathcal{A}^M$ is a DFA.*

The construction of $\mathcal{A}^M$ of a PNDFSM (or a CSFSM) M can be directly derived from M. The alphabet of $\mathcal{A}^M$ is $I \times O$. Every state of M is a final state, and the initial state is the same. For those (i, o) symbols which do not have specified transitions from a state p, a transition is added from state p to the dead state d, the only non-accepting state. This construction is pictorially explained in Figure 2.

Definition 9 *The set of permissible behaviors expressed by an ISFSM $M = (I, O, Q, T, q_0)$, a set of sequences $\subseteq (I \times O)^*$, is the language of a finite automaton $\mathcal{A}^M = (K, \Sigma, \delta, q_0, F)$, where $K = Q \cup \{u, d\}$, $\Sigma = I \times O$, $F = Q \cup \{u\}$, and $\delta = \{(p, (i, o), n) \mid (i, p, n, o) \in T\} \cup \{(p, (i, o), u) \mid ((i, p)$ are unspecified in $T) \wedge (o \in O)\} \cup \{(u, (i, o), u) \mid (i, o) \in I \times O\} \cup \{(p, (i, o), d) \mid ((i, p)$ are specified in $T) \wedge ((i, p, o)$ are unspecified in $T)\} \cup \{(d, (i, o), d) \mid (i, o) \in I \times O\}$. Note that $\mathcal{A}^M$ is a DFA.*

The set of permissible behaviors expressed by an ISFSM may not be equivalent to the defined behavior of an ISFSM, since an ISFSM may be input-incomplete, i.e., having unspecified transitions. By definition, an unspecified transition never happens, so it can be arbitrarily associated with any output value from that transition on. Consequently, unspecified transitions should be interpreted to be permissible. We pictorially explain the construction in Definition 9 in Figure 3. For the rest of the paper, the set of permissible behaviors expressed by a PNDFSM M (a CSFSM, or an ISFSM) is denoted as $\mathcal{L}(\mathcal{A}^M)$.

Definition 10 *A CSFSM R is a* **permissible realization** *(***implementation***) of a PNDFSM M (or an ISFSM) if $\mathcal{L}(\mathcal{A}^R) \subseteq \mathcal{L}(\mathcal{A}^M)$.*

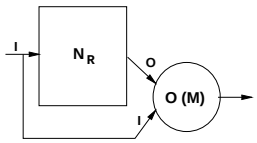

Figure 4: The FSM observability network $\mathcal{N}$ of $\mathcal{N}_R$ with a flexibility M.

Theorem 3.1 [1] *Let $M = (I, O, Q, T, q_0)$ be a PNDFSM. An FSM $R = (I, O, Q, T', q_0)$, where $T'(i, p, n, o) = \perp(\alpha_0, T(i, p, n, o))$ and α_0 is a minterm in (n, o) space, is a permissible realization of M.*

In this section, we do not concentrate on computing the flexibility which can be found in [30, 33, 17, 1]. After computing the flexibility, the optimization problem reduces to finding a permissible realization from a PNDFSM or an ISFSM with respect to some objective, such as area, testability, timing, power and etc. Conventional approaches employ state minimizers for PNDFSM's or ISFSM's [12, 14, 34, 8, 15] to explore such a permissible realization. However, no existing state minimizer can efficiently handle large PNDFSM's or ISFSM's. For example, the problem for exact state minimization of ISFSM's is NP-hard. To trade off between quality and efficiency, approximations on PNDFSM's and ISFSM's may be needed; consequently, much flexibility may be lost. How much approximation is needed hinges on the ability of state minimizers, since approximation needs to be performed so that the state minimization can be completed. If too much approximation is performed, it may turn out that very limited information can be actually exploited. This becomes a problem when we consider optimization of large FSM networks.

Furthermore, these state minimizers explore a permissible realization at the symbolic level where the logic implementation objective is hard to estimate. The optimization objective of standard state minimizers is to find an FSM with the minimum number of states. State minimality is only a heuristic and does not imply that the resultant logic circuit after state encoding is optimized. In fact, it is just regarded as a good starting point for state encoding. In addition, if a circuit implementation is given as the starting point which may be useful for further optimization, state minimizers will completely ignore it. In this sense, state minimization techniques are 'distant' to optimality at the net-list logic level. In comparison, sequential optimization techniques at the net-list 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 net-list logic level. In the rest of this section, we propose a general approach to exploit PNDFSM's and ISFSM's directly at the net-list logic level.

3.2 FSM Observability Networks

In [24], the observability network $\mathcal{N}$ of a combinational Boolean network $\mathcal{N}'$ was proposed for logic optimization of $\mathcal{N}'$ with a flexibility, say $\mathcal{O}(i, o)$ which is a Boolean relation (observability relation). Initially, $\mathcal{N}'$ is compatible with $\mathcal{O}(i, o)$. The **observability network** $\mathcal{N}$ is a derived network of $\mathcal{N}'$ by adding a Boolean node $\mathcal{O}$ to $\mathcal{N}'$; the logic function of node $\mathcal{O}$ is equal to $\mathcal{O}(i, o)$. With the notion of the observability network, optimization of $\mathcal{N}'$ with flexibility $\mathcal{O}(i, o)$ is reduced to optimization of $\mathcal{N}$. As a result, no special logic optimization techniques are required to optimize a combinational circuit $\mathcal{N}'$ with flexibility $\mathcal{O}(i, o)$, e.g., observability don't cares of nodes in $\mathcal{N}'$ with respect to a flexibility $\mathcal{O}(i, o)$ can be computed from $\mathcal{N}$. In the following, we generalize the notion of observability network to sequential circuits.

Definition 11 *Let $M = (I, O, Q, T, q_0)$ be a PNDFSM (an ISFSM), and the DFA accepting the set of permissible behaviors ex-*

[1] Detailed proofs of the theorems in this paper are given in [32].

pressed by M be $\mathcal{A}^M = (K, \Sigma, \delta, q_0, F)$ as defined in Definition 8 (Definition 9). The **observability FSM** *of M is a CSFSM $\mathcal{O}(M) = (\Sigma, B, K, T', q_0)$, where $B = \{0, 1\}$ and $T' = \{((i, o), p, n, 1) \mid (i, p, n, o) \in T, n \in F\} \cup \{((i, o), p, n, 0) \mid (i, p, n, o) \in T, n \notin F\}$.*

Let R be a permissible realization of M, i.e., R is a CSFSM and $\mathcal{L}(\mathcal{A}^R) \subseteq \mathcal{L}(\mathcal{A}^M)$, and $\mathcal{N}_R$ be a net-list logic implementation (i.e., a Boolean network) of R. The **FSM observability network** $\mathcal{N}$ is derived by adding an additional Boolean node $\mathcal{O}$ to $\mathcal{N}_R$, and $\mathcal{O}$ is a logic implementation of observability FSM of M, $\mathcal{O}(M)$. This is shown in Figure 4. Observability FSM $\mathcal{O}(M)$ is analogous to the Boolean relation (observability relation) in the case of combinational circuits. $\mathcal{N}$ has many interesting properties that can be used for optimization and verification of $\mathcal{N}_R$ with a flexibility M. This is stated in the following theorem.

Theorem 3.2 *The output of $\mathcal{N}$ is a tautology if and only if R is a permissible realization of M, i.e., $\mathcal{L}(\mathcal{A}^R) \subseteq \mathcal{L}(\mathcal{A}^M)$.*

Theorem 3.2 gives an approach to explore a logically optimized implementation of M. Consider the FSM observability network of M in Figure 4. Our goal is to optimize $\mathcal{N}_R$ while the $\mathcal{O}(M)$ is kept intact. This is the same as for combinational circuits; observability don't cares of nodes in $\mathcal{N}_R$ with respect to the flexibility M can be computed from $\mathcal{N}$.

Theorem 3.3 *A stuck-at-fault f in $\mathcal{N}_R$ is redundant with respect to the flexibility M if and only if f is redundant in $\mathcal{N}$.*

Theorem 3.3 implies that sequential ATPG techniques can directly exploit the flexibility M. A stuck-at-fault f in $\mathcal{N}_R$ may be testable, but if for every test sequence of f its corresponding output sequence is accepted by $\mathcal{A}^M$, then f becomes untestable in $\mathcal{N}$, and thus redundant. Thus, with the flexibility M, it is likely that $\mathcal{N}_R$ has more sequential redundancies. In the following, we consider three sets of logic optimization techniques in more detail.

1. **Don't-care-based approach**. This is the conventional approach [2], widely used in logic synthesis. This set of techniques includes *kernel extraction*, *re-substitution*, *elimination*, and *node simplification* [2]. These techniques normally can make a large improvement from a given initial circuit [2, 26, 27]. Note that unreachable states can be regarded as don't cares during node simplification. Besides the node simplification method, another powerful approach to exploit don't cares is Muroga's *transduction methods* [21].

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* to improve the quality of circuits. For example, techniques in [5, 6] are based on *redundancy removal*. Entrena and Cheng [11] 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 [21] but for sequential circuits.

3. **Re-encoding and re-synthesis**. After a few iterations of the above two optimization techniques, $\mathcal{N}_R$ 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 $\mathcal{N}_R$. There are good encoding algorithms for both two-level and multi-level logic implementations [29, 18, 10] 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.

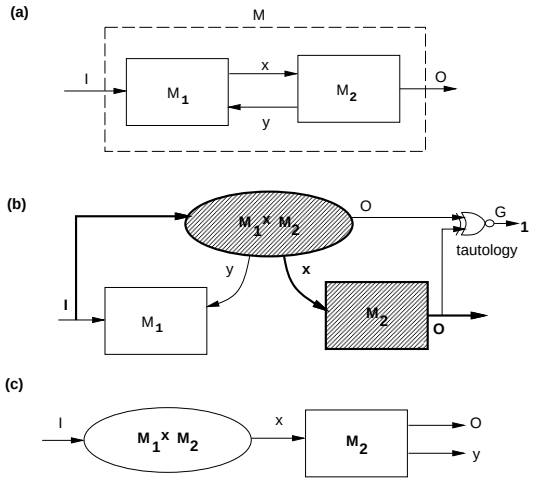

Figure 5: (a). A two-way-communication circuit $\mathcal{N}_2$. (b). $\mathcal{N}_2'$: An equivalent one-way-communication FSM network to $\mathcal{N}_2$. (c). An equivalent one-way-communication circuit for computing input don't care sequences of M_2.

It is easy to design a *local search* (*steepest decent*) algorithm which iteratively runs these logic optimization techniques in some order before a CPU run-time limit is reached or an acceptable logic realization is achieved. Re-encoding and re-synthesis can be used to jump out of a local optimum. Like state minimizers, our approach is also a heuristic of exploiting PNDFSM's and ISFSM's to find a good permissible realization with respect to some optimization objective. However, our approach has the following main advantages.

- The flexibility can be exploited using existing state-of-the-art sequential net-list logic optimization techniques. Most can deal with larger sequential circuits and produce good results. In comparison, state minimizers for ISFSM's or PNDFSM's can only handle much smaller circuits.

- Circuit implementation objectives, such as area, timing, power etc., can be considered directly during the exploitation. In comparison, it is much harder to estimate these objectives at the symbolic level.

- Even if state minimization and state encoding are first used, our approach can still be applied.

The computation step for E-machines [17, 1] and input don't care sequences [16, 23, 30] requires subset construction in the general case. The worst case complexity of subset construction is exponential in the number of states [22]. To trade off between quality and efficiency, approximations can be made using techniques in [23, 30]. However, some flexibility may be lost. With a similar formulation to the FSM observability network, in the next section we present a new procedure for input don't care sequences in general FSM networks. This new procedure does not use an ISFSM to express the flexibility due to input don't care sequences, and no subset construction is required. The exploitation is also performed at the net-list logic level. As a result, both computation and exploitation of input don't care sequences can be made efficient and effective.

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

In this section, we concentrate on both computation and exploitation of input don't care sequences in general FSM networks. First, we give an overview of previous work, and then discuss difficulties in computing and exploiting input don't care sequences. Afterwards, we present a new procedure for input don't care sequences in general FSM networks.

4.1 K-N Procedure

Consider the cascade machine $M_1 \rightarrow M_2$ in Figure 1(a). Kim and Newborn [16] 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 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 [22] and then state minimization for DFA [13]. 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.

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. State minimizers for ISFSM's are 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.

4.2 Input Don't Care Sequences in General FSM Networks

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 [30] 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 5. For example, the abstract driving machine to M_2 in Figure 1(a) is M_1, while the abstract driving machine to M_2 in Figure 5(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(a), 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 [30]. Therefore, with the notion of the abstract driving machine, the K-N procedure works in general FSM networks. In addition, in [30] 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).

4.3 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 $\mathcal{A}$) using subset construction is exponential in the number of states [22]. 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

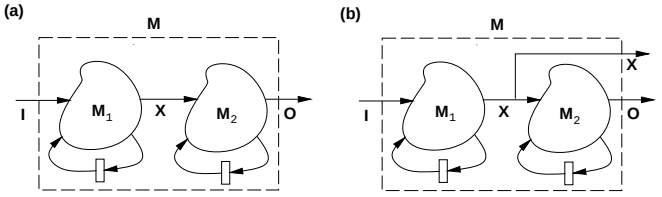

Figure 6: (a). $M \equiv M_1 \rightarrow M_2$, where I is input and O output. (b). M, where I is input, and X, O outputs.

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(a). 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 $\mathcal{L}(M_1^o) \subseteq \mathcal{L}' \subseteq {I_2}^*$ gives rise to a feasible subset $\overline{\mathcal{L}'}$ of input don't care sequences. Approximation methods in [23, 30] can be used.

Approximation needs to be performed so that the state minimization of $M_2{}'$ can be completed in the K-N procedure. As a consequence, even if 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. Also, even if we use the exploitation approach in Section 3, we still have the problem in the subset construction. In the rest of this section, we propose a new procedure to circumvent the subset construction; as a result, both computation and exploitation can be made efficient and effective.

4.4 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 6(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 can employ logic optimization techniques in Section 3.2 to optimize M_2.

We require that the behavior of M_1 be kept unchanged and that X is the only communicating variable between M_1 and M_2. That is, we are only concerned about the logic optimization of M_2. Therefore, some logic optimization technique as described in Section 3.2 may need 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 6(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 can be fed into SIS where sequential redundancy removal can be performed.

From the K-N procedure [16], the flexibility of implementing M_2 comes from input don't care sequences. Logic optimization techniques in Section 3.2 exploit input don't care sequences in different ways. In the next subsection, this is discussed in more detail.

4.5 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 3.2.

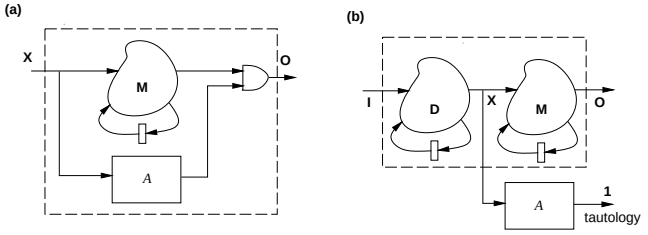

Figure 7: (a). The specified behavior of M with restricted input sequences $\mathcal{L}(\mathcal{A})$. (b). Construction of a driving machine D to M. The set of output sequences of D is equivalent to $\mathcal{L}(\mathcal{A})$.

We consider two exploitation techniques. Node simplification can exploit external don't cares both effectively and efficiently [25]. 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 $EDC(x) = \overline{\exists_{i,p_1,n_1} T_1(i, p_1, n_1, x)}$.

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 6(a). We assume that M_1 is deterministic.

Lemma 4.1 *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.*

Lemma 4.2 *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.*

Theorem 4.3 *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 6(a).*

Theorem 4.3 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, it is likely to have more sequential redundancies in M_2. This demonstrates that sequential ATPG-based techniques in Section 3.2 can directly exploit the flexibility of M_2 coming from input don't care sequences.

Theorem 4.4 *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 $M_1{}' \rightarrow M_2$.*

Theorem 4.4 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.

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

Theorems 4.3 and 4.4 lead to a method to optimize a machine M with input don't care sequences which, say, are not accepted by $\mathcal{A}$. Figure 7(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 CSFSM D whose set of output sequences is $\mathcal{L}(\mathcal{A})$ as the driving machine. This is shown in Figure 7(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 $\mathcal{L}(\mathcal{A})$.
- **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".

Since the above construction needs explicit enumeration which may not be efficient, in the following, we provide an implicit method for constructing such a CSFSM D directly from $\mathcal{A}$. Note that $\mathcal{A}$ has the following properties — every state except the dead state d is a final state, the input string in each transition is of length one, and there are no ϵ-transitions. Thus, we do not need to explicitly express the dead state d in the transition relation, since it is implicit from all unspecified transitions. We do not need to specify the set of final states, since every state is a final state. As a consequence, we can represent the transition relations of $\mathcal{A}$ in the same way as FSM's. Let the transition relation of $\mathcal{A}$ be $T_{\mathcal{A}}(p, i, n)$. To detect if $\mathcal{A}$ is deterministic is easy using BDD's; we compute ${T_{\mathcal{A}}}'(p, i, n) = \perp(\alpha, T_{\mathcal{A}}(p, i, n))$, where α is a reference next state vertex. For each pair (p, i) defined in $T_{\mathcal{A}}(p, i, n)$, $\perp$ assigns a unique n. However, if ${T_{\mathcal{A}}}'(p, i, n)$ equals $T_{\mathcal{A}}(p, i, n)$, nothing was changed, implying that T already had only one such candidate. Hence, $T_{\mathcal{A}}(p, i, n)$ is deterministic; otherwise nondeterministic.

Theorem 4.5 *If $\mathcal{A}$ is deterministic, the transition relation of one possible CSFSM D, $T_D(i, p, n, o)$ can be derived as follows:*

$$T_{D_1}(i, p, n, o) = T_{\mathcal{A}}(p, i, n) \cdot (i \equiv o). \quad (1)$$

$$T_{D_2}(i, p, n, o) = \{\overline{\exists_n T_{\mathcal{A}}(p, i, n)} \cdot \theta_{i,o} T_{\mathcal{A}}(p, i, n)\} + T_{D_1}(i, p, n, o). \quad (2)$$

$$T_D(i, p, n, o) = \perp(\alpha_0, T_{D_2}(i, p, n, o)) \quad (3)$$

where α_0 is a minterm in (n, o) space.

Theorem 4.6 *If $\mathcal{A}$ is non-deterministic, the transition relation of one possible CSFSM D, $T_D(i', p, n, o)$ can be derived as follows:*

$$T_{D_1}(i', p, n, o) = T_{\mathcal{A}}(p, i, n) \cdot (i \equiv o) \cdot (n \equiv i_1). \quad (4)$$

$$T_{D_2}(i', p, n, o) = \{\overline{\exists_{n,o} T_{D_1}(i', p, n, o)} \cdot \exists_{i'} T_{D_1}(i', p, n, o)\} + T_{D_1}(i', p, n, o). \quad (5)$$

$$T_D(i', p, n, o) = \perp(\alpha_0, T_{D_2}(i', p, n, o)) \quad (6)$$

ckt	I/X/O/S_1/S_2	M_2 lits	K-N procedure				Our procedure		
			SM		enc + opt		opt + red		
			$S_2{}'$	cpu	lits	cpu	$S_2{}'$	lits	cpu
ce1	9/19/7/20/47	248	7	0.2	95	8.9	12	**37**	20.4
ce2	2/2/3/8/27	348	15	0.1	**63**	3.7	16	75	19.0
ce3	18/19/7/25/47	248	8	0.3	39	2.4	16	**34**	29.2
ce4	18/19/7/25/47	248	4	0.1	**14**	1.0	5	15	17.3
ce5	7/7/2/16/19	314	18	1.3	193	53.2	18	**170**	40.3
ce6	7/2/3/19/27	348	19	0.1	120	11.7	20	**94**	41.4
ce7	19/7/2/47/19	314	15	7.5	178	41.5	16	**93**	107.4
ce8	11/9/19/32/20	280	8	83.8	239	59.3	9	**66**	552.5
ce9	7/7/19/16/48	617	-	sout	-	-	44	**454**	169.6
ce10	7/19/7/48/47	248	-	tout	-	-	35	**165**	534.7
ce11	19/7/19/47/48	617	-	tout	-	-	45	**441**	438.0
ce12	11/9/10/32/30	596	-	tout	-	-	27	**375**	405.6

Table 1: Experimental results of one-way-communication 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 **lits**:	number of literals (in factored form) of the initial M_2.
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, then optimized by running `script.rugged` twice.
opt + red:	optimized by running (`script.rugged` + `red_removal`) twice.
cpu:	CPU time in seconds on a DEC 3000/500 AXP (160MB memory).
tout:	set to 20,000 seconds of CPU time.
sout:	spaceout

ckt	M_2					
	after K-N procedure opt + red			after (opt + red) re-encoding + (opt + red)		
	lits(i)	lits(f)	cpu	lits(i)	lits(f)	cpu
ce1	95	74	13.2	37	**26**	9.0
ce2	63	63	4.0	75	**58**	4.7
ce3	39	33	15.8	34	**33**	17.5
ce4	14	**12**	7.1	15	14	7.5
ce5	193	152	48.9	170	**124**	20.9
ce6	120	101	34.9	94	**75**	27.6
ce7	178	108	62.5	93	**70**	14.9
ce8	239	113	323.0	66	**56**	111.2
ce9 *	-	-	-	454	**396**	1243.1
ce10	-	-	-	**165**	186	512.2
ce11 *	-	-	-	441	**377**	1154.9
ce12 *	-	-	-	375	**312**	1826.3

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

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

where $i' = (i, i_1)$ *and* α_0 *is a minterm in* (n, o) *space.*

Based on the discussion in Section 4.5, we can assign arbitrary state encoding to FSM D, and then have a logic implementation of FSM D. Subsequently, the methods in Section 3.2 can be employed to optimize M. Theoretically, the logic implementation of D will not affect the optimality of M. However, efficiency may be affected. For example, state-of-the-art sequential ATPG algorithms are based on BDD's, and state encoding of D will affect the size of BDD's for constructing the transition function of D. Currently, we are investigating this effect.

4.7 Generalized K-N Procedure in Logic Optimization of FSM Networks

Based on the discussions in Sections 4.5 and 4.6, we propose an approach for logic optimization of a component in a general FSM network using input don't care sequences. 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 [30]. 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 4.6, construct a CSFSM D whose set of output sequences is equivalent to $\mathcal{L}(\mathcal{A}')$. Then derive a logic implementation of CSFSM D.
4. Use the various optimization techniques in Section 3.2 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, but no subset construction is needed. This is because, in a sense, we do not use an ISFSM $M_2{}'$ to express the flexibility due to input don't care sequences. Exact input don't care sequences can be approximated. Many approximation methods for dealing with large FSM networks have been proposed in [30]. 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.

5 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. We have implemented the new procedure for input don't care sequences as described in Section 4.7.

Table 1 shows experimental results for some cascade circuits consisting of two FSM's. The circuit topology of these examples is shown in Figure 6(a). We employ both the K-N procedure and our procedure to optimize M_2 and then compare their results. Note that M_1 is a CSFSM, so in our approach we don't use the methods in Section 4.6 to construct a CSFSM D whose set of output sequences is equivalent to that of M_1. The logic optimizer used is SIS [27], and its standard optimization procedure is called `script.rugged` [26] 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 [30]; the bound on the number of states is set to 64. The state minimizer used here is STAMINA [12]. Afterwards, the state-minimized machine is encoded using JEDI [18], 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 then exploited in `script.rugged`. This corresponds to the first set of optimization techniques in Section 3.2. We then use the construction in Figure 6(b), and apply the

`red_removal` command in SIS to remove sequential redundancies. This corresponds to the second set of optimization techniques in Section 3.2. 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 `ce2` and `ce4`. For the third set of examples (`ce9`, `ce10`, `ce11` and `ce12`), 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 `ce4` and `ce10`.

In our experiments, only *redundancy removal* is used, and we expect that better results can be achieved if *redundancy addition and removal* in [11] is employed. These preliminary results indicate that our approach together with the notion of abstract driving machines [30] is promising for computing and exploiting input don't care sequences in general FSM networks.

We plan to integrate the algorithm for computing E-machines in [1] and our approach in Section 3, and then study various trade-offs about efficiency and effectiveness between input don't care sequences and E-machines in FSM networks.

6 Conclusion

We presented a novel approach for exploiting exact or approximate flexibility for a component in an FSM network directly at the net-list logic level. With our approach, many existing sequential net-list logic optimization techniques can be applied to exploit the flexibility. Moreover, we proposed a new procedure to facilitate both computation and exploitation of input don't care sequences in general FSM networks. Multi-level logic optimization of larger FSM networks can then be achieved. Preliminary results look promising but more FSM networks must be experimented on.

References

[1] A. Aziz, F. Balarin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Synthesis using S1S. In *IEEE International Conference on Computer-Aided Design*, November 1995.

[2] 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.

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

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] B. Lin and A. R Newton. Implicit Manipulation of Equivalence Classes Using Binary Decision Digrams. In *International Workshop on Logic Synthesis*, 1991.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] H.-Y. Wang and R. K. Brayton. Multi-level Logic Optimization of FSM Networks. Technical Report UCB/ERL M95/66, University of California, Berkeley, August 1995.

[33] 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.

[34] 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.