PERMISSIBLE OBSERVABILITY RELATIONS IN FSM NETWORKS

by

Huey-Yih Wang and Robert K. Brayton

Memorandum No. UCB/ERL M94/15

25 February 1994

PERMISSIBLE OBSERVABILITY RELATIONS IN FSM NETWORKS

by

Huey-Yih Wang and Robert K. Brayton

Memorandum No. UCB/ERL M94/15

25 February 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PERMISSIBLE OBSERVABILITY RELATIONS IN FSM NETWORKS

by

Huey-Yih Wang and Robert K. Brayton

Memorandum No. UCB/ERL M94/15

25 February 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Permissible Observability Relations in FSM Networks *

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

February 25, 1994

Abstract

Previous attempts to capture the phenomenon of output don't care sequences for a component in an FSM network have been incomplete. We demonstrate that output don't care sequences for a component can be expressed using a set of observability relations given that its state transition function is kept unchanged. Each observability relation is permissible in the sense that any implementation compatible with one of them is feasible. The representation for a set of permissible observability relations is not unique. We provide a method to find a set with the minimum number of permissible relations. We briefly discuss the exploitation of permissible observability relations in state minimization, circuit implementation and signal encoding. We have implemented these methods and present some preliminary results on a few small artificially constructed examples.

*This project was supported by DARPA under contract number JFBI90-073 and NSF under contract numbers EMC-84-19744 and MIP-87-19546.

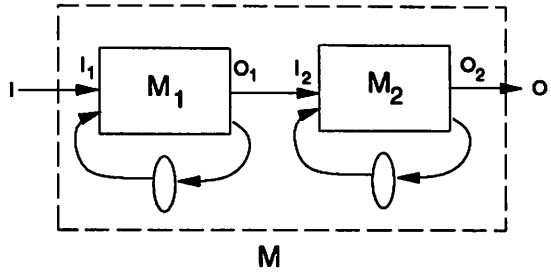

Figure 1: A cascade circuit of two FSM's.

1 Introduction

The flexibility in implementing an isolated combinational logic circuit can be expressed by don't cares. For an individual component in a hierarchically designed combinational logic circuit, a Boolean relation, an observability relation or a symbolic relation is required to express all possible implementations [5, 2, 13, 20]. This freedom in implementation is due to reduced controllability and observability from the environment. By exploiting this information, we often can achieve a better implementation for that component.

Similarly, sequential don't cares are important in the optimization of sequential circuits. Several approaches have been proposed. For example, in [14], unreachable states and equivalent states are exploited in the logic optimization of an isolated sequential circuit. Damiani *et al.* [8] introduced the notion of synchronous relations to deal with the logic optimization of pipelined sequential circuits. This approach is motivated by a circuit implementation point of view. On the other hand, a transition relation can be used to represent an isolated finite state machine (FSM). This allows us to deal with symbolic information, i.e. with unencoded machines. Incompletely specified information often refers to possible implementations. Exploiting this information may also change the state minimality of a machine. In general, a transition relation can be regarded as an observability relation or a symbolic relation. State minimization for an isolated machine has been well studied [16, 11]. Although state minimality does not imply that the resultant logic circuit after state encoding is minimized, it is generally regarded as a good starting point for state encoding to get smaller logic implementations.

In the case of sequential don't cares for an individual component in a network of FSM's,[1] we may need to consider sequences of don't cares. There are a few studies related to this problem [12, 9, 17, 25]. Although by flattening a network of FSM's into a composite machine we may perform global optimization, the composite machine is often too big to be handled by synthesis tools. To perform hierarchical synthesis, we must consider the interaction between components. The computation and exploitation of don't care information is crucial for the quality of the resultant circuit implementation.

The computation of don't care information for a component in an FSM network is much harder than its counterpart in combinational logic. We divide this problem into two parts: sequential input and output don't cares. In this paper, we deal with the latter. Consider the cascade machine in Figure 1. The flexibility in implementing M_1 when cascaded with M_2 is called *sequential output don't cares*. This was studied by Devadas [9], and later by Rho *et al.* [17] who generalized Devadas' procedure to compute *fixed-length output don't care sequences*.

In this paper, after reviewing previous work in section 3, we explain why the notion of *information lossyness* introduced in [17] can not completely characterize the phenomenon of output don't care sequences. Then, we discuss the difficulties in computing and expressing these. We demonstrate that output don't care sequences for a component can be expressed using a set of observability relations given that its state transition function is kept unchanged. In section 5, we propose an implicit enumeration algorithm which exactly computes them. Each observability relation is *permissible* in the sense that the behavior of the network is preserved. We describe how to exploit them in state minimization, circuit implementation and signal encoding. Finally, we give some preliminary results on some artificially constructed circuits.

[1] In this paper, only synchronous FSM networks with known initial states are considered.

2 Preliminaries

2.1 Finite Automata

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 nondeterministic 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 Machines

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 *distinguishing sequence* for two states $q_1, q_2 \in Q$ is a sequence of inputs such that when applied to M, the last input produces different outputs depending whether M started at q_1 or q_2. In a completely specified machine M, two states q_1 and q_2 are *equivalent* if there is no distinguishing sequence. In an incompletely specified machine M, two such states q_1 and q_2 are *compatible.*

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.* For $x \in Q_{M_1}$, its *co-reachable* states in M_2 are $\{y | y \in Q_{M_2}$ such that (x, y) is a reachable state in the cascade machine $M_1 \rightarrow M_2\}$. Similarly, a state in M_2 has co-reachable states in M_1.

2.3 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 :*

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

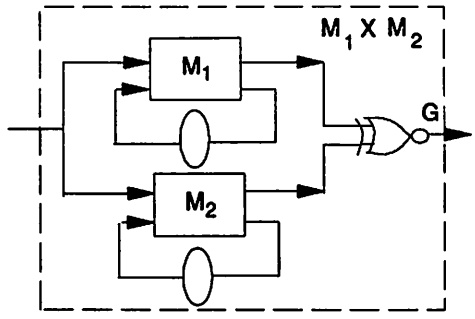

Figure 2: Product machine of M_1 and M_2

We can use F to obtain the image by f of $S_1 \subseteq B^n$, by computing the projection on B^m of the set $F \cap (S_1 \times B^m)$:

$$f(S_1)(y) = \exists_x (F(x,y) \cdot S_1(x)) .$$

Similarly, the inverse image by f of $S_2 \subseteq B^m$ can be computed as :

$$f^{-1}(S_2)(x) = \exists_y (F(x,y) \cdot S_2(y)) .$$

Reduced ordered binary decision diagrams (BDD's) [3] are well suited to represent the characteristic functions of subsets of a set, and efficient algorithms [3, 1] exist to manipulate them to perform all standard Boolean operations. As a result, the above set operations can be done efficiently.

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

We can use multiple-valued decision diagrams (MDD's) [22] 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 [22]. In the sequel, we just use the term BDD to interchangeably refer to characteristic functions of multiple-valued variables.

2.5 Finite State Machine Equivalence Checking

Suppose we want to check whether two FSM's M_1 and M_2 are equivalent. The general approach is as follows. Construct M, the product machine of M_1 and M_2, as shown in Figure 2. Now, reformulate this problem to that of checking whether the output of M, G, is a tautology for all states reachable from the initial state of M. The reachable states can be computed efficiently using implicit state enumeration techniques introduced by Coudert *et al.* [6]. These techniques are widely used in FSM verification [6, 7, 24], and in design verification [4, 23]. 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 an FSM implicitly by a characteristic function using BDD's.

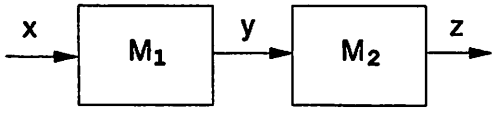

Figure 3: A cascade of two combinational circuits.

Definition 6 *The* **transition relation** *of an FSM* $M = (I, O, Q, \delta, \lambda, q_0)$ *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.*

A predicate transformer is a monotone function operating on the power set of a finite set. The set of states $R(p)$ containing all states reachable from a given set of initial states $I(p)$ can be viewed as the least fixed point of the function :

$$\mathcal{F} : c(p) \mapsto c(p) + \theta_{n,p}\exists_{i,p,o}(T(i, p, n, o) \cdot c(p)) .$$

At a fixed point, $R(p)$ satisfies :

$$R(p) = R(p) + \theta_{n,p}\exists_{i,p,o}(T(i, p, n, o) \cdot R(p)) .$$

The least fixed point of $\mathcal{F}$ can be computed as the limit of the following sequences :

$$\begin{aligned} R_0(p) &= I(p) \\ R_{m+1}(p) &= R_m(p) + \theta_{n,p}\exists_{i,p,o}(T(i, p, n, o) \cdot R_m(p)) \\ R_\infty(p) &= R_m(p) \text{ if } R_{m+1}(p) = R_m(p) . \end{aligned}$$

3 Previous Work

3.1 Observability Relation for Combinational Logic

In a hierarchically designed combinational logic circuit, all possible implementations can be represented by a single Boolean relation, an observability relation or a symbolic relation [2] [5, 2, 13, 20]. For example, as shown in Figure 3, M is a cascade machine $M_1 \rightarrow M_2$, where M_1 and M_2 are combinational circuits. Let $\mathcal{O}(x, z)$, $\mathcal{O}_1(x, y)$, and $\mathcal{O}_2(y, z)$ be the observability relations of M, M_1, and M_2, respectively. If $\mathcal{O}(x, z)$ and $\mathcal{O}_2(y, z)$ are given, $\mathcal{O}_1(x, y)$ can be computed as follows :

$$\mathcal{O}_1(x, y) = \exists_z(\mathcal{O}(x, z) \cdot \mathcal{O}_2(y, z)) .$$

$\mathcal{O}_1(x, y)$ captures all possible implementations of M_1 without violating the desired behavior of the cascade machine M. Exploiting this freedom in implementation often leads to better logic implementations.

3.2 Sequential Output Don't Cares

If M_1 and M_2 in Figure 3 are FSM's, computing the flexibility in implementing M_1 is much harder. Devadas [9] addressed this problem as computing *sequential output don't cares* for M_1 and proposed a simple heuristic to compute partial don't care information for M_1. Consider a transition edge e in the STG of M_1. Let the output symbol of e be v_1. Devadas' procedure first computes the co-reachable states in M_2 corresponding to the present state of transition e. If for every corresponding co-reachable state in M_2, an output symbol v_2 from M_1 drives machine M_2 to produce the same output and next state as the original output symbol v_1 from M_1 does, then the output part $\{v_1\}$ of e is **expanded** to $\{v_1, v_2\}$. This is repeated on e for each output symbol v_2 of M_1. Then the above process is repeated for each transition edge e in the STG of M_1. This **output expansion** procedure does not change state reachability of the composite machine $M_1 \rightarrow M_2$.

This procedure, in fact, is restrictive[3]. First, it considers only one transition edge of M_1 at a time, and excludes the possibility of **simultaneous output expansions** among all transition edges. That is, an expanded output symbol in a transition

[2] In this paper, we will not make a distinction among Boolean relations, observability relations and symbolic relations unless necessary.

[3] This is in contrast to the comment, "This is not restrictive, as long as we can assume that M_2 is state-minimal.", made in [18].

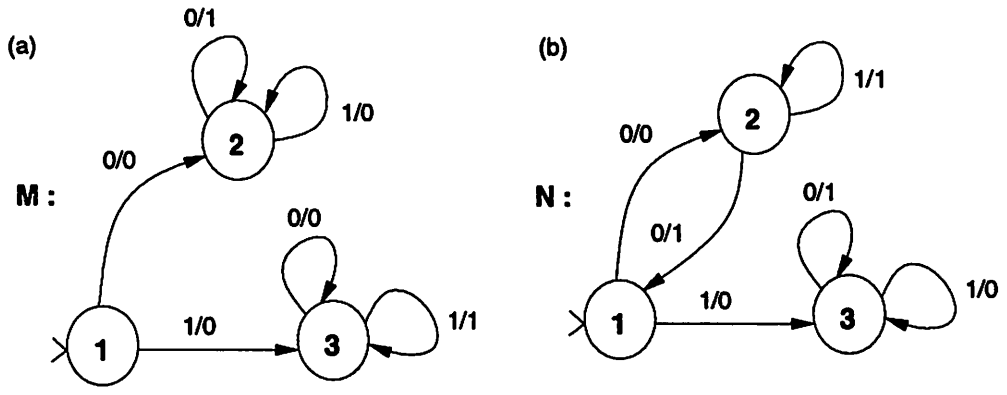

Figure 4: (a) An information lossless and state-minimal machine. (b) A lossy and state-minimal machine.

edge may depend on the expanded output symbols of the other transition edges. Furthermore, next states need not be the same when an output expansion takes place. To be more specific, the set of co-reachable states for each state in M_1 may change due to simultaneous output expansions. In the next subsection, we show this by example.

3.3 Output Don't Care Sequences

Later, Rho *et al.* [17] considered sequences of output don't cares and used the notion of **information lossyness** to explain the phenomenon of these sequences in a cascade machine. A machine is said to be **information lossless** if given the initial state, the final state and the output sequence, the corresponding input sequence can be uniquely determined. A machine that is not lossless is said to be **lossy**. A state s of a machine is said to be a *lossy state* if starting from s there exist two distinct finite-length input sequences such that their output sequences and final states are the same. An information lossless machine cannot contain any lossy states.

Consider the cascade machine $M_1 \rightarrow M_2$ in Figure 1. Rho *et al.* [17] interpreted that output don't care sequences for M_1 are due to the lossyness of M_2. Based on this explanation, if M_2 has no lossy states, there are no output don't care sequences for M_1. In this sense, only lossy states in M_2 need to be considered for computing output don't care sequences. Accordingly, they gave the following definitions for *equivalent sequences* and *equivalent machines*. Input sequences that lead a driven machine M_2 from the same initial state s to the same final state t and produce the same output sequences, are said to be *equivalent* with respect to state s. Thus, state s is a lossy state. Two machines M_1' and M_1'' are said to be *equivalent* with respect to M_2 if and only if for each input sequence they produce output sequences that are in the same equivalence class of input sequences of M_2, i.e. ending in the same final state if M_2 is state-minimal.

According to the above definitions, a heuristic was proposed in [17] to compute a subset of fixed-length output don't care sequences for M_1. This is an extension of Devadas' procedure [9]. To compute fixed-length, say k-length, *equivalent sequences* in M_2, this procedure first *unrolls* M_1 and M_2 for k-length time frames. Let the unrolled machines of M_1 and M_2 be M_1^k and M_2^k, respectively. An unrolled machine has the same state space as that of the original machine except that for each transition edge the input part is a k-length input sequence, and the output a k-length output sequence. By such a construction, k-length equivalent sequences starting at a state s of M_2^k can be computed. In contrast, Devadas' procedure computes *equivalent value* from a state s in M_2. Then the output part of each transition edge in M_1^k can be **expanded** using the same rationale in Devadas' procedure except that a *consistency check* needs to be performed between the input and output part of all transition edges in M_1^k. Finally, to construct a non-unrolled machine of M_1^k, M_1', a heuristic based on *state splitting* may be employed to accommodate this don't care information as much as possible. This procedure *explicitly* enumerates fixed-length output don't care sequences. In general, output don't care sequences are of infinite length. The complexity of this procedure may grow *exponentially* with the length of don't care sequences.

This procedure is an elegant extension of [9]. However, their interpretation of output don't care sequences is not general. By their definition, output don't care sequences for M_1 can be interpreted as those input sequences starting from the initial state of M_2 (assumed to be lossy and state-minimal), producing the same output sequence, and ending in the same final state. Their reasoning is as follows. If the output sequence of M_2 is altered, so is the overall behavior of the cascade. Also, if the final state is different, the behavior of the cascade changes, unless the new final state is equivalent to the original one.

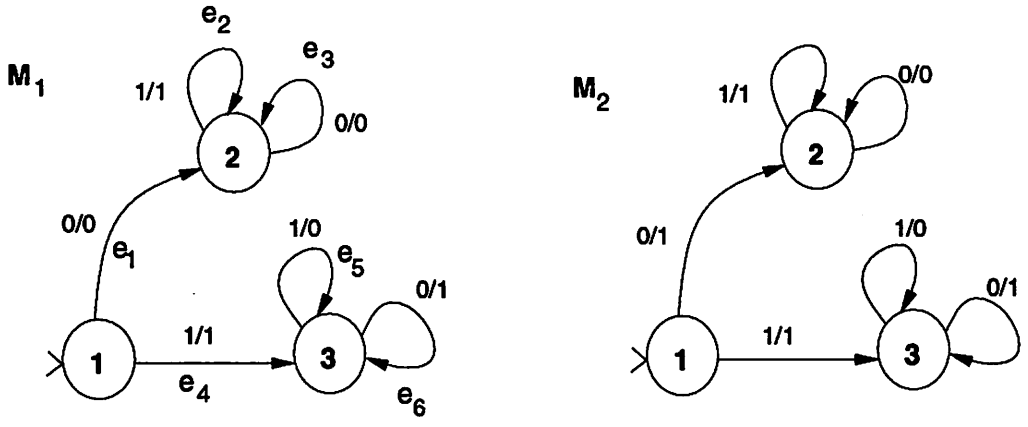

Figure 5: An example : $M \equiv M_1 \rightarrow M_2$, where both M_1 and M_2 are state-minimal and information lossless.

There are several factors not considered in this reasoning.

First, output sequences from M_1 may change the equivalence of states in M_2 simultaneously, since the complement of output sequences from M_1 are input don't care sequences for M_2, which may change the state minimality of M_2 [12]. During the process of exploiting output don't care sequences for M_1, the output function of M_1 is changed at the same time. Consequently, we are unable to assume that state minimality of M_2 is invariant.

Secondly, by definition, different input sequences applied to an information lossless machine may produce the same output sequences, but end in different final states. Consequently, if M_2 produces the same output sequences for different input sequences, it is not necessarily lossy. In other words, there might exist output don't care sequences even if M_2 is information lossless. For example, machine M in Figure 4(a) is information lossless and state-minimal. Input sequences (11^*) and (00^*) produce the same output sequences (01^*), but do not end in the same final state. As a consequence, it is not necessary that input sequences which produce the same output sequences end in the same final state even when the driven machine is state-minimal. Moreover, this argument is not valid even for a lossy and state-minimal machine. An example is machine N shown in Figure 4(b). N is lossy since state 1 is a lossy state[4]. Input sequences $(00)^*$ and $(10)^*$ produce the same output sequences $(01)^*$, but do not end in the same final state.

Consider the cascade machine $M \equiv M_1 \rightarrow M_2$ as shown in Figure 5. This is an example where there are output don't care sequences for M_1 even when M_2 is state-minimal and information lossless. Let v_i denote the output value of a transition edge e_i in the STG of M_1. The value of $(v_1, v_2, v_3, v_4, v_5, v_6)$ in Figure 5 is $(0,1,0,1,0,1)$. Since the output sequences of M_1 are $\{0,1\}^*$, there are no input don't care sequences for M_2. Thus, we cannot use any input don't care information to simplify M_2 first. If we apply Devadas' [9] or Rho's [17] procedures, we cannot find any sequential output don't cares for M_1. However, any one of the following values of $(v_1, v_2, v_3, v_4, v_5, v_6)$ preserves the same behavior as $M_1 \rightarrow M_2$.

$$(v_1, v_2, v_3, v_4, v_5, v_6) = \begin{cases} (0,1,0,1,0,1) \\ (0,1,0,0,1,0) \\ (1,0,1,0,1,0) \\ (1,0,1,1,0,1) \end{cases}$$

The reachable states of M_2 are $\{1,2,3\}$ when $(v_1, v_2, v_3, v_4, v_5, v_6) = (0,1,0,1,0,1)$. But when $(v_1, v_2, v_3, v_4, v_5, v_6) = (0,1,0,0,1,0)$, the reachable states of M_2 are $\{1,2\}$. So, the state reachability may change when we have different output sequences from M_1. This is why Devadas' procedure is restrictive.

Consequently, the previous definitions of machine equivalence for the driving machine do not include all possible machines which when cascaded by M_2 produce the same behavior as the original cascade machine. As a matter of fact, the general definition of **machine equivalence** should be the following : two machines M_1' and M_1'' are *equivalent* with respect to M_2 if and only if $M_1' \rightarrow M_2$ and $M_1'' \rightarrow M_2$ have the same input/output behavior. This specifies the full flexibility for implementing M_1. Therefore, this should be regarded as the general definition of *sequential output don't cares.*

[4]Input sequences 0010, 1010 from initial state 1 produce the same output sequence 0101, and end in the same state 3.

4 Permissible Observability Relations

As mentioned earlier, an incompletely specified FSM can be expressed by a symbolic relation. Using this representation for an isolated FSM, two kinds of don't care information can be conveyed. The first is known as *input-incompletely-specified don't cares* or *unspecified transitions*. They characterize the situation that a given input symbol never occurs when a machine is in a particular state, since there are limited kinds of sequences that can be applied to the machine. The other kind is called *output-incompletely-specified don't cares*. They occur when we are not interested in an output symbol associated with a given state or state transition. In the following, we investigate whether this representation is powerful enough to convey sequential don't cares for a component in an interacting FSM network.

In a cascade circuit $M_1 \rightarrow M_2$ as shown in Figure 1, we can compute input don't cares sequences for M_2 by keeping M_1 unchanged. The general procedure known to solve this problem is due to Kim and Newborn [12]. This procedure summarizes output sequences from M_1 by an NFA $\mathcal{A}'$, and then transforms $\mathcal{A}'$ into a minimal DFA $\mathcal{A}$. The equivalent machine to M_2 with input don't care sequences is the product machine $\mathcal{A} \times M_2$. The input don't cares sequences of M_2 are unspecified transitions in the resultant product machine. This product machine captures all input don't cares sequences, and we can represent it as an incompletely specified FSM. As a consequence, a single observability relation is sufficient to implicitly express input don't care sequences.

On the other hand, a single observability relation may not be sufficient to express output don't care sequences. We can compute output don't care sequences for M_1 by keeping M_2 unchanged. An FSM can be regarded as a *language transducer*, i.e. transforming a regular language to another regular language. Therefore, output sequences from M_1 can be expressed by a regular language, say $\mathcal{L}(M_1^o)$. We can define an equivalence class of languages with respect to M_2, $[\mathcal{L}(M_1^o)]_{M_2}$, such that any language in this equivalence class can be generated by a certain machine M_1' which preserves the same behavior as $M_1 \rightarrow M_2$ when cascaded with M_2. Next, we explain the intrinsic difficulties in computing sequential output don't cares even when we adopt the definition of *machine equivalence* from [17], i.e. *equivalent* input sequences end up in the same final state if M_2 is state-minimal in isolation. Rho's procedure [17] computes fixed-length equivalent sequences, and then *expands* these to be output don't care sequences for M_1. Exploiting this information, some equivalent machines may be derived. Let M_1' be an equivalent machine. Then $\mathcal{L}(M_1'^o)$, output sequences from M_1', is in $[\mathcal{L}(M_1^o)]_{M_2}$. However, the length of equivalent sequences may be arbitrarily large. Furthermore, the complexity of this computation may grow exponentially with the length of output don't care sequences. Therefore, it is hard to enumerate all languages in $[\mathcal{L}(M_1^o)]_{M_2}$.

From another point of view, we can enumerate all possible languages produced by M_1 with its state transition function unchanged. Let the cardinality of the transition edges of M_1 be k, and that of the output alphabet of M_1 be m. If we keep the state transition function of M_1 unchanged, there are m^k possible output functions, i.e. m^k possible regular languages may be produced by M_1 (some of them may be the same). For each output function λ_1', there is a corresponding machine M_1'. The feasibility of λ_1' can be checked by testing if $M_1' \rightarrow M_2$ preserves the same behavior as $M_1 \rightarrow M_2$. This is shown in Figure 6. Using this naive approach, we can check all possible output functions one by one to find all feasible solutions. We may need m^k invocations of FSM equivalence checking.

We can interpret the above approach as **simultaneous output expansions** among all transition edges. An expanded output symbol in a transition edge may be dependent on the expanded output symbols of other transition edges. Consequently, the flexibility of implementations captured by the above approach is more than for output-incompletely-specified don't cares in an isolated FSM. All feasible output functions possibly may not be expressed by a single observability relation. In fact, a set of observability relations is needed. We show this in the next section. Each observability relation is **permissible** in the sense that any output function **compatible** with one of the observability relations corresponds to a possible implementation.

In the next section, we present an implicit algorithm which finds all such feasible output functions by executing FSM equivalence checking *once*.

5 Computation of Permissible Observability Relations

Let $M_1 = (I_1, O_1, Q_1, \delta_1, \lambda_1, q_{10})$ and $M_2 = (I_2, O_2, Q_2, \delta_2, \lambda_2, q_{20})$ be two FSM's. $M = (I, O, Q, \delta, \lambda, q_{M0})$ is $M_1 \rightarrow M_2$, the cascade machine. Let Λ_1 denote the set of all possible output functions of M_1 while keeping its state transition function unchanged. Let $M_1|_{\lambda_1'}$ denote a machine which is the same as M_1 except with an output function $\lambda_1' \in \Lambda_1$. Suppose that the cardinality of transition edges in M_1 is k. For each transition edge e_j, we associate it with a symbolic variable v_j which takes values from O_1. Let V denote the space spanned by $v_0, v_1, \ldots, v_{k-1}$, i.e. O_1^k.

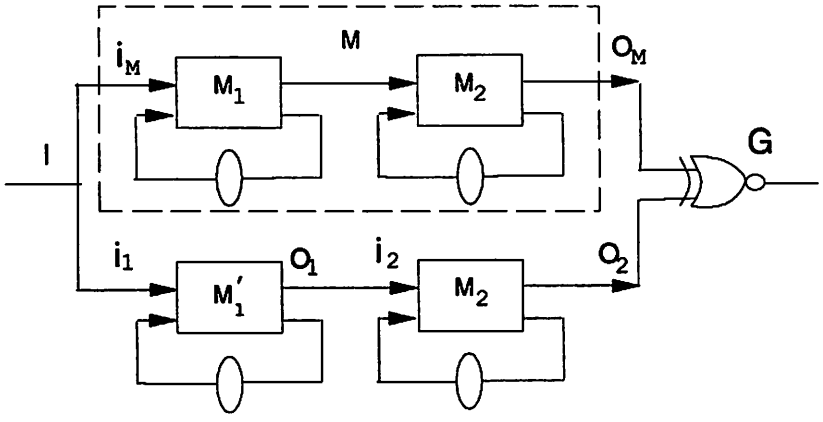

Figure 6: Feasibility testing using FSM equivalence checking.

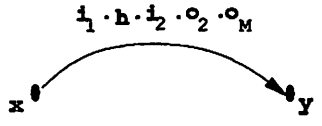

Figure 7: A transition defined in T.

Definition 7 *Any minterm in V is called an* **output assignment**. *An output assignment corresponds to an output function in Λ_1, and vice versa.*

Definition 8 *An* **assignment mapping** *is a bijective mapping $\mathcal{M} : V \rightarrow \Lambda_1$ which maps a minterm $v \in V$ to an output function λ_1' in Λ_1.*

Definition 9 *An output function $\lambda_1' \in \Lambda_1$ is* **feasible** *if and only if $M_1|_{\lambda_1'} \rightarrow M_2$ preserves the same behavior as $M_1 \rightarrow M_2$.*

Definition 10 *An output assignment v is* **feasible** *if and only if $\mathcal{M}(v)$ is a feasible output function.*

Definition 11 *The set of feasible output assignments is denoted as $f(v)$, the* **feasible output assignment function.**

Our goal is to compute all $\lambda_1' \in \Lambda_1$ such that $M_1|_{\lambda_1'} \rightarrow M_2$ preserves the same behavior as $M_1 \rightarrow M_2$. This is pictorially explained in Figure 6.

5.1 Reachability Relation

Here, we present an implicit enumeration method based on a generalization of implicit FSM equivalence checking. The most important step in the FSM equivalence checking is the computation of reachable states. The state space for our equivalence checking is $Q_1 \times Q_2 \times Q$, denoted as S. For an output assignment v, there is a corresponding output function $\mathcal{M}(v)$ and a set of reachable states which is a subset of S. Different output functions may result in different sets of reachable states. With this observation, we introduce the concept of *reachability relation.*

Definition 12 *A* **reachability relation** *is a function $\mathcal{F} : S \times V \rightarrow B$ such that $\mathcal{F}(s, v) = 1$ if and only if s is reachable from the initial state when the output function is $\mathcal{M}(v)$.*

For an output assignment v, the transition relation of $M_1|_{\mathcal{M}(v)}$ is $T_1|_{\mathcal{M}(v)}$. We can compose $T_1|_{\mathcal{M}(v)}$, T_2, and T_M, and then use implicit reachability computation to check whether G in Figure 6 is a tautology. However, this is an explicit enumeration method since we need to enumerate explicitly for all $v \in V$. To perform implicit enumeration, we construct an **abstract transition relation** for M_1, T_1', as follows. First, give a labeling for each transition edge in M_1. Let h be a k-valued variable. We substitute j, i.e. the literal $h^{\{j\}}$, for the output part of e_j. The abstract transition relation is $T_1'(i_1, p_1, n_1, h)$. Let the transition

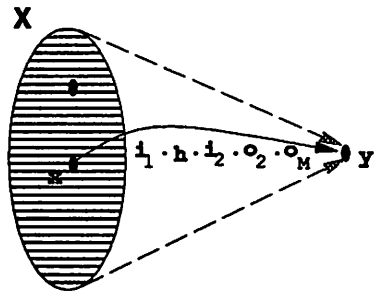

Figure 8: A state y is reachable from a set of states X at the m-th iteration.

relation of M_2 be $T_2(i_2, p_2, n_2, o_2)$ and that of M, $T_M(i_M, p_M, n_M, o_M)$. Define $T(i, h, i_2, p, n, o_2, o_M) = T_1' \cdot T_2 \cdot T_M$, where $i = i_1 = i_M$, $p = (p_1, p_2, p_M)$ and $n = (n_1, n_2, n_M)$. The initial state is $q_0 = (q_{10}, q_{20}, q_{M0})$. The motivation for constructing the abstract transition relation, T_1', is explained below. T contains all possible transitions for any composite machine of $M_1|_{\mathcal{M}(v)}$, M_2, and M. Consider a transition defined in $T(i, h, i_2, p, n, o_2, o_M)$ as shown in Figure 7. The expression associated with this transition is $(i \cdot h \cdot i_2 \cdot o_2 \cdot o_M)$. Let the value of h be j. It means that when the output value of e_j in M_1 is equal to that of i_2, this transition is made, and vice versa. Different output values of transition edge e_j result in different output assignments. Therefore, this provides a way to relate output assignments to transitions in T.

Since $\mathcal{F} \subseteq S \times V$, $\mathcal{F}$ is a finite set. In the following theorem, we demonstrate that the reachability relation can be computed by a least fixed point computation.

Theorem 5.1 *Let $\mathcal{P}(p, n, v)$ be defined as follows:*

$$\mathcal{P}(p, n, v) = \sum_{j=0}^{k-1} \theta_{i_2, v_j} \{[(\exists_{i, o_2, o_M} T)]_{(h=j)}\} . \tag{1}$$

$\mathcal{F}(p, v)$ is the limit of the following sequence:

$$\begin{aligned} \mathcal{F}_0 &= (p = q_0) \cdot 1 \\ \mathcal{F}_m &= \theta_{n,p} \exists_p \{\mathcal{F}_{m-1} \cdot \mathcal{P}(p, n, v)\} + \mathcal{F}_{m-1} \\ \mathcal{F}_\infty &= \mathcal{F}_m \ \ if \ \ \mathcal{F}_m = \mathcal{F}_{m+1} . \end{aligned}$$

Proof Induction on the number of iterations, m. Initially, q_0 is reachable for every output assignment; thus the construction of $\mathcal{F}_0$ is correct. Variables i, o_2, o_M have no contribution in computing reachability relation, so we can smooth them out from T in the first place. Suppose the reachability relation up to the $(m-1)$-th iteration is $\mathcal{F}_{m-1}$. Consider transitions traversed at the m-th iteration. Suppose a state y is reached at the m-th iteration from a set of states X, reached up to the $(m-1)$-th iteration. This is shown in Figure 8. For $x \in X$, by induction hypothesis, state x is reachable only when output assignments are $(\mathcal{F}_{m-1})_{p=x}$. The value of h, say j, in the transition from x to y means that the output value of transition edge e_j in M_1 contributes the transition from x to y. Consequently, the value of v_j should be equal to that of i_2 to make this transition from x to y. $\mathcal{P}(p, n, v)$ defined in Equation (1) characterizes all such conditions. State y is reachable when x is reachable under the output assignments $(\mathcal{F}_{m-1})_{p=x}$ and v_j equals the value of i_2. Then summing up over $x \in X$ and over all possible values of h (i.e. $\{0, \ldots, k-1\}$), we can get the corresponding output assignments of y at the m-th iteration. This corresponds to the expression : $\exists_p \{\mathcal{F}_{m-1} \cdot \mathcal{P}(p, n, v)\}$. By implicit enumeration, we can get all such y, the next state image at the m-th iteration. In order to get $\mathcal{F}_m$, the reachability relation up to the m-th iteration, we still need to add $\mathcal{F}_{m-1}$. This proves the correctness of constructing $\mathcal{F}_m$. Furthermore, we have $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \cdots \subseteq \mathcal{F}_m$, $\forall m$, and $\mathcal{F}_\infty$ is a finite set; consequently, when $\mathcal{F}_m = \mathcal{F}_{m+1}$, we have $\mathcal{F}_\infty = \mathcal{F}_m$. ∎

Corollary 5.2 *For an output assignment α, its corresponding set of reachable states up to the m-th iteration $(R_m(p))_\alpha$ is $(\mathcal{F}_m(p, v))_{v=\alpha}$. In particular, the set of reachable states is $(R_\infty(p))_\alpha = (\mathcal{F}_\infty(p, v))_{v=\alpha}$.*

Proof Follows directly from the Theorem 5.1. ∎

5.2 Feasible Output Assignment Function

An FSM can be regarded as a language transducer, i.e. transforming a regular language to another regular language. If M_2 is completely specified, M_2 maps I_2^* to another regular language. That is, for all input sequences, the behavior of M_2 is defined. First, we consider the case when M_2 is completely specified.

For each output assignment v, there is a corresponding set of reachable states in S. This results in $T|_{\mathcal{M}(v)}$, the transition relation of the composite machine of $M_1|_{\mathcal{M}(v)}$, M_2 and M. If there are different values between o_2 and o_M in a transition of $T|_{\mathcal{M}(v)}$, G in Figure 6 is not a tautology. Therefore, v is not a feasible output assignment. An output assignment v is **infeasible** if and only if $T|_{\mathcal{M}(v)}$ has a transition with $(o_2 \not\equiv o_M)$. Consider a transition in T as shown in Figure 7. Let the value of h be j, and the value of i_2 be r. Therefore, when the output value of e_j in M_1 is equal to that of i_2, this transition is made. State x is reachable if and only if the output assignments are in $(\mathcal{F}_\infty)_{p=x}$. If this transition is with $(o_2 \not\equiv o_M)$, $(\mathcal{F}_\infty)_{p=x} \cdot (v_j = r)$ are infeasible output assignments. We need to enumerate all transitions of T with $(o_2 \not\equiv o_M)$ to compute the set of all infeasible output assignments; the set of feasible output assignments is just the complement. In the following theorem, we present an implicit enumeration method to find the feasible output assignment function, $f(v)$.

Theorem 5.3 *(M_2 : completely specified)*
Let $\mathcal{W}(p, v)$ be defined as follows :

$$\mathcal{W}(p, v) = \sum_{j=0}^{k-1} \theta_{i_2, v_j}\{[\exists_{n, o_M, o_2}((\exists_i T) \cdot (o_2 \not\equiv o_M))]_{(h=j)}\} . \tag{2}$$

The feasible output assignment function $f(v)$ is :

$$f(v) = \overline{\exists_p(\mathcal{F}_\infty \cdot \mathcal{W}(p, v))} .$$

Proof Variable i has no contribution to compute $f(v)$, so it can be smoothed out from T first. Now, consider the transitions in T. A state x can transit to a set of states by different values of i_2. First, we can aggregate transition edges which start from x with different values between o_2 and o_M. Let the corresponding set of next states be Y. Any one of these transitions is infeasible in the sense that it causes G in Figure 6 not to be a tautology. Let $y \in Y$. The value of h, say j, in the transition from x to y means that transition edge e_j in M_1 contributes to this transition, and the value of v_j must be the same as that of i_2. This is the condition to make an infeasible transition from x. We can implicitly sum up over Y to get the condition to make infeasible transitions from x. $\mathcal{W}(p, v)$ is the relation which associates a state p with the corresponding condition to make infeasible transitions starting at p. For each state x, we intersect $\mathcal{W}(p, v)_{p=x}$ with $(\mathcal{F}_\infty)_{p=x}$, the output assignments for reaching x. These are infeasible output assignments. With summation over every possible reachable state, we get the set of infeasible output assignments; the set of feasible output assignments is just the complement. ∎

The feasible output assignment function $f(v)$ may be a relatively small subset of all output assignments, V. Moreover, it is not necessary to construct $\mathcal{F}$ first using the method in Theorem 5.1, and then remove all infeasible output assignments using Theorem 5.3. We may *incrementally* remove the infeasible output assignments during the construction of the reachability relation. In the next theorem, we present an incremental approach.

Theorem 5.4 *(M_2 : completely specified)*
The set of feasible output assignments $f(v)$ is the limit of the following sequence, where $\mathcal{C}_m(p, v)$ is the reachability relation restricted to $f_m(v)$, the feasible output assignment function up to the m-th iteration. $\mathcal{P}(p, n, v)$ and $\mathcal{W}(p, v)$ are defined in Equations (1) and (2), respectively.

$$\begin{aligned}
\mathcal{C}_0 &= (p = q_0) \cdot 1 , \quad f_0(v) = 1 \\
\mathcal{C}_m{}' &= \theta_{n,p}\exists_p\{\mathcal{C}_{m-1} \cdot \mathcal{P}(p, n, v)\} + \mathcal{C}_{m-1} \\
\overline{f_m(v)} &= \exists_p(\mathcal{C}_m{}' \cdot \mathcal{W}(p, v)) + \overline{f_{m-1}(v)} \\
\mathcal{C}_m &= \mathcal{C}_m{}' \cdot f_m(v) \\
\mathcal{C}_\infty &= \mathcal{C}_m, \ f_\infty = f_m \ \ if \ \ \mathcal{C}_m = \mathcal{C}_{m+1} .
\end{aligned}$$

In particular, $\mathcal{C}_\infty$ is the reachability relation restricted to $f(v)$, the feasible output assignment function.

Proof Induction on the number of iterations, m. Initially, q_0 is reachable for every output assignment, thus the construction of $\mathcal{C}_0$ and $f_0(v)$ is correct. Now suppose at the $(m-1)$-th iteration, the reachability relation restricted to the feasible output assignments up to the $(m-1)$-th iteration is $\mathcal{C}_{m-1}$. By the proof in Theorem 5.1, $\mathcal{C}_m{}'$ is the reachability relation up to the m-th iteration but may contain infeasible output assignments. However, from Theorem 5.3, $\overline{f_m(v)}$ is the set of infeasible output assignments up to the m-th iteration. Consequently, $\mathcal{C}_m$ gives the reachability relation restricted to the feasible output assignments up to the m-th iteration. When $\mathcal{C}_m$ reaches the fixed point, we have $\mathcal{C}_\infty$ and f_∞. ∎

When M_2 is input-incompletely-specified (i.e. with unspecified transitions), we need to modify the above theorem to compute $f(v)$. As explained in the previous section, input-incompletely-specified don't cares are due to the interaction with other machines (in our case, M_1). Let $\mathcal{L}(M_2^i)$ denote the input sequences of M_2 where the behavior is defined. We can construct an automaton $\mathcal{A}$ to accept $\mathcal{L}(M_2^i)$ as follows. Every state of M_2 is a final state. For each transition in the STG of M_2, remove the output part. Then for each state with unspecified input values, create a transition edge to the *dead* state (a non-final state), and associate those unspecified values to that transition. Any input sequences not in $\mathcal{L}(M_2^i)$ drive M_2 to exhibit undefined behavior. Suppose $M \equiv M_1 \rightarrow M_2$ does not have undefined behavior. Then a feasible output assignment v, $M_1|_{\mathcal{M}(v)} \rightarrow M_2$ should not have undefined behavior, either. In other words, if v is a feasible output assignment, the output sequences generated by $M_1|_{\mathcal{M}(v)}$ must be in $\mathcal{L}(M_2^i)$ (i.e. $\mathcal{L}(M_1^o)|_{\mathcal{M}(v)} \subseteq \mathcal{L}(M_2^i)$).

Theorem 5.5 *(M_2 : input-incompletely-specified)*
Let $\mathcal{W}(p,v)$ be defined as follows :

$$\begin{aligned} T_2^C(i_2,p_2) &= \overline{\exists_{n_2,o_2}(T_2)} \\ \mathcal{W}(p,v) &= \sum_{j=0}^{k-1} \theta_{i_2,v_j}\{[T_2^C + \exists_{n,o_M,o_2}((\exists_i T)\cdot(o_2 \not\equiv o_M))]_{(h=j)}\}\,. \end{aligned}$$

The feasible output assignment function $f(v)$ is the limit of the following sequence, where $\mathcal{P}(p,n,v)$ is defined in Equation (1) and $\mathcal{C}_m(p,v)$ is the reachability relation restricted to $f_m(v)$, the feasible output assignment function up to the m-th iteration.

$$\begin{aligned} \mathcal{C}_0 &= (p=q_0)\cdot 1\,, \quad f_0(v) = 1 \\ \mathcal{C}_m{}' &= \theta_{n,p}\exists_p\{\mathcal{C}_{m-1}\cdot\mathcal{P}(p,n,v)\} + \mathcal{C}_{m-1} \\ \overline{f_m(v)} &= \exists_p(\mathcal{C}_m{}'\cdot\mathcal{W}(p,v)) + \overline{f_{m-1}(v)} \\ \mathcal{C}_m &= \mathcal{C}_m{}'\cdot f_m(v) \\ \mathcal{C}_\infty &= \mathcal{C}_m,\ f_\infty = f_m \ \ if\ \ \mathcal{C}_m = \mathcal{C}_{m+1}\,. \end{aligned}$$

In particular, $\mathcal{C}_\infty$ is the reachability relation restricted to $f(v)$, the feasible output assignment function.

Proof We define $T_2^C(p_2,i_2) = 1$ if and only if i_2 is an unspecified value at state p_2 of M_2. We construct $T_2^C(p_2,i_2)$ as follows. We smooth out variable o_2 since it is irrelevant to the computation of T_2^C. Then we implicitly summarize the input values over transitions starting from a state of M_2 by smoothing out n_2. For each state, those values not defined on it are obtained by complementing the relation, and we get T_2^C. An output assignment v that causes transitions in T_2^C will force $M_1'|_{\mathcal{M}(v)} \notin \mathcal{L}(M_2^i)$. Consequently, $\mathcal{W}(p,v)$ needs to include T_2^C. The rest of the proof is the same as that of Theorem 5.4. ∎

We can compute the feasible output assignment function $f(v)$ using Theorem 5.4 or 5.5 depending on whether M_2 is input completely specified. As an example, consider the cascade machine in Figure 5, where M_2 is completely specified. The feasible output assignments $f(v)$ can be computed by the above algorithms.

$$f(v_1,v_2,v_3,v_4,v_5,v_6) = \overline{v_1}\,v_2\,\overline{v_3}\,v_4\,\overline{v_5}\,v_6 + \overline{v_1}\,v_2\,\overline{v_3}\,\overline{v_4}\,v_5\,\overline{v_6} + v_1\,\overline{v_2}\,v_3\,\overline{v_4}\,v_5\,\overline{v_6} + v_1\,\overline{v_2}\,v_3\,v_4\,\overline{v_5}\,v_6\,.$$

5.3 Relationship Between Feasible Output Assignments and Permissible Observability Relations

The feasible output assignment function, $f(v)$, is a multiple-valued function, thus it can be expressed in terms of a multiple-valued sum-of-products.

Lemma 5.6 *Let c be a multiple-valued cube in the space of V. $T_1|_{\mathcal{M}(c)}$ can be expressed using a single symbolic relation. Conversely, the output assignments of a symbolic relation $T_1(i_1,p_1,n_1,o_1)$ can be expressed in terms of a multiple-valued cube in the space of V.*

Proof Let c be $v_0^{L_0} v_1^{L_1} \cdots v_{k-1}^{L_{k-1}}$, where $L_0, L_1, \ldots, L_{k-1}$ are subsets of O_1. $T_1|_{\mathcal{M}(c)}$ can be achieved by substituting every L_j in the output of transition e_j of M_1. Consequently, it can be expressed in terms of a single symbolic relation. Conversely, let the output values of a transition edge e_j in T_1 be L'_j, a subset of O_1. We can express the output assignments of T_1 as $v_0^{L'_0} v_1^{L'_1} \cdots v_{k-1}^{L'_{k-1}}$, which is a cube in the space of V. ■

Definition 13 *Let c be a cube of $f(v)$, the feasible output assignment function. $T_1|_{\mathcal{M}(c)}$ is called a* **permissible observability relation** *in the sense that any implementation compatible with $T_1|_{\mathcal{M}(c)}$ is feasible.*

Definition 14 *Let p be a prime of $f(v)$, the feasible output assignment function. Then $T_1|_{\mathcal{M}(p)}$ is called a* **prime permissible observability relation** *in the sense that it is not contained in any other permissible observability relations.*

Theorem 5.7 *Let $f(v)$ be the feasible output assignment function. $\{T_1|_{\mathcal{M}(f(v))}\}$ is a set of permissible observability relations.*

Proof Let $f(v)$ be expressed in terms of a sum-of-products, $\{c_1, c_2, \ldots, c_n\}$. From Lemma 5.6, a cube c_i of $f(v)$ corresponds to a permissible observability relation $T_1|_{\mathcal{M}(c_i)}$. Therefore, $\{T_1|_{\mathcal{M}(f(v))}\}$ corresponds to a set of permissible observability relations, $\{T_1|_{\mathcal{M}(c_1)}, T_1|_{\mathcal{M}(c_2)}, \ldots, T_1|_{\mathcal{M}(c_n)}\}$. ■

Therefore, all feasible output functions can be expressed in terms of a set of permissible observability relations, and vice versa. By Lemma 5.6, a permissible observability relation covers some feasible output functions. Therefore, the representation for a set of permissible observability relations is not unique. *The minimum set of permissible observability relations* is a cover which covers all feasible output functions with the minimum number of permissible observability relations.

Theorem 5.8 *The cardinality of the minimum set of permissible observability relations is equal to the cardinality of the minimum sum-of-products cover of $f(v)$, the corresponding feasible output assignment function.*

Proof Any feasible output assignment is contained in a minimum sum-of-products cover of $f(v)$, $C_{f(v)} = \{p_1, p_2, \ldots, p_n\}$. The corresponding set of permissible observability relations, $C_{\mathcal{M}(f(v))} = \{T_1|_{\mathcal{M}(p_1)}, T_1|_{\mathcal{M}(p_2)}, \ldots, T_1|_{\mathcal{M}(p_n)}\}$, is a minimum cover. Suppose $C_{\mathcal{M}(f(v))}$ is not a minimum cover of permissible observability relations, then there exists a cover $C'_{\mathcal{M}(f(v))}$ with smaller cardinality. By Lemma 5.6, this implies $C_{f(v)}$ is not the minimum cover of $f(v)$. This is a contradiction. ■

In contrast, we only need to use a single observability relation, say $\mathcal{O}(i, o)$ (where i is the input, and o is the output), to express all the flexibility of implementation for a component in a hierarchically designed combinational logic circuit. A minterm of i may map to several minterms of o, and it is *independent* of other minterms of i. However, this is not true in the sequential case. That is, a minterm of i may map to several minterms of o, but it is *dependent* on the other minterms of i. Consequently, the notion of Boolean relations must be generalized for hierarchical designed sequential circuits, i.e. *sets of permissible observability relations.*

5.4 Permissible Observability Relations vs. Output Don't Care Sequences

In the next theorem, we state the relationship between the set of permissible observability relations and output don't care sequences.

Theorem 5.9 *If the transition function of M_1 is not changed, and $f(v)$ is the feasible output assignment function, then the corresponding set of permissible observability relations, $\{T_1|_{\mathcal{M}(f(v))}\}$, exactly captures output don't care sequences for M_1.*

In a sense, our algorithms compute a class of *permissible machines*, and these machines can be expressed in terms of a set of permissible observability relations.

Although we have only considered one-way-communication circuits (i.e. cascade machines) so far, the above algorithms can be directly applied to compute permissible relations for a component in a *two-way-communication* circuit as shown in Figure 9. An interesting extension is to use these algorithms to compute permissible relations for both submachines M_1 and M_2 in Figure 9 simultaneously. We simply keep both state transition functions of M_1 and M_2 intact, and then use the above algorithms to get the feasible output function assignments of both machines simultaneously, i.e. the new output assignment space is the Cartesian product of output assignment spaces of M_1 and M_2. Note that the flexibility in implementing one submachine is *dependent* on that of the other. That is, their flexibility of implementation need to be **compatible**. With this approach, we are able to compute this **compatible flexibility** in implementing these submachines.

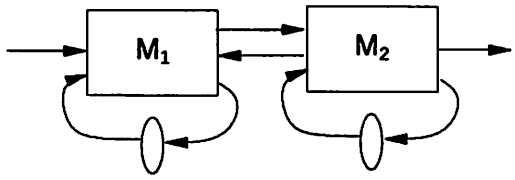

Figure 9: A two-way-communication FSM network.

5.5 Further Discussion

A restriction on our algorithms is that during the computation step the state transition function of M_1 is kept unchanged, although the state transition function may change after the exploitation of output don't care information using state minimization procedures for incompletely specified FSM's. This may limit possible exploitation of other output don't care information. Therefore, the don't care information derived using our algorithms may be affected by the given state transition function of M_1. In contrast, Rho's procedure [17] employs a heuristic based on *state splitting* to accommodate don't care information (a subset of output don't care sequences with a fixed-length) as much as possible. Therefore, the state transition function of the component may change. This limits the exploitation of other possible output don't care information as well since the definitions of *equivalent sequences* in [17] is not general as explained in Section 3.3. Moreover, our algorithms implicitly enumerate *infinite-length* output don't care sequences using FSM equivalence checking, while Rho's procedure is limited to *fixed-length* sequences. Besides, our algorithms can be directly applied to handle two-way-communication circuits as shown Figure 9, and can be extended to compute *compatible flexibility* in implementing these two interacting components.

Consider the following case. Let M_1' be a machine with the same input and output alphabets as those of M_1 but with a different state space Q_1' and a different transition function δ_1'. Our algorithms can be applied to check if there exists an output function λ_1' such that $M_1' \rightarrow M_2$ and $M_1 \rightarrow M_2$ have the same I/O behavior.

Based on the above discussion, a possible approach to effectively compute and exploit output don't care sequences is as follows. Some fixed-length output don't sequences can be exploited by using state splitting. Therefore, the state transition function of M_1 changes. Infinite-length output don't care sequences can be implicitly enumerated to check if there exist better feasible output functions under this new transition function. Thus, how to perform state splitting properly such that as much output don't care information as possible can be exploited is of interest. Currently, we are investigating this problem.

6 Exploitation of Permissible Observability Relations

In the following, we discuss different aspects of exploiting permissible observability relations.

6.1 State Minimization

Limited observability may be beneficial in minimizing the number of states of a machine. A machine with the minimum number of states may not have the best implementation. However, it can be a good starting point for state assignment if the machine is not encoded yet, or for sequential logic optimization if the machine is encoded.

However, current state minimization algorithms only manipulate one observability relation (transition relation) at a time. By Lemma 5.6, a cube of $f(v)$ corresponds to a permissible observability relation. So in order to fully exploit the set of permissible observability relations, one must run state minimization procedures several times. Let p be a prime of $f(v)$. Then $T_1|_{\mathcal{M}(p)}$ is a prime permissible observability relation which can not be contained in any other permissible observability relations. *Therefore, a state minimization procedure needs to be executed as many times as the number of primes of* $f(v)$.

6.2 Implementation

In the case of combinational logic, a Boolean relation is sufficient to capture all the freedom of implementation. However, in the case of FSM networks, a set of permissible observability relations may be needed to represent output don't care sequences for a component. Each minterm of $f(v)$ corresponds to a feasible implementation. Now, suppose the machine is encoded. In order to find the best implementation, we need to consider every minterm of $f(v)$. From Theorem 5.8, *the minimum number of times we need to run the Boolean relation minimizer to find the best implementation is equal to the number of terms in a*

circuit	driving			driven			feasible	permissible	CPU
	PI	PO	states	PI	PO	states	assignments	relations	time
P	1	1	3	1	1	3	4	4	0.1
bbara-bbtas	4	2	7	2	2	6	203584	6	51.5
bbtas-ex5	2	2	6	2	2	8	2096	5	1.9
bbtas-ex7	2	2	6	2	2	8	160512	4	1.6
dk27-lion	1	2	7	2	1	4	499941	18	41.1
ex2-ex5	2	2	19	2	2	8	32768	1	57.8
ex2-ex7	2	2	19	2	2	8	8704	2	20.4
ex3-bbtas	2	2	11	2	2	6	81	1	4.2
ex3-ex7	2	2	11	2	2	8	12	1	7.9
ex5-bbtas	2	2	8	2	2	6	243	1	2.6
ex5-ex7	2	2	8	2	2	8	128	1	3.2
ex3-ex5	2	2	11	2	2	8	1	1	10.9
ex2-ex3	2	2	19	2	2	11	1	1	20.9

Table 1: Experimental results

states : number of states in isolation
feasible assignments : number of feasible output assignments of the driving machine
permissible relations : minimum number of permissible observability relations
CPU time : in seconds on a DEC 5000/260 with 128 MB

minimum sum-of-products for $f(v)$. The rationale is as follows : A product term in the minimized sum-of-products form of $f(v)$ corresponds to a Boolean relation, and every minterm of $f(v)$ is covered by the sum-of-products form with minimum cardinality. Consequently, we can pick the best result from all Boolean minimization executions.

6.3 Encoding of Interconnection Signals

Another direct application to permissible observability relations is the encoding of signals between interacting FSM's. We may convert the set of interacting binary signals between components into a symbolic variable for the purpose of *re-encoding*. For example, let us consider a cascade machine $M_1 \rightarrow M_2$. A good output encoding of M_1 may be a bad input encoding for M_2, and vice versa. *Re-encoding* can be imagined as a means of moving logic between two interacting machines [9].

Shen *et al.* [21] give a formulation to this problem but without experimental results. They simply combine the I/O constraints from each individual component and convert them into a dichotomy covering problem with some conflict resolution techniques. Then they try to satisfy as many constraints as possible. However, this is not an exact formulation since it does not consider the exact output encoding (e.g., GPI's [10]) or any don't care information (e.g. symbolic relation). Moreover, they did not consider sequential don't cares (e.g. permissible observability relations and input don't care sequences are not used). The general formulation is still an open problem.

Permissible observability relations allow us to have many feasible output functions, and in theory these should be useful for encoding the interacting signals.

7 Experimental Results

In this section, we present some preliminary results. Most of the examples are obtained by interconnecting FSM's from the MCNC benchmarks. These FSM's are completely specified and state-minimal in isolation. Example `P` is shown in Figure 5.

Table 1 shows the results of some cascade circuits consisting of two FSM's. The minimum number of permissible relations is obtained using ESPRESSO-MV [19]. The CPU time indicated includes both the computation of feasible output assignments and ESPRESSO-MV. Although the feasible output functions of examples `ex2-ex5`, `ex3-bbtas`, `ex3-ex7`, `ex5-bbtas` and `ex5-ex7` can be expressed by a single relation individually, they cannot be computed using either Devadas' or Rho's procedures. In examples `P`, `bbara-bbtas`, `bbtas-ex5`, `bbtas-ex7`, `dk27-lion` and `ex2-ex7`, the minimum

number of permissible relations to express the feasible output functions is more than one. The number of feasible output functions in examples ex3-ex5 and ex2-ex3 is one.

Our implicit algorithm based on BDD's deals with all output assignments at a time. With our current implementation, we can handle small-size examples in a reasonable amount of CPU time as shown in Table 1. In contrast, the explicit algorithm which enumerates all possible output assignments one by one is very inefficient since the number of all output assignments is too large. For instance, there are 2^{152} possible output assignments in example ex2-ex7, but only 8704 of them are feasible. The feasible output assignment function, $f(v)$, is normally a relatively small subset of V, the set of all output assignments. Therefore, proper BDD variable ordering or use of 0-Sup-BDDs [15] should enhance the ability and efficiency of our algorithms. Currently, we are studying a good BDD variable ordering to handle larger examples.

The output part of a transition edge in a permissible relation is a multiple-valued literal. As pointed out in [9], pairwise compatibility of a set of states S does not imply S is compatible. Thus, additional checking has to be performed during state minimization. At the present time, there are no state minimization programs with this ability in our logic synthesis system.

8 Conclusion

We discussed intrinsic difficulties in computing output don't care sequences for a component in an FSM network. We pointed out that these can not be explained using information lossyness [17]. We demonstrated that output don't care sequences for a component can be expressed using a set of permissible observability relations given that its state transition function is kept unchanged. We presented a novel approach to exactly compute them. The representation for a set of permissible observability relations is not unique. We provided a method to find a set with the minimum number of permissible observability relations. We also discussed the applications of permissible observability relations in different contexts, such as state minimization, circuit implementation and signal encoding.

9 Acknowledgements

The authors are thankful to Dr. June-Kyung Rho for providing valuable information. We thank Dr. Rajeev Murgai for reading and improving this manuscript. Also special thanks to Szu-Tsung Cheng and Thomas Shiple for helpful discussions on the BDD package.

References

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[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] S. Devadas and A. R. Newton. Exact Alogrithms for Output Encoding, State Assignment and Four-level Boolean Minimization. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 13–27, January 1991.

[11] G. 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.

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

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

[14] 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.

[15] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *30th ACM/IEEE Design Automation Conference*, pages 272–277, June 1993.

[16] M. C. Paull and S. H. Unger. Minimizing the Number of States in Incompletely Specified Sequential Circuits. In *IRE Transactions on Electronic Computers*, pages 356–366, September 1959.

[17] J. K. Rho, G. 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.

[18] J. K. Rho, G. Hachtel, and F. Somenzi. Don't Care Sequences and the Optimization of Interacting Finite State Machines. In *International Workship on Logic Synthesis*, May 1991.

[19] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. In *IEEE Transactions on Computer Aided Design of Integrated Circuit and Systems*, pages 727–750, 1987.

[20] 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.

[21] J. J. Shen, Zafar Hasan, and M. J. Ciesielski. State Assignment for General FSM Networks. In *The European Conference on Design Automation*, pages 245–249, 1992.

[22] 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.

[23] 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.

[24] 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.

[25] 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.