A New Algorithm for Exact Reduction of Incompletely Specified Finite State Machines

Jorge M. Pena and Arlindo L. Oliveira, *Member, IEEE*

***Abstract*— We propose a new algorithm for the problem of state reduction in incompletely specified finite state machines. Unlike the most commonly used algorithms for this problem, our approach is not based on the enumeration of compatible sets, and, therefore, its performance is not dependent on its number. Instead, the algorithm uses techniques for finite state machine identification that are well known in the computer science literature, but have never been applied to this problem. We prove that the algorithm is exact and present results that show that, in a set of hard problems, it is much more efficient than both the explicit and implicit approaches based on the enumeration of compatible sets. We also present a complexity analysis for the special cases where worst case polynomial time bounds can be obtained and present experiments that validate empirically the bounds obtained.**

I. Introduction

THE reduction of finite state machines (FSM's) is a well-known problem of great importance in sequential circuit synthesis, since the complexity of the final implementation of a finite state controller is related with the number of states in its state transition diagram (STG). This is specially important when synthesis tools are used in the synthesis process because suboptimal STG representations are commonly generated by this approach.

For completely specified FSM's, the state reduction problem can be solved in polynomial time [1], [2], the most efficient algorithm being due to Hopcroft [3]. For incompletely specified FSM's (ISFSM's), the problem is known to be NP-complete [4]. Nevertheless, exact and heuristic algorithms are commonly used in practice [5], [6], and it is possible, in many cases of practical importance, to obtain exact solutions.

The standard approach for this problem is based on the identification of sets of compatible states, (or compatibles) and the solution of a binate covering problem. A number of practical systems based on this approach has been proposed, based on both explicit [7] and implicit enumeration [8] of the compatibles. However, some problems cannot be solved using this approach for one of two reasons: 1) the set of prime compatibles is too large to be listed, either explicitly or implicitly or 2) the binate covering problem can be formulated but takes too long to solve. We review these algorithms and concepts in Section III, after introducing the basic definitions in Section II.

The approach we propose is based on a different paradigm, thus avoiding altogether the identification of prime compatibles and the solution of a covering problem. Our method uses techniques well known in the computer science community for the identification of FSM's consistent with a given set of input/output mappings. These techniques were originally aimed at identifying the minimum state deterministic finite automator (DFA) that generates a given grammar, given a set of examples of strings in the grammar, together with examples of strings not in the grammar. Although all these results and algorithms have used the DFA formalism, they can be easily translated to similar results formulated in terms of FSM's.

The complexity of the problem of DFA identification (and, therefore, of minimum FSM identification) varies with the ability of the algorithm to control the set of input sequences for which outputs are known. If this set is given and fixed, it has been proved that given a finite alphabet Σ, two finite subsets $S, T \subseteq \Sigma^*$ and an integer k, determining if there is a k-state DFA that recognizes L such that $S \subset L$ and $T \subset \Sigma^* - L$ is NP-complete [9]. In fact, even finding a DFA with a number of states polynomially larger than the number of states of the minimum solution is NP-complete [10]. Under these conditions, the most efficient search algorithms for this problem are based on the approach proposed by Bierman [11], [12].

The problem becomes easier if the algorithm is allowed to make queries or experiment with the unknown machine. Hennie was the first to address this problem [13] and Angluin proposed the L* algorithm [14] that solves the problem in polynomial time by allowing the algorithm to ask membership queries, based on the approach described by Gold [15].

Since we make extensive use of both Bierman's approach and Angluin's L* algorithm, we describe these algorithms in some detail in Section IV.

Section V contains the central contribution of this work. We show that it is possible to use a modified version of Angluin's technique to identify the minimum FSM equivalent with a given ISFSM without enumerating the set of compatibles. We also show that the algorithm runs in time polynomial on the number of states, for the special case where the original machine is completely specified.

Section VI generalizes the algorithm to handle the case where unspecified input values are used in the description of the original ISFSM, a case that is of practical interest although it is not central for understanding the key ideas of the algorithm.

Manuscript received May 29, 1998; revised January 5, 1999. This paper was recommended by Associate Editor D. Dill.

J. M. Pena is with the Department of Informatics of IST, Lisbon Technical University, 1000 Lisboa, Portugal.

A. L. Oliveira is with the Department of Informatics of IST, Lisbon Technical University, 1000 Lisboa, Portugal, and is also with INESC, R. 1000 Lisboa, Portugal, and the Cadence European Laboratories, 1000 Lisboa, Portugal (e-mail: aml@inesc.pt).

Publisher Item Identifier S 0278-0070(99)09478-6.

Section VII describes the results we obtained in a set of FSM's that have been used to evaluate the performance of state reduction algorithms [8]. These results show that the algorithm can be much more effective than alternative methods in problems that exhibit a very large number of prime compatibles. This section also presents empirical evidence that for completely specified FSM's the run time complexity of the algorithm is bounded by a low-degree polynomial.

II. Definitions

This section introduces some general definitions that will be used throughout the paper. Other more specific definitions will be introduced as they are needed.

Definition 1: A FSM is a tuple $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ where $\Sigma \neq \emptyset$ is a finite set of input symbols, $\Delta \neq \emptyset$ is a finite set of output symbols, $Q \neq \emptyset$ is a finite set of states, $q_0 \in Q$ is the initial "reset" state, $\delta(q,a) : Q \times \Sigma \rightarrow Q \cup \{\phi\}$ is the transition function, and $\lambda(q,a) : Q \times \Sigma \rightarrow \Delta \cup \{\tau\}$ is the output function.

We will use $q \in Q$ to denote a particular state, $a \in \Sigma$ a particular input symbol and $b \in \Delta$ a particular output symbol. A FSM is incompletely specified if the destination or the output of some transition is not specified. For incompletely specified machines, ϕ denotes an unspecified next state while τ denotes an unspecified output.

Definition 2: An output b_i is compatible with an output b_j ($b_i \equiv b_j$) iff $b_i = b_j$ or $b_i = \tau$ or $b_j = \tau$.

We will always use quoted symbols (M', Q', etc.) to refer to the original, unreduced machines, and unquoted symbols to refer to the final, reduced, machines.

The domain of the second variable of functions λ and δ is extended to strings of any length in the usual way. Let $s = (a_1, \ldots, a_k)$ be a string of input symbols and the notation $\lambda(q,s)$ denote the final output of the FSM after sequence s is applied in state q. The output of such a sequence is defined to be $\lambda(q,s) = \lambda(\delta(\delta(\cdots\delta(q,a_1)\cdots),a_{k-1}),a_k)$. Similarly, $\delta(q,s)$ denotes the final state reached by a FSM after a sequence of inputs $(a_1, \ldots, a_k)$, is applied in state q. The final state is defined to be $\delta(q,s) = \delta(\delta(\cdots\delta(\delta(q,a_1),a_2)\cdots),a_k)$. To avoid unnecessary notational complexities, $\lambda(\phi,a)$ is defined to be equal to τ and $\delta(\phi,a) = \phi$.

Definition 3: A tree FSM (TFSM) M is a FSM satisfying Definition 1 and the following additional requirements:

$$\forall q \in Q \backslash q_0, \quad \exists^1 s \in \Sigma^z \quad \text{s.t. } \delta(q_0,s) = q$$
$$\forall q \in Q, \quad \forall a \in \Sigma \delta(q,a) \neq q_0.$$

These requirements specify that there exists one and only one string taking the machine from state q_0 to any other state q and that state q_0 is not reachable from any other state. This is the same as saying that the graph that describes the TFSM is a tree rooted at state q_0. We say that a TFSM M contains a string s iff the application of string s in state q_0 leads to a state in M, i.e., iff $\delta(q_0,s) \in Q$. Fig. 1 shows an example of a TFSM.[1]

[1]It is useful to view ϕ as another state (not shown) that is reached by all strings not contained in M.

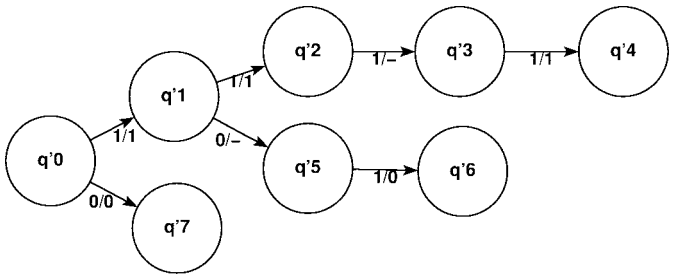


Fig. 1. Example of a TFSM.

Definition 4: A completely specified FSM $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ is compatible with an incompletely specified FSM $M' = (\Sigma, \Delta, Q', q_0', \delta', \lambda')$ iff, for every input string s, the output of M is compatible with the output of M', i.e., $\lambda(q_0,s) \equiv \lambda'(q_0',s)$.

Note that we define compatibility only for strings applied at the reset state. It is possible to define compatibility in a slightly different form that is not strictly equivalent to this one for STG's that have states unreachable from the reset state. We believe that for the purposes of FSM reduction, this definition is the appropriate one.

Definition 5: Two states are compatible iff the output sequences of the FSM initialized in the two states are compatible for any input sequence.

If two states q_i and q_j are not compatible, they are incompatible and we write $I(q_i,q_j) = 1$. Otherwise, $I(q_i,q_j) = 0$. Finally, we have the definition of a compatible set [5], that will be used in the description of the state reduction algorithms in Section III.

Definition 6: A set of states is a compatible iff for each input sequence there is a corresponding output sequence which can be produced by each state in the compatible.

It is known that, for FSM's, it is sufficient to have pairwise compatibility between each state in the compatible ([5], Theorem 4.3).

Theorem 1: A set of states is a compatible iff all the states in the set are pairwise compatible.

Proof: see [5]. □

III. Classic Methods for the Reduction of ISFSM's

The standard approach for the reduction of incompletely specified FSM's is based on the enumeration of the set of compatibles [16] and on the satisfaction of a set of restrictions that can be viewed as a covering problem. The objective of this section is not to give a complete overview of this field, but only to introduce the basic ideas underlying the well-known techniques for FSM reduction. Readers interested in a comprehensive treatment of this topic are referred elsewhere for details [5]. It is known that the minimum compatible FSM can be found through the enumeration of the set of compatibles and the identification of a minimum-cardinality closed cover of compatibles.

Definition 7: Compatible cover: a set of compatibles $S = \{C_1, C_2, \ldots, C_n\}$ is said to be a cover for the states in $M' = (\Sigma, \Delta, Q', q_0', \delta', \lambda')$ iff every state in Q' belongs to some compatible in S.

To solve the ISFSM reduction problem, it is not enough to select a set of compatibles that cover the states in the original ISFSM since an additional closure condition also has to be imposed. This closure condition is based on the definition of *implied sets*.

Definition 8: The implied set for a compatible C under input a is the set of states $D_a(C)$ that are the next states reachable from the states in C under input a.

The closure condition that needs to be imposed is the following.

Definition 9: Closed cover: a set of compatibles S is called a closed cover iff for each compatible C_j in S, each of its implied sets is covered by some C_k in S.

Obtaining the minimum compatible FSM is, therefore, equivalent to the selection of a minimum cardinality cover that obeys the covering and closure requirements stated above. This can be formulated as a binate covering problem, and solved using one of the many proposed approaches for the solution of this type of problems [17], [5].

There are several optimizations that can be applied to the problem and used to reduce the size of the binate table. It is trivial to notice that one can ignore all implied sets of cardinality one and all implied sets that are covered by the compatible that implied them. Grasselli and Luccio proved [18] that only a subset of the compatibles needs to be considered, the compatibles that are not dominated by any other compatible.

Definition 10: A compatible C' dominates a compatible C if $C' \supset C$ and $\forall a D_a(C') \subseteq D_a(C)$.

Compatibles that are not dominated by any other compatible are called prime compatibles and only these need to be considered in the binate covering problem. Compatibles that are not properly contained in any other compatible are called maximal compatibles, and they are always prime compatibles.

Even with these optimizations, many state reduction problems imply the consideration of a very large number of compatibles. In the worst case, the number of compatibles grows exponentially with the number of states in the ISFSM. For this reason, the possibility of using an implicit algorithm to enumerate the compatibles and formulate the covering problem has received a great deal of attention. In particular, Kam and Villa have proposed a fully implicit approach [8] that performs all the necessary computations without ever listing, in an explicit form, the compatibles. This approach has the ability to handle problems that exhibit a very large number of compatibles, although this ability depends on the existence of a compact representation for the set of compatibles. The results of this approach, as well as the best-known implementation of the explicit approach [7] are compared to our algorithm in Section VII.

It is also known that the covering clauses are not necessary, and that only the reset state needs to be covered [19].[2] In fact, the closure clauses are such that a solution to the problem necessarily covers all other reachable states, as long as the reset state itself is covered.

[2]Note that if the covering clauses are not included, it is not possible to ignore implied sets of cardinality one.

TABLE I
EXAMPLE OF A SET OF I/O MAPPINGS

1/1	1/1	1/-	1/1
0/0			
1/1	0/0	1/0	

IV. FSM IDENTIFICATION FROM INPUT–OUTPUT ((I/O) SEQUENCES

A. Identifying FSM's from Specified I/O Sequences

Consider the situation where the behavior of a FSM is specified by a given I/O mapping, where an I/O mapping is defined in the following way:

Definition 11: An I/O mapping is a sequence of input/output pairs $((a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)) \in (\Sigma \times (\Delta \cup \{\tau\}))^k$ that specify the value observed in the outputs of a FSM under a specific sequence of inputs.

Consider now the following question: given a set of I/O mappings R, what is the FSM with minimum number of states that exhibits a behavior compatible with that set of I/O mappings?

This problem is NP-complete, and is equivalent to the problem addressed by Gold [9] of identifying the minimum state DFA that accepts a given set of strings and rejects another set. It turns out that there is a trivial equivalence between this problem and the problem of reducing FSM's of a special type, TFSM's.

B. TFSM's

From a set of I/O mappings, it is straightforward to generate a FSM that exhibits a behavior compatible with it. Simply generate the trie (or prefix tree) that corresponds to the input strings in the set, and label the transitions in accordance with the given I/O mapping.

As an example, consider the set of (three) I/O mappings shown in Table I. It is trivial to observe that the TFSM shown in Fig. 1 generates the desired outputs for this set of I/O mappings. Since the TFSM generated in this way exhibits a behavior compatible with the given set of I/O mappings, we can select the minimum FSM consistent with this set by reducing the TFSM, i.e., by selecting the minimum FSM compatible with this TFSM. It turns out that TFSM's can be reduced using algorithms other than the ones described in Section III. This happens because, unlike general FSM's, TFSM's can be reduced by selecting a valid mapping function from the states in the TFSM to the final, reduced, FSM.

Definition 12: Let $M' = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$ be an incompletely specified FSM and $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ be completely specified. A function $F : Q' \rightarrow Q$ is a valid mapping function iff it satisfies

$$\forall q' \forall a \lambda'(q', a) \equiv \lambda\Big(F(q'), a\Big) \tag{1}$$

$$\forall q' \forall a F\Big(\delta'(q', a)\Big) = \delta\Big(F(q'), a\Big). \tag{2}$$

The first equation states that F is a valid mapping function only if it maps each state q' in M' to a state $q = F(q')$ in M that exhibits an output compatible with q' for every possible input. The second equation states that, under each possible input, the next state is also correctly mapped. The importance of a mapping function is given by the following theorem.

Lemma 1: If F is a valid mapping function between M' and M, then machine M is compatible with M'.

Proof: By Definition 4, we need to prove that, if there exists a valid mapping function F, then $\lambda(q_0, s) \equiv \lambda'(q'_0, s)$, for all strings $s = \{a_1, a_2, \ldots, a_k\}$. Assuming that $q_0 = F(q'_0)$, we have

$$\begin{aligned}
\lambda(q_0, s) &= \lambda(\delta(\delta(\cdots\delta(q_0, a_1)\cdots), a_{k-1}), a_k) \\
&= \lambda(\delta(\delta(\cdots\delta(F(q'_0), a_1)\cdots), a_{k-1}), a_k) \\
&= \lambda(\delta(\delta(\cdots F(\delta'(q'_0, a_1))\cdots), a_{k-1}), a_k) \\
&= \cdots = \lambda(\delta(F(\delta(\cdots\delta'(q'_0, a_1))\cdots), a_{k-1}), a_k) \\
&= \lambda(F(\delta'(\delta'(\cdots(\delta'(q'_0), a_1)\cdots), a_{k-1})), a_k) \\
&\equiv \lambda'(\delta'(\delta'(\cdots(\delta'(q'_0), a_1)\cdots), a_{k-1}), a_k) \\
&= \lambda'(q'_0, s).
\end{aligned}$$

□

It is well known that the existence of a mapping function is not a necessary condition for compatibility between two FSM's. For example, machine M in Fig. 2 is compatible with machine M', but no mapping function exists between M' and M. To see why this is the case, note that states q'_0 and q'_1 in machine M' cannot be compatible. Therefore, the reduced machine M needs to have at least two states, with the value of the corresponding outputs being defined by the transitions in M'. Therefore, q'_0 maps to q_3 and q'_1 maps to q_4. Now, q'_2 cannot map to q_3 because, in that case, $q_3 = F(q'_2) = F(\delta'(q'_2, 1)) \neq \delta(F(q'_2), 1) = \delta(q_3, 1) = q_4$ and cannot map to q_4 because then $q_4 = F(q'_2) = F(\delta'(q'_2, 1)) \neq \delta(F(q'_2), 1) = \delta(q_4, 1) = q_3$. This happens because state q'_2 in machine M' becomes *split*, its functionality being partially performed by state q_3 and partially by state q_4 in machine M.

It turns out that, if certain restrictions are imposed on machine M', a mapping function between M' and a compatible machine M will always exist.

Let $M' = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$ be a TFSM and $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ an arbitrary FSM. Consider now a relation F between the states of M' and the states of M defined as follows.

Definition 13: Let $F : Q' \rightarrow Q$ be defined by $F(\delta'(q'_0, s)) = \delta(q_0, s)$ for each string s contained in M'.

Then, the following lemma applies.

Lemma 2: F is a many to one mapping, mapping each state in M' to one unique state in M.

Proof: Since M' is an TFSM each state in M' can be reached by **one and only one** string. Therefore, the definition of F will assign a unique state in M to each state in M'.

□

Note that if such a definition of F is applied to a machine M' that is not an TFSM, as, for example, the one in Fig. 2(a), this lemma is not true. Because a state in M' can be reached my more than one string, there is no warranty that F is a function. In particular, consider the strings 100 and 101 and try to apply Definition 13: string 100 leads to state q'_2 in M' and to state q_3 in M while string 101 leads to state q'_2 in M' and to state q_4 in M.

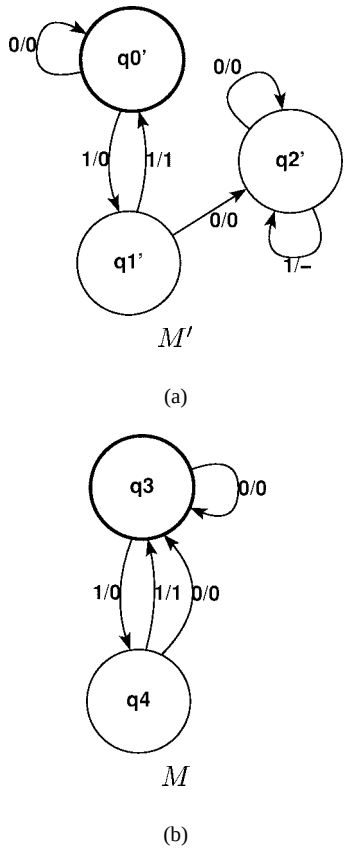


Fig. 2. Machine M is compatible with M', but no mapping function between M' and M exists.

Theorem 2: Let M' be a TFSM. Then, for any machine M compatible with M', the function F as defined above is a valid mapping function between the states of M' and the states of M.

Proof: Since M is compatible with M', it gives an output compatible with M', for every string s applied at the reset state, i.e., $\lambda(q_0, s) \equiv \lambda'(q'_0, s)$. Consider two states $q' \in Q'$ and $q \in Q$ and assume that $F(q') = q$ because $\delta'(q'_0, s) = q'$ and $\delta(q_0, s) = q$ (Definition 13). Now, $\lambda'(q', a) = \lambda'(q'_0, sa) \equiv \lambda(q_0, sa) = \lambda(\delta(q_0, s), a) = \lambda(q, a)$ and, therefore, (1) is respected. On the other hand, $F(\delta'(q', a)) = F(\delta'(q'_o, sa)) = \delta(q_0, sa) = \delta(\delta(q_0, s), a) = \delta(q, a) = \delta(F(q'), a)$ and, therefore, (2) is also respected. Therefore, F is a valid mapping function. □

The result of this theorem is critically important, because is strongly limits the solution space for machines that are TFSM's. This result has been implicitly used by Bierman [12] (although not stated in this form) and a slightly different proof has been presented in [20]. The fact that a solution can be found by selecting a mapping function enables us to use totally different algorithms to solve TFSM's than the ones used to solve standard ISFSM's. Theorem 2 says that, to select a minimum sized FSM compatible with an TFSM machine M', one needs not consider all possible relations F, but only those that are many to one mappings.

If machine M' has m states and is compatible with a machine M with n states, there are a maximum of n^m possible mapping functions, while there exist 2^{m^n} possible mapping relations. Although both numbers are extremely large, it is possible to design an algorithm that searches for the right mapping function in a way that can be more efficient than the standard state-reduction algorithm that has to search for a valid mapping relation.

C. Reduction of TFSM's

As shown in the previous sections, the reduction of a TFSM can be performed by selecting a valid mapping function F that has a range of minimum cardinality. To simplify the exposition, let S_i denote the index of the state in M that will be the mapping of q'_i, i.e., $q_{S_i} = F(q'_i)$. The constraints that need to be obeyed by the mapping function can be restated as follows.

1) If two states q'_i and q'_j in the original TFSM are incompatible, then $S_i \neq S_j$.
2) If two states q'_i and q'_j have successor states q'_k and q'_l for some input u, respectively, then $S_i = S_j \Rightarrow S_k = S_l$.

These two constraints can be rewritten as

$$I(q'_i, q'_j) = 1 \Rightarrow S_i \neq S_j \tag{3}$$

and

$$q'_k = \delta'(q'_i, a) \wedge q'_l = \delta'(q'_j, a) \Rightarrow S_i \neq S_j \vee S_k = S_l. \tag{4}$$

The basic search algorithm for this problem was proposed by Bierman [11]. Later, the same author proposed an improved search strategy that is much more efficient in the majority of the complex problems [12]. Although a full description of the method is outside the scope of this paper, we present here the basic idea. The search algorithm assigns a value to S_i following the depth first state order, and backtracks when some assignment is found to be in error. The number of states of the target machine, n, is estimated from the size of one large clique in the incompatibility graph. If a search for a machine of that size fails, n is increased and the search restarted. Assume, for the moment, that a search is being performed by a machine with n states.

The basic search with backtrack procedure iterates through the following steps

1) Select the next variable to be assigned S_i from among the unassigned variables.
2) Extend the current assignment by selecting a value from the range $0 \cdots n-1$ and assigning it to S_i. If no more values exist, undo the assignment made to the last variable chosen and backtrack to that level.
3) If the current assignment leads to a contradiction, undo it and goto Step 2. Else goto Step 1.

This search process can be viewed as a search tree. Consider the example in Fig. 1. For this TFSM, the set of constraints and the conditions that generated these constraints are shown in Table II.

If one selects the variables in breadth first order, one is lead to the search tree depicted in Fig. 3, yielding the solution

TABLE II
CONSTRAINTS GENERATED FROM THE TFSM IN FIG. 1

Constraint	Condition imposing constraint
$S_0 \neq S_5$	$I(q'_0, q'_5) = 1$
$S_1 \neq S_5$	$I(q'_1, q'_5) = 1$
$S_3 \neq S_5$	$I(q'_3, q'_5) = 1$
$S_0 \neq S_1 \vee S_1 = S_2$	$q'_1 = \delta'(q'_0, 1) \wedge q'_2 = \delta'(q'_1, 1)$
$S_0 \neq S_1 \vee S_7 = S_5$	$q'_7 = \delta'(q'_0, 0) \wedge q'_5 = \delta'(q'_1, 0)$
$S_0 \neq S_2 \vee S_1 = S_3$	$q'_1 = \delta'(q'_0, 1) \wedge q'_3 = \delta'(q'_2, 1)$
$S_0 \neq S_3 \vee S_1 = S_4$	$q'_1 = \delta'(q'_0, 1) \wedge q'_4 = \delta'(q'_3, 1)$
$S_1 \neq S_2 \vee S_2 = S_3$	$q'_2 = \delta'(q'_1, 1) \wedge q'_3 = \delta'(q'_2, 1)$
$S_1 \neq S_3 \vee S_2 = S_4$	$q'_2 = \delta'(q'_1, 1) \wedge q'_4 = \delta'(q'_3, 1)$
$S_2 \neq S_3 \vee S_3 = S_4$	$q'_3 = \delta'(q'_2, 1) \wedge q'_4 = \delta'(q'_3, 1)$

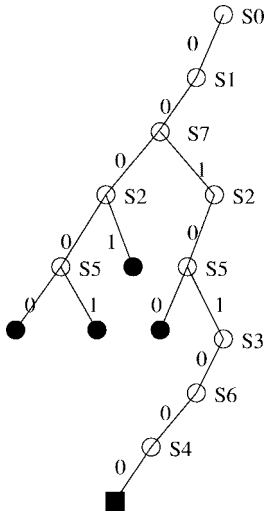


Fig. 3. Search tree for the reduction of the TFSM in Fig. 1.

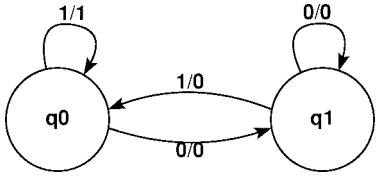


Fig. 4. Minimum FSM compatible with the TFSM in Fig. 1.

$S_0 = S_1 = S_2 = S_3 = S_4 = S_6 = 0$ and $S_5 = S_7 = 1$. In this tree, filled circles represent points where backtrack was forced, while a filled square means that a solution was found. Note that this solution is not unique, and that a different choice for the value of S_6 leads to a different machine. Once the mapping is obtained, the generation of the minimum compatible FSM is straightforward. There will be one state for each value in the range of the mapping function, and transitions will be labeled in accordance with the transitions specified in the TFSM. In this example, this mapping generates the minimum FSM shown in Fig. 4.

In more complex examples, finding an assignment to the variables S_i that satisfies the set of constraints may involve a considerable amount of search, given the need to backtrack expressed in Step 2 of the TFSM reduction algorithm.

Bierman noted [12] that a more effective search strategy can be applied if some bookkeeping information is kept and used to avoid assigning values to variables that will later prove to generate a conflict. This bookkeeping information can also be used to identify variables that have only one possible assignment left and should, therefore, be chosen next. This procedure can be viewed as a generalization to the multivalued domain of the unit clause resolution of the Davis–Putnam procedure [21] and can be very effective in the reduction of the search space that needs to be explored.

This can be done by keeping, for each state q'_i in Q', a table with the possible states that it can map to, i.e., the possible values that S_i can take. Every time a value is assigned to some S_j, the tables for every state are updated. In some cases, this will lead to a unique choice of assignment for a given node, and that node should be selected next. Recently, the use of dependency-directed backtracking has been shown to improve considerably the efficiency of Bierman's search algorithm [22].

D. The L* Algorithm

The L* algorithm [14] is an algorithm that identifies the canonical (and, therefore, minimum-state) DFA that accepts a given language. We present here a modified version that identifies the minimum state Mealy machine that provides output consistent with a given set of input/output strings. Let M' be the completely specified machine to be reduced.[3] The algorithm works as follows

- Keep a table T filled in using queries. Table T has one row for each element of $s \in (S \cup Sa)$, $a \in \Sigma$ and one column for each $e \in E$ where S is prefix closed and E is suffix closed.
- $T(se) = \lambda'(q'_0, se)$.
- T is **closed** if $\forall t \in Sa, \exists s \in S$ s.t. $\mathrm{row}(t) = \mathrm{row}(s)$.
- T is **consistent** if $\forall s_1, s_2 \in S, \forall a \in \Sigma$ $\mathrm{row}(s_1) = \mathrm{row}(s_2) \Rightarrow \mathrm{row}(s_1 a) = \mathrm{row}(s_2 a)$.

The table T is updated in the following way.

1) Initialize S with the empty string ϵ and E with all input combinations.
2) If table is not closed select a string s that violates closeness and move it from Sa to S. Extend table using queries.
3) If table is not consistent because for strings s_1 and s_2, $\mathrm{row}(s_1) = \mathrm{row}(s_2)$ but $\mathrm{row}(s_1 a) \neq \mathrm{row}(s_2 a)$, add ae to E, where e is the label of the column that causes the inequality. Extend table using queries.
4) If the table is closed and consistent, then it uniquely defines a completely specified FSM. Generate this machine and make a compatibility query.
5) If a counter-example s is received, add s and all its prefixes to S and goto 2. Otherwise, the generated machine is the minimum FSM compatible with M'.

A complete analysis of the correctness of this algorithm is outside the scope of this paper. For the details of this algorithm, the author is referred to Angluin's original work [14]. We will simply illustrate the algorithm with a small example, and state the most important results. Suppose that the target machine M' is the one given in Fig. 5.

[3]Later, we will apply this algorithm to the reduction of ISFSM's, but here we should view M' as a completely specified FSM.

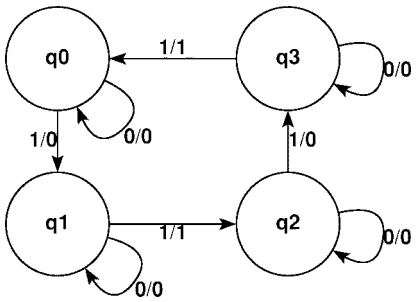


Fig. 5. Example of target machine for the L* algorithm.

TABLE III
INITIAL TABLE FOR THE L* ALGORITHM

	0	1
ϵ	0	0
0	0	0†
1	0	1

TABLE IV
EXAMPLE OF CLOSED AND CONSISTENT TABLE FOR THE L* ALGORITHM

	0	1
ϵ	0	0
1	0	1
0	0	0
11	0	0
10	0	1

The algorithm is initialized with the observation table shown in Table III. The entries in this table are initialized by simulating the strings in M'. For example, the entry marked with † is obtained by noticing that M' outputs 0 on the string 01, i.e., $\lambda'(q'_0, 01) = 0$. Table III is not closed, since there is no row in S that equals $\mathrm{row}(1)$. Therefore, move $\mathrm{row}(1)$ to S and extend table using queries, obtaining Table IV. This table is closed and consistent. Notice that there are no equal rows in S implying that the table is consistent and that all the rows in Sa are present in S. We can use this example to illustrate how a closed and consistent table defines a unique FSM. Consider the values in the rows as encodings for the states. In this case, we have two different labelings: 0, 0 and 0, 1. Now, it is simply a matter of filling the transitions between states in accordance with the row labels: for instance, string 1 takes us to state 0, 1; string 11 takes us to state 0, 0, and so on. The output values are filled in by looking at the values in the table. This procedure is linear on the size of the table. By applying it, we obtain the minimum machine compatible with the input/output relations specified in this table, as shown in Fig. 6. Since this machine is compatible with the original machine, the algorithm stops.

The fundamental results concerning this algorithm are the following.

Theorem 3: The L* algorithm runs in time polynomial on the number of states in the target machine and the maximum length of the counterexamples provided.

Proof: See [14]. □

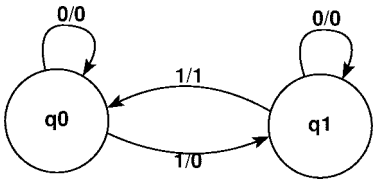


Fig. 6. Result of the L* algorithm in the machine of Fig. 5.

E. FSM Compatibility Checking

The compatibility check required by the L* algorithm is a problem that has been well studied and both implicit and explicit algorithms are known for this problem [23]. The compatibility check is based on the computation of the product machine.

Definition 14: The product machine of two completely specified FSM's $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ and $M' = (\Sigma, \Delta, Q', q_0', \delta', \lambda')$ is the FSM M'' defined by

$Q'' = Q \times Q'$, where the state $q_{ij}'' \in Q''$ corresponds to the pair of states q_i, q_j'.
$\delta''(q_{ij}'', a) = q_{kl}''$ where $q_k = \delta(q_i, a)$ and $q_l' = \delta'(q_j', a)$.
$\lambda''(q_{ij}'', a) = 1$ if $\lambda(q_i, a) = \lambda'(q_j', a)$, 0 otherwise

Once the product machine is known, compatibility can be checked by verifying if it is possible to reach a state that has an output value of 0. If this is the case, the machines are not compatible and the string that exhibits as a final output the value 0 is the proof that the machines are not compatible.

The computation of the product machine can be performed in a variety of ways. If network realizations of the two FSM's are available, the computation of the output function can be trivially accomplished with the addition of XOR gates and one NOR gate [23].

In the most interesting case for our purposes, we have only descriptions of the STG's of the two FSM's, and the computation of the STG of the product machine is required. This computation can be done either explicitly or implicitly. Implicit computation is critical when compatibility of FSM's with very large STG's is required. In this case, an implicit representation of the STG's is used to avoid enumeration of the states in the product machine [24].

For the application described in this work, the explicit computation of the product machine is not a limiting factor, and was the approach selected. However, in our case, it is important to generalize the compatibility check algorithm for the case where one or both the machines are incompletely specified. Note that in the L* algorithm, all compatibility checks are performed between two completely specified FSM's. However, in the sequence, we will need to perform compatibility checks between two incompletely specified FSM's and, therefore, the notion of FSM compatibility has to be extended.

First, let us consider the case where machine M is a completely specified FSM and machine M' is an incompletely specified FSM. In this case, the question *are the two machines compatible* can be answered using Definition 4. We will accept machine M as compatible with M' iff, for every possible input string, the output of M is compatible with the output of M'. Note that this definition of compatibility is, in fact, asymmetric. Machine M can be used in any application where machine M' could be used, but the reverse is not true. This is so because any actual implementation of machine M' would have to choose specific values for the undefined outputs and transitions, and these could be incompatible with the values specified in machine M.

Consider now the case where a compatibility check between an incompletely specified FSM M and an incompletely specified FSM M' is to be performed. The question we want to answer is: machine M a suitable replacement for machine M'? If this is so, machine M has to define the value of the output for every string that machine M' does so. To capture this requirement, the definition of compatibility between two ISFSM's that will be used in the sequence is the following:

Definition 15: An ISFSM M is said to be compatible with M' iff for every string s, the output of M is compatible with the output of M' and the output of M is defined whenever the output of M' is defined, i.e., iff

$$\forall s, (\lambda(q_0, s) \equiv \lambda'(q_0', s)) \wedge (\lambda'(q_0', s) \neq \tau \Rightarrow \lambda(q_0, s) \neq \tau).$$

Note that, again, compatibility between M and M' is not a symmetric relation. M is a valid replacement for M', but M' is not a valid replacement of M. This asymmetric relation might use a more descriptive word, but we will use the well accepted notion of FSM compatibility, keeping in mind that when we say M is compatible with M' referring to ISFSM's, we should say *M is a valid replacement for M'*.

Compatibility between two ISFSM's can also be checked using the product machine. However, the product machine uses a slightly different definition.

Definition 16: Let $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ and $M' = (\Sigma, \Delta, Q', q_0', \delta', \lambda')$ be two incompletely specified FSM's. Assume that $q_k = \delta(q_i, a)$ and $q_l' = \delta'(q_j', a)$. The product machine is the FSM M'' defined by.

$Q'' = Q \times Q'$, where the state $q_{ij}'' \in Q''$ corresponds to the pair of states q_i, q_j'
If $q_k \neq \phi$ and $q_l' \neq \phi$. then $\delta''(q_{ij}'', a) = q_{kl}''$
If $q_k = \phi$ or $q_l' = \phi$. then $\delta''(q_{ij}'', a) = \phi$
$\lambda(q_{ij}'', a) = 0$. if $\lambda(q_i, a) \not\equiv \lambda'(q_j', a)$
$\lambda(q_{ij}'', a) = 0$. if $\lambda(q_i, a) = \tau \wedge \lambda'(q_j', a) \neq \tau$
$\lambda(q_{ij}'', a) = 0$. if $q_k = \phi \wedge q_l' \neq \phi$
$\lambda(q_{ij}'', a) = 1$. otherwise

This definition forces the output of the product machine to be zero whenever the outputs are incompatible, when the output of M' is defined but the output of M is not and, finally, when the next state of M' is defined but the next state of M is not.

V. Reduction of Incompletely Specified FSM's

Given the background presented in Section IV, the description of the basic algorithm for the reduction of ISFSM's is now straightforward.

1) Generate a set of I/O mappings, R, by simulating a set of strings[4] in M'.

[4]When M' does not specify the value of the output, the I/O mapping generated also has a don't care in the corresponding output value.

2) Build the TFSM M'_R from the I/O mappings in R (as described in Section IV-B).
3) Select M, a minimum size FSM compatible with M'_R, using the exact algorithm from Section IV-C.
4) Check if M is compatible with M'. If they are compatible, stop. Otherwise, let s be a string such that $\lambda(q_0, s) \not\equiv \lambda'(q'_0, s)$, a certificate of noncompatibility.
5) Let $R = R \cup \{(s, \lambda'(q'_0, s)\}$ and goto 2.

Theorem 4: This algorithm always terminates and outputs a machine M compatible with M' with minimum number of states.

Proof: Proof that if it terminates, it always returns a minimum sized FSM. Assume, for contradiction, that it returns an FSM larger than the minimum compatible FSM X. The final results must have been returned by Step 3 of the algorithm. Therefore, either the algorithm used in Step 3 is not exact (a contradiction) or there is at least one string in $s \in R$ for which $X(s) \not\equiv M'(s)$, a contradiction since X is compatible with M'.

Proof that it always terminates: Every time a compatibility query is performed, a string that is a certificate of noncompatibility between the solution found and M' is selected and added to R. Therefore, the same machine is never generated twice. Since there exists only a finite number of machines with less than n states, the algorithm must terminate. □

Although this algorithm works, it has a serious disadvantage. Consider the case where the machine M' is a completely specified FSM. Since no restrictions are imposed on the structure of M'_R, the identification of a machine M compatible with M'_R is an NP-hard problem. Therefore, the reduction of M', a completely specified FSM requires the solution of several NP-hard problems, an undesirable situation.

By using a more judicious criterion to generate the I/O mappings in R, it is possible to avoid this problem. The solution is to generate the I/O mappings in R according to a modified version of the L* algorithm.

In any case, the algorithm needs to perform a number of reductions of TFSM's, a problem that is, in itself, NP-complete. Given this, it may not be clear why, in some cases, it would be faster than the standard approach based on compatible enumeration. There are two reasons why this will be the case.

In the first place, many of the TFSM reduction problems may be easy to solve. In fact, whenever the observation table contains no unfilled entries, reduction of the corresponding TFSM can be done in polynomial time, since in this case this is just one step of the L* algorithm.

In the second place, even for instances where the TFSM's are hard to reduce, they can be reduced without ever enumerating compatible sets. If the original machine to be reduced exhibits a very high number of compatible sets, it may still be more efficient to perform a few complex reductions of TFSM's using the basic search procedure detailed in Section IV-C than to enumerate all the prime compatible sets.

A. Improving the Reduction Algorithm

We can now improve the algorithm described in the beginning of this section, by adopting a string generation strategy inspired by the L* algorithm. The critical difference with the L* algorithm is that, for some strings, the output will be undefined. This means that we will not be able, in general, to fill completely the observation table T. We also need to generalize the concepts of closeness and consistency, when applied to a table that has unfilled entries. The fact that we will be unable to fill in some of the table entries means that the generation of an FSM from the table is no longer a straightforward procedure, since choices must be made. In fact, we have to generate a TFSM from the observation table and reduce it using the algorithm from Section IV-C. To generalize the concepts of closeness and consistency, start by defining a compatibility relation between rows:

TABLE V
EXAMPLE OF AN INCOMPLETELY FILLED OBSERVATION TABLE THAT IS CLOSED AND CONSISTENT

	0	1
ϵ	-	0
0	1	1
1	1	1
00	-	0
01	-	0

Definition 17: Two rows r_1 and r_2 are compatible (denoted by $r_1 \equiv r_2$) if the outputs are compatible for each and every column.

The definition of closeness and consistency can now be rephrased using this relation:

Definition 18: Table T is **closed** iff $\forall t \in Sa, \exists s \in S$ s.t. $\text{row}(t) \equiv \text{row}(s)$.

Definition 19: Table T is **consistent** iff $\forall s_1, s_2 \in S, \forall a\ \text{row}(s_1) \equiv \text{row}(s_2) \Rightarrow \text{row}(s_1 a) \equiv \text{row}(s_2 a)$.

Note that these definitions are consistent with the previous definitions of closeness and consistency and represent, in fact, an extension of the original ones. According to these definitions, there is a direct mapping between the observation table and a TFSM. The TFSM can be derived from the table by constructing the trie that corresponds to the set of words in $S \times E$, and labeling the outputs with the entries in the table. As an example, consider the (not totally filled) observation table in Fig. V. This observation table is closed and consistent, according to Definitions 18 and 19. This table defines an I/O mapping and, therefore, a TFSM, that is derived following the procedure illustrated in Table I and Fig. 1.

The improved version of the algorithm that represents the central contribution of this work is, therefore, the following.

1) Initialize an observation table T using the procedure defined in Section IV-D.
2) Generate an observation table T that is closed and consistent, according to Definitions 18 and 19 and the algorithm described in Section IV-D. This table defines a set of I/O mappings R.
3) Build the TFSM M'_R that corresponds to the set of I/O mappings specified in T.
4) Select M, a minimum size FSM compatible with M'_R, using the exact algorithm from Section IV-C.

5) Check if M is compatible with M'. If they are compatible, stop. Otherwise, let s be a string such that $\lambda(q_0, s) \not\equiv \lambda'(q_0, s)$.
6) Add the counterexample s to table T. Goto 2.

Theorem 4 also applies to this case, since the only change is the specific way strings are generated in steps 1 and 2. A closed and consistent table can always be generated, since our definitions of closeness and consistency are less restrictive than the original ones by Angluin. Therefore, this algorithm always terminates and outputs the minimum FSM compatible with M'.

The key point here is that the algorithm will have a guaranteed polynomial runtime **if** the original FSM is a completely specified FSM, since in that case it reduces to Angluin's original algorithm (adapted for Mealy machines). The hope is that for ISFSM's it will also be more efficient, although we know that, in this case, it cannot always work fast.

B. Complexity Analysis

Clearly, if the original FSM M' is incompletely specified, the TFSM M'_R can be arbitrarily hard to reduce and the algorithm can take exponential time. Our objective is to prove that, if M' is a completely specified FSM, the algorithm will run in time polynomial on the size of M'.

For the purposes of this analysis, let k be the cardinality of the input alphabet, n the number of states in the **final** FSM and m the number of states in the original ISFSM. Angluin has proved [14] that, during the construction of the table.

1) No more than $n + n^2$ rows are added to the top half of the table, S. Therefore, the table has no more than $n(n+1)(k+1)$ rows, $n(n+1)$ in S and $kn(n+1)$ in Sa.
2) There are at most n different row labelings, since each different row represents one state.
3) No more than n columns are added to the table, since every time a column is added, a different row is created.
4) No more than n compatibility queries are performed.
5) By choosing the smallest possible counter-example, the maximum length of any string in S is $2n$.

In our setting, the difference between DFA and Mealy FSM's implies that there will be, potentially, $k + n$ columns, since we initialize the table with columns corresponding to the k different values of the input. Therefore, the total complexity will be given by the following.

1) At most $(k+n) \times n(n+1)(k+1) \times 2n$ operations are necessary to fill up the final table (built incrementally by the algorithm).
2) At most $n \times knm$ operations are needed to compute the result of the compatibility queries, by computing the product machine at most n times.
3) At most n calls to the TFSM reduction algorithm are performed, and this TFSM can have up to as many states as the size of the table, $2(k+n) \times n(n+1)(k+1)$.

The critical factors determining complexity are the last two above. In the case where the observation table is totally filled in and complete, an appropriate choice of variable ordering in Bierman's algorithm leads to a linear time algorithm, since no decisions need to be made. However, in our implementation, the same algorithm is used for completely and incompletely filled observation tables. Although the TFSM has $(k + n) \times 2n(n + 1)(k + 1)$ states, assignments only need to be computed to the $n(n + 1)(k + 1)$ TFSM states reachable by the strings in $S \cup Sa$. For these states, the incompatibility relation is known from the observation table, but the computation of the restrictions in (4) is quadratic on the number of states. Therefore, each call to the TFSM reduction algorithm has complexity $O(k^2n^4)$ and this algorithm can be called up to n times. This leads to a total complexity given by $O(\max(k^2n^5, kn^2m))$. If one uses the linear time algorithm to solve special cases of TFSM's described by the completely filled observation tables, the complexity decreases to kn^2m, since $n \leq m$.

Although, in general, this **worst** case complexity will be much higher than the $km \log m$ complexity of the partition-refinement approach for completely specified FSM's [3], it still gives a polynomial time warranty in the particular case where M' is a completely specified FSM. It must be noted that this a worst case bound that may not be tight, since the assumptions made in the analysis of the table construction may be too pessimistic. In the very special case where M' is a large completely specified FSM with a compatible n state machine that is very small, $n \ll m$ and this algorithm may actually be asymptotically faster, as long as $n^2 < \log m$.

We must note that, in general, this analysis is of strictly theoretical interest, since we are mainly concerned with the reduction of ISFSM's, an NP-complete problem. For ISFSM's, one expects that its run-time will be exponential in many of the problems, but that the cases where it behaves poorly will not necessarily match the cases where the standard algorithms of Section III fail.

VI. Generalizations and Extensions

The previous sections described an algorithm for the reduction of incompletely specified FSM's that is applicable to any ISFSM. However, in practice, FSM's are specified using description languages that can describe concisely machines that require very large representations if one were to be restricted to the definition of FSM's presented in Section II. More specifically, FSM's are usually described using languages that allow the user to do the following.

- Use unspecified output values of only a subset of the outputs. For example, one transition may have an output value described as `00--`, meaning that the first two outputs should be zero, but the other two outputs have undefined values.
- Use unspecified input values as a shorthand to describe a potentially very large number of transitions taking place in a number of input combinations. For example, in an FSM with eight inputs, one may specify that on input `01------` the machine should change from state A to state B, thereby avoiding the need to explicitly describe every one of the 64 input combinations that cause that transition.

The first generalization is easily taken care of. Although the definitions in Sections II and IV assume that the value of the output either belongs to the output alphabet Δ or is undefined, it is relatively straightforward to extend the algorithms to the case where a subset of the outputs may be undefined. In fact, all that is required is a re-definition of the compatibility relation between output values that takes this into account. In particular, if the output consists of more than one multivalued variable, the outputs of two transitions are compatible iff the corresponding values of the outputs are compatible.

The situation is less simple in the case of unspecified input values. In this case, a more careful analysis needs to be performed since the extension of the algorithms to handle this case is not straightforward. Clearly, if there are k binary input variables, a straightforward procedure is to replace every transition labeled with unspecified input values (m of them) by the 2^m transitions with completely specified input combinations. Although this solution works, it is impractical for many cases of interest, since it leads to an explosion in the number of existing transitions. We show below that it is possible to preserve, up to some point, the unspecified input combinations and the compactness that they allow. Consider an STG for an FSM without loops where a transition is labeled with a don't care in the input part, as the example in Fig. 7(a). Clearly, this machine is a TFSM, but the TFSM it represents is the one in Fig. 7(b). We can extend slightly the concept of TFSM's to include this case.

Definition 20: An extended TFSM (ETFSM) is a TFSM where state transitions can also be labeled with unspecified input combinations.

Therefore, Theorem 2 applies to machine M_2, but not to machine M_1, which should be viewed only as a shorthand representation for machine M_2. The function F_2, a mapping from the states in M_2 to the states in some compatible machine M needs not be a many to one mapping from the states in M_1 to the states in M. In fact, since some states in M_1 correspond to more than one state in M_2, a function F_2 mapping the states in M_2 to the states in M defines a **relation** F_1 from the states in M_1 to the states in M.

For instance, Fig. 8 shows a FSM M that is compatible with both M_1 and M_2. However, the relation F_1 defined by Definition 13 is not a many to one mapping, since it maps state q_1 of M_1 to both states q_1 and q_2 in M. Note, however, that states q_1 and q_2 in M will necessarily give the same output for all strings, i.e., $\lambda(q_1, s) = \lambda(q_2, s)$.

However, it is possible to prove that if F_2 is not a function, then machine M is not minimum with respect to the number of states. It is always the case that an FSM that can be obtained by a many-to-one mapping from an TFSM and not from the related ETFSM is not minimum. This result is formalized by the following theorem.

Theorem 5: Let M' be a ETFSM. Then, for any FSM M compatible with M', there exists at least one machine M_2 with no more states than M such that F, as defined in Definition 13 is a mapping function, mapping each state in M' to one and only one state in M_2.

Proof: Let M'' be the TFSM represented by ETFSM M'. Each state in M' corresponds to one or more states in M'',

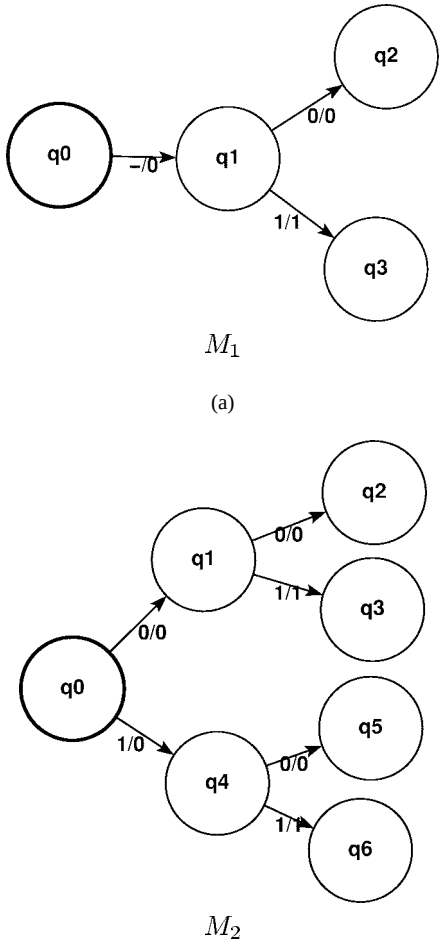


Fig. 7. (a) A TFSM with unspecified input values (ETFSM) and (b) its corresponding TFSM.

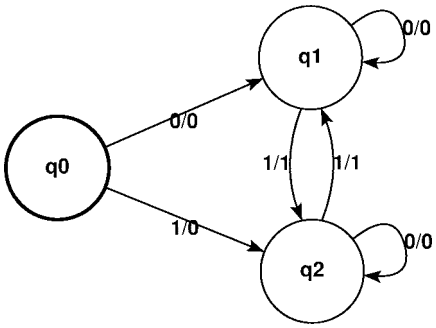


Fig. 8. FSM M compatible with both machines M_1 and M_2 in Fig. 7.

since M' is a compact representation for M''. Consider now a state q' in M' that corresponds to states $Z = \{q''_1, \ldots, q''_k\}$. By Theorem 4 there exists a function F'' that maps each state in Z to a state in M. Let $F''(q''_1) = q_1, F''(q''_2) = q_2 \cdots F''(q''_k) = q_k$. Now, states $q_1, q_2 \cdots q_k$ have compatible outputs for each string starting at state $q''_1, q''_2 \cdots q''_k$. Therefore, the function $F''' : Q'' \rightarrow Q$ defined by $F'''(q''_1) = q_1, F'''(q''_2) = q_1, \ldots F'''(q''_k) = q_1$ is also a valid mapping function for TFSM M'' to machine M and, therefore, a function $F : Q' \rightarrow Q$ defined (for state q') by $F(q') = q_1$ and in an analog way

for all the other states will be a valid mapping function for M'. □

This result means that in the presence of unspecified input values, one needs not consider all possible input combinations. It is sufficient to construct the ETFSM and reduce it using the same algorithm used for TFSM's.

Although this result shows that one needs not unroll the unspecified input combinations, it should be noted that the L* procedure described in Section IV-D and used in the algorithm in Section V assumes that different inputs are nonoverlapping. In particular, the concepts of closeness and consistency require that input sequences be disjoint, in the sense that no input sequence can be contained in more than one of them. There are two possible solutions to this limitation. The first one, adopted in the prototype system we developed, is to apply the *disjoint sharp*[5] operator [25] to the original set of input combinations in order to obtain a disjoint cover of the used combinations. The second possibility that is still being studied is to generalize the concepts of closeness and consistency of observation tables to the case where overlapping input combinations exist. All the experiments described in the next section were performed using the first alternative, that was found to work well except in a few isolated cases where the multiplication of input combinations leads to a degradation of the CPU times used.

VII. Experimental Results

To evaluate empirically our algorithm we used the set of problems used by Kam and Villa to evaluate their implicit algorithm (`ism`) against an explicit implementation [8]. This choice is justified because the most commonly used benchmarks for sequential circuits (e.g., MCNC or ISCAS benchmarks) are inappropriate for the task at hand. As an example, all but one of the MCNC sequential benchmarks require less than one second to be solved by a modern computer.[6] The set of problems used by Kam and Villa ([8], Table I) comes from a variety of sources: standard benchmarks, asynchronous synthesis, learning problems, synthesis of interacting FSM's and FSM's that have been constructed to exhibit a large number of compatible pairs.

Table VI describes some statistics of the problems and the results obtained. Columns 2, 3, 4, and 5 list the number of inputs for each FSM, the number of states in the initial, unreduced FSM, the number of states in the final, reduced, FSM and the number of compatible sets (as computed by `ism`). The execution times of our method (`bica`) were compared with those of `ism` and `stamina`, a popular implementation of the explicit version of the standard state reduction algorithm [7].[7] The CPU times were obtained in a DECStation 5000/260 with 440 MB of memory. Columns 6, 7, and 8 show the time required for the three programs under comparison: `ism`, `stamina` and `bica`. Values marked with * mean that only the first solution was computed and, therefore, the problem was not totally solved. In the first set of problems, `ism` was not used to solve the covering problem and, therefore, those times are not listed.

TABLE VI
Statistics and Results Obtained with `ism`, `bica`, and `stamina`

FSM	Statistics			#	FSM Reduction		
	PI	init	fin	compat	ism	stam	bica
alex1	5	42	6	55928	N.A	16	34
intel_edge	3	28	4	9432	N.A	0	3
isend	7	40	4	22207	N.A	1	18
rcv-ifc	8	46	2	1.52e11	N.A	0	16
rcv-ifc.m	8	27	2	1.79e6	N.A	0	8
send-ifc	8	70	2	5.07e17	N.A	1	58
send-ifc.m	8	26	2	8.98e6	N.A	0	20
vbe4a	6	58	3	1.75e12	N.A	173	47
vmebus	11	32	2	5.05e7	N.A	1	1555
th.20	2	21	4	97849	*547	fails	2
th.40	2	41	8	1.45e6	*6862	fails	3
th.55	2	55	8	3.62e7	fails	fails	4794
fo.20	2	21	3	42193	*33	fails	1
fo.50	2	51	6	3.64e7	fails	fails	9
fo.70	2	71	?	9.62e10	fails	fails	fails
ifsm0	7	38	3	1.0e6	fails	0	5
ifsm1	2	74	14	43006	*413	1294	12
ifsm2	7	150	9	497399	403	694	474
rubin18	1	18	3	2^{12}-1	fails	fails	0
rubin600	1	600	3	2^{400}-1	fails	fails	51
rubin1200	1	1200	3	2^{800}-1	fails	fails	382
rubin2250	1	2250	3	2^{1500}-1	fails	fails	2518
e271	2	19	2	393215	22	0	4
e285	2	19	2	393215	13	0	1
e304	2	19	2	393215	556	0	1
e423	2	19	2	204799	*443	fails	1
e680	2	19	2	327679	984	0	1

Table VI shows that, for this set of hard problems, `bica` is more robust and the unique algorithm that is able to solve some of the problems exhibiting a very large number of compatibles. However, in a considerable number of cases, `stamina` is able to solve the problems faster, specially when enumeration of all the primes can be avoided because a cover consisting only of maximal compatibles is closed. `Ism` is able to complete in some cases where `stamina` fails and is almost always able to compute the compatibles, being faster than both `stamina` and `bica` in a number of examples. Although a machine with 440 MB of memory was used for the comparisons, `bica` does not usually require much memory, with all examples except one requiring much less than 64 MB of memory.

The diverse behavior of the three algorithms analyzed is to be expected since the three algorithms use very different approaches.

It is also interesting to analyze the relative cost of the several operations performed by our algorithm. Table VII shows the CPU times for the same set of problems, broken down by phases. The second column, preprocessing time, lists the time spent to process the input data and compute the incompatibility graph. This time is usually negligible, with the exception of the larger `rubin` examples, where the suboptimal algorithm

[5] The disjoint sharp operator, applied to two cubes, returns a set of disjoint cubes that cover the same input points. It can applied iteratively to obtain a set of disjoint cubes that cover the same points as a given set of cubes.

[6] The remaining example `ex2` cannot be solved using standard methods and is easily solvable by our algorithm. However, as a whole, the benchmark is uninteresting for state reduction algorithms.

[7] The descriptions of the state machines used in the experimental evaluation are available at the home page of the second author, and the source code for *bica*, written in C++, is available upon request.

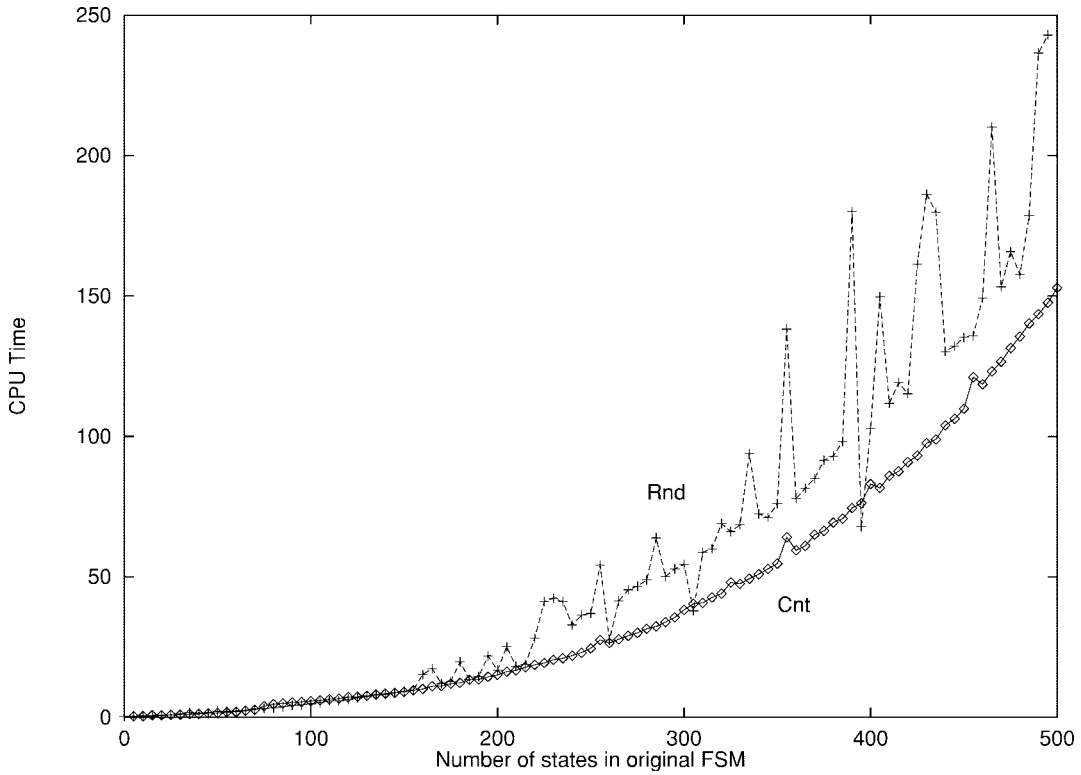


Fig. 9. Growth of CPU time for two parameterized families of completely specified FSM's.

TABLE VII
COMPONENTS OF `bica` EXECUTION TIME AND NUMBER OF ITERATIONS

FSM	CPU time				Iter.
	Pre Proc.	Compat. Check	Table Comp.	TFSM Red.	
alex1	0.22	9.52	14.88	8.50	25
intel_edge	0.16	1.62	0.18	0.54	10
isend	0.22	5.26	10.00	1.80	21
rcv-ifc	0.36	5.84	7.66	1.44	18
rcv-ifc.m	0.26	3.38	3.04	0.84	13
send-ifc	0.70	15.92	35.54	5.10	24
send-ifc.m	0.56	8.06	7.66	2.84	14
vbe4a	0.46	12.04	28.44	5.58	26
vmebus	7.82	981.50	365.80	193.20	18
th.20	0.10	0.58	0.02	0.12	7
th.40	0.06	1.68	0.48	0.54	11
th.55	0.10	3.32	1.62	4788.38	14
fo.20	0.12	0.62	0.10	0.08	6
fo.50	0.16	2.44	0.54	4.86	13
fo.70*	*0.12	*3.40	*2.14	*4415.64	10
ifsm0	0.52	2.22	0.24	1.52	6
ifsm1	0.30	5.16	0.50	5.88	12
ifsm2	2.60	92.54	16.48	352.10	16
rubin18	0.06	0.10	0.00	0.02	2
rubin600	48.40	0.50	0.00	0.06	2
rubin1200	375.82	1.40	0.00	0.22	2
rubin2250	2502.16	3.72	0.00	0.96	2
c271	0.10	1.64	0.68	1.30	10
c285	0.10	0.78	0.04	0.16	6
c304	0.12	0.66	0.04	0.20	7
c423	0.12	0.46	0.00	0.08	5
c680	0.10	0.76	0.10	0.24	7

used to compute the compatible pairs makes this component dominant. The third column lists the total time required to compute the compatibility checks required by the algorithm. The fourth column lists the time required to compute the observation table while the last column lists the total time spent in the TFSM reduction algorithm. The last column lists the number of iterations needed in the main loop of the algorithm, which coincides with the number of TFSM reductions that was required.

This table shows that no single step of the algorithm dominates the execution time for the majority of the examples, although the TFSM reduction time dominates for the examples that take a long time to solve (including the one that `bica` cannot solve, `fo.70`, marked with *). This is not surprising, since the TFSM reduction step is the only one for which no polynomial time bounds can be asserted.

We also run an experiment targeted at evaluating the behavior of the algorithm in a set of completely specified FSM's. The objective was to validate the worst case complexity analysis presented in Section V-B and verify the tightness of the bounds obtained. For this purpose, we created parameterized families of completely specified FSM's. These families of FSM's were designed to exhibit the following properties.

- They represent completely specified FSM's.
- The FSM's are always reducible.
- The size of the original FSM is controlled by a parameter that can be varied between a small number and an arbitrarily high value.

Random generation of FSM's is not appropriate for the task at hand, given the requirement that the FSM's have to be reducible. If this is not the case, the incompatibility graph has no edges and the algorithm terminates rapidly with the information that the machine is irreducible. Given these

limitations, we performed experiments with two families of FSM's: The first family (`Cnt`) is composed of state machines that count the number of ones in its input, parameterized by a parameter n. The nth counter has $5n$ states and outputs 1 after n ones have been presented at its input. The second family (`Rnd`) was obtained by randomly generating an FSM with n states and making it redundant by performing STG replication and changing the appropriate transitions, obtaining a reducible machine of size $5n$. The CPU times obtained in these two families of FSM's are shown in the graph in Fig. 9. An analysis of the growth rate for these two families shows that, empirically, the CPU time is roughly proportional to $n^{2.5}$ (or $n^{1.5}m$ since $m = 5n$). Although this result proves nothing about the general case, it raises the question of whether the bounds obtained by the theoretical analysis (k^2n^5) are tight. We believe they are not and that the analysis can be considerably improved.

VIII. Conclusions and Future Work

This work presented a new approach for the problem of reducing incompletely specified FSM's. Although inherently slower when applied to completely specified FSM's, the algorithm has the advantage of using a new approach that does not suffer from the limitations of the standard approach based on the computation of compatibles. Since the problem is NP-complete, one does not expect a fast execution on all the instances of the problem. However, the experiments have shown that the cases where our approach does not work well are distinct from the cases where the standard approach fails, thereby making this algorithm a very interesting alternative in the instances where enumeration of the compatibles is infeasible.

This work opened several interesting directions for future research. As an immediate direction for future research, it would be very interesting to generalize the concepts underlying the observation tables of the L^* algorithm, with the objective of avoiding the need to make the inputs disjoint. This could lead to considerable improvements in execution time for some examples, as discussed in Section VI.

It may also be possible to improve considerably the total execution time by developing better TFSM reduction techniques than the ones used in this work. In particular state merging techniques [26] have shown great promise in a recent contest [27] held to evaluate the performance of DFA identification algorithms. Although all state merging techniques proposed to date are heuristic, it is possible to modify them in order to obtain exact algorithms, a line of research that is very interesting in itself and that is being actively pursued by the researchers in the field [26].

A less immediate and more open line for future research is the application of techniques similar to these ones to the computation of other problems in similar domains like DFA property checking and NDFA reduction.

Acknowledgements

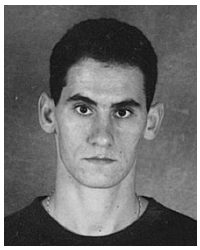
The authors would like to thank T. Villa, who helped greatly in the preparation of this manuscript and contributed with many helpful suggestions. They would also like to thank T. Kam for help in setting up the comparisons, the anonymous reviewers for their useful and detailed comments and Prof. Sangiovanni-Vincentelli for his support of this line of research.

References

[1] D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.*, part I, vol. 257, no. 3, pp. 161–190, Mar. 1954.
[2] E. F. Moore, "Gedanken experiments on sequential machines," in *Automata Studies*, C. E. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton Univ. Press, 1956.
[3] J. E. Hopcroft, "$n \log n$ algorithm for minimizing states in finite automata," Stanford Univ., Satnford, CA, Tech. Rep. CS 71/190, 1971.
[4] C. F. Pfleeger, "State reduction in incompletely specified finite state machines," *IEEE Trans. Comput.*, vol. C-22, pp. 1099–1102, 1973.
[5] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of FSM's: Functional Optimization*. Norwell, MA: Kluwer Academic, 1997.
[6] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw Hill, 1994.
[7] J.-K. Rho, G. Hachtel, F. Somenzi, and R. Jacoby, "Exact and heuristic algorithms for the minimization of incompletely specified state machines," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 167–177, Feb. 1994.
[8] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "A fully implicit algorithm for exact state minimization," in *Proc. ACM/IEEE Design Automation Conf.*, 1994, pp. 684–690.
[9] E. M. Gold, "Complexity of automaton identification from given data," *Inform. Contr.*, vol. 37, pp. 302–320, 1978.
[10] L. Pitt and M. Warmuth, "The minimum consistent DFA problem cannot be approximated within any polynomial," *J. ACM*, vol. 40, no. 1, pp. 95–142, 1993.
[11] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. Comput.*, vol. C-21, pp. 592–597, June 1972.
[12] A. W. Biermann and F. E. Petry, "Speeding up the synthesis of programs from traces," *IEEE Trans. Comput.*, vol. C-24, pp. 122–136, 1975.
[13] F. C. Hennie, "Fault detecting experiments for sequential circuits," in *Proc. 5th Annu. Symp. Switching Circuit Theory and Logical Design*, Princeton, NJ, Nov. 1964, pp. 95–110.
[14] D. Angluin, "Learning regular sets from queries and counterexamples," *Inform. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.
[15] E. M. Gold, "System identification via state characterization," *Automatica*, vol. 8, pp. 621–636, 1972.
[16] M. Paull and S. Unger, "Minimizing the number of states in incompletely specified state machines," *IRE Trans. Electron. Comput.*, vol. EC-8, pp. 356–367, Sept. 1959.
[17] O. Coudert and J. C. Madre, "New ideas for solving covering problems," in *Proc. ACM/IEEE Design Automation Conf.*, ACM Press, June 1995, pp. 641–646.
[18] A. Grasselli and F. Luccio, "A method for minimizing the number of internal states in incompletely specified sequential networks," *IRE Trans. Electron. Comput.*, vol. EC-14, no. 3, pp. 350–359, June 1965.
[19] M. Damiani, "State reduction of nondeterministic finite-state machines," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 1278–1291, Nov. 1997.
[20] A. L. Oliveira and S. Edwards, "Limits of exact algorithms for inference of minimum size finite state machines," in *Proc. 7th Workshop on Algorithmic Learning Theory*, no. 1160 in *Lecture Notes in Artificial Intelligence*. Sydney, Australia: Springer-Verlag, Oct. 1996, pp. 59–66.
[21] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.
[22] A. L. Oliveira and J. P. M. Silva, "Efficient search techniques for the inference of minimum size finite automata," in *Proc. 5th String Processing and Information Retrieval Symp.*, IEEE Press, Sept. 1998, pp. 81–89.
[23] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Norwell, MA: Kluwer Academic, 1996.
[24] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, vol. 407 of *Lecture Notes in Computer Science*, J. Sifakis, Ed. Berlin, Germany: Springer-Verlag, June 1989, pp. 365–373.
[25] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA: Kluwer Academic, 1984.

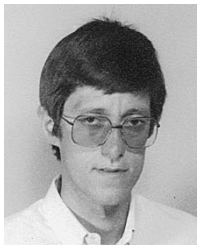
[26] K. J. Lang, "Random DFA's can be approximately learned from sparse uniform examples," in *Proc. 5th Annu. Workshop Comput. Learning Theory*, 1992, pp. 45–52.

[27] K. J. Lang, B. A. Pearlmutter, and R. Price, "Results of the Abbadingo one DFA learning competition and a new evidence driven state merging algorithm," in *Fourth International Colloquium on Grammatical Inference (ICGI-98), Lecture Notes in Computer Science*. Ames, IA: Springer-Verlag 1998, pp. 1–12.



Jorge M. Pena received the B.Sc. degree from Lisbon Technical University. He developed this work while working toward his degree in informatics, with a specialization in artificial intelligence.

His scientific interests include logic synthesis, artificial intelligence and search techniques.



Arlindo L. Oliveira (S'86–M'95) received the B.Sc. degree in engineering and the M.Sc. degrees in electrical and computer engineering from Lisbon Technical University, Lisboa, Portugal. He received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1994.

He is currently an Assistant Professor at Lisbon Technical University, and is also affiliated with INESC and the Lisbon Center of the Cadence European Laboratories. His research interests include combinatorial optimization, computer architecture, logic synthesis, machine learning, low-power design, and sequential systems optimization and automata theory.