A New Algorithm For The Reduction Of Incompletely Specified Finite State Machines

Jorge M. Pena
IST-INESC
Rua Alves Redol, 9
1000 Lisboa, Portugal
jmgp@algos.inesc.pt

Arlindo L. Oliveira
Cadence European Labs/IST-INESC
Rua Alves Redol, 9
1000 Lisboa, Portugal
aml@inesc.pt

Abstract

We propose a new algorithm to the problem of state reduction in incompletely specified finite state machines. This algorithm is not based on the enumeration of compatible sets, and, therefore, its performance is not dependent on the number of prime compatibles. 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.

1 Introduction

The reduction of finite state machines is a well known problem of great importance in sequential circuit synthesis.

For completely specified finite state machines (FSM), the state reduction problem can be solved in polynomial time [13]. For incompletely specified finite state machines (ISFSM), the problem is known to be NP-complete [15]. Nonetheless, exact and heuristic algorithms are commonly used in practice, 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 have been proposed, based on both explicit [16] and implicit enumeration [10] of the compatibles. However, some problems cannot be solved using this approach for one of two reasons: a) the set of prime compatibles is too large to be listed, either explicitly or implicitly or b) the binate covering problem can be formulated but takes too long to solve. We review these algorithms and concepts in section 3.

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 DFAs consistent with a given set of input/output mappings. Although all the results and algorithms have used the DFA formalism, they can be easily translated to similar results formulated in terms of finite state machines.

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 labels are known. If this set is fixed, the problem is NP-complete [5]. Under these conditions, the most efficient search algorithms for this problem are based on the approach proposed by Bierman [3].

The problem becomes easier if the algorithm is allowed to make queries or experiment with the unknown machine. Angluin proposed an algorithm [1] that solves the problem in polynomial time by allowing the algorithm to ask membership queries.

Since we make extensive use of both Bierman's approach and Angluin's algorithm, we describe these algorithms in some detail in section 4.

Section 5 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 to 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 6 describes the results we obtained in a set of finite state machines that have been used by other authors to evaluate the performance of state reduction algorithms [10]. 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.

2 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 finite state machine is a tuple $M = \left(\Sigma, \Delta, Q, q_0, \delta, \lambda\right)$ where $\Sigma \neq \emptyset$ is a finite set of input symbols, $\Delta \neq 0$ 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 $\lambda(q, a) : Q \times \Sigma \rightarrow \Delta \cup \{\epsilon\}$ 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 finite state machine 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 ϵ denotes an unspecified output. We say that an output b_i is compatible with an output b_j (and write $b_i \equiv b_j$) if $b_i = b_j$ or $b_i = \epsilon$ or $b_j = \epsilon$. We will always use quoted symbols (M', Q', etc) to refer to the original, un-reduced 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 finite state machine after sequence s is applied in state q. The output of such a sequence is defined to be $\lambda(q, s) = \lambda\big(\delta(\delta(\cdots\delta(q, a_1)\cdots), a_{k-1}), a_k\big)$. Similarly, $\delta(q, s)$ denotes the final state reached by a finite state machine after a sequence of inputs $(a_1, \ldots, a_k)$, is applied in state q. The final state is defined to be $\delta(q, s) = \delta(\delta(\ldots\delta(\delta(q, a_1), a_2)\ldots), a_k)$. To avoid unnecessary notational complexities, $\lambda(\phi, a)$ is defined to be equal to ϵ and $\delta(\phi, a) = \phi$.

Definition 2 *A Loop-Free Finite State Machine (LFFSM) M is a finite state machine satisfying definition 1 and the following additional requirements:*

$$\forall q \in Q \setminus q_0,\ \exists^1 s \in \Sigma^* \text{ s.t. } \delta(q_0, s) = q$$

$$\forall q \in Q,\ \forall a \in \Sigma\ \delta(q, a) \neq q_0$$

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 LFFSM is a tree rooted at state q_0. We say that an LFFSM M contains a string s iff the application of string s in state q_0 leads to a state in M, i.e., iff $\delta(q_0, s) \in Q$.

Definition 3 *A completely specified FSM $M = \big(\Sigma, \Delta, Q, q_0, \delta, \lambda\big)$ is equivalent to an incompletely specified finite state machine $M' = \big(\Sigma, \Delta, Q', q_0', \delta', \lambda'\big)$ iff, for every input string s, the output of M is compatible with the output of M', i.e., $\lambda(q_0, s) \equiv \lambda'(q_0', s)$.*

Definition 4 *Two states are compatible iff the output sequences of the finite state machine 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 [9], that will be used in the description of the state reduction algorithms described in section 3.

Definition 5 *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 FSMs, it is sufficient to have pairwise compatibility between each state in the compatible ([9], theorem 4.3):

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

Proof: see [9]
□

3 Reduction of ISFSMs

The standard approach for the reduction of incompletely specified finite state machines is based on the enumeration of the set of compatibles and on the solution of a covering problem. Only a very brief description will be given here. Readers interested in a comprehensive treatment of this topic are referred to [9] for details. It is known that the minimum equivalent FSM can be found through the enumeration of the set of compatibles and the identification of a minimum-cardinality closed cover of compatibles.

Definition 6 *Compatible cover: a set of compatibles $Z = \{C_1, C_2 \ldots C_n\}$ is said to be a cover for the states in M' iff every state in Q' belongs to some compatible in Z.*

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 7 *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 8 *Closed cover: a set of compatibles Z is called a closed cover iff for each compatible C_j in Z, each of its implied sets is covered by some C_k in Z.*

Obtaining the minimum equivalent 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 [4, 9].

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 [6] that only a subset of the compatibles needs to be considered, the compatibles that are not dominated by any other compatible:

Definition 9 *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 [10] 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 [16] are compared with our algorithm in section 6.

4 FSM identification from IO sequences

This section describes the basic techniques that are used by the algorithm described in section 5, namely FSM identification from IO sequences and state reduction of LFFSMs.

4.1 Identifying FSMs from specified IO sequences

Consider the situation where the behavior of a finite state machine is specified by a given input/output mapping (IO mapping), where an IO mapping is defined in the following way:

Definition 10 *An IO mapping is a sequence of input/output pairs $((a_1, b_1), (a_2, b_2) \ldots (a_k, b_k)) \in \big(\Sigma \times (\Delta \cup \{\epsilon\})\big)^k$ that specify the value observed in the outputs of a finite state machine under a specific sequence of inputs.*

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

This problem is NP-complete, and is equivalent to the problem addressed by Gold [5] 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 finite state machines of a special type, loop-free finite state machines.

4.2 Loop-Free Finite State Machines

From a set of IO mappings, it is straightforward to generate a finite state machine 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 IO mapping.

As an example, consider the set of (three) IO mappings shown in table 1. Is is trivial to observe that the LFFSM shown in figure 1 generates the desired outputs.

Table 1: Example of a set of IO mappings.

1/0	1/1	1/0
0/0	1/0	
0/0	0/0	0/0

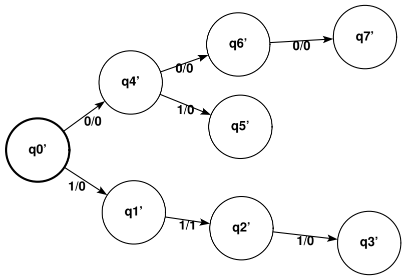

Figure 1: LFFSM generated from the IO mapping in table 1.

Since the LFFSM generated in this way exhibits a behavior compatible with the given set of IO mappings, we can select the minimum FSM consistent with this set by reducing the LFFSM, i.e., by selecting the minimum FSM equivalent to this LFFSM. It turns out that LFFSMs can be reduced using algorithms other than the ones described in section 3. This happens because, unlike general FSMs, LFFSMs can be reduced by selecting a mapping function from the states in the LFFSM to the final, reduced, FSM.

Definition 11 *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 $F : Q' \to Q$ is a valid mapping function iff it satisfies:*

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

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

The first equation states that F is a mapping function only if it maps each state q' in M' to a state 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 mapping function between M' and M, then machine M is equivalent to M'.*

Proof: We need to prove that, if there exists a 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
$\lambda(q_0, s) = \lambda(\delta(\delta(\ldots\delta(q_0, a_1)\ldots), a_{k-1}), a_k) =$
$\lambda(\delta(\delta(\ldots\delta(F(q'_0), a_1)\ldots), a_{k-1}), a_k) =$
$\lambda(\delta(\delta(\ldots F(\delta'(q'_0, a_1))\ldots), a_{k-1}), a_k) =$
$\lambda(F(\delta'(\delta'(\ldots(\delta'(q'_0), a_1)\ldots), a_{k-1})), a_k) \equiv$
$\lambda'(\delta'(\delta'(\ldots(\delta'(q'_0), a_1)\ldots), a_{k-1}), a_k) = \lambda'(q'_0, s)$
□

It is well known [6] that the existence of a mapping function is not a necessary condition for equivalence between two finite state machines. For example, machine M in figure 2 is equivalent to 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 equivalent. 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 $q_3 = F(\delta'(q'_2, 1)) \neq \delta(F(q'_2), 1) = \delta(q_3, 1) = q_4$ and cannot map to q_4 because $q_4 = F(\delta'(q'_2, 1)) \neq \delta(F(q'_2), 1) = \delta(q_4, 1) = q_3$. 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.

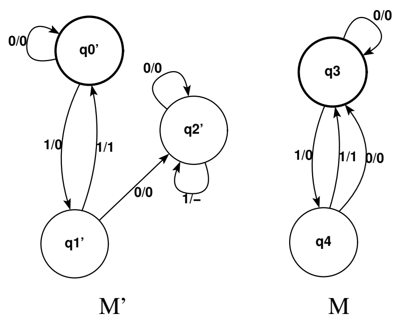

Figure 2: Machine M is equivalent to M', but no mapping function between M' and M exists.

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

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

Definition 12 *Let $F : Q' \to Q$ be defined by $F(\delta'(q'_0, s)) = \delta(q_0, s)$ for each string s contained in M'.*

If M' is a LFFSMs the following lemma applies:

Lemma 2 *F is a many to one mapping, mapping each state in M' to one and only one state in M.*

Proof: since M' is an LFFSM 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 LFFSM, as, for example, the one on the left of figure 2, 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 12: 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.

Theorem 2 *Let M' be a LFFSM. 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 equivalent to 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 12). Now, $\lambda'(q', a) = \lambda'(q'_0, sa) \equiv \lambda(q_0, sa) = \lambda(\delta(q_0, s), a) = \lambda(q, a)$, and therefore equation 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 equation 2 is also respected. Therefore, F is a valid mapping function.
□

This result is critical, because is strongly limits the solution space for machines that are LFFSMs. In fact, it enables us to use totally different algorithms to solve LFFSMs than the ones used to solve standard FSMs. Theorem 2 says that, to select a minimum sized finite state machine equivalent to an LFFSM machine M', one needs not consider all possible relations F, but only those that are many to one mappings.

4.3 Reduction of LFFSMs

As shown in the previous sections, the reduction of a LFFSM can be performed by selecting a 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 1 and 2 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 LFFSM 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 conditions can be rewritten as:

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

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

The basic search algorithm for this problem was proposed by Bierman [2]. Later, the same author proposed an improved search strategy that is much more efficient in the majority of the complex problems [3]. 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 $1 \dots n$ and assigning it to S_i. If no more values exist, undo the assignment made to the last variable chosen.
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 an hypothetical example where the original LFFSM has 10 states ($\{q'_0 \dots q'_9\}$) and a search is being performed by a machine with 3 states. Under these conditions, each S_i can assume only the values 0, 1 or 2. Suppose that variables will be assigned in the order $S_0 \dots S_9$ and that the following restrictions exist in this problem:

$$S_1 \neq S_2 \vee S_8 = S_9 \tag{5}$$

$$S_8 \neq S_9 \vee S_2 = S_3 \tag{6}$$

The section of the search tree depicted in figure 3 is obtained by the basic search algorithm described above. In every leaf of this tree a conflict was detected and backtracking took place.

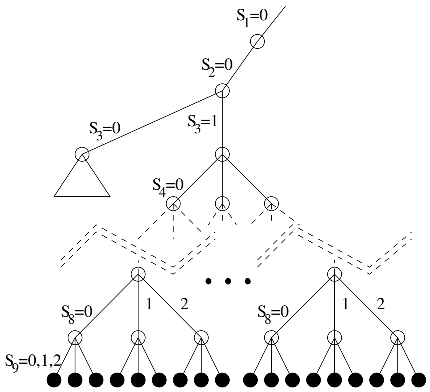

Figure 3: Search tree for the simple backtrack algorithm

Bierman noted [3] 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 multi-valued domain of the unit clause resolution of the Davis-Putnam procedure 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 [14]. However, for the purposes of the present work, this description of the basic search procedure should be sufficient.

4.4 The L* algorithm

The L* algorithm [1] 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[1]. The algorithm uses an observation table T and two sets of strings, S and E. The algorithm updates this table in accordance with the following rules:

- 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.
- $T(s, e) = \lambda'(q'_0, se)$
- T is **closed** iff $\forall t \in Sa, \exists s \in S$ s.t. $\mathrm{row}(t) = \mathrm{row}(s)$
- T is **consistent** iff $\forall s_1, s_2 \in S, \forall a \in \Sigma$ $\mathrm{row}(s_1) = \mathrm{row}(s_2) \Rightarrow \mathrm{row}(s_1 a) = \mathrm{row}(s_2 a)$

The table T is updated in the following way:

1. Initialize S with the empty string and E with all input combinations.
2. If table is not closed select a string s that violates closeness and move it from 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 finite state machine. Generate this machine and make an equivalence 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 equivalent to 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 [1]. 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 figure 4.

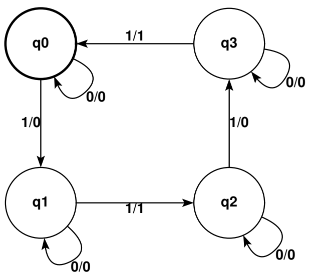

Figure 4: Example of target machine for the L* algorithm

The algorithm is initialized with the observation table shown in table 2. 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 2 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 3. 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. Applying this procedure, we obtain the minimum machine equivalent to the input/output relations specified in this table, as shown in figure 5. Since this machine is equivalent to the original machine, the algorithm stops.

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

Table 2: Initial table for the L* algorithm

	0	1
λ	0	0
0	0	0†
1	0	1

Table 3: Example of closed and consistent table for the L* algorithm

	0	1
λ	0	0
1	0	1
0	0	0
11	0	0
10	0	1

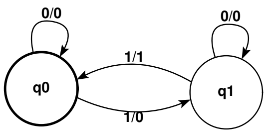

Figure 5: Result of the L* algorithm in the machine of figure 4

The fundamental result concerning this algorithm is 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 [1].
□

5 Reducing Incompletely Specified Finite State Machines

Given the background presented in the previous section, the description of the basic algorithm for the reduction of ISFSMs is now straightforward:

1. Generate a set of IO mappings, R by simulating a set of strings in M'.
2. Build the LFFSM M'_R from the IO mappings in R (as described in section 4.2).

3. Select M, a minimum size finite state machine equivalent to M'_R, using the exact algorithm from section 4.3.
4. Check if M is equivalent[2] to M'. If they are equivalent, stop. Otherwise, let s be a string such that $\lambda(q_0, s) \not\equiv \lambda'(q'_0, s)$, a witness of unequivalence.
5. Let $R = R \cup \{(s, \lambda'(q'_0, s)\}$ and goto 2.

Theorem 4 *This algorithm always terminates and outputs a machine M equivalent to 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 equivalent FSM X. The final result 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 equivalent to M'.

Proof that it always terminates: Every time an equivalence query is performed, a string that is a witness of unequivalence between the solution found and M' is selected and added to R. Therefore, the same machine is never generated twice. Since there are 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 finite state machine. Since no restrictions are imposed on the structure of M'_R, the identification of a machine M equivalent to M'_R is an NP-hard problem. Therefore, the reduction of M', a completely specified finite state machine requires the solution of several NP-hard problems, an undesirable situation.

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

5.1 The improved 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 won't 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 LFFSM from the observation table and reduce it using the algorithm from section 4.3. To generalize the concepts of closeness and consistency, start by defining a compatibility relation between rows:

Definition 13 *Two rows r_1 and r_2 are compatible ($r_1 \equiv r_2$) iff the table entries are compatible for each and every column.*

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

Definition 14 *Table T is* **closed** *iff*
$\forall t \in Sa, \exists s \in S \text{ s.t. } \mathrm{row}(t) \equiv \mathrm{row}(s)$

Definition 15 *Table T is* **consistent** *iff*
$\forall s_1, s_2 \in S, \forall a \ \mathrm{row}(s_1) \equiv \mathrm{row}(s_2) \Rightarrow \mathrm{row}(s_1 a) \equiv \mathrm{row}(s_2 a)$

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 LFFSM. The LFFSM 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 figure 4. This observation table is closed and consistent, according to definitions 14 and 15. This table defines an IO mapping, and therefore a LFFSM, that is derived following the procedure illustrated in table 1 and figure 1.

Table 4: Example of an incompletely filled observation table that is closed and consistent.

	0	1
λ	-	0
0	1	1
1	1	1
00	-	0
01	1	-

The improved version of the algorithm is therefore the following:

1. Initialize an observation table T as defined in section 4.4.
2. Generate an observation table T that is closed and consistent, according to definitions 14 and 15 and the algorithm described in section 4.4. This table defines a set of IO mappings R.
3. Build the LFFSM M'_R that corresponds to this set of IO mappings.
4. Select M, a minimum size finite state machine equivalent to M'_R, using the exact algorithm from section 4.3.
5. Check if M is equivalent to M'. If they are equivalent, stop. Otherwise, let s be a string such that $\lambda(q_0, s) \not\equiv \lambda'(q_0, s)$. Add the counterexample s to set S in table T and 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 equivalent to 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 ISFSMs it will also be more efficient, although we know that, in this case, it cannot always work fast.

5.2 Complexity analysis

Clearly, if the original FSM M' is incompletely specified, the LFFSM 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 [1] that, during the construction of the table:

[2] The equivalence check between the two machines is performed by computing the product machine. For an extended discussion of this issue, we refer the reader to an extended version of this paper.

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 equivalence 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 FSMs 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

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 equivalence queries, by computing the product machine at most n times.

3. At most n calls to the LFFSM reduction algorithm are performed, and this LFFSM 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 LFFSM has $(k+n) \times 2n(n+1)(k+1)$ states, assignments only need to be computed to the $n(n+1)(k+1)$ LFFSM 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 equation 4 is quadratic on the number of states. Therefore, each call to the LFFSM 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 LFFSMs 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 FSMs [8], 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 an equivalent n state machine that is very small, $n << m$ and this algorithm may actually be asymptotically faster, as long as $n^2 < \log m$.

For ISFSMs, 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 3 fail.

The algorithm presented in this section is only applicable to finite state machines that do not exhibit partially specified inputs or outputs. In many cases, finite state machine descriptions usually accept transitions labeled by input values where one or more Boolean inputs are unspecified (e.g. –1-0). It is possible to extend the theory presented to this case, but we omit this discussion here, for space reasons, and present it in an extended version of this work, in preparation.

6 Experimental results and conclusions

To compare our algorithm in a set of problems that are known to be hard, we used the set of problems used by Kam and Villa to evaluate their implicit algorithm (`ism`) against an explicit implementation [10] and also used by other researchers to evaluate heuristic approaches [7]. 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[3] The set of problems used by Kam and Villa ([10], table 1) come from a variety of sources: standard benchmarks, asynchronous synthesis, learning problems, synthesis of interacting FSMs and FSMs that have been constructed to exhibit a large number of compatible pairs.

We compared the execution times of `ism` with those of `bica`, a C++ implementation of the algorithm described in section 5 and `stamina`, a popular implementation of the explicit version of the standard state reduction algorithm [16]. The CPU times obtained in a DECStation 5000/260 with 440MB of memory are reported in table 5, rounded to the nearest second. For `stamina` and `bica`, timeouts were set at 2 hours of CPU time. For `ism` and for the prime generation times, we used the values reported in the literature [10]. In this table, the number of compatibles (column 2) is the total number of compatible sets exhibited by the problem.

Table 5: Results obtained with `ism`, `bica` and `stamina`.

FSM	#	FSM Reduction			Prime Gen.	
	compat	ism	stam	bica	ism	stam
alex1	55928	N.A	16	34	24	16
intel_edge	9432	N.A	0	3	37	3
isend	22207	N.A	1	18	13	failed
rcv-ifc	1.52e11	N.A	0	16	114	failed
rcv-ifc.m	1.79e6	N.A	0	8	3	147
send-ifc	5.07e17	N.A	1	58	571	failed
send-ifc.m	8.98e6	N.A	0	20	3	312
vbe4a	1.75e12	N.A	173	47	109	167
vmebus	5.05e7	N.A	1	1555	26	failed
th.30	97849	*547	failed	2	21	17256
th.40	1.45e6	*6862	failed	3	75	failed
th.55	3.62e7	failed	failed	4794	1273	failed
fo.20	42193	*33	failed	1	2	1369
fo.50	3.64e7	failed	failed	9	216	failed
fo.70	9.62e10	failed	failed	failed	22940	failed
ifsm0	1.0e6	failed	0	5	43	4253
ifsm1	43006	*413	1294	12	25	466
ifsm2	497399	403	694	474	267	356
rubin18	2^{12}-1	failed	failed	0	0	751
rubin600	2^{400}-1	failed	failed	51	1978	failed
rubin1200	2^{800}-1	failed	failed	382	2.7e4	failed
rubin2250	2^{1500}-1	failed	failed	2518	2.7e5	failed
e271	393215	22	0	4	21	failed
e285	393215	13	0	1	13	failed
e304	393215	556	0	1	93	failed
e423	204799	*443	failed	1	102	failed
e680	327679	984	0	1	151	failed

Columns 3, 4 and 5 show the time required for the three programs under comparison to solve a given FSM reduction problem: `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

[3] 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.

the covering problem, and therefore those times are not listed. The last two columns show the time required to generate the list of prime compatibles for this problem. These times have been reported by the authors of `ism`, and it is interesting to notice that, in some cases, stamina can find a minimum solution in less time than it takes to generate all the primes. This happens because if the cover found using only maximal compatibles is closed, there is no need to generate all the primes. We assume that the same approach can be applied to `ism`[4] but in the table of results presented, the binate covering step of `ism` uses the table generated from the full set of primes.

Table 5 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. `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 440MB of memory was used for the comparisons, `bica` does not usually require much memory, with all examples except one requiring much less than 64MB of memory.

7 Conclusions and future work

This work presented a new approach for the problem of reducing incompletely specified finite state machines. Although inherently slower when applied to completely specified finite state machines, the algorithm has the advantage of using a radically new approach that doesn't 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 algorithm for 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. The current version handles partially undefined inputs by making the inputs disjoint, a procedure that, in some examples, degrades the performance by a large factor.

It may also be possible to improve considerably the total execution time by developing better LFFSM reduction techniques than the ones used in this work. In particular state merging techniques [11] have shown great promise in a recent contest held to evaluate the performance of DFA inference algorithms [12]. Although these techniques, as presented by the authors, are heuristic, it is possible to modify them in order to obtain exact algorithms, a line of research that is very interesting in itself.

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, NDFA reduction and DFA equivalence.

Acknowledgements

The authors would like to thank Tiziano Villa, who helped greatly in the preparation of this manuscript and contributed with many helpful suggestions. They also thank Timothy Kam and Stephan Edwards for help in setting up the experimental comparisons, and Prof. Sangiovanni-Vincentelli for his support of this line of research.

[4] Since `ism` follows the same sequence of steps, it can presumably be used to look first for a cover consisting only of maximal compatibles, as `stamina` does, stopping if this cover is closed.

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.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, November 1987.

[2] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21:592–597, June 1972.

[3] A. W. Biermann and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. on Computers*, C-24:122–136, 1975.

[4] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *Proc. Design Automation Conference*, pages 641–646, June 1995.

[5] E. M. Gold. Complexity of automaton identification from given data. *Inform. Control*, 37:302–320, 1978.

[6] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[7] H. Higuchi and Y. Matsunaga. A fast state reduction algorithm for incompletely specified finite state machines. In *Proc. Design Automation Conference*, pages 463–466, New York, June 1996. ACM Press.

[8] J.E. Hopcroft. n log n algorithm for minimizing states in finite automata. Technical Report CS 71/190, Stanford Univiversity, 1971.

[9] T. Kam, T. Villa, R. Brayton, and A. S. Vincentelli. *Synthesis of FSMs: functional optimization*. Kluwer Academic Publishers, 1997.

[10] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni Vincentelli. A fully implicit algorithm for exact state minimization. In *Proc. Design Automation Conference*, pages 684–690, 1994.

[11] K. J. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *Proc. 5th Annu. Workshop on Comput. Learning Theory*, pages 45–52. ACM Press, New York, NY, 1992.

[12] 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, 1998.

[13] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[14] A. L. Oliveira and J. P. M. Silva. Efficient search techniques for the inference of minimum size finite automata. In *Proceedings of the 1998 South American Symposium on String Processing and Information Retrieval*, Santa Cruz de La Sierra, Bolivia, September 1998. IEEE Computer Society Press.

[15] C. F. Pfleeger. State reduction in incompletely specified finite state machines. *IEEE Trans. Computers*, C-22:1099–1102, 1973.

[16] J.-K. Rho, G. Hachtel, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Transactions on Computer-Aided Design*, 13(2):167–177, February 1994.