INFERENCE OF STATE MACHINES FROM EXAMPLES OF BEHAVIOR

by

Arlindo Oliveira and Stephen Edwards

Memorandum No. UCB/ERL M95/12

18 February 1995

INFERENCE OF STATE MACHINES FROM EXAMPLES OF BEHAVIOR

by

Arlindo Oliveira and Stephen Edwards

Memorandum No. UCB/ERL M95/12

18 February 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

INFERENCE OF STATE MACHINES FROM EXAMPLES OF BEHAVIOR

by

Arlindo Oliveira and Stephen Edwards

Memorandum No. UCB/ERL M95/12

18 February 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Inference of State Machines from Examples of Behavior

Arlindo Oliveira[1] Stephen Edwards[2]

February 18, 1995

[1] aml@eecs.berkeley.edu This work was developed while this author was supported by the Joint Services Electronics Program under contract number F49620-94-C-0038.

[2] sedwards@eecs.berkeley.edu This work was developed while the author was supported, in part, by a National Science Foundation Graduate Research Fellowship. Additional support was provided by the Semiconductor Research Corporation under grant number 94-DC-008.

Abstract

Often, the desired behavior of a system is known before a design of the system is known. It usually falls to the designer to correctly translate this behavior into a system that exhibits it.

This report describes two algorithms that design systems guaranteed to exhibit specified behavior. Specifically, our algorithms identify a state machine with the fewest states exhibiting behavior specified by a set of input/output strings. One algorithm builds the machine explicitly by fitting together transitions that correspond to input/output pairs. The other implicitly considers all machines of a given size and discards those not exhibiting the desired behavior.

Although the problem is NP-complete, our algorithms behave exponentially only in the number of states in the minimum machine; other state-minimization algorithms behave exponentially in the size of the behavior specification. This advantage has allowed machines of up to fourteen states to be identified exactly within an hour.

Contents

Chapter 1

Introduction and Related Work

This report addresses the problem of selecting a deterministic finite state machine (FSM) with a minimum number of states consistent with a given sample of input/output behavior. This sample can consist of one or more input sequences or strings. Each input sequence is associated with an output sequence obtained by simulating an unknown (but fixed) FSM on that input sequence. We call a set of input/output sequences a *specification*.

To make the problem more general, we assume that some of the outputs produced by the FSM in response to the input string may be unavailable. In particular, if the machine has a Boolean-valued output and that, for each string, only the final output is known, then the algorithm will find the deterministic finite automaton (DFA) with minimum number of states that accepts the strings whose output was one and rejects those whose output was zero.

The reverse also works. Given an algorithm that computes the minimum DFA accepting a set of strings and rejecting another, it is easy to select the minimum Moore machine consistent with the observed data.

The problem of selecting the minimum DFA consistent with a set of labeled strings is NP-complete. Gold [Gol78] 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.

If *all* strings of length n or less are given (a *uniform-complete* sample), the problem can be solved in polynomial time [GH66, TB73, PF88]. However, Angluin has shown [Ang78] that even if an arbitrarily small fixed fraction $(|\Sigma^{(n)}|)^{\epsilon}$, $\epsilon > 0$ is missing, the problem remains NP-complete.

The problem becomes easier if the algorithm is allowed to make queries or experiment with the unknown machine. Angluin [Ang87] proposes an algorithm based on the approach described by Gold [Gol72] that solves the problem in polynomial time by allowing the algorithm to ask membership queries. Schapire [Sch92] proposes an interesting approach that does not require the availability of a reset signal to take the machine to a known state.

All these algorithms, however, solve easier versions of the problem addressed here. Our algorithms take a set of labeled strings and do not make queries or experiment with the machine. The best algorithms known for the problem addressed here, where the learner has no control over the training set, remain those proposed by Bierman et al. [BK76, BP75]. Based on an explicit search algorithm guaranteed to obtain the exact solution, they require, in general, exponential time. Section 3.2 details an explicit enumeration method similar to Bierman's, and uses a data structure that makes the implementation very efficient.

Recently, connectionist approaches to learning from a given set of strings have been proposed. These have had limited success. Polack [Pol91], Giles et al. [GMC+92] and Das and Mozer [DM93] propose different approaches based on gradient descent algorithms for neural network training, but their results show this strategy does not have any important advantages over search-based methods (e.g., those described in this report). Being heuristic algorithms, they are not guaranteed to find the exact solution. Moreover, the size of problems they can handle is very limited. For example, they are not able to solve some of the Tomita grammars [Tom82], none of which require larger than five-state DFAs. The main purpose of the connectionist work, however, was not to beat discrete search algorithms, but to evaluate the feasibility of such an approach to problems of this kind.

Lang [Lan92] describes a much more promising heuristic approach. Although it fails to find the smallest

DFA in many cases, it is a very efficient algorithm and and can find approximate solutions for machines with several hundred states.

The problem of selecting the minimum automaton consistent with a set of strings can be transformed into the problem of minimizing an incompletely-specified finite state machine[1]. Pfleeger [Pfl73] showed minimizing these is NP-complete. However, since this problems is of great practical importance, many different algorithms have been developed for its solution. Paull and Unger [PU59] were the first to propose a method based on the selection of compatibility classes, or compatibles. A compatible is a set of states equivalent in the sense that they can be merged without affecting the behavior of the machine. The minimum machine can be found by selecting a minimum set of compatibles that satisfies two simple requirements. This method was improved by Grasselli and Luccio [GL65], who showed that only a subset of the compatibles, the prime compatibles, need be considered. Hachtel et al.'s [HRSJ91] `stamina` program provides an efficient implementation of this algorithm.

This algorithm is still the state-of-the-art in finite state machine reduction for the majority of cases. Some problems, however, exhibit exponentially large numbers of compatibles, rendering an explicit enumeration approach such as `stamina`'s ineffective. In particular, incompletely-specified finite state machines generated from a behavior specification as described in the next chapter tend to have an extremely large number of compatibles. In this case, a version of the Grasseli and Luccio algorithm based on the implicit enumeration of the compatibles is more efficient. Kam et al. describe such a scheme [KVBSV94]. Comparisons between techniques presented in this report and these two algorithms are presented in Section 4.1.

[1] The exact way in which this reduction is performed is described in the next chapter.

Chapter 2

Problem Description

This chapter describes the form of inputs and outputs to our algorithms and defines their relationships. Section 2.1 defines the format we use to specify behavior, and Section 2.2 defines the type of systems our algorithms produce. Section 2.3 defines the two requirements we satisfy to ensure that a system behaves as specified, and shows they are sufficient. The section concludes with a theorem showing there is a trivially-constructed machine for each behavior specification.

2.1 Behavior Specification

Figure 2.1 shows the input/output behavior of a deterministic finite-state machine. From the reset state, it produces a 0 after being given an A. Then, it produces a 1 after being given another A.[1] Then, it produces a 0 after a B, and so forth.

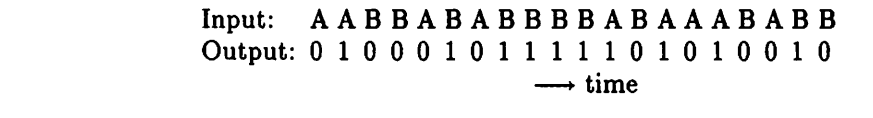


Figure 2.1: A simple specification of input/output behavior: each input produces the output beneath it. The machine starts in the reset state before the leftmost input and proceeds through other states.

Figure 2.2 shows a more elaborate specification of input/output behavior. Here, the machine should produce a 1 after the the input strings in the "accept" column, and a 0 after the input strings in the "reject" column, but the machine is allowed to produce anything between the beginning and end of each string.

Our algorithms accomodate both of these specifications through a labeled tree. Figures 2.3 and 2.4 show how the two example behaviors are translated into our format, described below.

Definition 1 (Behavior Specification) *A* specification *is a 5-tuple:*

$$S = (\Sigma, \Delta, N, n_0, T)$$

where

$\Sigma \neq \emptyset$ *is a finite set of input symbols (we will use a to denote a particular input symbol)*

$\Delta \neq \emptyset$ *is a finite set of output symbols (we will use b to denote a particular output symbol)*

$N \neq \emptyset$ *is a finite set of nodes (we will use n to denote a particular node)*

[1] Here, the machine cannot be in the reset state, since in the reset state, an A produces a 0. Considerations like this lead us to a lower bound on the number of states in the minimum machine.

accept	reject
1	0
11	10
111	01
1111	00
11111	011
111111	110
1111111	11111110
11111111	10111111

Figure 2.2: A specification of input/output behavior: strings in the "accept" column produce a 1 at the end, those in the "reject" column produce a 0. From Tomita [Tom82].

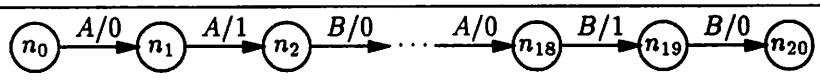


Figure 2.3: A specification for the behavior in Figure 2.1: each node will be part of some state in the minimal machine. Each transition is labeled with its input/output.

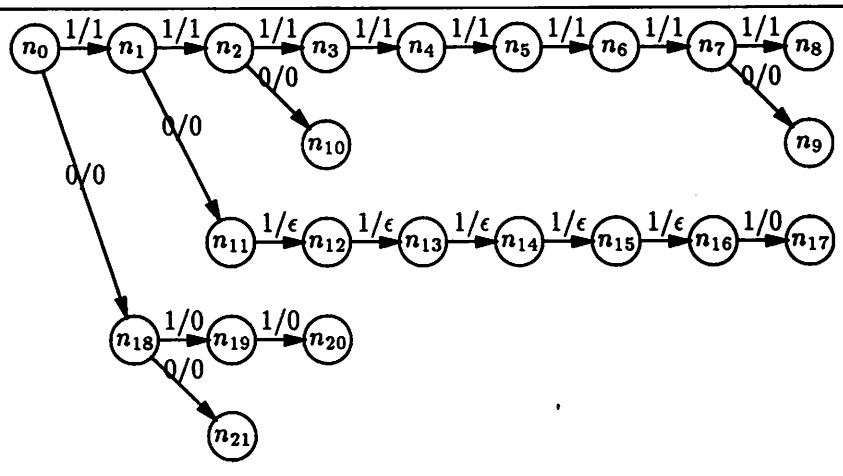


Figure 2.4: A specification for the behavior of Figure 2.2: each string is a path in the graph, and strings with identical prefixes have been merged. An output of ϵ indicates it is unspecified.

$n_0 \in N$ *is the initial node*

T *is a set of transitions of the form* (a, n_s, n_d, b) *where*

$a \in \Sigma$ *is the input*

$n_s \in N$ *is the source node*

$n_d \in N$ *is the destination node*

$b \in \Delta \cup \{\epsilon\}$ *is the output (ϵ denotes unspecified)*

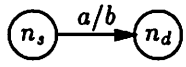
that satisfies

For all $a \in \Sigma$, $n_s \in N$, *there is at most one* $t = (a, n_s, n_d, b) \in T$. *($T$ is* deterministic*)*

For all $a \in \Sigma$, $n_d \in N \neq n_0$ *there is exactly one* $t = (a, n_s, n_d, b) \in T$;

there is no $t = (a, n_s, n_0, b) \in T$. *($T$ is* tree-structured*)*

The transitions and nodes in a behavior specification form a tree, with the reset state on the left, and using the following notation



represents $t = (a, n_s, n_d, b)$

Definition 2 (Behavior Containment) *An input sequence* $(a_{s_1}, a_{s_2}, \ldots, a_{s_k})$, $a_{s_i} \in \Sigma$ *is* part of a specification S *if there exists a sequence of nodes* $(n_{r_0}, n_{r_1}, \ldots, n_{r_k})$, $n_{r_i} \in N$, $n_{r_0} = n_0$, *such that for* $i = 1, \ldots, k$ *there is a* $t \in T$ *such that*

$$t = (a_{s_i}, n_{r_{i-1}}, n_{r_i}, b_{t_i})$$

The output of this sequence is $(b_{t_1}, b_{t_2}, \ldots, b_{t_k})$.

Definition 2 tells us that paths in Figure 2.4 correspond to strings in Figure 2.2. For example, the rejected string 110 corresponds to the path connecting nodes n_0, n_1, n_2, and n_{10}. The transition $(0, n_2, n_{10}, 0)$ with output 0 indicates that this string was not accepted. Also, since the set of strings in Figure 2.2 is not prefix-closed (e.g., we know nothing of the string 101), not every output is specified. For example, the transition $(1, n_{11}, n_{12}, \epsilon)$ has an unspecified output.

Our specifications can be interpreted as incompletely-specified finite state machines. Theorem 2 on Page 9 shows that a specification is equivalent to a machine whose behavior is contained in the specification, but this rarely has the minimum number of states.

The minimum machine can, in theory, be obtained by sending the specification directly to a traditional state-minimizer, but this is infeasible for all but the smallest behavior specifications: see our experimental results in Section 4.1.

2.2 State Machines

We consider two types of state machines, Moore and Mealy. A Moore machine is a Mealy machine whose output does not directly depend on its input.

Definition 3 (Mealy Machines) *A* Mealy Machine *is a 6-tuple* $M = (\Sigma, \Delta, Q, q_0, \delta(a, q), \lambda(a, q))$ *where*

$\Sigma \neq \emptyset$ *is a finite set of input symbols (we will use a to denote a particular input symbol)*

$\Delta \neq \emptyset$ *is a finite set of output symbols (we will use b to denote a particular output symbol)*

$Q \neq \emptyset$ *is a finite set of states (will use q to denote a particular state)*

$q_0 \in Q$ *is the initial "reset" state*

$\delta(a, q) : \Sigma \times Q \rightarrow Q \cup \{\phi\}$ *is the transition function*

$\lambda(a, q) : \Sigma \times Q \rightarrow \Delta \cup \{\epsilon\}$ *is the output function*

ϕ *denotes an unspecified transition,* ϵ *denotes an unspecified output.*

Definition 4 (Moore Machines) *A* Moore Machine *is a Mealy Machine where* $\lambda(a_1, q) = \lambda(a_2, q)$ *for all* $a_1, a_2 \in \Sigma$, *i.e., the output of a Moore Machine does not depend on the input, only the state.*

Definition 5 (Output of a Sequence) *The notation* $\lambda(a_k, a_{k-1}, \ldots, a_1)$ *denotes the output of a Moore or Mealy machine after a sequence of inputs* $(a_1, \ldots, a_k)$, $a_i \in \Sigma$, *when all transitions are specified, i.e.,*

$$\begin{aligned} \delta(a_1, q_0) &\neq \phi \\ \delta(a_2, \delta(a_1, q_0)) &\neq \phi \\ &\vdots \quad \vdots \\ \delta(a_k, \delta(a_{k-1}, \delta(\cdots, \delta(a_1, q_0)\cdots))) &\neq \phi \end{aligned}$$

The output is defined to be

$$\lambda(a_k, a_{k-1}, \ldots, a_1) \equiv \lambda\big(a_k, \delta(a_{k-1}, \delta(\cdots, \delta(a_1, q_0)\cdots))\big) \tag{2.1}$$

By definition, if M *is a Moore Machine, then* $\lambda(a_k, \ldots, a_1)$ *is independent of* a_k.

2.3 The Satisfying Criteria

Our aim is to construct a machine M which exhibits behavior consistent with the specification S. We define "consistent" in terms of a *satisfying machine*:

Definition 6 (Satisfying Machine) *A machine* M *is said to* satisfy *a behavior specification* S *if, for any input sequence* $(a_{s_1}, a_{s_2}, \ldots, a_{s_k})$ *which is part of the specification* S,

$$b_{t_k} \neq \epsilon \text{ implies } \lambda(a_{s_k}, \ldots, a_{s_1}) = b_{t_k} \tag{2.2}$$

where b_{t_k} *is the output of the last transition in the sequence according to Definition 2.*

Definition 7 (Output Requirement) *A machine* M satisfies the output requirement *of a specification* S *if there exists a function* $F : N \rightarrow Q$ *such that*

$$\lambda(a, F(n_s)) = b \qquad \forall t = (a, n_s, n_d, b) \in T \text{ s.t. } b \neq \epsilon \tag{2.3}$$

Definition 8 (Transition Requirement) *A machine* M satisfies the transition requirement *of a specification* S *if there exists a function* $F : N \rightarrow Q$ *with* $F(n_0) = q_0$ *such that*

$$\delta(a, F(n_s)) = F(n_d) \qquad \forall t = (a, n_s, n_d, b) \in T \tag{2.4}$$

The following theorem shows the utility of the output and transition requirements. Informally, the transition requirement ensures that the sequences contained in the behavior go to the "right" state in the machine, and the output requirement ensures that the output of that state is correct.

Theorem 1 (Output and Transition Requirements are Sufficient) *If a machine* M *satisfies both the output requirement (2.3) and the transition requirement (2.4) of a specification* S *(with the same node-to-state mapping function* F*), then* M *satisfies the behavior specification* S.

Proof. We need to show that if there is a function F that satisfies (2.3) and (2.4), then any input sequence $(a_{s_1}, a_{s_2}, \ldots, a_{s_k})$ present in the behavior has the output of (2.2).

Assume $(a_{s_1}, a_{s_2}, \ldots, a_{s_k})$ is present in the behavior, i.e., there is a sequence of nodes $(n_{s_0}, \ldots, n_{s_k})$ such that for $i = 1, \ldots, k$ there exists a $t \in T$ such that

$$t = (a_{s_i}, n_{s_{i-1}}, n_{s_i}, b_{s_i})$$

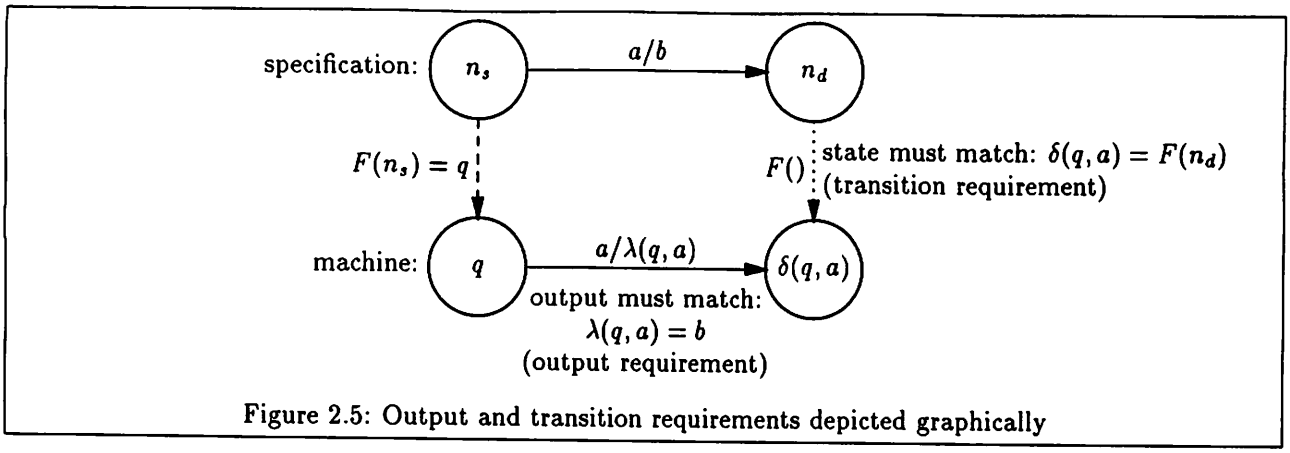


Figure 2.5: Output and transition requirements depicted graphically

From (2.1), the output of the machine M after the input sequence is

$$\lambda(a_{s_k}, \delta(a_{s_{k-1}}, \delta(\cdots, \delta(a_{s_1}, q_0)\cdots)))$$

The transition requirement, (2.4), implies

$$\begin{aligned} \delta(a_{s_1}, F(n_{s_0})) &= F(n_{s_1}) \\ \delta(a_{s_1}, q_0) &= F(n_{s_1}) \end{aligned} \tag{2.5}$$

Similarly, if

$$\delta(a_{s_{i-1}}, \delta(a_{s_{i-2}}, \ldots, \delta(a_{s_1}, q_0))) = F(n_{s_{i-1}})$$

then

$$\begin{aligned} \delta(a_{s_i}, \delta(a_{s_{i-1}}, \ldots, \delta(a_{s_1}, q_0))) &= \delta(a_{s_i}, F(n_{s_{i-1}})) \\ &= F(n_{s_i}) \end{aligned} \tag{2.6}$$

Finally, (2.5) and (2.6) tell us that (2.6) holds for all sequences, so

$$\begin{aligned} \lambda(a_{s_k}, a_{s_{k-1}}, \ldots, a_{s_1}) &= \lambda(a_{s_k}, \delta(a_{s_{k-1}}, \delta(\cdots, \delta(a_{s_1}, q_0)\cdots))) \\ &= \lambda(a_{s_k}, F(n_{s_{k-1}})) \\ &= b_{s_k} \text{ if } b_{s_k} \neq \epsilon \end{aligned}$$

This shows that the machine satisfies the behavior for any sequence present in the behavior, completing the proof. □

Consider the machine

$$M = (\Sigma, \Delta, Q, F(n_0), \delta(a,q), \lambda(a,q)) \tag{2.7}$$

constructed by mapping each node in the specification

$$S = (\Sigma, \Delta, N, n_0, T) \tag{2.8}$$

to a unique state, i.e.,

$$F(n_i) = q_i \ \forall i \tag{2.9}$$

and converting each transition in the specification to a transition in the machine

$$\delta(a, F(n_s)) = \begin{cases} F(n_d) & \text{when there exists a } t = (a, n_s, n_d, b) \in T \\ \phi & \text{otherwise} \end{cases} \tag{2.10}$$

$$\lambda(a, F(n_s)) = \begin{cases} b & \text{when there exists a } t = (a, n_s, n_d, b) \in T \\ \epsilon & \text{otherwise} \end{cases} \tag{2.11}$$

Theorem 2 *The machine (2.7) created from the specification (2.8) using (2.9), (2.10), and (2.11) satisfies the specification.*

Proof.

The transition requirement (2.4) is satisfied,

$$\begin{aligned} \delta(a, F(n_s)) &= \delta(a, n_s) \\ &= n_d \\ &= F(n_d) \end{aligned}$$

and the output requirement (2.3) is satisfied,

$$\begin{aligned} \lambda(a, F(n_s)) &= \lambda(a, n_s) \\ &= b \end{aligned}$$

so by Theorem 1, M satisfies the specification S. □

Chapter 3

Solution Techniques

In this chapter, we present two algorithms for finding machines that satisfy the output and transition requirements. Theorem 1 showed satisfying these is sufficient for a machine to satisfy the specification.

Presented first is an explicit algorithm that builds a machine by traversing the transitions in the specification, fitting them into the machine so the output and transitions requirements are satisfied. This is very similar to Bierman et al.'s work [BK76, BP75].

Presented second is an implicit algorithm that simultaneously considers all possible mapping functions F, and eliminates those not satisfying the output and transition requirements.

Both algorithms look for a minimum machine in stages. They start with a lower bound on the number of states in the minimum machine and look for a machine with that many states. If no machine is found, they allow an additional state and continue searching until a satisfying machine is found. This always terminates since there is at least one machine satisfying any specification (Theorem 2).

The lower bound on the number of states in the minimal machine is computed using the incompatibility graph, which has an arc between pairs of nodes in the specification that cannot be mapped to the same state in a satisfying machine. Nodes in a clique in this graph must all be mapped to different states, so the size of any clique is a lower bound on the size of the minimum machine. We find a large clique in the graph with a fast heuristic and use its size as a lower bound.

3.1 The Incompatibility Graph

The *incompatibility graph* represents information about which nodes in the behavior specification can be merged, i.e., can be mapped to the same state in a satisfying machine. It is an undirected graph with one node per node in the behavior specification and edges between pairs of incompatible (unmergeable) nodes.

The incompatibility graph is represented by a function $I : N \times N \rightarrow \{1, 0\}$. $I(n_i, n_j)$ is 1 if and only if nodes n_i and n_j are incompatible. As mentioned above, this is an undirected graph, so that $I(n_i, n_j) = I(n_j, n_i)$ for all $n_i, n_j \in N$. A node is never incompatible with itself, i.e., $I(n_i, n_i) = 0$.

We define two types of incompatibility. (see Figure 3.1)

- Output Incompatibility

  Two nodes are *output incompatible* if merging them would immediately violate the output requirement. For a Mealy machine, this happens when, for some particular input, the two nodes would produce a different output. For a Moore machine, this happens when the two nodes produce different outputs for any input.

- Transitive Incompatibility

  Two nodes are *transitively incompatible* if, on the same input, they lead to incompatible nodes. If the transition requirement (2.4) holds, merging the two nodes would lead to a violation of the output requirement.

We need a notion of inequality for outputs that takes into account unspecified outputs.

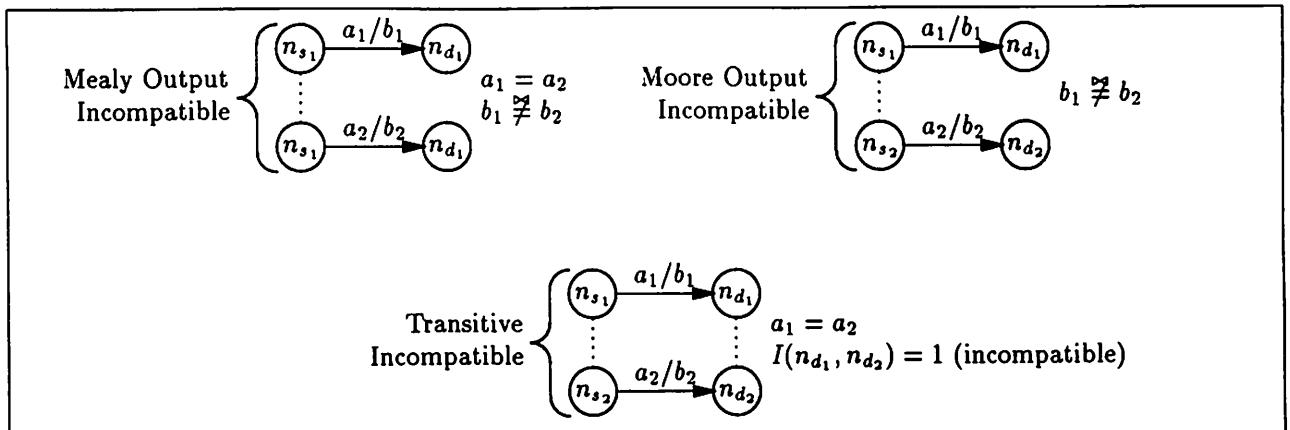


Figure 3.1: A graphical depiction of output and transitive incompatibility.

Definition 9 (Output Inequality) *The relation* $\overset{\text{out}}{\neq}$ *for* $b_1, b_2 \in \Delta \cup \{\epsilon\}$ *is*

$$b_1 \overset{\text{out}}{\neq} b_2 \quad = \begin{cases} \textit{false} & \textit{if } b_1 = b_2 \neq \epsilon \\ \textit{false} & \textit{if } b_1 = \epsilon \textit{ or } b_2 = \epsilon \textit{ or both} \\ \textit{true} & \textit{otherwise} \end{cases}$$

Definition 10 (Incompatibility Graph) *The incompatibility graph is*

$$I(n_{s_1}, n_{s_2}) = \begin{cases} 1 & \textit{if } \exists t_1 = (a_1, n_{s_1}, n_{d_1}, b_1) \in T, \\ & t_2 = (a_2, n_{s_2}, n_{d_2}, b_2) \in T \textit{ s.t.} \\ & a_1 = a_2^{\dagger} \textit{ and } b_1 \overset{\text{out}}{\neq} b_2 \textit{ (output)} \\ 1 & \textit{if } \exists t_1 = (a_1, n_{s_1}, n_{d_1}, b_1) \in T, \\ & t_2 = (a_2, n_{s_2}, n_{d_2}, b_2) \in T \textit{ s.t.} \\ & a_1 = a_2 \textit{ and } I(n_{d_1}, n_{d_2}) = 1 \textit{ (transitive)} \\ 0 & \textit{otherwise} \end{cases} \tag{3.1}$$

For Moore machines, the condition marked with a dagger (†) is omitted, that is, a_1 *and* a_2 *may differ.*

3.1.1 A Clique in the Incompatibility Graph

A clique in the incompatibility graph gives a lower bound on the size of the minimum machine. By definition, pairs of incompatible nodes cannot be merged, so a clique corresponds to a group of nodes that must be assigned to different states in the machine.

We locate a large clique in the incompatibility graph using a slightly modified version of an exact algorithm by Carraghan and Pardalos [CP90], shown in Figure 3.2. The algorithm takes a set of nodes, forms subsets incompatible with another node from the set, and calls itself on these subsets. Each node from each subset is considered in turn, although only nodes which are "later" in the set (according an ordering imposed at the beginning) are considered to be part of the new subset.

Our heurstic modification to this algorithm places a limit on the amount of time the algorithm may spend looking for a clique. When a clique is located, the algorithm is given twice as much time as it took to find that clique. The time for finding the initial clique is set arbitrarily to two seconds, although we observe that the first clique is always found long before this timeout.

Our timeout scheme arose from observing that in most cases, a clique of maximum cardinality was found quickly, but the algorithm would spend a long time after this convincing itself there was none larger.

3.1.2 Using a Clique to Search for the Minimum Machine

Since we expect the minimum machine to have far fewer states than the number of nodes in the specification, our algorithms look first for small machines, starting at the lower bound on the number of states given by

```
MaxClique(C = {c_1, ..., c_n} ⊆ N, depth)
  for i := 1, ..., n − depth                          nodes being examined
    O := {c : c ∈ {c_{i+1}, ..., c_n}, I(c_i, c) = 1}   node to consider next
    if O ≠ ∅
      MaxClique(O, depth + 1)
    else if depth ≥ maxclique
      maxclique := depth + 1                          found a clique larger than any known
```

Figure 3.2: Exact clique-finding algorithm [CP90], called with MaxClique(N, 0). Result is maxclique.

```
FindSmallestMachine(S)
  Sort the transitions in S in a breadth-first order
  Form the incompatibility graph I from the specification S
  C := large clique in I
  for n := |C|, |C| + 1, ...
    if FindMachine(S, I, n) finds a machine
      return that smallest machine
```

Figure 3.3: The general algorithm

the size a the large clique in the incompatibility graph. If a machine with this many states cannot be found, an additional state is allowed and the search is continued.

This scheme has proven effective. An early attempt of ours used a branch-and-bound technique that looked for machines no larger than the smallest-known satisfying machine. Usually, a large satisfying machine (roughly twice the size of the minimum) was found quickly, and the algorithm spent a long time looking for machines much larger than the minimum. Even starting the existing algorithm with a lower bound of one state was faster than the branch-and-bound algorithm. Finally, looking for machines with a fixed number of states leads to faster data structures, another reason to prefer the presented algorithms to the branch-and-bound approach.

3.2 The Explicit Algorithm

The explicit algorithm attempts to build a satisfying machine with no more than m states by traversing the specification and fitting its transitions into the machine. If the algorithm traverses the entire specification, the constructed machine satisfies the specification, otherwise there is no satisfying machine with m states.

The explicit algorithm fits transitions in the specification in a breadth-first order (sorted by the algorithm of Figure 3.5). The reasons for this are twofold. First, it ensures that each transition's source node is mapped to a state when the transition is considered. Second, it forces the algorithm to use important information as soon as possible, which is important for a backtracking algorithm.

There are two possibilities when fitting a transition:

1. There is no corresponding transition in the machine:

   In Figure 3.4a, node n_s is mapped to state q_s, but there is no transition from q_s with input a_1. The algorithm must create a new transition with input a_1 and output b_1, but there are many possibilities for the destination state.

   The incompatibility graph reduces the number of destination states to consider. If a node incompatible with n_d has been mapped to a state, choosing that state would lead to an inconsistent machine, so none of these states are considered.

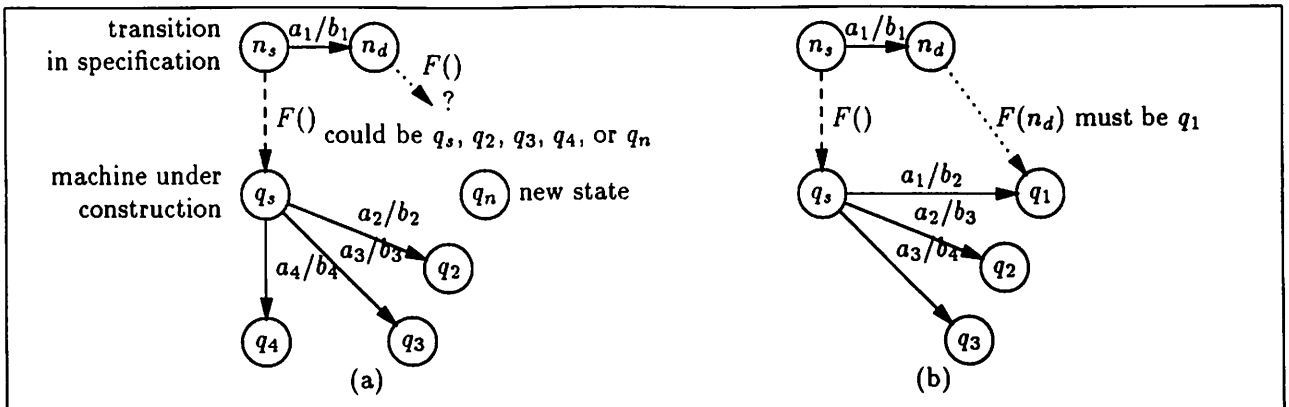


Figure 3.4: The two main cases in the explicit algorithm: (a) there is no corresponding transition, so there is a choice of destination state (b) there is a corresponding transition, so b_1 must equal b_2 for the transition to fit.

In addition to existing states, if there are fewer than m states in the machine, the correct destination state could be a new state. The algorithm also considers this.

The algorithm tries all possible choices of destination state. It chooses one and tries to satisfy the remainder of the specification. If this works, the choice was correct and the algorithm terminates successfully. Otherwise, the next untried destination state is considered. If none of the possible destination states leads to a satisfying machine, then some earlier choice was incorrect or there are no satisfying machines of size m. In either case, the algorithm indicates it was unable to satisfy the remainder of the specification.

In choosing a destination state, the most-recently added states are considered first, followed by a new state, if allowed. This order was chosen to keep the number of possible destination states low. However, this order does not affect the runtime when there is no satisfying machine with m states. Since all possibilities must be considered before the algorithm can conclude there is no machine, the order in which they are considered is irrelevant.

2. There is already a corresponding transition in the machine:

   In Figure 3.4b, node n_s is mapped to state q_s, which has a transition with the input a_1. There are two cases, depending on whether the outputs differ:

   (a) If the outputs match ($b_1 = b_2$), then node n_d is mapped to state q_1 and the next transition can be fitted.

   Because this step is simple, any transitions considered after the machine satisfies the entire specification are fitted very rapidly. This gives the algorithm a linear time-dependence on the size of the specification.

   (b) If the outputs differ ($b_1 \neq b_2$), then a previous decision was incorrect and the algorithm indicates it was unable to satisfy the specification. It will backtrack to its last choice of destination state and consider another.

The explicit algorithm uses a single recursive routine (Figure 3.6) for backtracking. At each recursive step, the routine is given

- a partially-constructed machine M
- a partially-constructed mapping function F
- the breadth-first sorted specification S
- a depth d: an index into the breadth-first sorted list of transitions in S

```
BREADTHFIRSTORDERING(S)
  i := 0                                          have not ordered any transitions
  C[n_1 ... n_m] := unconsidered                  have not considered all nodes
  h := 1                                          set head of queue to one past the beginning
  t := 0                                          set tail of queue to be beginning
  Q[0] := n_0                                     put the initial node at the beginning of the queue
  C[n_0] := considered                            mark the initial node as considered
  while h > t                                     head beyond tail
    foreach t = (a, Q[t], n_d, b) ∈ T             transition from the node at the tail
      T[i] := t                                   add the transition
      i := i + 1
      if C[n_d] = unconsidered
        Q[h] := n_d
        h := h + 1                                add the unconsidered node to the queue
        C[n_d] := considered
    t := t + 1                                    look at the next node in the queue
```

Figure 3.5: The breadth-first transition ordering algorithm

- the maximum number of allowed states m

By construction, on entry, M satisfies the output and transition requirements for transitions 1, 2, ..., $d-1$, and the source node of transition d has been mapped to some state in M.

The routine is first called with the node-to-state mapping function $F : N \rightarrow Q \cup \{\phi\}$ only defined for the mapping between the initial node and the reset state, i.e., $F(n_0) = q_0$, and $F(n \neq n_0) = \phi$.

The explicit algorithm was implemented in C++ using classes for variable-sized behavior specifications, fixed-sized incompatibility graphs, and fixed-sized machines under construction. A stack representation was used for both the mapping function F and the machine M, removing the need for a separate copy of these for each recursive invocation of the routine. This is a significant memory saving because finding the minimal machine requires as many invocations as transitions in the specification. At most, one machine, one mapping function, the behavior specification, the incompatibility graph, and as many stack frames as transitions in the behavior specification are present in memory.

The current implementation of the explicit algorithm uses a simple-minded $O(n^3)$ scheme for forming the incompatibility graph. Surprisingly, for some examples this turned out to be the slowest part of the program. A production version of the algorithm would use a more efficient scheme.

3.3 The Implicit Algorithm

The implicit approach described in this section avoids the need to explicitly search for a satisfying mapping function. It does so by keeping an implicit description of all the mapping functions that satisfy the output and transition requirements.

The discrete function manipulation needed to keep this implicit list of possible mappings is performed by a multi-valued decision diagram (MDD) package. Discrete function manipulation using MDDs is briefly described in Section 3.3.1.

This approach makes the implicit algorithm considerably simpler to describe, but incurs the overhead imposed by the use of discrete function manipulation routines. This overhead can be recovered if the regularities of the problem make the use of an implicit enumeration technique more efficient than an explicit one.

```
FitTransition(M, F, S, d, m)
  if d = |T|                                                    fitted all transitions
    minimal machine := M
    return success                                              machine satisfies the specification

  (a, n_s, n_d, b_S) := T[d]                                    get the transition from the specification
  (a, q_s, q_d, b_M) := (a, F(n_s), δ(a, F(n_s)), λ(a, F(n_s)))   get the corresponding transition from the machine

  if b_S ≱ b_M
    return failure                                              output requirement violated
  if F(n_d) ≠ φ and q_d ≠ φ and F(n_d) ≠ q_d
    return failure                                              transition requirement violated

  M' = M                                                        machine to be modified
  F' = F                                                        mapping function to be modified

  if q_d = φ                                                    machine's destination state undefined
    if b_S ≠ ε and b_M = ε                                      specification's output defined, machine's not
      λ'(a, q_s) := b_S                                         set machine's output to specification's
    foreach q ∈ Q s.t. ∄n s.t. F(n) = q, I(n, n_d) = 1 for all states in the machine which have not been
      assigned nodes that are incompatible with the destination of the transition
      δ'(a, q_s) := q                                           set the destination state to that compatible state
      F'(n_d) := q
      if FitTransition(M', F', S, d + 1, m) ≠ failure
        return success
    if |Q| < m
      Q' := Q ∪ q                                               add a new state q
      δ'(a, q_s) := q
      F'(n_d) := q                                              try q as the destination state
      if FitTransition(M', F', S, d + 1, m) ≠ failure
        return success
    return failure
  else                                                          machine's destination state defined
    F'(n_d) := q_d                                              assign the destination node
    if b_M = ε and b_S ≠ ε
      λ'(a, q_s) := b_S                                         set the machine's output
    return FitTransition(M', F', S, d + 1, m)
```

Figure 3.6: The explicit algorithm's recursive routine. The notation $T[i] = (a_i, n_{s_i}, n_{d_i}, b_i)$ represents the ith transition in the breadth-first ordering of the transitions in the behavior specification.

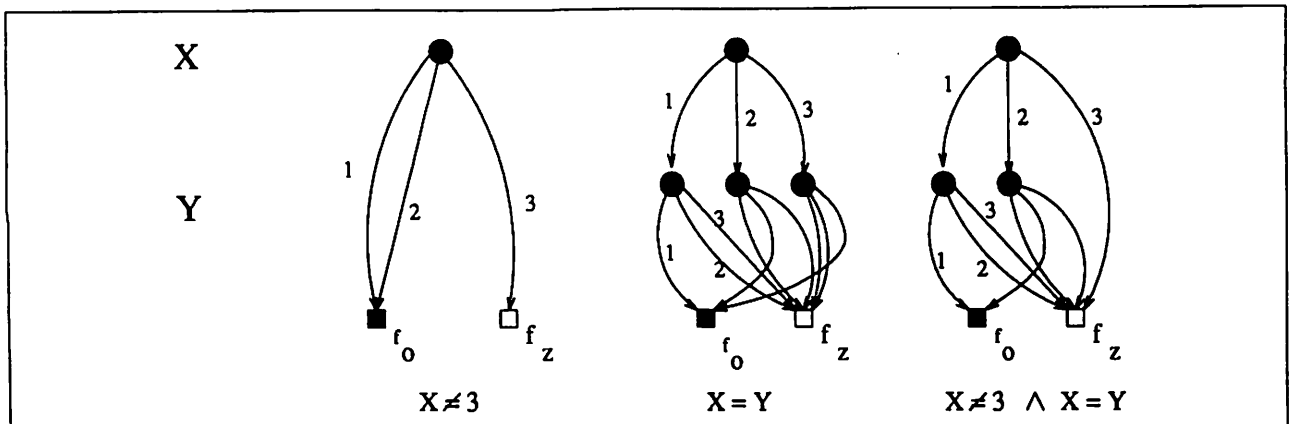


Figure 3.7: Graphic representation of the MDDs for function $X \neq 3$, $X = Y$ and $X \neq 3 \wedge X = Y$.

3.3.1 Multi-valued Decision Diagrams

Any Boolean function of k discrete variables, $x_1, x_2, \ldots, x_k$:

$$F : P_1 \times P_2 \times \cdots \times P_k \rightarrow \{0, 1\} \tag{3.2}$$

can be represented by a Multi-valued Decision Diagram [KB90] (MDD). An MDD is a rooted, directed, acyclic graph where each non-terminal node is labeled with the name of one variable. An MDD for F has two terminal nodes f_z and f_o that correspond to the leaves of the graph. Every non-terminal node f_i, labeled with variable x_j, has $|P_j|$ outgoing edges labeled with the possible values of x_j. Each of these edges points to one child node. The value of $\mathcal{F}$ for any point in the input space can be computed by starting at the root and following, at each node, the edge labeled with the value assigned to the variable tested at that node. The value of the function is 0 if this path ends in node f_z and 1 if it ends in node f_o.

An MDD is *reduced* if no two nodes that branch exactly in the same way exist and no node exists that has all the edges pointing to the same node. An MDD is *ordered* if there is an ordering of the variables such that, for all paths in the MDD, the variables are always tested in that order, possibly skipping some of them.

For a given variable ordering, reduced, ordered MDDs are canonical representations for functions defined over that domain, thus implying that two functions can easily be checked for equivalence.

Our implicit algorithm uses the MDD package described by Kam and Brayton [KB90]. This MDD package provides an array of primitives for function manipulation. The reader is referred to that reference for a more detailed description of these primitives.

We use the following MDD package primitives:

1. Boolean operations (*and, or, not*) between two functions: E.g., $f := g \wedge h$ returns the *and* of g and h, a function that is 1 only if both g and h are 1.

2. Primitive functions that express relations between variables: E.g., $f := (x_i = x_j)$ returns the function that is 1 for all points of the input space where $x_i = x_j$.

3. Existential quantification: E.g., $f := \exists x_i g$ returns a function $f(x_1 \ldots x_{i-1}, x_{i+1} \ldots x_n)$ that is 1 if function $g(x_1 \ldots x_{i-1}, x_i, x_{i+1} \ldots x_n)$ is 1 for some value of x_i.

4. Variable substitution: E.g., $f := [x_i \rightarrow x_j]g$ returns a function f that is obtained from g by replacing the occurrences of variable x_i by variable x_j.

Figure 3.7 depicts the MDDs for the function $f := (x \neq 3)$, $g := (x = y)$ and $h := f \wedge g$, all defined over $P \times P$, $P = \{1, 2, 3\}$.

```
MAINLOOP()
  F := 1
  R := ∅                                          Stores the processed transitions
  foreach t_i := (a_i, n_{s_i}, n_{d_i}, b_i) ∈ T
    R := R ∪ t_i                                  Add this transitions to the list
    foreach t_j := (a_j, n_{s_j}, n_{d_j}, b_j) ∈ R
      if a_i = a_j ∧ b_i ≠ b_j                    Output determinism restriction
        F := F ∧ (x_{s_i} ≠ x_{s_j})
      if a_i = a_j                                Next state determinism restriction
        F := F ∧ ((x_{s_i} ≠ x_{s_j}) ∨ (x_{d_i} = x_{d_j}))
  return F
```

Figure 3.8: The implicit algorithm basic loop

3.3.2 The Implicit Enumeration Algorithm

An implicit list of the valid mapping functions $F : N \rightarrow Q$ can be directly manipulated using simple Boolean operations. This list is kept by considering a function $\mathcal{F} : Q^{|N|} \rightarrow \{0,1\}$ defined as follows:

Definition 11 *$\mathcal{F}(x_0, x_1, \ldots, x_{|N|-1}) = 1$ for the point $v_0, v_1, \ldots, v_{|N|-1}$ if the mapping function F defined by $F(n_0) = v_0, F(n_1) = v_1, \ldots, F(n_{|N|-1}) = v_{|N|-1}$ produces a machine M that satisfies the transition and output requirements (2.4) and (2.3).*

There is a one-to-one correspondence between each variable x_i in the support of $\mathcal{F}$ and each node $n_i \in N$. Therefore, restrictions on valid mapping functions can be written as restrictions on the variables x_i. For instance, if two nodes in the specification, n_i and n_j, have to be mapped to different states, this is equivalent to the statement that $\mathcal{F}$ can only be true for points where $x_i \neq x_j$.

The transition and output requirements impose restrictions on the function $\mathcal{F}$. Let $t_i = (a_i, n_{s_i}, n_{d_i}, b_i) \in T$ and $t_j = (a_j, n_{s_j}, n_{d_j}, b_j) \in T$. For any two transitions that take place on the same input and have different outputs, output determinism requires that the source states of the transition should be assigned different states. For Mealy machines this can stated as

$$(a_i = a_j \wedge b_i \neq b_j) \Rightarrow x_{s_i} \neq x_{s_j}. \tag{3.3}$$

For Moore machines, different outputs imply different states

$$b_i \neq b_j \Rightarrow x_{s_i} \neq x_{s_j}. \tag{3.4}$$

Next-state determinism implies that, for any two transitions in the original machine that take place on the same input, the same assignment for the initial states implies the same assignment for the final states

$$(a_i = a_j \wedge x_{s_i} = x_{s_j}) \Rightarrow (x_{d_i} = x_{d_j}). \tag{3.5}$$

This can be rewritten as

$$(a_i = a_j) \Rightarrow (x_{s_i} \neq x_{s_j} \vee x_{d_i} = x_{d_j}). \tag{3.6}$$

For Mealy machines, (3.3) and (3.6) can be used to form $\mathcal{F}$ using the algorithm in Figure 3.8. For Moore machines, the lines that impose the output determinism restriction are changed to employ (3.4) instead of (3.3).

3.3.3 Performance Improvement Techniques

The above description of the implicit algorithm is deceptively simple, since all the complex manipulation of Boolean functions is performed by the MDD package. However, for complex problems, the storage requirements of the MDD package limit the usability of the algorithm.

The techniques described in the following sections can be used to reduce the storage and time requirements of the algorithm, extending its applicability to larger problems.

```
COMPUTEALLOWED(C)
  foreach n_i ∈ N
    A_i := 1
  foreach n_{c_i} ∈ C
    A_{c_i} := (x_{c_i} = q_i)          Nodes in the clique are assigned a unique state
  foreach n_j ∈ N, n_j ∉ C
    foreach n_{c_i} ∈ C
      if I(n_{c_i}, n_j) = 1            If a node is incompatible with a node in the clique
        A_j := A_j ∧ (x_j ≠ q_i)        it should be assigned a different value
```

Figure 3.9: Computation of the allowed mappings functions

Using the Incompatibility Graph

Although (3.3) and (3.5) contain enough to fully specify $\mathcal{F}$, the algorithm can be made more efficient by making use of the information contained in the incompatibility graph.

In particular, if $I(n_i, n_j) = 1$, then $F(n_i) \neq F(n_j)$. This implies that (3.3) and (3.4) can be replaced by:

$$(I(n_i, n_j) = 1) \Rightarrow x_i \neq x_j. \tag{3.7}$$

As described in Section 3.3.2, the resulting function $\mathcal{F}$ is 1 for all points in $Q^{|N|}$ that represent a valid mapping. In general, many mappings exist that satisfy the output and transition requirements. In particular, if a mapping $F : N \rightarrow Q$ exists, at least $|Q|!$ mappings exist (simply renumber the states in the final machine).

Since $\mathcal{F}$ implicitly keeps track of all these redundant mappings, it makes sense to preassign the mapping of some of the nodes. This can be done by observing that the nodes in a large clique in the incompatibility graph have to be assigned to different states, so assigning these to arbitrary (but different) states does not discard any simpler solution and makes $\mathcal{F}$ much simpler.

Once the mapping of these nodes has been performed, some mappings for other nodes can be removed from consideration. In particular, let $C = \{n_{c_0}, n_{c_1}, \ldots, n_{c_l}\}$ be a clique in the incompatibility graph. Then we can force $F(n_{c_0}) = q_0, F(n_{c_1}) = q_1, \ldots, F(n_{c_l}) = q_l$. Furthermore, if n_i is a node such that $I(n_i, n_{c_0}) = 1$, then we know that $F(n_i) \neq q_{c_0}$.

This information can be incorporated into the algorithm by defining a family of functions $\mathcal{A}_i : Q^{|N|} \rightarrow \{0, 1\}$ that describe the values allowed for each of the variables x_i. These functions can be computed by the procedure shown in Figure 3.9.

Selection of Variable Ordering

Two ordering problems need to be addressed in the algorithm. One is the order in which the nodes are processed. Experiments showed none was a significant improvement over the breadth-first ordering described in Section 3.2, so that is currently the default.

The ordering in which variables are stored internally in the MDD package is also important. The best results were obtained by sorting the nodes in the specification according to the degree of their nodes in the incompatibility graph. States that corresponded to nodes with higher degree in the incompatibility graph are earlier in the ordering. The intuitive justification for this is that states that are incompatible with a larger number of other states have fewer degrees of freedom and restrict the branching of the MDD used to represent $\mathcal{F}$.

Dynamic reordering algorithms were also tried. Although this technique reduced somewhat the memory requirements of the algorithm, it also unfavorably increased the running time. It is, therefore, not used in any of the experiments described in Chapter 4.

```
SIMULATE(n, G, R)
  foreach t_i := (a_i, n, n_{d_i}, b_i) ∈ T
    foreach t_j := (a_j, n_{s_j}, n_{d_j}, b_j) ∈ R        Transition already processed
      if a_i = a_j                                          Same input as a processed transition
        G := G ∧ (v ≠ x_{s_j} ∨ w = x_{d_j})                 Transition requirement must be satisfied
      if I(n, n_{d_j})                                      Use incompatibility information
        G := G ∧ (w ≠ x_{d_j})                              to limit the possible next state
    G := ∃v G
    G := [w → v]G                                           Make the new state the current state
    H := SIMULATE(n_{d_i}, G, R)                            Simulate the machine from the new destination state
    G := G ∧ H
  G := ∃v G
return G
```

Figure 3.10: Symbolic simulation of partially defined machines

Discarding Assignments Using Implicit Simulation

If the specification is sufficient to uniquely define one finite state machine of minimal size, the final size of the MDD required to represent $\mathcal{F}$ will be small, since $\mathcal{F}$ will be one for a small number of points in the input space. However, the storage requirements needed for intermediate steps may be high. These can be somewhat reduced if one performs implicit simulation of the machines defined by the assignments present in $\mathcal{F}$ and removes the inconsistent ones from consideration.

The implicit simulation can be performed by noting that even a partial definition of the mapping function F defines some transitions in the resulting machine M that can be inconsistent with input-output relations that have not been considered.

Let $R \subseteq T$ be the set of transitions taken into consideration so far, and let $\mathcal{F}$ be the function that represents all the mappings consistent with the information provided by the transitions in R.

We consider a function $\mathcal{G} : Q^{|N|+2}$, $\mathcal{G}(x_0, ...x_{|N|-1}, v, w)$ that will be used to implicitly keep the present and future state of all the machines contained in $\mathcal{F}$. Variable v will be used to to keep the present state in the implicit simulation, and variable w will be used to store the next state.

The code in Figure 3.10 performs the symbolic simulation and removes from consideration all machines inconsistent with the specification. Since the implicit simulation algorithm is time-consuming, it only makes sense to use it when memory requirements grow very large, not at every iteration of the main loop.

The Improved Algorithm

Using the techniques described in section 3.3.3, the main loop of the algorithm is shown in Figure 3.11. A large clique of the incompatibility graph is selected and the family of functions $\mathcal{A}_i$ is computed. The call to the implicit simulation algorithm only takes place if the memory requirements of the MDD package are exceeding some prespecified limit.

The implicit algorithm is implemented in C++ and makes use of the MDD package described by Kam and Brayton [KB90]. This package uses the CMU Boolean Decision Diagram package described by Brace [BRB89].

```
MAINLOOP()
    F := 1
    R := ∅                                          Stores the processed transitions
    C := LargeClique()
    COMPUTEALLOWED(C)
    foreach t_i := (a_i, n_{s_i}, n_{d_i}, b_i) ∈ T
        R := R ∪ t_i                                Add this transitions to the list
        F := F ∧ A_{s_i}
        foreach t_j := (a_j, n_{s_j}, n_{d_j}, b_j) ∈ R
            if I(n_{s_i}, n_{s_j})                  Output determinism restriction
                F := F ∧ (x_{s_i} ≠ x_{s_j})
            if a_i = a_j                            Next state determinism restriction
                F := F ∧ ((x_{s_i} ≠ x_{s_j}) ∨ (x_{d_i} = x_{d_j}))
        if Memory use too large
            F := F ∧ (w = x_0)                      Start all machines in the reset state
            F := SIMULATE(n_0, F, R)                Remove non-conforming machines
    return F
```

Figure 3.11: Optimized version of the implicit algorithm

Chapter 4

Experimental Results

4.1 Comparison with Existing Approaches

As described in Chapter 1, the problem can be viewed as the minimization of an incompletely-specified finite state machine. Algorithms for finite state minimization have been the subject of extensive studies in the past and several implementations of these algorithms are available. In this section, we compare the performance of the explicit algorithm, `mmm`, described in Section 3.2 and the implicit version, `iasmin`, described in Section 3.3 with two algorithms that solve the problem using the FSM reduction paradigm: `stamina` [HRSJ91] and `ism` [KVBSV94].

We first selected three target machines, shown in Figure 4.1. For each generating machine, we generated a number of specifications. Each specification consisted of one string. The specifications used had strings with lengths between 10 and 65 input/output pairs.

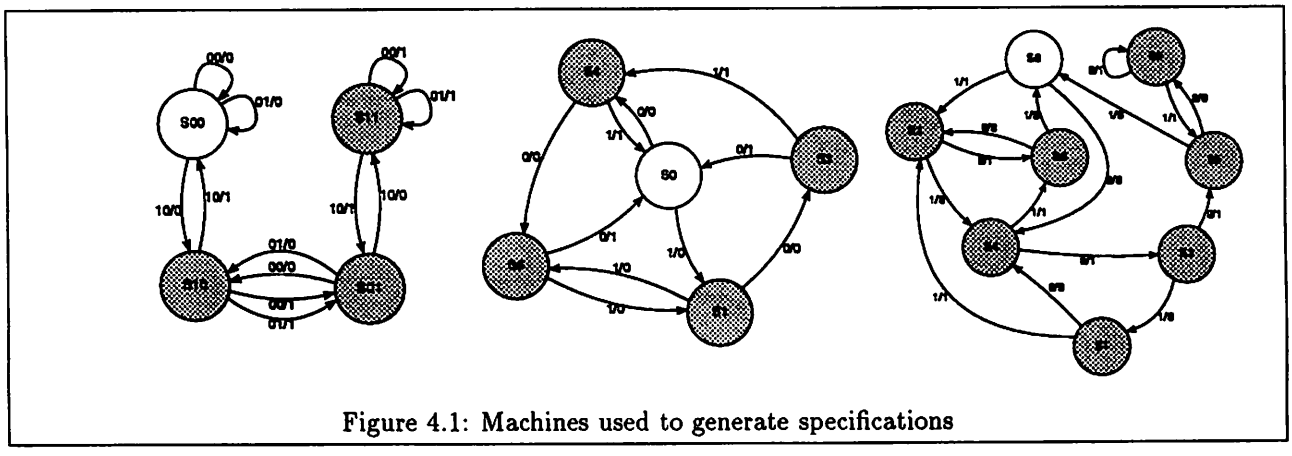


Figure 4.1: Machines used to generate specifications

For each length considered, five specifications were generated by simulating randomly generated inputs on the generating machine. The various programs were then used to find the minimum machine consistent with each of the specifications.

Figures 4.2, 4.3, and 4.4 show the times required. Each point represents the average over the five different specifications generated for each given length. These figures show that, in all cases, the state minimization algorithms require a time that increases exponentially in the length of the string. Our algorithms, `mmm` and `iasmin`, show a less drastic increase; the curves for these two algorithms tend to level off for longer strings. The different behavior observed illustrates well the distinct exponential dependences of the different approaches: traditional state minimization algorithms require time exponential in the size of the original specification, while the algorithms described in this report require time exponential in the size of the final machine.

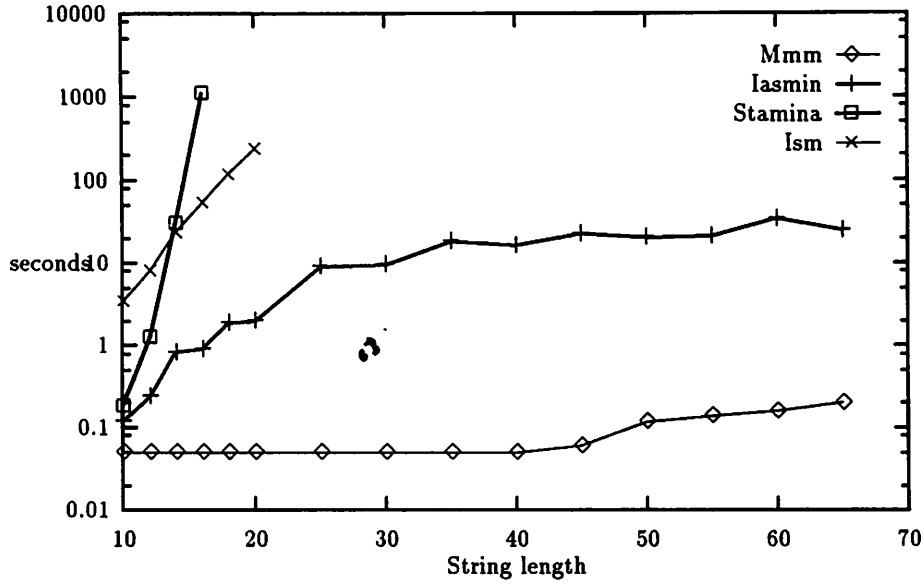


Figure 4.2: Run-time comparison for specifications generated with the first machine

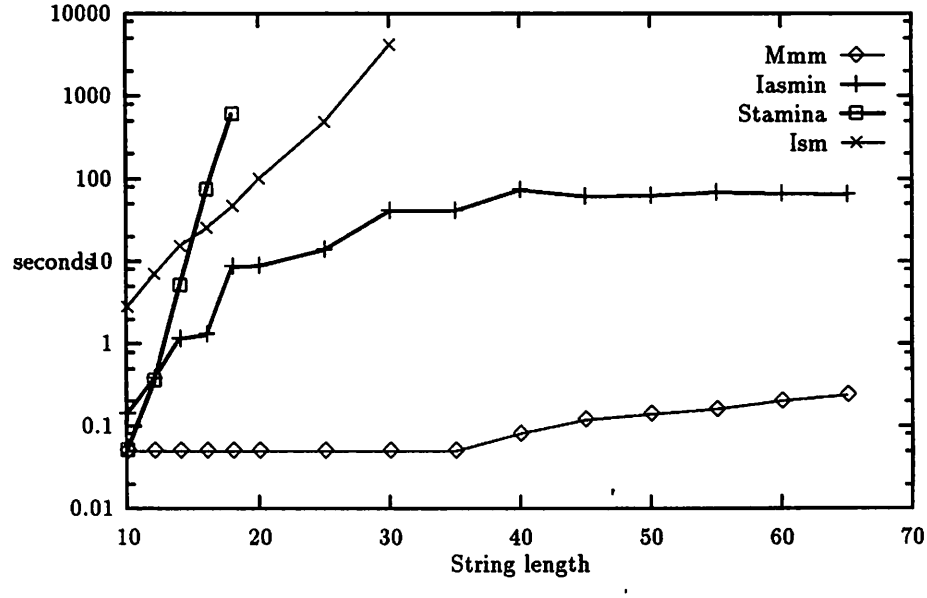


Figure 4.3: Run-time comparison for specifications generated with the second machine

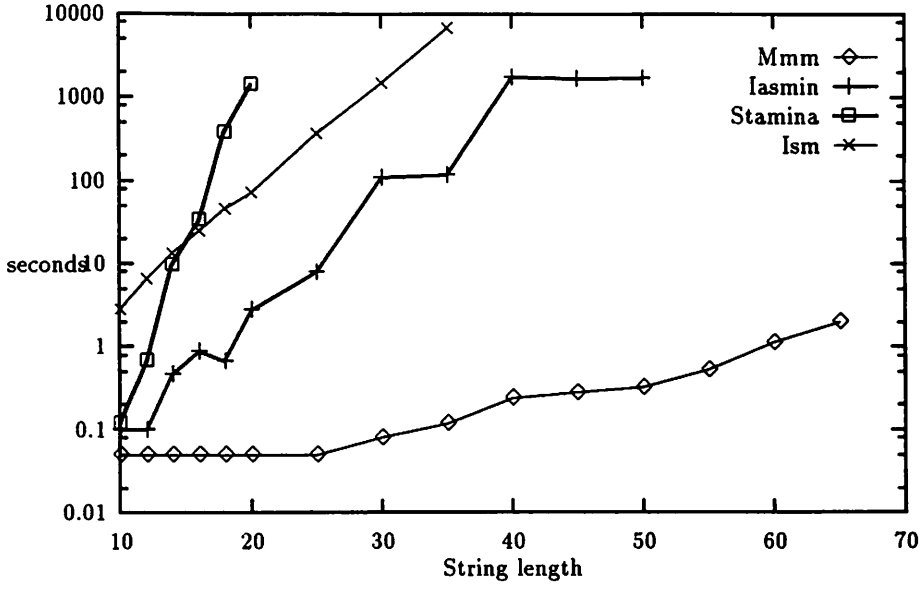


Figure 4.4: Run-time comparison for specifications generated with the third machine

4.2 Inferring Randomly-Generated Machines

4.2.1 Searching Time Versus Number of States

To examine the performance of the two algorithms on specifications with differently-sized minimum satisfying machines, 575 specifications[1] were generated (see Appendix A) from 115 randomly-generated state machines and presented to each algorithm. Each program was given an hour and 150 MB of memory to find the minimum satisfying machine.

Figure 4.5 shows the fraction of the specifications each algorithm was able to complete in the allotted time/space, plotted as a function of the number of states in the minimum satisfying machine[2]. We tried at least fifteen specifications for each number of states.

The explicit algorithm was successful more often than the implicit, although the implicit did not go from success to failure as quickly as the explicit. The explicit algorithm succeeded on 394 of the 575 specifications, and the implicit succeeded on 374.

Figure 4.6 shows the times taken, in seconds, for the two algorithms to find the minimum satisfying machine. The points plotted are only for those runs which were successful. Two conclusions can be drawn from this graph: the time appears to be growing exponentially with the number of states in the final machine, and the times required are widely distributed, with roughly a normal density in log scale.

4.2.2 Searching Time Versus String Length

Figures 4.7 and 4.8 show the searching time required to find the minimum satisfying machine for specifications with a string of varying length. Two 500-step strings were generated randomly from a 9-state generating machine. Prefixes of these strings of varying length were given to the explicit algorithm to produce the graphs. Although the second example requires thirty times more time than the first, the overall behaviors are very similar.

After 50 or so steps in the first example, the minimum satisfying machine suddenly changes (the 51st transition cannot be fitted into the machine) and causes a large increase in search time. Similar behavior

[1] Each specification contained twenty strings of thirty steps each. Each state machine had two inputs (0 and 1), and two outputs (0 and 1).

[2] This number is easily obtained when one of the algorithms complete. When neither did, we used the scheme described in Appendix A.

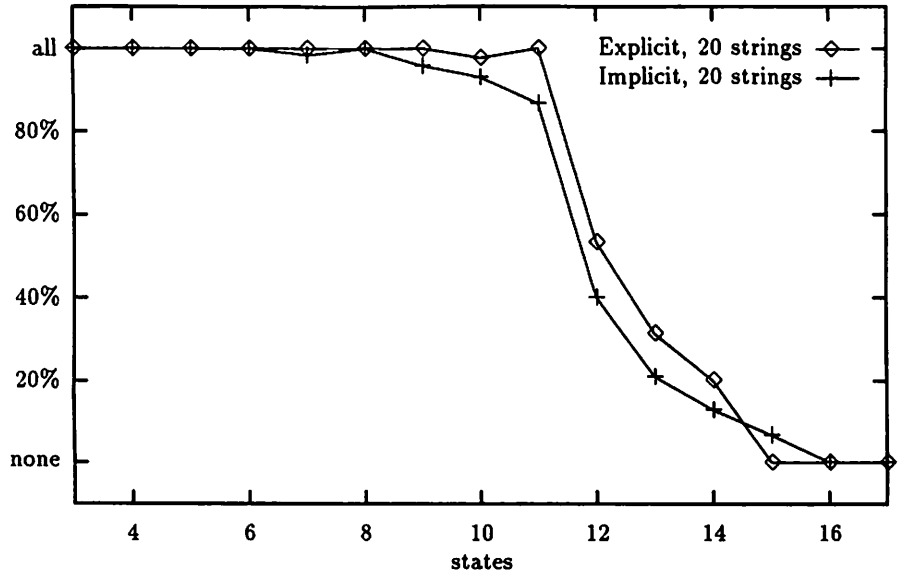


Figure 4.5: Fraction of runs completed

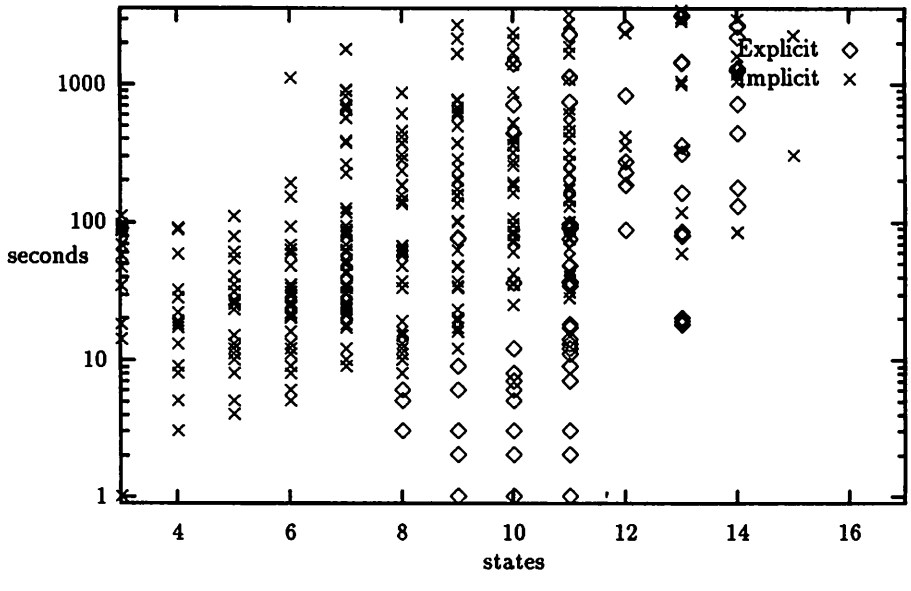


Figure 4.6: Searching times for successful runs

occurs in the second example, this time after about 100 steps.

The most important thing to note about these graphs, however, is that they are linear after a point. When the minimum satisfying machine has been found, the explicit algorithm decays into an efficient simulation. This behavior is distinctly different from that of more traditional state-minimizers.

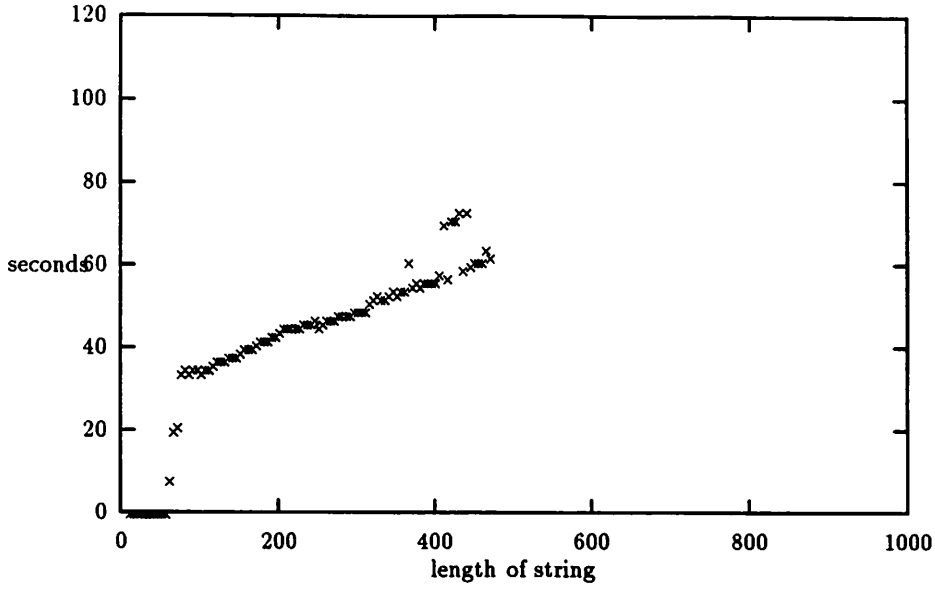


Figure 4.7: Searching times as a function of string length for the explicit algorithm.

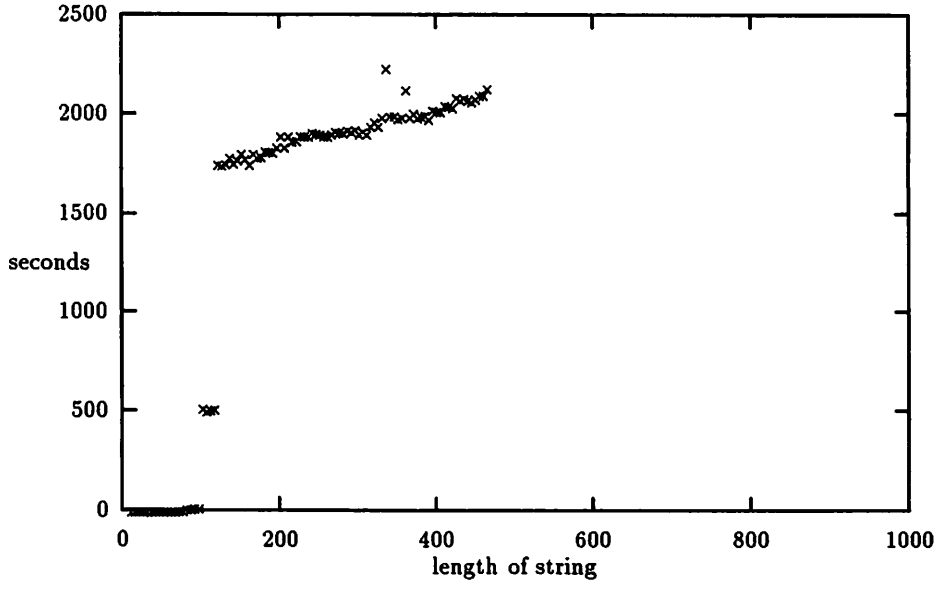


Figure 4.8: Searching times as a function of string length for the explicit algorithm.

4.2.3 Searching Time Versus Number of Strings

The graph in Figure 4.9 depicts the effect the number of strings has on the total run time for the explicit algorithm. For all data points, the product of steps and strings is constant (1000), so in some sense, the amount of information in the specification is constant. Each line represents specifications that are subsets of

a single specification, a single machine. It appears that above six strings, the problem can be solved much more rapidly.

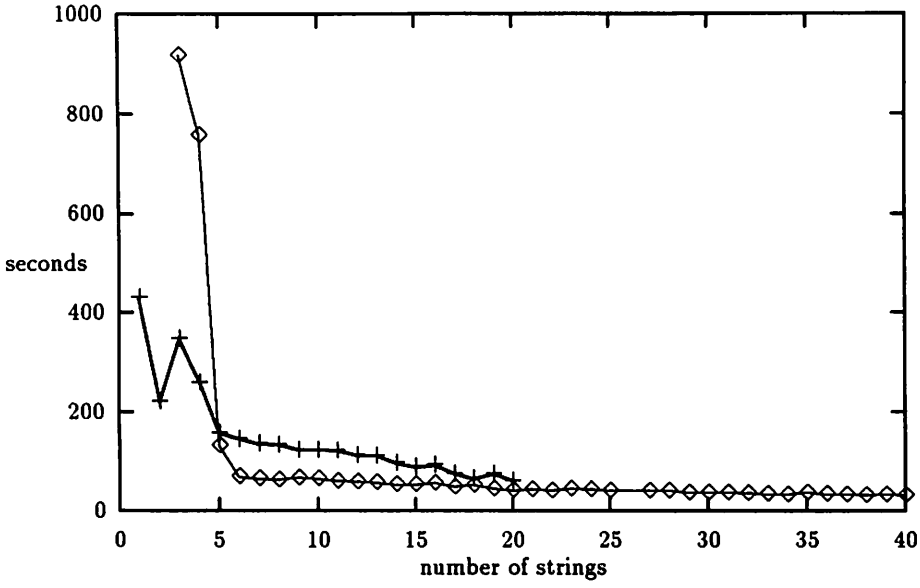


Figure 4.9: Total execution time as a function of the number of strings for the explicit algorithm.

4.3 Inferring Machines from Structured Domains

Although the experiments with random machines performed in the previous section give a clear idea of the potential and limitations of the algorithms studied, it is also interesting to evaluate their performance on more structured problems. In fact, problems from structured domains tend to be more regular and exhibit a higher level of symmetry, making them potentially harder to learn.

In this section, the target machines are the finite state machines that correspond to the following robot-worlds:

1. N-Rooms: The robot is in a circular house with N rooms. At each point in time, the robot has 3 possible actions (inputs to the finite state machine): toggle the light switch, move to the room on the right or move to the room on the left. The output is 1 if the light in the current room is on, 0 otherwise.

2. N×N Checkerboard: The robot is in an N×N checkerboard field that wraps around in a torus-like fashion. There are 4 possible actions: move left, move right, move up or move down. The output is related to the square the robot is on: the white squares have the same output and each black square has a distinct output.

3. N-Counters: The robot is in a circular house with N rooms. There are two possible actions: move to the next room on the right or stay in the current room. The output is one only in the room immediately to the left of the starting room.

Figure 4.10 shows an example of one machine from each of the families listed above.

For each problem in these three families, we performed five experiments. Each specification consisted of twenty strings, each of length thirty. Table 4.1 lists the number of successful runs. We used the same time and memory limits as in the previous section. Runs that failed to complete within the allotted time and memory requirements were considered failures.

These results seem to imply that inferring machines with a high level of symmetry may be more difficult than inferring randomly-generated machines. The data is, however, somewhat sparse and more experiments

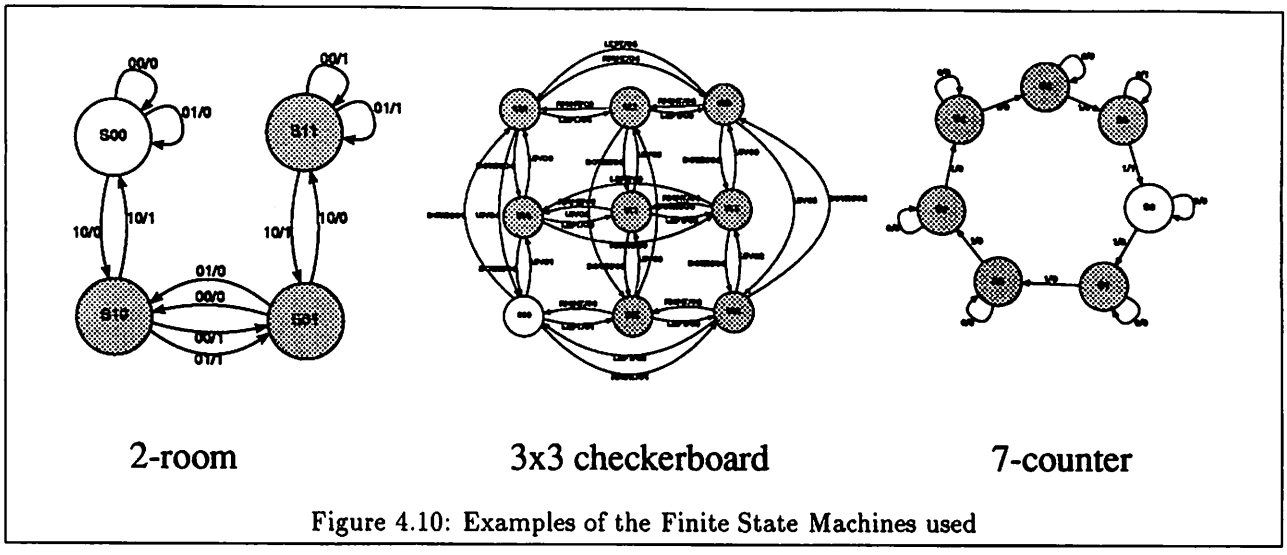


Figure 4.10: Examples of the Finite State Machines used

Problem	Iasmin	Mmm
2-room	5	5
3-room	1	3
4-room	0	0
2x2 board	5	5
3x3 board	5	5
4x4 board	0	0
4-counter	5	5
5-counter	5	5
6-counter	5	5
7-counter	5	5
8-counter	0	0

Table 4.1: Number of successful runs.

are required to establish a firm conclusion. In fact, some of these problems have multi-valued inputs or outputs, thereby making a direct comparison impossible.

The results for the 8-counter problem, a finite state machine with binary valued inputs and outputs, seem to show that these machines generate harder problems than the randomly-generated machines studied in the previous section. However, the increased difficulty may be related with other characteristics of the machine, such as the imbalance between the number of times the machine outputs a 1 versus the number of times the machine outputs a 0.

Chapter 5

Conclusions and Future Work

The experimental results described in the previous section illustrate the potentialities and limitations of an exact approach to the finite state machine inference problem.

On the plus side, it is true that the constructive strategy used by both the implicit and the explicit algorithms represents a big gain over strategies based on the reduction of incompletely specified finite state machines. In fact, while the run time for the approaches based on the reduction of FSMs has an exponential dependence on the size of the original specification, both the implicit and explicit approaches have a very weak (approximately linear) dependence on the size of the specification. They depend, however, exponentially on the size of the reduced machine, and this makes them inapplicable to problems where the minimal equivalent machine has more than about twelve states. These problems remain outside the reach of any exact algorithms published to date, so it appears that heuristic approaches are required for problems of larger sizes.

The search for effective heuristic algorithms is, therefore, one area for future research. This is especially true because little has been done on this to date.

The number of states in the final machine may not be an adequate objective for some problems. In some cases, it may be more reasonable to search for a system of communicating finite state machines that is minimal, according to some specific complexity measure. For example, a counter of a given length has a very regular structure but may exhibit a very large number of states. As of this writing, effective algorithm for the inference of minimal systems of communicating state machines are not known to the authors. This appears to be a promising area for future research.

The explicit approach has the advantage that it requires a very small amount of memory and is considerably faster for small problems because of the overhead incurred by the implicit algorithms. On the other hand, the CPU time used by the implicit algorithm seems to increase more gradually, which may make it better for larger problems. The main conclusion, however, is that neither of the approaches is clearly better than the other: their performance is comparable.

Since the implicit algorithm uses the MDD and BDD packages, its performance is strongly dependent on the ordering selected for the variables[1]. Further research on the use of different variable orderings may lead to significant savings in run-time and memory usage.

[1] This problem is fundamental to all BDD-based algorithms.

Appendix A

Generation of Example Specifications

A.1 Generating Random Machines

The following algorithm was used to generate small completely-specified machines. As many states as desired are created, one designated the reset state. Then for each state and each possible input, a random output is chosen uniformly from all possible outputs and a random next state is chosen uniformly from all states.

This does not guarantee that all states are reachable or that the machine is irreducible. However, for our purposes, it has proven sufficient. As discussed in the next section, the true number of states is only bounded by the number of states chosen originally.

A.2 Generating Specifications

To test the algorithms, randomly-generated specifications were created by simulating a known, small state machine. For each string in the specification, the machine is placed in the reset state, a randomly-chosen input is applied, and the output recorded. Another input is applied, the output recorded, and so on.

The number of states in the minimum specification is clearly bounded above by the number of states in the generating machine. However, the number of states in the minimum specification can be smaller when, for example, not every state in the generating machine is visited or not every transition is taken during the simulation. Another possibility is that the generating machine is itself non-minimal.

To obtain a tighter bound on the number of states in the minimal satisfying machine for a given specification, all states and transitions not visited are discarded. The resulting machine is sent through the traditional state minimizer `stamina` [HRSJ91] and the number of states in this minimized machine is used as an estimate. In two of 394 specifications generated this way, this bound was off by one (i.e., the minimum satisfying machine had one fewer states).

Bibliography

[Ang78] D. Angluin. On the complexity of minimum inference of regular sets. *Inform. Control*, 39(3):337–350, 1978.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, November 1987.

[BK76] A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Trans. on Software Engineering*, SE-2:141–153, 1976.

[BP75] A. W. B. R. I. Biermann and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. on Computers*, C-24:122–136, 1975.

[BRB89] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 26th Design Automation Conference*, June 1989.

[CP90] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, November 1990.

[DM93] S. Das and M. Mozer. A unified gradient-descent/clustering algorithm architecture for finite state machine induction. In *Advances in Neural Information Processing Systems 6*, Denver, CO, 1993. Morgan Kaufmann.

[GH66] James N. Gray and Michael A. Harrison. The theory of sequential relations. *Information and Control*, 9, 1966.

[GL65] 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.

[GMC+92] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4:393–405, 1992.

[Gol72] E. M. Gold. System identification via state characterization. *Automatica*, 8:621–636, 1972.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Inform. Control*, 37:302–320, 1978.

[HRSJ91] G. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *Proceedings of the European Design Automation Conference*, 1991.

[KB90] T. Kam and R.K. Brayton. Multi-valued decision diagrams. *Tech. Report No. UCB/ERL M90/125*, December 1990.

[KVBSV94] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. Proceedings of the Design Automation Conference, 1994.

[Lan92] 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.

[PF88] S. Porat and J. A. Feldman. Learning automata from ordered examples. In *Proc. 1st Annu. Workshop on Comput. Learning Theory*, pages 386–396, San Mateo, CA, 1988. Morgan Kaufmann.

[Pfl73] C. F. Pfleeger. State reduction in incompletely specified finite state machines. *IEEE Trans. Computers*, C-22:1099–1102, 1973.

[Pol91] Jordan B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:123–148, 1991.

[PU59] M. Paull and S. Unger. Minimizing the number of states in incompletely specified state machines. *IRE Transactions on Electronic Computers*, September 1959.

[Sch92] R. E. Schapire. *The Design and Analysis of Efficient Learning Algorithms.* MIT Press, Cambridge, MA, 1992.

[TB73] B. A. Trakhtenbrot and Y. M. Barzdin. *Finite Automata.* North-Holland, Amsterdam, 1973.

[Tom82] M. Tomita. Dynamic construction of finite-state automata from examples using hill-climbing. In *Proc. Fourth Annual Cognitive Science Conference*, page 105, 1982.