# Matrix-Based Algorithms and an Analysis of System Structure for Partially-Observable Discrete-Event Systems

by

Adrian Victor Payne

A thesis submitted to the
Department of Electrical and Computer Engineering
in partial fulfillment of the requirements for
the degree of Master of Applied Science

Queen's University
Kingston, Ontario, Canada
August 1997

## Abstract

The complexity issues associated with the finite state machine (FSM) framework for analyzing partially-observable discrete-event system (DES) control problems are reviewed. Methods that take advantage of system structure to provide better estimates of the size of the state space of FSMs which recognize the projection of partially-observable systems are presented. The computational advantages or disadvantages of each method are discussed. The applicability and effectiveness of these methods are illustrated using a number of simple, yet illustrative, examples. The example problems illustrate cases where our methods are both effective and ineffective in improving upon the standard complexity results.

The existing set of DES software tools is reviewed, and used to form a basis for the development of a new more flexible and intuitive DES environment which may be used to design, analyze and solve DES problems. The design of this tool is such that it can be implemented in a reasonably simple manner using common proven computational tools, and graphical user interface (GUI) building tools.

In conjunction with the development of a new DES software tool, matrix-based data structures and DES operations are presented and developed for a selection of common DES functions. This approach is designed to take advantage of high-level matrix operations available in a number of commercial off-the-shelf (COTS) software applications, and to take advantage of sparse-matrix data structures which allow DES information to be stored and processed in an efficient manner. Finally, the set of matrix–based DES operations is designed so that it is straightforward to write high-level scripts which perform more complex DES analysis tasks.

## Acknowledgements

I would like extend a special thank you to my thesis supervisor Dr. Karen Rudie for her guidance, encouragement, and friendship throughout the past two years, without whom this work would not have been possible.

Great appreciation goes to the School of Graduate Studies at Queen's University for its financial support through the Queen's Graduate Awards program, and to Dr. K. Rudie for the financial support she provided through her NSERC operating grant.

I would like to thank the members of my examining committee, specifically Dr. N. Manjikian, Dr. R. Dawes, Dr. S. Tavares, and Dr. E. W. Grandmaison for their very helpful comments and suggestions, regarding the content of this thesis.

I would also like to thank my family, especially my mother, Matthew and Annette, and friends such as Moira, Justin, Lucy, and Michelle who helped me maintain a positive outlook on life, and who were a constant source of moral support.

I thank my fellow students in the DES group, particularly Laurie, Gino, and Kari, for sharing some of their ideas with me and generally making the DES group an interesting place to work and learn.

Finally I want to extend a special thank you to my good friend Wade who, through endless movies, pool nights and Wellington outings, ensured that I kept a proper balance in my life.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The Discrete-Event System (DES) framework can be used to model an increasing number of engineering problems arising in industry today. Applications including flexible manufacturing systems [LMMB88], communications protocols [RW92a], failure diagnosis systems [SSL+95], [SSL+96], task scheduling, and database management [Laf88] illustrate why a system that focuses on the discrete nature of these applications is in increasing demand. While a body of research pioneered by [RW82] and discussed in [RW89], [CLO95], [Thi96] has provided a basis for this type of system modeling and control, a number of issues remain that prevent the broad acceptance of this research in industry applications.

One of the key issues in the field of DES is the problem of state-space explosion, which occurs when modeling large (typical) systems. A number of methods have been developed that attempt to address the problem of computational complexity in large and/or partially-observable systems [BH93], [CLL92], [HL94], [LW93], [LW94], [OW90]. In general, state-space explosion may occur while modeling the problem or while devising a suitable controller. In various instances, the number of states of the system that a controller must keep track of becomes intractable, making the control problem difficult to solve in a reasonable amount of time, or using a reasonable amount of computer resources.

Our work represents an attempt to understand the reasons for state-space explosion, notably in the worst case scenario where a controlling agent is only aware of a subset of the events occurring within a system—and thus must keep track of all possible states of the system. We examine the structure of the modeled system, and present reasons why, when developing the model, state-space explosion occurs or does not occur. We then take advantage of this knowledge to devise a series of tests that are performed while modeling the system to estimate the extent of the state-space explosion. The results of these tests identify how the system model can be modified to minimize the problem. This would correspond to recommending that additional sensors be installed in the system, or that various events be prevented from occurring or eliminated entirely from the system. In devising these tests, it is necessary to take care not to introduce computationally expensive procedures into our test algorithms, thereby possibly invalidating the usefulness of the test.

Computational complexity issues are generally associated with the development of efficient algorithms that can then be implemented as software tools. Therefore, as part of this thesis, we develop matrix-based data structures and algorithms which can be used to model and analyze DES problems. By coupling complexity analysis with these data structures and algorithms, we provide an efficient and flexible environment for building and analyzing partially-observable discrete-event systems.

## 1.1 Partial Observability and DES

The work contained in this thesis focuses specifically on cases where only a subset of the events in a modeled system are observed by controlling agents [LW88], [CDFV88], [Tsi89], [RW90], [RW92b], [RW95]. Since in cases such as these, the controlling agent needs to keep track of all the possible states the system may be in at any one time, the resulting supervisor may need to store control information for all possible combinations of states of the system. This results in a supervisor which could require up to $2^n$ amount of time to generate, and which could take up to $2^n$ amount of storage space, where $n$ is a measure of the size of the partially-observable system. The exponential growth of the supervising agent is computationally intractable when working with problems where $n$ is large.

Partially-observable systems are common to DES problems, specifically in the areas of decentralized control and fault diagnosis. The structures created when solving these types of problems can exhibit the exponential growth described above. It has been noted [SSL$^+$95], [SSL$^+$96], [OW90] that this type of exponential growth is seldom observed.

## 1.2 Tools for Computing and Displaying Discrete Event Systems

Throughout the process of creating and analyzing example problems, we noted that existing DES software tools were of limited use in a number of areas. In general, there existed no simple method for processing large numbers of problems; no method for writing high level scripts for solving specific types of DES problems that require the use of a number of different basic DES operations; no method for easily expanding the functionality of existing tools by adding new DES functions as required; and finally no method for displaying plants or controllers (finite-state machines) in a simple and understandable format.

In this thesis, we provide some background on a number of existing DES tools for reference, and then proceed to propose a DES software design based on prototype software that we believe satisfies a number of our requirements. We also develop in detail some matrix implementations of a subset of DES operations—specifically those which relate to modeling partially-observable systems.

## 1.3 Research Contributions

The following list summarizes the research contributions of the material presented in this thesis.

- Matrix-Based Tools: Matrix-based data structures, together with a set of matrix-based implementations of existing DES operations have been provided. These data structures and operations have been implemented using MATLAB [Mat92] software.

- Front-End Graphical User Interface (GUI) Requirements: A set of requirements has been developed and a prototype GUI has been implemented using Tcl/Tk [Ous94] software.

- Complexity Analysis for Partially-Observable DES Problems: A set of DES structural properties that tighten the complexity bounds for solutions to partially-observable problems has been identified.

## 1.4 Thesis Outline

- Chapter 2 provides a brief review of the areas of research which form the basis for the results presented in this thesis. The Discrete-Event Systems (DES) framework first introduced in [RW82] is presented, followed by some selected background topics in the fields of Computational Complexity, Automata Theory and Graph Theory.

- Chapter 3 describes the specific problem of state-space explosion when considering partially-observable DESs, and presents methods of analysis which take advantage of the structure of the system to make worst-case estimates about the resulting state-space explosion. A detailed analysis of the sensitivity of DESs to various structural properties is presented. Finally, methods for identifying and modifying problem structures within a system to minimize the resulting state-space explosion are presented.

- Chapter 4 provides a review of existing DES software tools, proposes some high–level architectural requirements for a new set of tools, and presents algorithms used to implement the structure-based analysis presented in Chapter 3, including a DES MATLAB toolbox with specific procedures for testing and analyzing partially-observable systems.

- Chapter 5 illustrates how typical DES problems may be analyzed and, in some cases, modified for control based on partial observation using the tools presented in this thesis.

- Chapter 6 provides conclusions about the work presented in this thesis.

# Chapter 2

# A Review of DES and Automata Theory

## 2.1 Automata Theory

In general, the work done in DES does not require that the system be modeled using any single methodology. Typically, however, much of the work done in the field to date borrows models from the body of work in computer science on automata theory.

### 2.1.1 Deterministic Finite-State Automata

A *Deterministic Finite-State Automaton* (DFA) is formally denoted by the 5-tuple $(Q, \Sigma, \delta, q_o, Q_m)$, where $Q$ is a finite set of states, $\Sigma$ is a finite input alphabet, $\delta$ is a partial transition function mapping $Q \times \Sigma$ to $Q$, $q_o$ is an initial state, and $Q_m$ is a set of terminal states. In DES theory, terminal states are often called *marked* or *marker* states. Figure 2.1 shows a simple example of a DFA[1] where an initial state is indicated by a left-pointing arrow ($\leftarrow$) in the state box, and marked states are indicated by right-pointing arrows ($\rightarrow$) in the state box. If the initial state is also a marked state, then a double-headed arrow ($\leftrightarrow$) is used in place of the left-pointing arrow.

[1]All the Finite-State Machines (FSMs) with labeled states are generated using the prototype DES software toolkit presented in this thesis.

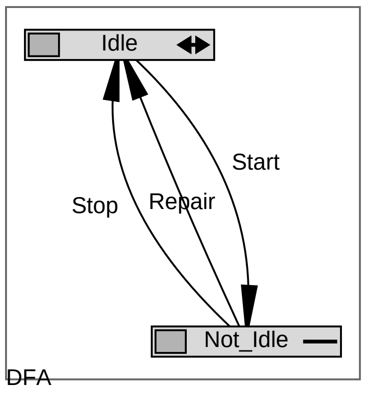


**Figure 2.1:** A Simple Deterministic Finite-State Automaton

### 2.1.2 Nondeterministic Finite-State Automata

A *Nondeterministic Finite-State Automaton* (NFA) is formally denoted by the 5-tuple $(Q, \Sigma, \delta, q_o, Q_m)$, where $Q$, $\Sigma$, $q_o$, and $Q_m$ have the same meaning as for a DFA, and where $\delta$ maps $Q \times \Sigma$ to $2^Q$. Whereas the transition function in a DFA maps $Q \times \Sigma$ to single elements in $Q$ (e.g., $\delta(q_1, \sigma) = q_2$), the transition function for an NFA maps $Q \times \Sigma$ to subsets of $Q$ (e.g., $\delta(q_1, \sigma) = \{q_2, q_4, q_5\}$). It follows that a DFA is a special case of an NFA, where $\delta$ maps $Q \times \Sigma$ to single-element subsets of $Q$. Figure 2.2 shows a simple example of an NFA. Note that the transition $\delta(\text{Idle}, \text{Start}) = \{\text{Working}, \text{Broken}\}$ is the source of the nondeterminism in the automaton.

### 2.1.3 Nondeterministic Automata with $\varepsilon$-Transitions

Nondeterministic Automata with *$\varepsilon$-transitions* (NFA$\varepsilon$) are automata defined in the same manner as NFAs, with the additional property that the automaton may make a transition on the empty input $\varepsilon$. Refer to Figure 2.3 for an example of a simple nondeterministic automaton with $\varepsilon$-transitions.

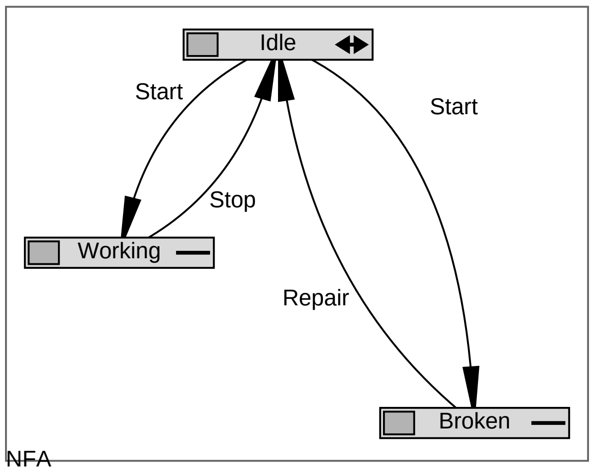


**Figure 2.2:** A simple nondeterministic finite-state automaton

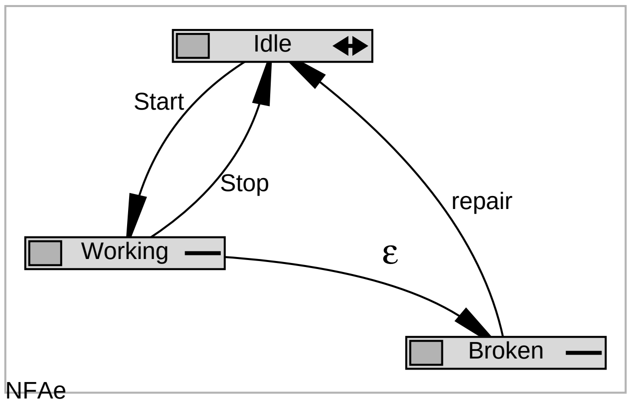


**Figure 2.3:** A simple NFA with an $\varepsilon$-transition

### 2.1.4 Minimum-State DFA

Theorem 2.1 [HU79] (see below) together with Algorithm 2.1 (also from [HU79]) provide a polynomial-time method for constructing an output DFA which is a minimum-state recognizer for the language recognized by an input DFA. It should be noted that no such algorithm exists for NFAs. Indeed, it can be proven [JR93] that the decision problem associated with the conversion of a DFA to a minimum-state NFA is PSPACE-complete [2].

**Theorem 2.1 [HU79]** *The DFA constructed using Algorithm 2.1, with inaccessible states removed (trim), is the minimum state DFA for its language.*

**Algorithm 2.1 : Minimum-State DFA Construction**

```
1. for $q_m \in Q_m$ and $q \in Q - Q_m$ do flag$(q_m, q)$;

2. for each unordered pair of states $(q_i, q_j), i \neq j$
       define an empty list $L_{(q_i,q_j)}$
   end

3. for each pair of distinct states $(q_i, q_j)$
   in $Q_m \times Q_m$ or $(Q - Q_m) \times (Q - Q_m)$ do
       if for some input symbol $\sigma$, $(\delta(q_i,\sigma), \delta(q_j,\sigma))$ is flagged then
           RecursiveFlag$(q_i, q_j)$ (Algorithm 2.2)
       else
           for all input symbols $\sigma$ do
               if $\delta(q_i,\sigma) \neq \delta(q_j,\sigma)$
                   put $(q_i, q_j)$ on $L_{(\delta(q_i,\sigma),\delta(q_j,\sigma))}$
               end
           end
       end
   end
```

[2]Section 2.3.1 provides some discussion on PSPACE-complete problems

**Algorithm 2.2 : The RecursiveFlag Function**

```
1. input unordered pair $(q_i, q_j)$

2. flag $(q_i, q_j)$

3. for each unordered pair $(q_m, q_n)$ in the list $L_{(q_i,q_j)}$
      if the unordered pair $(q_m, q_n)$ is not flagged then
         RecursiveFlag$(q_m, q_n)$
      end
   end
```

## 2.2 Discrete-Event Systems

A Discrete-Event System model can be thought of as a representation of a real system which exhibits asynchronous, event-driven behaviour. Typically such a system can be described using a state-transition structure. Abstractly, this model can be represented by a five-tuple deterministic automaton (DFA)

$$G = (Q, \Sigma, \delta, q_o, Q_m)$$

where $Q$ is a set of states,
$\Sigma$ is a set of event labels,
$\delta : Q \times \Sigma \rightarrow Q$, is a partial function defined for some states $q \in Q$,,
and for some events $\sigma \in \Sigma$, such that $\delta(\sigma, q) = q'$ where $q' \in Q$,
$q_o$ is the initial state,
and $Q_m \subseteq Q$ is the set of marked states.

Let $\Sigma^*$ denote the set of all strings over $\Sigma \cup \{\varepsilon\}$. We extend the definition of $\delta$ in the usual manner, as follows:

$$\delta^* : Q \times \Sigma^* \rightarrow Q, \text{ where}$$

$$\begin{aligned}
\delta^*(q,\varepsilon) &= q \text{ for } q \in Q,\\
\delta^*(q,\sigma) &= \delta(q,\sigma) \text{ for } q \in Q, \sigma \in \Sigma,\\
\delta^*(q,s\sigma) &= \delta(\delta^*(q,s),\sigma) \text{ for } q \in Q, \sigma \in \Sigma, s \in \Sigma^*.
\end{aligned}$$

For simplicity, we use $\delta$ to represent both $\delta$ and $\delta^*$, recognizing that when $\delta$ operates on a state and string (of length greater than 1), then we are implicitly using $\delta^*$.

Given $\Sigma^*$ and $\delta$ as defined above, the languages generated by $G$ and marked by $G$ (denoted by $L(G)$ and $L_m(G)$, respectively) are

$$L(G) := \{s \in \Sigma^* \mid \delta(q_o, s) \text{ is defined } \},$$

$$L_m(G) := \{s \in \Sigma^* \mid \delta(q_o, s) \in Q_m\}.$$

An example of a typical finite-state plant is shown in Figure 2.4. For this example, the event set is $\Sigma = \{$start_job, finish_job, repair, break_down$\}$, the labels for the state set $Q$ are $\{$Idle, Working, Broken$\}$, the initial state is $q_o =$ Idle and the marked state set is $Q_m = \{$Idle$\}$. The partial transition function is defined for this example as

$$\begin{aligned}
\delta(\text{Idle,start\_job}) &= \text{Working},\\
\delta(\text{Working,finish\_job}) &= \text{Idle},\\
\delta(\text{Working,break-down}) &= \text{Broken},\\
\delta(\text{Broken,repair}) &= \text{Idle}.
\end{aligned}$$

The event set of a plant $G$ can be partitioned for the purposes of supervisor design into two disjoint subsets, the first set $\Sigma_c$ composed of all "controllable" events, and the second set $\Sigma_{uc}$ composed of all "uncontrollable" events. Note that $\Sigma = \Sigma_c \cup \Sigma_{uc}$. Controllable events are those events which a supervising agent ($S$ in Figure 2.5) may enable or disable in accordance with some control strategy. Uncontrollable events are considered to always be enabled. Enabled events are those events which may occur in the plant, whereas disabled events are those events which are prevented from occurring.

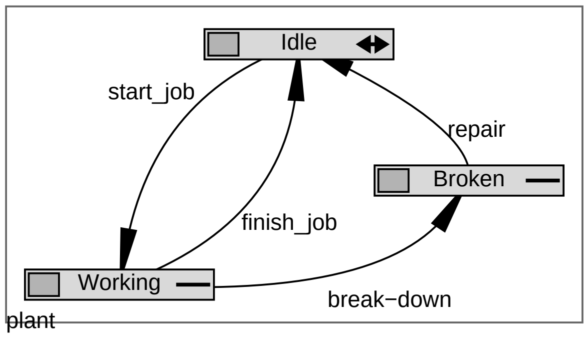


**Figure 2.4:** A simple plant

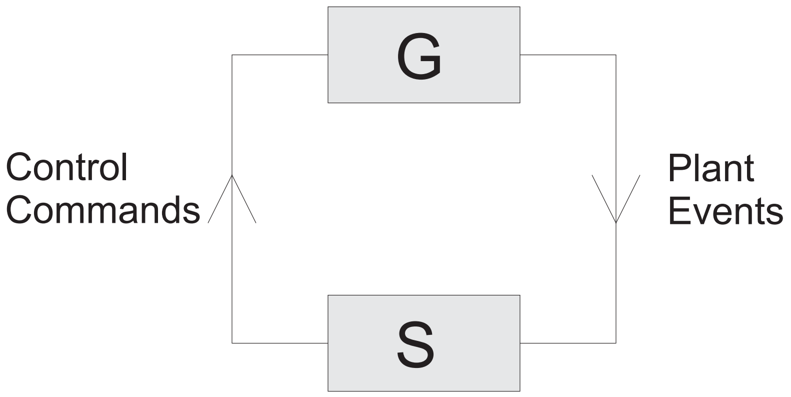


**Figure 2.5:** A plant/supervisor system

A (nonempty) plant $G$ may be "controlled" by a supervising agent $S$, where the supervising agent sends control commands to $G$ which serve to enable or disable events based upon the observed sequences of events occurring in the plant (Figure 2.5). These actions by the supervising agent limit the general behaviour of $G$ to some specified legal or desired behaviour in a closed-loop system $S/G$. This is equivalent to saying that $S$ restricts the language $L(G)$ to some legal sublanguage $L(S/G)$.

Now, suppose that the legal behaviour is represented by the language $K \subseteq \Sigma^*$. Before a supervisor may be constructed, it is necessary to determine if it is possible to restrict $L(G)$ to $K$. The language $K$ is said to be *controllable* with respect to $G$ if and only if

$$\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K}$$

where $\overline{K}$ represents the prefix-closure of $K$ (i.e., the language composed of all prefixes of $K$), and the notation $\overline{K}\Sigma_{uc}$ stands for the set $\{k\sigma \mid k \in \overline{K}, \sigma \in \Sigma_{uc}\}$. Thus, controllability states that given any prefix of $K$, there is no uncontrollable event which when appended to the prefix of $K$, generates a string which is contained in $L(G)$, but which is not contained in $\overline{K}$.

We now define the supervising agent model as the pair $S = (T, \psi)$. The supervisor $S$ is represented by an automaton

$$T = (X, \Sigma, \xi, q_o, X_m),$$

and a control mapping $\psi : \Sigma \times X \rightarrow \{\text{enable}, \text{disable}\}$. The automaton $T$ accepts as input the sequences of symbols in $\Sigma$ generated by $G$, and then generates control commands based on the control mapping $\psi$, which enables or disables controllable events in $G$.

### 2.2.1 DES Building Blocks

The following sections review a set of operations which are used to construct and manipulate DES models [Won96]. The DES plants shown in Figure 2.6 will be used in examples of how the meet and synchronous product operations work.

#### 2.2.1.1 Trim

A *trim* automaton is defined to be an automaton with all states being "reachable" and "coreachable". For a state $q$ to be *reachable*, there must exist a path (possibly of length zero) from the initial state $q_o$ to the state $q$. For a state $q$ to be *coreachable*, there must exist a path (possibly of length zero) from $q$ to a marked state $q_m \in Q_m$.

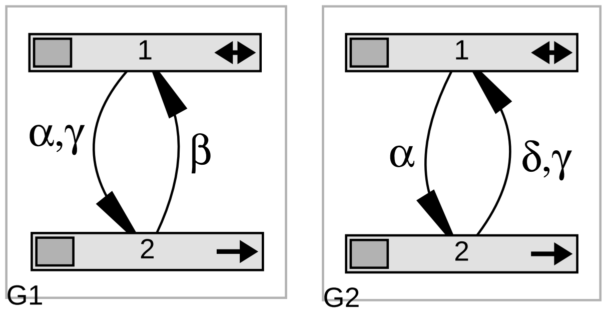


**Figure 2.6:** Plants $G_1$ and $G_2$

#### 2.2.1.2 Meet

The *meet* of $n$ languages $L_1, L_2, \ldots, L_n$ is defined to be

$$L_{meet} = L_1 \cap L_2 \cap \ldots \cap L_n.$$

This definition can be used to construct a generator $G_{meet}$ which generates the language $L_{meet}$ based on generators $G_i = (Q_i, \Sigma_i, \delta_i, q_{o_i}, Q_{m_i})$ for $i = 1$ to $n$:

$$\begin{array}{lcl} Q_{meet} & = & \{(q_1, q_2, \ldots, q_n) \mid q_i \in Q_i, i = 1, \ldots, n\}, \\ \Sigma_{meet} & = & \bigcap_{i=1}^{n} \Sigma_i, \\ \delta_{meet} & = & \{((q_1, q_2, \ldots, q_n), \sigma, (p_1, p_2, \ldots, p_n)) \mid \\ & & p_i, q_i \in Q_i, i = 1, \ldots, n \ \bigwedge_{i=1}^{n} \delta_i(q_i, \sigma) = p_i\}, \\ q_{o_{meet}} & = & \{(q_{o_1}, q_{o_2}, \ldots, q_{o_n})\}, \\ Q_{m_{meet}} & = & \{(q_{m_1}, q_{m_2}, \ldots, q_{m_n}) \mid q_{m_i} \in Q_{m_i}, i = 1, \ldots, n\}. \end{array} \tag{2.1}$$

Thus, the automaton $G_{meet}$ generates (resp., recognizes) only those strings which can be generated (resp., recognized) by all the $G_i$ automata. In many applications, it is desirable that $G_{meet}$ be a trim automaton. In this case, $G_{meet}$ only recognizes those strings which are recognized by all the $G_i$ automata. No similar claim can be made for generated strings.

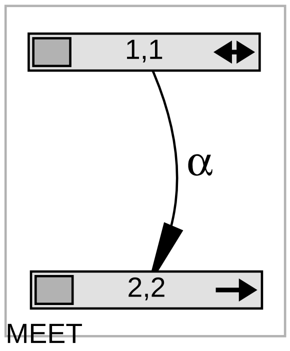


**Figure 2.7:** The meet of $G_1$ and $G_2$

An example of the meet of the plants $G_1$ and $G_2$ in Figure 2.6 is shown in Figure 2.7. Note that for this example, in plant $G_1$, $\alpha$ and $\gamma$ are distinct events which take the system from state 1 to state 2. For simplicity, we only show one arrow for these two transitions. A similar simplification is made in the plant $G_2$.

#### 2.2.1.3 Synchronous Product

Whereas the meet of a group of languages captures only those strings which are contained in all the languages, the *synchronous product* contains all possible interleavings of strings in the group of languages. A generator $G_{sync}$ can be constructed which generates the language $L_{sync}$ based on the generators $G_i = (Q_i, \Sigma_i, \delta_i, q_{o_i}, Q_{m_i})$ for $i =$ 1 to $n$ as follows:

$$\begin{aligned}
Q_{sync} &= \{(q_1, q_2, \ldots, q_n) \mid q_i \in Q_i, i = 1, \ldots, n\}, \\
\Sigma_{sync} &= \textstyle\bigcup_{i=1}^{n} \Sigma_i, \\
\delta_{sync} &= \{((q_1, q_2, \ldots, q_n), \sigma, (p_1, p_2, \ldots, p_n)) \mid \\
&\quad\; p_i = \delta_i(q_i, \sigma) \text{ if } \sigma \in \Sigma_i, \\
&\quad\; p_i = q_i \text{ if } \sigma \notin \Sigma_i\}, \\
q_{o_{sync}} &= \{(q_{o_1}, q_{o_2}, \ldots, q_{o_n})\}, \\
Q_{m_{sync}} &= \{(q_{m_1}, q_{m_2}, \ldots, q_{m_n}) \mid q_{m_i} \in Q_{m_i}\}.
\end{aligned} \tag{2.2}$$

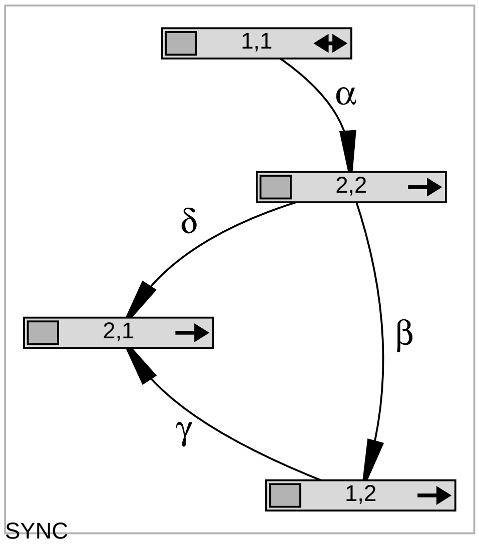


**Figure 2.8:** The synchronous product of $G_1$ and $G_2$

As with meet, in general, it is desirable to express the result of the synchronous product operation as a trim automaton.

An example of the synchronous product of the plants $G_1$ and $G_2$ in Figure 2.6 is shown in Figure 2.8.

### 2.2.2 Supervisory Control with Partial Observation

In many applications, full knowledge about all the events occurring in a plant $G$ is not available to the supervising agent $S$. In these cases, it is useful to partition the event set into two disjoint subsets, $\Sigma_o$ representing the set of observable events, and $\Sigma_{uo}$ representing the set of unobservable events. Note that $\Sigma = \Sigma_o \cup \Sigma_{uo}$, and also that there is no particular relationship between $\Sigma_o$ and the set of controllable events $\Sigma_c$. Practically, such a system would correspond to a plant where there exists an array of sensors that are capable of detecting a subset of all the possible events that may occur. In such cases, it may not be economically or practically feasible

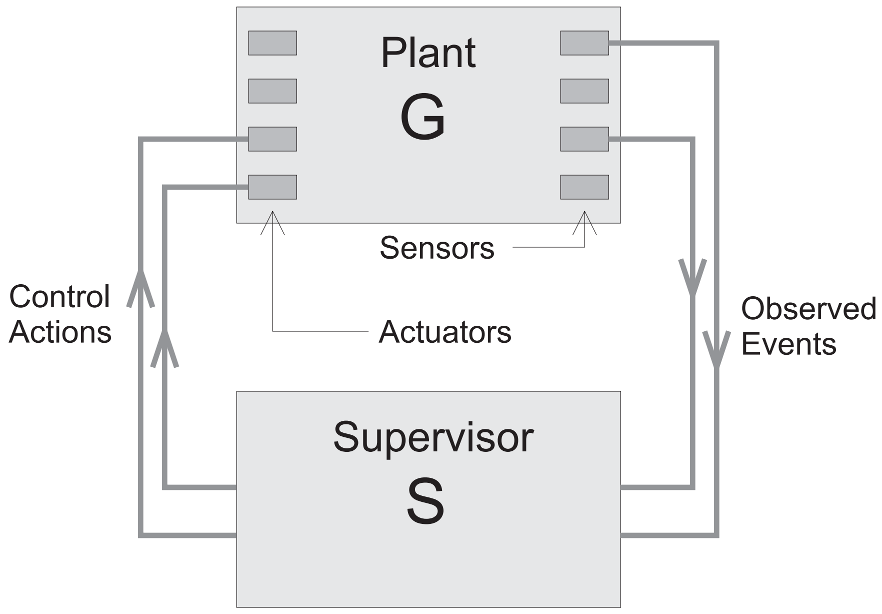


**Figure 2.9:** A plant/supervisor system with partial observation

to install enough sensors to monitor all plant events. Figure 2.9 illustrates such a system, where a subset of the events occurring in the plant are passed (via sensors) to the supervising agent.

Informally, if a plant is modeled using a finite state machine (FSM) $G$, containing states $q_1, q_2 \in Q$, and $\delta(q_1, \sigma_{uo}) = q_2$, where $\sigma_{uo}$ is an unobservable event, then any supervising agent upon seeing the plant enter state $q_1$ must provide for the possibility that the plant could be in either state $q_1$ or state $q_2$, since it is impossible to detect the occurrence of the unobservable event $\sigma_{uo}$. Thus, it is useful to construct a modified model of the plant $G$, which erases all occurrences of unobservable events, and which contains states (in the modified model) which correspond to subsets of states in $G$ which are indistinguishable to the supervisor $S$.

Formally, the process of removing events from strings contained in a language is called *natural projection*

$$P : \Sigma^* \to \Sigma_o^*$$

and can be recursively defined on strings as

$$\begin{aligned} P(\varepsilon) &= \varepsilon, \\ P(\sigma) &= \begin{cases} \sigma & \text{if } \sigma \in \Sigma_o \\ \varepsilon & \text{otherwise} \end{cases}, \\ P(s\sigma) &= P(s)P(\sigma) \text{ for } s \in \Sigma^*, \sigma \in \Sigma. \end{aligned} \tag{2.3}$$

Since in many cases, problems in DES are described in terms of FSMs which generate languages, it is useful to apply the concept of projection directly to FSMs. To do this, first it should be observed that all partially-observable FSMs used in DES applications can be thought of as nondeterministic finite state automata with $\varepsilon$-transitions, where the $\varepsilon$-transitions represent transitions in FSMs that cannot be observed.

In [HU79] induction on the length of strings is used to prove the following two theorems:

**Theorem 2.2 [HU79]** *If $L$ is accepted by a nondeterministic finite automaton (NFA) with $\varepsilon$-transitions, then $L$ is accepted by an NFA without $\varepsilon$-transitions.*

**Theorem 2.3 [HU79]** *Let $L$ be a set accepted by an NFA. Then there exists a deterministic finite automaton (DFA) that accepts $L$.*

Using these two theorems in combination, it is possible to convert an NFA with $\varepsilon$-transitions to a DFA (without $\varepsilon$-transitions). By labeling all unobservable events in the plant DFA as $\varepsilon$-transitions, thereby creating an NFA with $\varepsilon$-transitions, we can then convert the resulting NFA with $\varepsilon$-transitions to a DFA without $\varepsilon$-transitions. Thus, we have "erased" the occurrence of unobservable events. A construction method based on the proofs given in [HU79] is provided below. Let $G = (Q, \Sigma, \delta, q_o, Q_m)$ be a DFA with $\Sigma \dot{\cup} \Sigma_{uo}$ where $\dot{\cup}$ stands for disjoint union. For a state $q \in Q$, define

$$\varepsilon\text{-CLOSURE}(q) = \{r \mid (\exists s \in \Sigma^*)\delta(q, s) = r \text{ and } P(s) = \varepsilon\},$$

and

$$\eta(q, \sigma) = \{r \mid \delta(q, s) = r, \text{ where } p(s) = \sigma\}.$$

Construct NFA $G' = (Q, \Sigma_o, \delta', q_o, Q'_m)$. The marked state set $Q'_m$ and $\delta'$ are constructed as follows.

$$Q'_m = \begin{cases} Q_m \cup \{q_o\} & \text{if } \varepsilon - CLOSURE(q_o) \text{ contains a state of } Q_m \\ Q_m & \text{otherwise} \end{cases}$$

and $\delta'(q, \sigma) = \eta(q, \sigma)$ for $q \in Q$ and $\sigma \in \Sigma_o$. Note that the size of the state space for $G'$ remains the same.

The second part of the construction requires that the NFA be converted to an equivalent DFA. The construction is: Let $G' = (Q, \Sigma_o, \delta', q_o, Q'_m)$ be the above NFA. Construct $G_p = (Q_p, \Sigma_o, \delta_p, q_{o_p}, Q_{m_p})$ where:

$$\begin{aligned} Q_p &= 2^Q \text{ (the power set of } Q\text{)}, \\ q_{o_p} &= q_o, \\ Q_{m_p} &= \{q_p \in Q_p \mid \exists q \text{ where } q \text{ is contained in the label of } q_p \text{ and } q \in Q_m\}. \end{aligned}$$

Thus, single states $q_p \in Q_p$ use some subset of states $q \in Q$ as labels. For example, the label of some $q_p \in Q_p$ could be $\{q_1, q_2, \ldots, q_k\}$. Now define:

$$\delta'(\{q_1, q_2, \ldots, q_k\}, \sigma) = \cup_{i=1}^{k} \eta(q_i, \sigma).$$

As it is rarely the case that all $2^Q$ states are reachable in $G_p$, the construction can be made more efficient [Rud88] using an iterative approach as follows:

- Flag the initial state $q_{o_p}$ as a "new."
- For each "new" state, remove the "new" flag from the state, and construct all states reachable from that state via some $\sigma \in \Sigma_o$. If these states do not already exist, flag them as "new" states.
- Repeat the second step until no states with a "new" flag remain.

In this way, only those states which are reachable from the initial state are generated. Note however, that it can be shown by example that the resulting DFA is not necessarily a minimum-state DFA. To obtain a minimum-state DFA, the algorithm presented in 2.1 can be used. If $G_p$ is a DFA that recognizes the language $P(L_m(G))$

for some DFA $G$, we use the notation $G_p = p(G)$. A complete pseudo-code algorithm for constructing $p(G)$ is provided in Algorithm 2.3. Note that for step 3(c), line 6 of Algorithm 2.3 (which contains the statement "for each $q \in q_p$"), the state-set $q_p$ is itself a subset of $Q$.

**Algorithm 2.3 : A Projection Algorithm for FSMs**

```
1. Inputs:  Automaton $G = \{Q, \Sigma, \delta, q_o, Q_m\}$

2. Define a new automaton $G' = \{Q, \Sigma_o, \delta', q_o, Q'_m\}$

   (a) if there exists $q_m \in Q_m$ such that $q_m \in \varepsilon$-CLOSURE($q_o$) then
           $Q'_m = Q_m \cup \{q_o\}$
       else
           $Q'_m = Q_m$
       end

   (b) set $\delta'(\cdot,\cdot) = \emptyset$
       for each $q_i \in Q$ do
           for each $q_j \in Q$ do
               for each $\sigma \in \Sigma_o$ do
                   if $\delta(q_i, \sigma) = q_j$ then
                       $\delta'(q_i, \sigma) = q_j$
                   else if $\delta(q_i, s) = q_j$ for some $s \in \Sigma^*$
                       such that $P(s) = \sigma$ then
                       $\delta'(q_i, \sigma) = \delta'(q_i, \sigma) \cup \{q_j\}$
                   end
               end
           end
       end

3. Convert the NFA $G'$ to a DFA $G_p$

   (a) Set $q_{o_p} = q_o$
```

```
(b) flag $q_{o_p}$ as ``new''
    set $Q_p = \{q_{o_p}\}$
    set $Q_{m_p} = \emptyset$

(c) while  states flagged as ``new'' exist, do
            for  each state $q_p \in Q_p$ flagged as ``new'', do
                 remove the ``new'' flag
                 for  each $\sigma \in \Sigma_o$ do
                      set $q_{p_{new}} = \emptyset$
                      for  each $q \in q_p$ do
                           $q_{p_{new}} = q_{p_{new}} \cup \delta'(q, \sigma)$
                      end
                      if  $q_{p_{new}} \notin Q_p$ and $q_{p_{new}} \neq \emptyset$ then
                           let $Q_p = Q_p \cup \{q_{p_{new}}\}$
                           flag $q_{p_{new}}$ as ``new''
                           if  $\exists\, q \in q_{p_{new}}$ such that $q \in Q_m$ then
                                let $Q_{m_p} = Q_{m_p} \cup \{q_{p_{new}}\}$
                           end
                      end
                      if  $q_{p_{new}} \neq \emptyset$ then
                           define $\delta_p(q_p, \sigma) = q_{p_{new}}$
                      end
                 end
            end
    end

4. Output the DFA $G_p = (Q_p, \Sigma_o, \delta_p, q_{o_p}, Q_{m_p})$ as $p(G)$
```

The Algorithm 2.3 is based upon the two existing algorithms discussed in [HU79] and [Rud88]. While this algorithm constructs an output automaton that recognizes the projection (as defined by (2.3)) of the language recognized by an input automaton, some state information that can be useful to the observer is lost. We consider an example FSM $G$ where the initial state is $q_o$, and where for some $\sigma \in \Sigma_{uo}$, $\delta(q_o, \sigma) = q$ is defined for some $q \in Q$ where $q \neq q_o$. In this case, the initial state in $G_p = p(G)$ is labeled by $q_{o_p} = q_o$. However, if the supervising agent does not observe any events (and thus remains in the initial state), the plant $G$ could be in state $q_o$ or in state $q$. The labeling of the initial state using Algorithm 2.3 does not provide this type of information. In order to construct an automaton that both generates the projection language and contains useful state label information, we present an algorithm from [Rud88] in Algorithm 2.4. Algorithm 2.4 labels states so that each label identifies the states the plant could be in after the observation of a sequence of events.

**Algorithm 2.4 : A Modified Projection Algorithm for FSMs**

```
1. Inputs:  Automaton $G = \{Q, \Sigma, \delta, q_o, Q_m\}$

2. Define a new automaton $G' = \{Q, \Sigma_o, \delta', Q'_o, Q'_m\}$ with a set of initial
   states $Q'_o$

   (a) let $Q'_o = \{q_o, q_1, q_2, \ldots, q_n \mid q_i \in Q \wedge \exists s \in \Sigma^*$ such that $\delta(q_o, s) =$
       $q_i, P(s) = \varepsilon\}$

   (b) set $\delta'(\cdot, \cdot) = \emptyset$
       for each $q_i \in Q$ do
           for each $q_j \in Q$ do
               for each $\sigma \in \Sigma_o$ do
                   if $\delta(q_i, s) = q_j$ for some $s \in \Sigma^*$
                    such that $P(s) = \sigma$ then
                       $\delta'(q_i, \sigma) = \delta'(q_i, \sigma) \cup \{q_j\}$
                   end
               end
           end
       end
```

```
3. Convert the NFA $G'$ to a DFA $G_p$

  (a) Set $q_{o_p} = Q'_o$

  (b) flag $q_{o_p}$ as ``new''
      set $Q_p = \{q_{o_p}\}$
      set $Q_{m_p} = \emptyset$

  (c) while states flagged as ``new'' exist, do
            for each state $q_p \in Q_p$ flagged as ``new'', do
                remove the ``new'' flag
                for each $\sigma \in \Sigma_o$ do
                    set $q_{p_{new}} = \emptyset$
                    for each $q \in q_p$ do
                        $q_{p_{new}} = q_{p_{new}} \cup \delta'(q, \sigma)$
                    end
                    if $q_{p_{new}} \notin Q_p$ and $q_{p_{new}} \neq \emptyset$ then
                        let $Q_p = Q_p \cup \{q_{p_{new}}\}$
                        flag $q_{p_{new}}$ as ``new''
                        if $\exists\, q \in q_{p_{new}}$ such that $q \in Q_m$ then
                            let $Q_{m_p} = Q_{m_p} \cup \{q_{p_{new}}\}$
                        end
                    end
                    if $q_{p_{new}} \neq \emptyset$ then
                         define $\delta_p(q_p, \sigma) = q_{p_{new}}$
                    end
                end
            end
      end

4. Output the DFA $G_p = (Q_p, \Sigma_o, \delta_p, q_{o_p}, Q_{m_p})$ as $p(G)$
```

It should be noted that the check done at line 9 in step 3(c) of Algorithm 2.4 could cause computational problems if not implemented efficiently. There exist a number of methods ("path compression" [CLR90], for example) that can efficiently check for set inclusion.

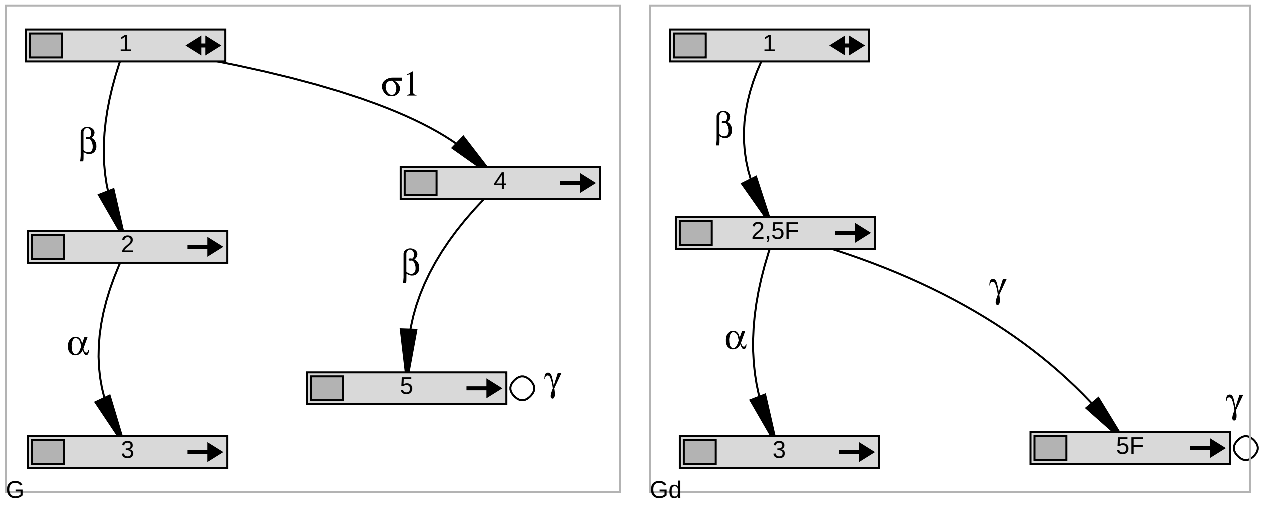
## 2.2.3 Diagnosability

Diagnosability is a branch of DES theory which addresses the problem of fault detection and isolation in large complex systems. In [SSL+95] and [SSL+96], a systematic procedure for analyzing systems and constructing FSM diagnosers for the purposes of fault detection is developed, with specific emphasis on application to heating, ventilation and air–conditioning (HVAC) systems. This section reviews the fundamental concepts relating to diagnosability, and discusses why the application of some of the theoretical results of this thesis are of interest in fault detection and isolation applications.

When analyzing a system to determine if that system is *diagnosable*, we first need to understand what it is that we are "diagnosing." We start with a FSM $G$ with event set $\Sigma$ which represents a plant containing observable and unobservable events ($\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$). A subset of events in $\Sigma$ are considered to be "failure" events (call the subset $\Sigma_f$) in the system. The event $\sigma 1$ event in Figure 2.10(a) is an example of such a failure event. We are not concerned with the failure events which are observable (i.e., $\sigma \in \Sigma_f \cap \Sigma_o$), since by definition a supervisor can observe these events, and therefore "diagnose" that they have occurred. Thus, without loss of generality, we can consider only those cases where all the failure events are unobservable (i.e., $\Sigma_f \subseteq \Sigma_{uo}$). Diagnosability theory considers the behaviour of a system after the occurrence of a failure event, and determines if it is possible to know in some finite amount of time that the failure event has occurred. A more generalized scenario can be achieved by partitioning the set of failure events into "classes" of failure events

$$\Sigma_f = \Sigma_{f1} \dot{\cup} \ldots \dot{\cup} \Sigma_{fm}.$$

In this case, for all $\sigma_f \in \Sigma_f$, the diagnosing agent need only determine in a finite



(a) The System (b) The Diagnoser

**Figure 2.10:** A system/diagnoser pair

amount of time that a failure of type $\Sigma_{fn}$, where $\sigma_f \in \Sigma_{fn}$, has occurred. It does not need to determine exactly which failure event occurred. Figure 2.10 shows an example system ($G$) and diagnoser ($G_d$). The diagnoser is a FSM that records the possible states the system may be in after observing a string of events, and infers what failures may have occurred.

For the example system $G$ shown in Figure 2.10(a)[3], $\sigma 1$ is the only failure event. Initially, the diagnoser $G_d$ only knows that the system has started in state 1. After observing the $\beta$ event, the diagnoser knows that the system could be in state 2, with no failures having occurred, or state 5 with failure $\sigma 1$ having occurred. Thus, at this stage the diagnoser is not able to determine if failure $\sigma 1$ has occurred or not. However, after observing the string $\beta\alpha$, the system can only be in state 3, and therefore the failure $\sigma 1$ cannot have occurred. If instead, the diagnoser observes the string $\beta\gamma$, then it knows that the failure $\sigma 1$ must have occurred. Since it is possible to detect the occurrence of all failure events in a finite amount of time (i.e., after a finite number of

[3]The circle to the right of state 5 indicates that a self-loop of event $\gamma$ may occur.

events have occurred) in this system, then the system is considered to be diagnosable, with diagnoser $G_d$.

Note that the the diagnoser is a DFA which recognizes the language given by the projection of the language recognized by the system. The method used to construct the diagnoser DFA is slightly different from the methods described in previous sections. In this case, each state in the diagnoser represents the set of states in the system which can be reached from an existing set of states via string $s$ where $s = s_{uo}\sigma_o$, where $s_{uo}$ is a string of (possibly zero) unobservable events, and $\sigma_o$ is an observable event. In all the other constructions presented in this thesis, the string $s$ is constructed in the opposite manner (i.e., $s = \sigma_o s_{uo}$). It can be shown that both methods of construction recognize $L_m(p(G))$ [SSL+95], [Rud88].

Diagnosability theory can be extended to cover systems which are considered to be "i-diagnosable." I-diagnosability is a looser condition than diagnosability in that after the occurrence of a failure event, the diagnoser need only identify that a failure of that type has occurred after the occurrence of an observable indicator event. Thus, for set of failure event types $\{\Sigma_{f1}, \ldots, \Sigma_{fm}\}$ there is a corresponding set of indicator event types $\{I_1, \ldots, I_m\}$.

It is apparent from the informal description of the diagnoser presented above, that diagnosability theory presents a direct application of the projection operation, combined with a set of rules for the labeling of states in the diagnoser such that they contain information relevant to the failure status of the system, and with a set of conditions placed on the system to determine if such a system is diagnosable or i-diagnosable. It is mentioned in [SSL+96] that the "two crucial issues regarding the applicability of our theory to HVAC units or other classes of systems are: 1) building the system model and 2) dealing with the computational complexity of the diagnostics process." It is also noted however, that with regards to the computational complexity issue: "our experience so far, while limited in scope, tends to indicate that the system often has enough structure so that the worst case computational bounds may be rarely attained." Finally, [SSL+96] states that if an approach which constructs diagnoser states on-line [HL94] is adopted, the problem can be solved with a computation of polynomial complexity at each observed transition of the system. Unfortunately, if a system is not diagnosable or i-diagnosable, and if an off-line analysis of the system is not done, the on-line diagnoser may arrive at states where it will never be possible to know if a failure has or has not occurred. We attempt to address this problem by analyzing the previously-mentioned "structure" of the system to make better estimates of the complexity of the computation required to construct a full diagnoser.

## 2.3 Computational Complexity

Since the motivation for developing DES theory is to be able to solve control problems in real systems, it is necessary to examine the efficiency with which DES operations can be implemented as algorithms. It may be a simple task to understand how an algorithm which implements a DES operation works. However, when the solution is actually computed, if the algorithm which does the computing takes an unreasonably long period of time, or uses an unreasonably large amount of computing resources, then the DES formalism becomes less useful as a control tool for real systems. The following sections outline some of the key ideas and tools in complexity theory which can be used to better understand the efficiency of algorithms which are used to solve common DES problems.

### 2.3.1 Complexity Classes: Background

Current research in complexity theory allows us to make some initial observations about the computational difficulty associated with DES problems. To provide a motivating example for why it is useful to group problems into complexity classes, consider the cases presented in Table 2.1 taken from [GJ79], where the time for each operation on some CPU is $1\mu s$. For example, if a problem is of "size" $n = 20$ and its solution is $O(n^2)$ complexity, then it would take $1\mu s \cdot 20^2 = 0.0004s$ to compute the solution on a CPU.

**Table 2.1:** An example of computational complexity

| Time Complexity Function | 20 | 40 | 60 |
|---|---|---|---|
| $n$ | 0.00002 s | 0.00004 s | 0.00006 s |
| $n^2$ | 0.0004 s | 0.0016 s | 0.0036 s |
| $n^3$ | 0.008 s | 0.064 s | 0.216 s |
| $n^5$ | 3.2 s | 1.7 min | 13.0 min |
| $2^n$ | 1.0 s | 12.7 days | 366 centuries |
| $3^n$ | 58 min | 3855 centuries | $1.3 \times 10^{13}$ centuries |

If the values presented in Table 2.1 are interpreted to correspond to the amount of time required to solve a problem of size 20, 40, and 60, where the solution takes either a polynomial or exponential amount of time, it becomes clear that in general, problems which require an exponential amount of time to solve become *intractable* when the size of the problem gets large. What complexity theory allows us to do is:

- determine if problems are intractable, and
- suggest methods for simplifying intractable problems by examining approximate solutions, or subproblems which can be solved in a polynomial amount of time.

Formally, decision problems can be grouped into complexity classes. An inclusion diagram for these classes is provided in Figure 2.11, taken from [GJ79]. Decision problems are placed in group $P$ if there exists an algorithm which can solve the decision problem in polynomial time. A decision problem is placed in the larger[4] $NP$ group if there exists an algorithm which can check the correctness of a "yes" answer to that decision problem in polynomial time. In order to understand the concept of the NP-complete (NPC) group, we first discuss the idea of problem transformations. Given two languages $L_1$ and $L_2$, we say that $L_1 \subseteq \Sigma_1^* \propto L_2 \subseteq \Sigma_2^*$ if there exists a

[4]It is widely believed, but has not been proven, that $P$ is a strict subset of $NP$.

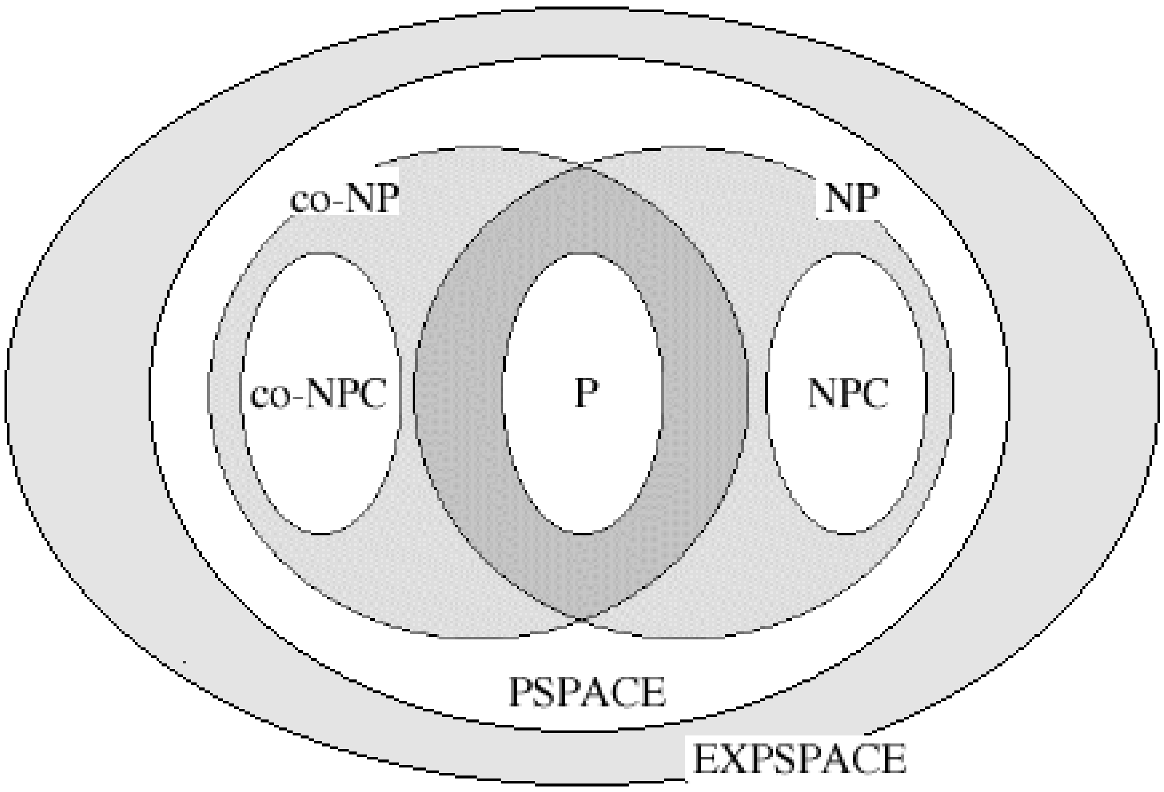


**Figure 2.11:** Complexity classes (assuming $P \neq NP$ and $NP \neq co\text{-}NP$)

function $f$ such that $f : \Sigma_1^* \rightarrow \Sigma_2^*$, and $s \in L_1$ iff $f(s) \in L_2$, and where $f$ can be computed in polynomial time. By extending the idea of the polynomial transformability relation from languages to decision problems (refer to [GJ79] for details), then for two decision problems $\Pi_1$ and $\Pi_2$, the relationship $\Pi_1 \propto \Pi_2$ can be interpreted to mean "$\Pi_2$ is at least as hard as $\Pi_1$". The two problems are considered to be *polynomially equivalent* if $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_1$. Since it can also be proven that polynomial transformability is transitive, then the relation "$\propto$" imposes a partial ordering on all equivalence classes of decision problems in NP, where P represents the computationally "easiest" problems, and NPC represents the computationally "hardest" problems. Thus, a decision problem $\Pi \in NP$ can be proven to be NP-complete if for some $\Pi' \in NPC$, $\Pi' \propto \Pi$. This method is used in [Tsi89] where Tsitsiklis reduces an instance of the "3-satisfiability" problem which is known to be NP-complete to a specific class of partial-observation DES problems.

The complexity class co-NP represents the complement decision problems for all the decision problems which comprise the class NP. Given a decision problem in NP such as "Given I, is X true for I?", the complementary decision problem would be "Given I, is X false for I?". It has not yet been proven that co-NP≠NP. Indeed if this could be proven, then it would have to be the case that $P \neq NP$.

While the $P$ versus $NP$ complexity classes focus primarily on the time which algorithms take to solve problems, the $PSPACE$ and $EXPSPACE$ complexity groups focus on the amount of memory required to solve problems. Specifically $PSPACE$ (resp., $EXPSPACE$) decision problems require a polynomial (resp., exponential) amount of memory to solve. By adopting a similar method as used to define NP–complete problems, problems can be ordered such that a subset of problems in $PSPACE$ (resp., $EXPSPACE$) represents the computationally most difficult problems in the set. These subsets are referred to as $PSPACE$ (resp., $EXPSPACE$) complete problems. Again, as with the NP-complete class of decision problems, a decision problem $\Pi \in PSPACE$ (resp., $EXPSPACE$) can be proven to be $PSPACE$–complete (resp., $EXPSPACE$–complete) if for some $\Pi' \in PSPACE$ (resp., $EXPSPACE$), $\Pi' \propto \Pi$.

### 2.3.2 Working with NP-Complete Problems

If a problem is proven to be NP–complete (or $PSPACE/EXPSPACE$-complete), then a method needs to be devised for solving that type of problem in a computationally feasible manner. For example, it may be possible to construct an heuristic algorithm which produces a correct result in most cases. However, current DES formalisms primarily model safety-critical systems, and therefore require correct results all of the time. Alternatively, it may be possible to restrict the set of problems to a subset of allowable problems which we know (and can prove) to be solvable in a polynomial amount of time. If it is computationally feasible to test whether a problem belongs to this subset, and if the subset captures a large enough class of DES applications, then we will have found a computationally feasible method for solving a

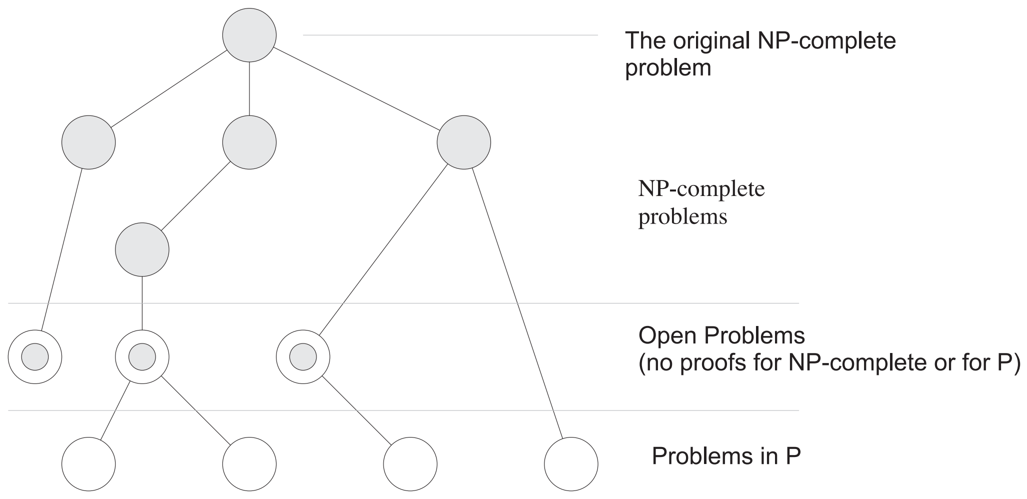

The original NP-complete problem
NP-complete problems
Open Problems (no proofs for NP-complete or for P)
Problems in P

**Figure 2.12:** An example of the hierarchical breakdown of an NP-complete problem

subset of partial-observation DES problems. Furthermore, for those problems which do not fall into this subset of computationally feasible problems, it would be useful if there were methods for identifying the characteristics of the problem which disqualify it for inclusion in the subset. If we could identify "problem areas," it is possible that the component DES models could be modified such that the solution can be computed in a computationally feasible manner. Figure 2.12 taken from [GJ79] provides an example of the hierarchy of subproblems for some NP-complete problem.

### 2.3.3 Complexity Theory and DES

There are two areas in DES Theory where computational complexity issues make the solutions to large problems intractable. First, as noted in [Won96] and [WR88], when the synchronous product or meet of $n$ FSMs $(G_1, G_2, \ldots, G_n)$ is computed, then it is possible that the state space of the resulting FSM $G = MEET(G_1, G_2, \ldots, G_n)$ could have a state space as large as $k^n$, where $k$ is the maximum of the sizes of the state-spaces of $G_1, \ldots, G_n$. Since the number of states in $G$ increases exponentially with $n$, the problem becomes intractable for large n (i.e., for a large number of component models in a typical DES problem). A Petri–net method for efficiently modeling a class of problems where many of the $n$ component models are identical (i.e., parallel or additive machines) is presented in [LW93] and [LW94]. The results to be presented in Chapter 3 of this thesis do not focus on DES complexity problem where the components are identical FSMs.

The second area of DES theory which presents us with computationally intractable problems is in the area of partially-observable and/or decentralized DES problems. While in many cases it has been noted that the actual results obtained while working with these types of DES problems are good, it was proven by Tsitsiklis [Tsi89] that for a specific class of partial-observation problems, there exists a polynomial transformation which maps an instance of the Boolean logic "three satisfiability" problem (a restricted version of Cook's theorem for the satisfiability problem), which has been proven to be NP–complete [GJ79], to this class of partial-observation DES problems. This means that unless it is proven that $P = NP$, there is no polynomial–time algorithm which can construct a FSM which marks the projection of the language marked by a given FSM. However, even though this type of problem has been proven to be NP–complete, the favourable results obtained in many partial-observation applications suggest that there may exist a class of sub-problems (i.e., a class of FSMs) for which there exists an algorithm which takes significantly less than an exponential amount of time to complete the same task. Chapter 3 of this thesis attempts to identify some of the properties of this type of FSM.

# Chapter 3

# Structure-Based DES Analysis

While in theory projection can lead to an exponential increase in the number of states, in practice it has been noted [SSL$^+$95], [SSL$^+$96], and [OW90] that in many cases the number of states generated is typically much less than the exponential limit. This suggests that there exist subproblems which can be solved efficiently. If these types of subproblems can be characterized and identified in a simple and time-efficient manner, then we would have a test we could run on large systems which would identify whether the system will project efficiently, or could identify problem structures within the system which could cause exponential explosion.

In designing this type of test, two things need to be considered. First, an algorithm (or series of algorithms) needs to be designed which calculates upper limits for the projection state-space. If such an algorithm can show that the projection state-space is going to be small relative to the exponential limit, then we can go ahead and calculate the projection, knowing that the resulting automaton can be found in a reasonable amount of time.

When the upper bound algorithms do not show any significant reduction on the exponential limit on the projection state space, algorithms that indicate lower bounds are helpful (for flagging problem areas). In these cases, we need to understand what structures within the system are causing problems, in order to effect changes in the model (perhaps by adding more sensors in the physical system) so that when the upper bound algorithms are re-run, the resulting estimate of the projection state-space is significantly better than the exponential limit. These lower bound algorithms would indicate that the projection state space will be at least a certain size, and would identify the structures which are primarily responsible for this lower bound.

It is conceivable that the results of the upper bound algorithms show no significant reduction relative to the exponential limit, and that the lower bound algorithms can identify no structures which cause state-space explosion. To date, it is not clear how many DES models which are based on physical problems fit into this category.

An important consideration when designing algorithms which establish either upper or lower bounds is that all algorithms must be able to identify properties or structures within plant models in an efficient manner. Algorithms which run in low-order polynomial time will result in tests which are simple and fast to run on systems, whereas high-order polynomial or exponential time algorithms serve no purpose, since the projection operation itself is an exponential time algorithm. In effect, we would be better off just running the projection algorithm itself, rather than running exponential–time algorithms that test how quickly projection can be done.

## 3.1 Structures: Upper Limits on State-Space Explosion

In establishing algorithms which can be used to identify upper limits on the projection state space, we first consider how the presence of unobservable events affects projection. For a system to be partially observable, there must exist at least one unobservable event. We examine the properties of arbitrary systems containing single and double unobservable event transitions, and establish some upper bounds for the size of the projection state-space. While these upper bounds don't significantly reduce the projection state-space estimate below the exponential limit, they at least serve as a starting point in our attempt to understand the effect of unobservable event structures within DES models.

We then define a property of a plant we call *$n\sigma$-reachability* to be the set of states which can be reached by starting at any state in the plant, and following the string $s \in \Sigma^*$ where $P(s)$ is a string of $n$ observable events. We use this concept to calculate subsets of states which can be used to define all possible subset-labels in the projection state space. This has the effect of reducing the exponent used to calculate the projection state space, and thus allows for some significant reductions in the estimated size of the state space (relative to the exponential limit) to be made. The computational complexity of this algorithm is $O(|\Sigma_o|^n \cdot |Q|^2)$, where $\Sigma_o$ is the set of observable events, and $Q$ is the set of states.

Furthermore, by examining the cases when the estimated projection state size is significantly reduced, some conjectures may be drawn about desirable structural properties of the system. These may be used as a guide when modifying systems which do lead to exponential explosion.

### 3.1.1 The Significance of Automata with $\varepsilon$-Transitions

It has been shown that when an NFA is converted to an DFA which recognizes the same language, the state set for the DFA is a subset of $2^Q$, where $Q$ is the state set of the NFA. If $|Q| = n$ it is not necessarily the case that all $2^n - 1$ nonempty states will be generated using the subset-construction method given in Algorithm 2.4. However, it can be shown that in some cases, not only are all $2^n - 1$ nonempty states generated, but also that the DFA is, in fact, a minimal-state DFA according to [HU79].

In DES theory, plants are represented as DFAs. Nondeterminism is introduced when the unobservable events are relabeled as $\varepsilon$-transitions, and the resulting NFA with $\varepsilon$-transitions is converted to an NFA without $\varepsilon$-transitions. We want to take advantage of the structure in this NFA to make some observations about the upper limit on the number of states generated when converting the NFA to an equivalent DFA. The first way this can be done is by examining the structure of the $\varepsilon$-transitions. The following list provides five separate $\varepsilon$-transition structures (Figure 3.1) which will be discussed in further detail:

(a) $\delta(q_i, \varepsilon) = q_j$, $i \neq j$,

(b) $\delta(q_i, \varepsilon) = q_j$ and $\delta(q_i, \varepsilon) = q_k$, $i \neq, j \neq k$

(c) $\delta(q_j, \varepsilon) = q_i$ and $\delta(q_k, \varepsilon) = q_i$, $i \neq, j \neq k$

(d) $\delta(q_i, \varepsilon) = q_j$ and $\delta(q_k, \varepsilon) = q_l$, $i \neq, j \neq k \neq l$

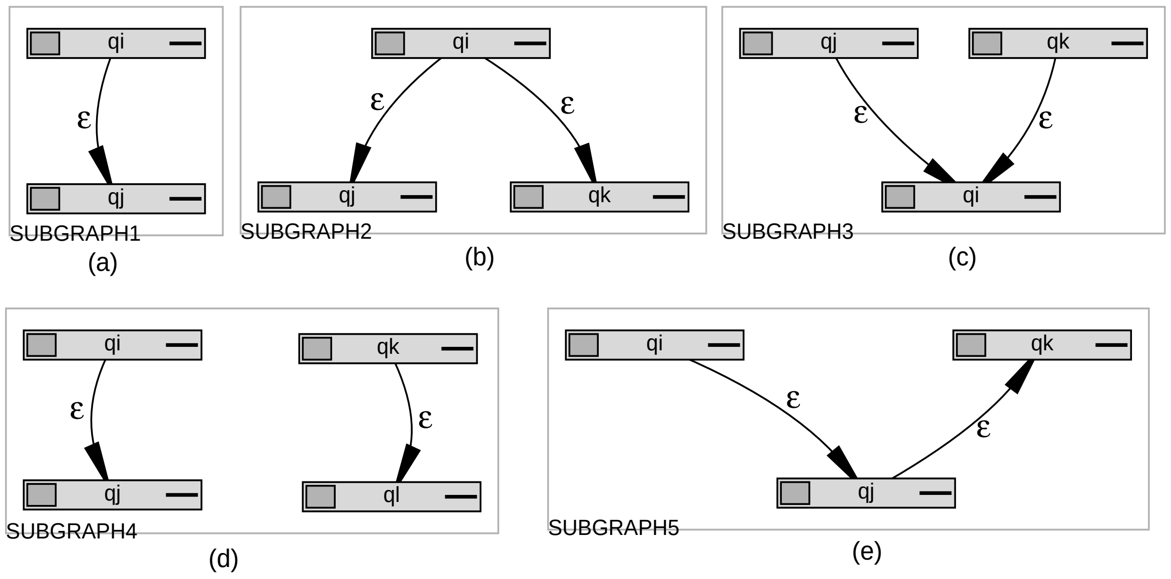


**Figure 3.1:** Unobservable event transitions

(e) $\delta(q_i, \varepsilon) = q_j$ and $\delta(q_j, \varepsilon) = q_k$, $i \neq .j \neq k$

#### 3.1.1.1 The Single $\varepsilon$-Transition

An almost immediate observation that can be made about the construction in Algorithm 2.4 is that if there is at least one transition labeled by an unobservable event (Figure 3.1(a)), then the state-space of the projection will never exceed $3/4 \cdot 2^{|Q|}$.

**Proposition 3.1** *Given plant $G$ with event set $Q$, unobservable event set $\Sigma_{uo}$, and transition function $\delta$, and FSM $G_p$ with event set $Q_p$ such that $G_p = p(G)$. If $\delta(q_i, \sigma) = q_j$ for some $q_i, q_j \in Q$, $i \neq j$ and for some $\sigma \in \Sigma_{uo}$ then:*

$$|Q_p| \leq \frac{3}{4} \cdot 2^{|Q|} - 1.$$

*Proof:* From Algorithm 2.4 lines 5–8 of step 2(b) and lines 6-8 of step 3(b), the set $Q_p$ can not contain states with state-set labels that contain $q_i$ but do not contain $q_j$.

Now, if $Q_p$ cannot contain state-sets which include $q_i$ but not $q_j$, all we need to do is count the number of states of this type. Simple combinatorics shows us that this number is $2^{|Q|-2}$. Discounting the empty state-set, we can show that $|Q_p| \leq 3/4 \cdot 2^{|Q|} - 1$.

*Example:*

Take an automaton $G$, with state set $Q = \{q_1, q_2, q_3, q_4\}$, and with an unobservable event $\sigma$ where $\delta(q_1, \sigma) = q_2$ is defined. Observe that any state label in $Q_p$ which includes $q_1$ must also include $q_2$ since if the automaton $G$ could be in state $q_1$, then it may also be in state $q_2$ via $\delta(q_1, \sigma) = q_2$. Thus the set of subsets of $2^Q$ which cannot appear as labels in $Q_p$ are:

$$\{\{q_1\}, \{q_1, q_3\}, \{q_1, q_4\}, \{q_1, q_3, q_4\}\},$$

and the set of labels which may be included in $Q_p$ is at most:

$\{\{q_2\}, \{q_1, q_2\}, \{q_1, q_2, q_3\}, \{q_1, q_2, q_4\}, \{q_1, q_2, q_3, q_4\}, \{q_3\},$
$\{q_4\}, \{q_2, q_3\}, \{q_2, q_4\}, \{q_3, q_4\}, \{q_2, q_3, q_4\}\}.$

Since $|Q| = 4$ we can see that $Q_p$ contains at most $3/4 \cdot 2^{|Q|} - 1 = 11$ elements.

#### 3.1.1.2 Double $\varepsilon$-Transition Geometries

In this section, we attempt to improve upon the results presented for single $\varepsilon$-transition geometries by applying those results to the four geometries which can result when at least two unobservable events appear in an arbitrary plant (Figure 3.1(b)–(e)).

Formally, we present Propositions 3.2–3.5, which provide upper bounds for plants which contain structures illustrated in Figure 3.1(b)–(e), respectively.

**Proposition 3.2** *Given plant $G$ with event set $Q$, unobservable event set $\Sigma_{uo}$, and transition function $\delta$, and FSM $G_p$ with event set $Q_p$ such that $G_p = p(G)$. If $\delta(q_i, \sigma_1) = q_j$ and $\delta(q_i, \sigma_2) = q_k$ for some $q_i, q_j, q_k \in Q$, $i \neq j \neq k$ and for some $\sigma_1, \sigma_2 \in \Sigma_{uo}$ (Figure 3.1(b)) then:*

$$|Q_p| \leq \frac{5}{8} 2^{|Q|} - 1.$$

*Proof:* As claimed earlier, if for some $G$, $\delta(q', \sigma) = q''$ and $\sigma \in \Sigma_{uo}$, then $Q_p$ as constructed by Algorithm 2.4 can contain no states with state-set labels that contain $q'$, and that do not contain $q''$. We can now break down a problem where $\delta(q_i, \sigma_1) = q_j$ and $\delta(q_i, \sigma_2) = q_k$ for $\sigma_1, \sigma_2 \in \Sigma_{uo}$. The transition $\delta(q_i, \sigma_1) = q_j$ implies that $Q_p$ will contain no state-sets containing $q_i$ but not $q_j$. Further, the transition $\delta(q_i, \sigma_2) = q_k$ implies that $Q_p$ will contain no state-sets containing $q_i$ but not $q_k$. By applying combinatorics together with a counting argument, and providing that $i \neq j \neq k$, it can be easily shown that

$$|Q_p| \leq 2^{|Q|} - 2^{|Q|-2} - 2^{|Q|-2} + 2^{|Q|-3},$$

where the first term represents all state-sets in the power set, the second term represents all the state sets which include $q_i$ but not $q_j$, the third term represents all the state-sets which include $q_i$ but not $q_k$, and finally the fourth term represents all the state-sets which include $q_i$, but do not contain $q_j$ or $q_k$ (and therefore have been counted twice when calculating second and third terms). These calculations result in Proposition 3.2 when the null state-set is discarded.

*Example:*

Take an automaton $G$, with state set $Q = \{q_1, q_2, q_3, q_4\}$, and with unobservable events $\sigma_1$ and $\sigma_2$ where $\delta(q_1, \sigma_1) = q_2$ and $\delta(q_1, \sigma_2) = q_3$ are defined. Observe that any state label in $Q_p$ which includes $q_1$ must also include $q_2$ and $q_3$ since if the automaton $G$ could be in state $q_1$, then it may also be in state $q_2$ (resp. $q_3$) via $\delta(q_1, \sigma_1) = q_2$ (resp. $\delta(q_1, \sigma_2) = q_3$). Thus the set of subsets of $2^Q$ which cannot appear as labels in $Q_p$ are:

$$\{\{q_1\}, \{q_1, q_3\}, \{q_1, q_2\}, \{q_1, q_4\}, \{q_1, q_3, q_4\}\},$$

and the set of labels which may be included in $Q_p$ is at most:

$$\{\{q_1, q_2\}, \{q_1, q_2, q_3\}, \{q_1, q_2, q_4\}, \{q_1, q_2, q_3, q_4\}, \{q_3\}, \{q_4\}, \{q_3, q_4\}\}.$$

Note that this set contains exactly $3/4 \cdot 2^{|Q|} - 1$ elements since the empty set is not counted.

**Proposition 3.3** *Given plant $G$ with event set $Q$, unobservable event set $\Sigma_{uo}$, and transition function $\delta$, and FSM $G_p$ with event set $Q_p$ such that $G_p = p(G)$. If $\delta(q_j, \sigma_1) = q_i$ and $\delta(q_k, \sigma_2) = q_i$ in $G$ for some $q_i, q_j, q_k \in Q$, $i \neq j \neq k$ and for some $\sigma_1, \sigma_2 \in \Sigma_{uo}$ (Figure 3.1(c)) then:*

$$|Q_p| \leq \frac{5}{8}2^{|Q|} - 1.$$

*Proof:* If for some $G$, $\delta(q', \sigma) = q''$ and $\sigma \in \Sigma_{uo}$, then $Q_p$ as constructed by Algorithm 2.4 can contain no states with state-set labels that contain $q'$, and that do not contain $q''$. Therefore for this example, $\delta(q_j, \sigma_1) = q_i$ implies that no states (in $Q_p$) can exist which contain $q_j$ but not $q_i$. Similarly, $\delta(q_k, \sigma_2) = q_i$ implies that no states (in $Q_p$) can exist which contain $q_k$ but not $q_i$. Therefore, as with the previous proposition, providing that $i \neq j \neq k$ we have

$$|Q_p| \leq 2^{|Q|} - 2^{|Q|-2} - 2^{|Q|-2} + 2^{|Q|-3},$$

where the first term represents all state-sets in the power set, the second term represents all the state sets which include $q_j$ but not $q_i$, the third term represents all the state-sets which include $q_k$ but not $q_i$, and finally the fourth term represents all the state-sets which include $q_j$ and $q_k$ but not $q_i$ (and therefore have been counted twice when calculating the second and third terms). These calculations result in Proposition 3.3 when the null state-set is discarded.

**Proposition 3.4** *Given plant $G$ with event set $Q$, unobservable event set $\Sigma_{uo}$, and transition function $\delta$, and FSM $G_p$ with event set $Q_p$ such that $G_p = p(G)$. If $\delta(q_i, \sigma_1) = q_j$ and $\delta(q_k, \sigma_2) = q_l$ in $G$ for some $q_i, q_j, q_k, q_l \in Q$, $i \neq j \neq k \neq l$ and for some $\sigma_1, \sigma_2 \in \Sigma_{uo}$ (Figure 3.1(d)) then:*

$$|Q_p| \leq \frac{7}{16}2^{|Q|} - 1.$$

*Proof:* If for some $G$, $\delta(q', \sigma) = q''$ and $\sigma \in \Sigma_{uo}$, then $Q_p$ as constructed by Algorithm 2.4 can contain no states with state-set labels that contain $q'$, and that do not contain $q''$. Therefore for this example, $\delta(q_i, \sigma_1) = q_j$ implies that no states (in $Q_p$)

can exist which contain $q_i$ but not $q_j$. Similarly, $\delta(q_k, \sigma_2) = q_l$ implies that no states (in $Q_p$) can exist which contain $q_k$ but not $q_l$. Therefore we have

$$|Q_p| \leq 2^{|Q|} - 2^{|Q|-2} - 2^{|Q|-2} + 2^{|Q|-4},$$

where the first term represents all state-sets in the power set, the second term represents all the state sets which include $q_i$ but not $q_j$, the third term represents all the state-sets which include $q_k$ but not $q_l$, and finally the fourth term represents all the state-sets which include $q_i$ and $q_j$ but not $q_k$ or $q_l$ (and therefore have been counted twice when calculating the second and third terms). These calculations result in Proposition 3.4 when the null state-set is discarded.

**Proposition 3.5** *Given plant $G$ with event set $Q$, unobservable event set $\Sigma_{uo}$, and transition function $\delta$, and FSM $G_p$ with event set $Q_p$ such that $G_p = p(G)$. If $\delta(q_i, \sigma_1) = q_j$ and $\delta(q_j, \sigma_2) = q_k$ in $G$ for some $q_i, q_j, q_k \in Q$, $i \neq j \neq k$ and for some $\sigma_1, \sigma_2 \in \Sigma_{uo}$ (Figure 3.1(e)) then:*

$$|Q_p| \leq \frac{1}{2} 2^{|Q|} - 1.$$

*Proof:* If for some $G$, $\delta(q', \sigma) = q''$, and $\sigma \in \Sigma_{uo}$, then $Q_p$ as constructed by Algorithm 2.4 can contain no states with state-set labels that contain $q'$, and that do not contain $q''$. Therefore for this example, $\delta(q_i, \sigma_1) = q_j$ implies that no states (in $Q_p$) can exist which contain $q_i$ but not $q_j$. Similarly, $\delta(q_j, \sigma_2) = q_k$ implies that no states (in $Q_p$) can exist which contain $q_j$ but not $q_k$. Therefore we have

$$|Q_p| \leq 2^{|Q|} - 2^{|Q|-2} - 2^{|Q|-2},$$

where the first term represents all state-sets in the power set, the second term represents all the states which include $q_i$ but not $q_j$, the third term represents all the state-sets which include $q_j$ but not $q_k$. Note that in this case, no terms are double-counted since all the states counted by the first term do not contain $q_j$, and all the states counted by the second term do contain $q_j$. These calculations result in Proposition 3.5 when the null state-set is discarded.

### 3.1.2 Tree Structures

We define a *tree* to be an automaton that contains a unique path between any two states. Note that if automata of this type are finite, then they mark only finite languages. This is true for the following reason: If $\delta(q_i, s) = q_j$ is defined for some $s \in \Sigma^*$, then there is no other $s' \in \Sigma^*$ for which $\delta(q_i, s') = q_j$ is also defined (true by definition of a tree structure). Therefore, if the tree structure $G$ is trim, then for each $q_m \in Q_m$, there exists a unique $s \in \Sigma^*$ such that $\delta(q_o, s) = q_m$. Thus, the marked language is composed of exactly $|Q_m|$ unique strings, and is therefore finite.

Given a tree structure $G$, some observations can also be made about the size of the state space of $p(G)$.

**Proposition 3.6** *Given a tree structure $G$ with state set $Q$ and $G_p$ with state set $Q_p$ such that $G_p = p(G)$, then the following is true:*

$$|Q_p| \leq |Q|.$$

*Proof:*

*Claim:* Any state $q \in Q$ in tree $G$ may appear in at most one of the labels of $Q_p$.

Assume otherwise: take some state $q \in Q$ such that $q$ appears in the labels of states $q_1'$ and $q_2'$ where $q_1', q_2' \in Q_p$. Since $G_p$ is by definition a DFA, there must be two strings $s_1', s_2' \in \Sigma_o^*$ where $s_1' \neq s_2'$ such that $\delta_p(q_{o_p}, s_1') = q_1'$ and $\delta_p(q_{o_p}, s_2') = q_2'$. Thus by definition there must be two strings $s_1, s_2 \in \Sigma^*$ where $p(s_1) = s_1'$ and $p(s_2) = s_2'$ such that $\delta(q_o, s_1) = q$ and $\delta(q_o, s_2) = q$. Since $s_1' \neq s_2'$ then $s_1 \neq s_2$. If this is the case, there are two distinct paths between the initial state $q_o$ and state $q$, thereby contradicting the definition of a tree.

Finally, using a simple counting argument, it can be shown that the size of the state space $|Q_p| \leq |Q|$. Each state $q' \in Q_p$ must be labeled by a nonempty subset of $Q$. By the above claim, each state $q \in q'$ is unique to $q'$, and appears in no other state in $Q_p$. If this is the case, then there must exist at least $|Q_p|$ unique states in $Q$, i.e., $|Q| \geq |Q_p|$.

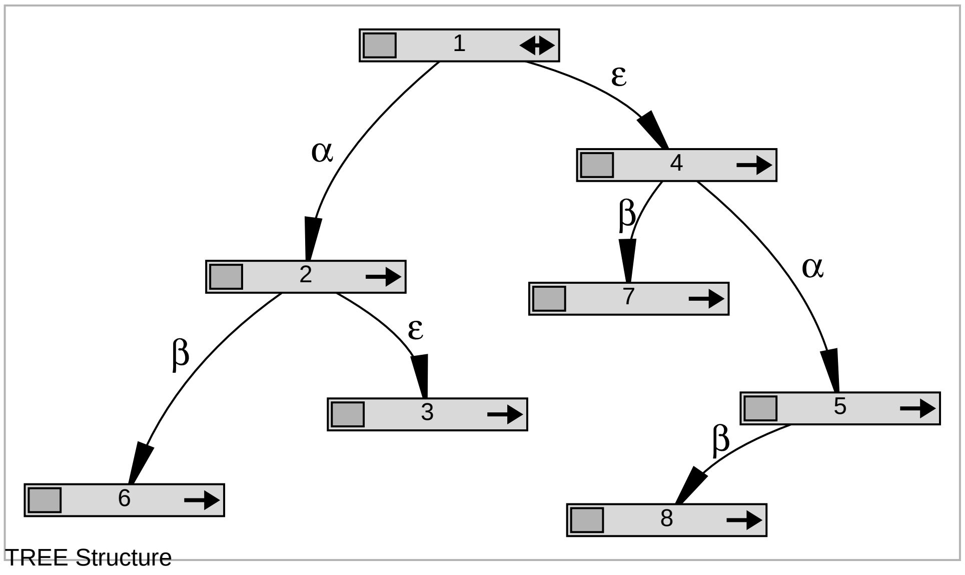


**Figure 3.2:** Tree structure example

*Example:*

An example of a tree-structure is presented in Figure 3.2. In this example, $Q = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $\Sigma_{uo} = \{\varepsilon\}$. By inspection, it can be seen that the state-space of the projection of the tree structure is $Q_p = \{\{1, 4\}, \{2, 3, 5\}, \{7\}, \{6, 8\}\}$.

### 3.1.3 $\sigma$-Reachability

While the concepts of single and double $\varepsilon$ geometries provide us with some upper bounds which apply to all FSMs, because the reduction is a simple constant varying from 3/4 to 1/2, the resulting effect on complexity results is negligible when dealing with large systems. By contrast, if the system we are considering is a tree (as defined in Section 3.1.2), then we have shown that the complexity of the projection algorithm is $O(n)$ or linear. While this result is computationally good, the tree FSM structure captures a very small subset of possible FSMs. What is needed to make practical analysis of DES partial-observation problems possible is something which can be applied to a large subset of FSMs, and which provides a significant (i.e., $O(2^n)$ or exponential) improvement on complexity results. In the following sections we present a method which we believe provides significant improvements on complexity estimates and which is not limited in applicability to tree structures.

#### 3.1.3.1 $1\sigma$-Reachability

The following intuition can be used to tighten the upper bound for the size of the state space of a projected DES. Any state subset, say $q_p$, that appears as a state label in a projected FSM resulting from Algorithm 2.4 is, by construction, a reachable state. This means that some observable event $\sigma$ leads from some set of states in the original DES to the set $q_p$. The set $q_p$ cannot be any larger than the set of states that could be reached via $\sigma$ from the set of *all* states in the original DES. This observation leads to a concept called "$\sigma$-reachability," defined as follows. Given FSMs $G = (Q, \Sigma, \delta, q_o, Q_m)$ and $G_p = (Q_p, \Sigma_o, \delta_p, q_{o_p}, Q_{m_p})$, where $G_p = p(G)$, then we say that the set of all nonempty subsets of the set of states $Q_\sigma = \{q' \mid \exists q \in Q, \delta(q, \sigma) = q'\}$ are $1\sigma$-*reachable* states in $G$. That is, $2^{Q_\sigma}$ contains all the states in $Q_p$ which the system could be in after observing the $\sigma$ event. For example, for the FSM $G$ given in Figure 3.3 with $Q = \{1, 2, 3, 4, 5\}$, the $\alpha$-reachable set is $2^{Q_\alpha} = \{\{1, 4\}, \{1\}, \{4\}\}$, and the $\beta$-reachable set is $2^{Q_\beta} = \{\{2, 3, 5\}, \{2, 3\}, \{2, 5\}, \{3, 5\}, \{2\}, \{3\}, \{5\}\}$, and finally the $\gamma$-reachable set is $2^{Q_\gamma} = \{\{5\}\}$. Finally, we need to include the initial state $\{1, 4\}$, since it may not be included as a subset of any of the $Q_\sigma$ sets. Therefore, a new estimate on the maximum number of states in $p(G)$ is 12, compared with the $3/4 \cdot 2^5 - 1 = 23$ state estimate using the upper limit defined in Proposition 3.1. Note, however that in calculating this number, no attempt is made to account for the duplication of state sets. Thus, for this example, the state sets $\{5\}$ and $\{1, 4\}$ are counted twice. The actual limit (i.e., without double counting) for the number of state sets is 10. For all $\sigma$-reachable based state-estimates presented in Chapter 5, note that state set duplication has not been accounted for, and thus, the estimates could be smaller than indicated. Theorem 3.1 formalizes the $\sigma$-reachability concept for the case described above, where a single observable event is seen by a supervising agent.

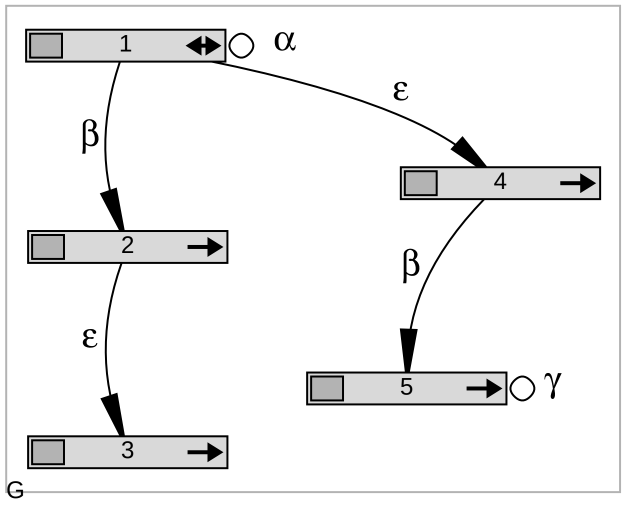


**Figure 3.3:** $\sigma$-reachability example

**Theorem 3.1** *Given FSMs* $G = (Q, \Sigma, \delta, q_o, Q_m)$ *and* $G_p = (Q_p, \Sigma_o, \delta_p, q_{o_p}, Q_{m_p})$, *where* $G_p = p(G)$, *then*

$$Q_p \subseteq \bigcup_{\sigma \in \Sigma} 2^{Q_\sigma} \cup q_{o_p}$$

*Proof:*

Recall that the states in $Q_p$ are subsets of $Q$. Consider an element $q_p \in Q_p$.

Case 1: $q_p = q_{o_p}$. Then by observation, $q_p \in Q_p$.

Case 2: $q_p \neq q_{o_p}$.

Then there is a $q'_p \in Q_p$ and $\sigma \in \Sigma_o$ such that $\delta_p(q'_p, \sigma) = q_p$ (since $q_p \in Q_p$ means $q_p$ is reachable from $q_{o_p}$ via $\delta_p$). Since $q'_p \in Q_p$, by definition, $q'_p \subseteq Q$.

According to the construction given in Algorithm 2.4, $\delta_p(q'_p, \sigma') \subseteq Q_{\sigma'}$ for all $\sigma' \in \Sigma_o$. Therefore, in particular, $\delta_p(q'_p, \sigma) \subseteq Q_\sigma$. That is, $q_p \subseteq Q_\sigma$, which implies that $q_p \in 2^{Q_\sigma}$.

#### 3.1.3.2 Multiple $\sigma$-Reachability

The $\sigma$-reachability property described in Section 3.1.3.1 while providing useful properties by itself, can be iteratively applied in such a way as to provide improved estimates for some FSMs. The key observation is as follows: $1\sigma$-reachability dictates that after the occurrence of some event $\sigma_1$, the set of states which the plant may be in must be a subset of $Q_{\sigma_1}$. Now consider the following: call the next observable event (following $\sigma_1$) $\sigma_2$. We have established that after $\sigma_1$, the system must be in some subset of the states contained in the set $Q_{\sigma_1}$. We can now further reduce the subset of states that the system could be in by substituting $Q_{\sigma_1}$ for $Q$ in the expression $Q_{\sigma_2} = \delta_p(Q, \sigma_2)$. Thus, the possible subsets of states which the system could be in after observing two transitions must be subsets of the following:

$$\forall \sigma_1 \in \Sigma_o, \forall \sigma_2 \in \Sigma_o : Q_{\sigma_1\sigma_2} = \{q' \mid \exists q \in Q_{\sigma_1}, \delta(q, \sigma_2) = q'\}.$$

For the example given in Figure 3.3, the $Q_{\sigma_1\sigma_2}$ subsets are:

$$\begin{aligned}
Q_{\alpha\alpha} &= \{1, 4\}, \\
Q_{\alpha\beta} &= \{2, 3, 5\}, \\
Q_{\alpha\gamma} &= \emptyset, \\
Q_{\beta\alpha} &= \emptyset, \\
Q_{\beta\beta} &= \emptyset, \\
Q_{\beta\gamma} &= \{5\}, \\
Q_{\gamma\alpha} &= \emptyset, \\
Q_{\gamma\beta} &= \emptyset, \\
Q_{\gamma\gamma} &= \{5\}.
\end{aligned}$$

For this example, the total for all the nonempty subsets for all the $Q_{\sigma_1\sigma_2}$ sets (without correcting for state-set duplication) is 16 (including the additional state-sets representing the initial state-set and the three state-sets which can be reached after observing the first transition). If state-set duplication is accounted for, the estimate reduces to 10. Thus, for this example, no improvement is obtained by iterating the $\sigma$-reachability procedure.

Note that if we iterate $I$ times, then $\sum_{i=1}^{I} |\Sigma_o|^i$ computations must be done. Thus, the complexity of this type of test is exponential in $I$. Based on the results presented in Chapter 5, we have found that for the systems analyzed, the best results (without correcting for state-set duplication errors) are achieved with two or three iterations, and therefore, only a small number of computations is required.

### 3.1.4 Related Work

After completing the work presented in this thesis on $\sigma$-reachability, it came to our attention that work by Özveren and Willsky [OW90] uses a very similar approach for analyzing the structure of FSMs and making improved estimates for projection state-space.

Specifically, [OW90] shows that if we have some system with state space $Q$, then $Q$ can be partitioned into $n$ distinct subsets $Q_1, \ldots, Q_n$. A notion called the *persistent* part of the state-space of some FSM can be informally defined as that part of the state-space which captures the long-term behaviour of the FSM. The size of the *persistent* part of the projection state-space $Q_{p_R}$ is given by

$$|Q_{p_R}| = \sum_{i=1 \ldots n} 2^{|Q_n|}.$$

Since this method partitions the event set $Q$ into disjoint subsets, the double-counting problem which we encounter in $\sigma$-reachability (discussed in Section 3.1.3.2) is avoided.

## 3.2 Structures: Lower Limits on State-Space Explosion

We have shown in the previous sections that $\varepsilon$-geometries, $\sigma$-reachability and (in special cases) tree structures can be used to establish upper limits on the possible size of the projection state-space. We now proceed to structures which can be proven to produce at *least* a certain number of states in the state-space of the projected DES. If it is possible to efficiently identify such structures within plant models, then we would be able to modify the plant so that the structure no longer causes the problem to be considered computationally intractable.

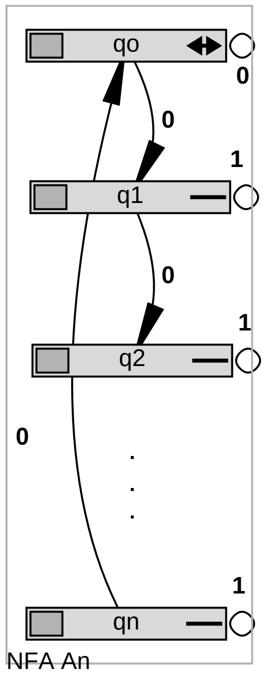


**Figure 3.4:** The cyclic NFA $A_n$

### 3.2.1 Cyclic Structures

We first present a result from [Leu93] regarding the class of automata presented in Figure 3.4. It is proven in [Leu93] that for NFA $A_n$ with states $Q$, the smallest DFA which recognizes $L_m(A_n)$ has $2^n$ states. First it is shown that for such an automaton, all the $2^Q$ states are generated using a standard subset construction method. Second, it is shown that no two different subsets of states are "equivalent" in the sense identified by the Myhill-Nerode theorem [HU79], and therefore the DFA is a minimum-state recognizer for the language.

Now, it remains to show that there exists a DFA with a nonempty set of unobservable events $\Sigma_{uo}$ such that when the unobservable events are converted to $\varepsilon$-moves, the resulting automaton recognizes the same language as the NFA in Figure 3.4. A DFA of this type is shown in Figure 3.5.

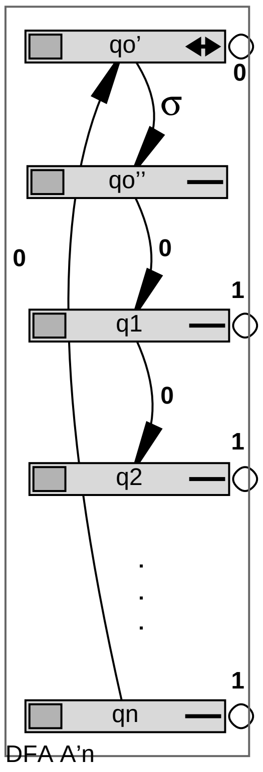


**Figure 3.5:** A DFA $A'_n$ with $\sigma \in \Sigma_{uo}$

Thus, for an NFA $A_n$ with $n+1$ states, a DFA $A'_n$ whose projection is $A_n$ has $n+2$ states. More generally, for a DFA of this type with states $Q$, we have shown that the size of the projection state-space $Q_p$ is

$$\begin{aligned} |Q_p| &= 2^{|Q|-1} - 1 \\ &= 1/2 \cdot 2^{|Q|} - 1. \end{aligned}$$

We conjecture that by choosing a slightly different type of cyclic structure (Figure 3.6) for the plant, it is possible to get exactly

$$|Q_p| = 3/4 \cdot 2^{|Q|} - 1. \tag{3.1}$$

This structure was chosen since it intuitively allows for all single-state state-sets, all double-state state-sets, etc... to be generated using Algorithm 2.4. Note that (as is the case in this example), if there exist two or more transitions between two states (for example, $\delta(q_1, 1) = q_2$, $\delta(q_1, 0) = q_2$, and $\delta(q_1, \alpha 2) = q_2$), the transitions are indicated by a single arrow, and a label containing a list of all the events (for example $0, 1, \alpha 2$) is attached to the arrow. The $n$ distinct $\alpha 1, \alpha 2, \ldots, \alpha n$ events appear to prevent DFA reduction via Algorithm 2.1.

### 3.2.2 Acyclic Structures

The results presented in the previous section suggest that particular types of cyclic structures cause computational problems when computing projections. If we exclude all those FSMs which contain cycles, we are left with *acyclic* FSMs. Formally, we define a plant $G = \{Q, \Sigma, \delta, q_o, Q_m\}$ to be *acyclic* if there does not exist $s \in \Sigma^*$ and there does not exist $q \in Q$ such that $\delta(q, s) = q$.

Tsitsiklis constructs such an acyclic type of plant in [Tsi89]. Tsitsiklis goes on to prove that a supervising agent would require an exponential number of states to keep track of all the possible states the plant could be in. The example in [Tsi89] (Figure 3.7) is constructed so that for the parameter $n$, the number of states in the plant is on the order of $n^2$, and the number of states in the projection automaton

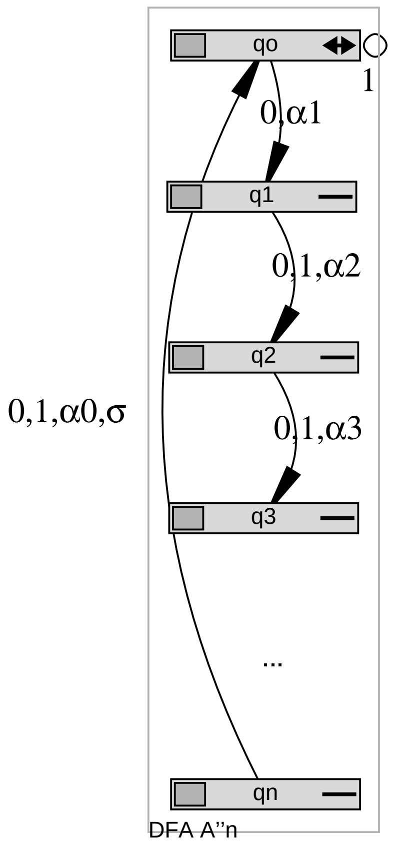


**Figure 3.6:** A modified DFA with $\sigma \in \Sigma_{uo}$

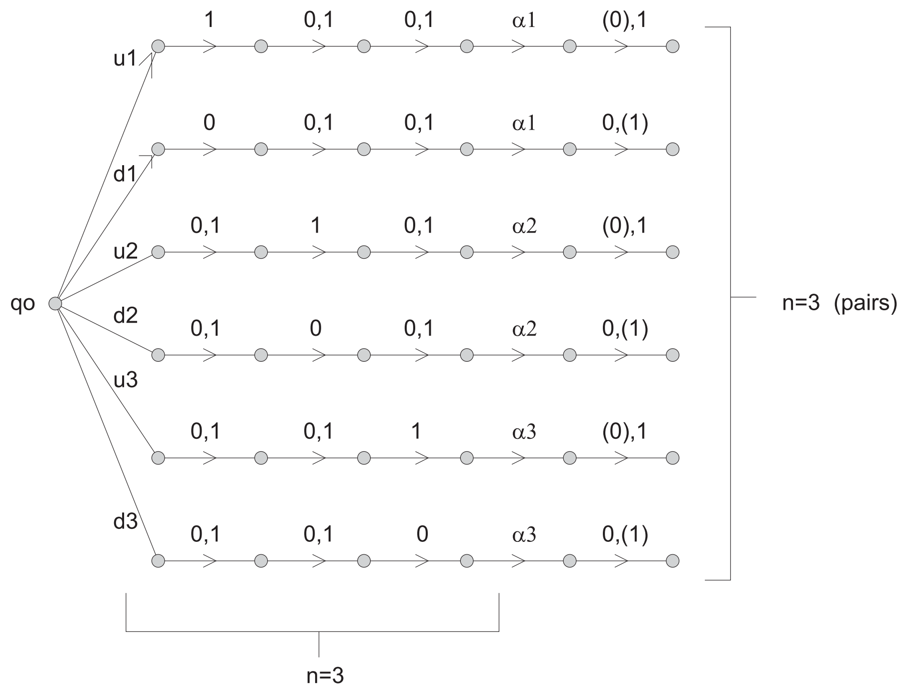


**Figure 3.7:** A "$n \times n$" construction for $n = 3$

is on the order of $2^n$. It is shown in [Tsi89] that no reduction in the size of the supervisor is possible. Note that in Figure 3.7, the transitions labeled in brackets indicate transitions which are defined in the plant, but which are not defined in the legal language.

# Chapter 4

# Software Implementation

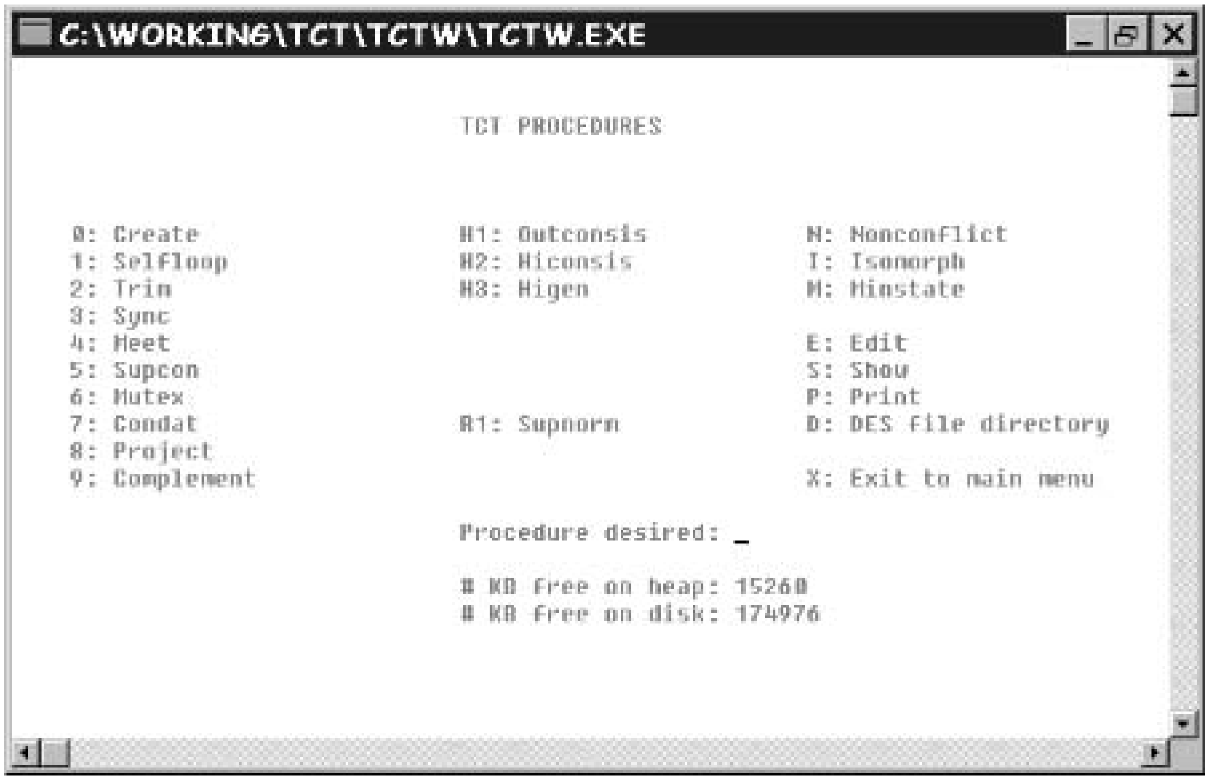
In practice, discrete-event models describing real systems may require hundreds or thousands of states. In order to effectively manipulate these large plant models in an efficient manner, we need to make use of algorithms which can be implemented as software programs. While efficient algorithms have been identified for many of the operations which are required to solve DES problems [Rud88], to the best of our knowledge there does not exist a software implementation which provides these operations in a flexible, intuitive manner.

In this chapter, we review some of the currently available DES software packages, we present a number of architectural and functional requirements for a new software implementation based on a prototype package developed to aid in the research presented in this thesis, and finally we present a series of DES functions implemented in MATLAB (a commercial software package) [Mat92] which would be the computational core of the proposed software implementation.

## 4.1 A Review of Current DES Software Tools

### 4.1.1 TCT and Object TCT

The software package TCT [Won96] represents the first DES software tool to be developed. It provides a wide variety of basic DES operations (Figure 4.1) and an interactive environment where these operations can be used. Recent development efforts have focused on making the software capable of reliably working with large DESs. One immediate drawback of the TCT software tool can be seen in Figure 4.2.

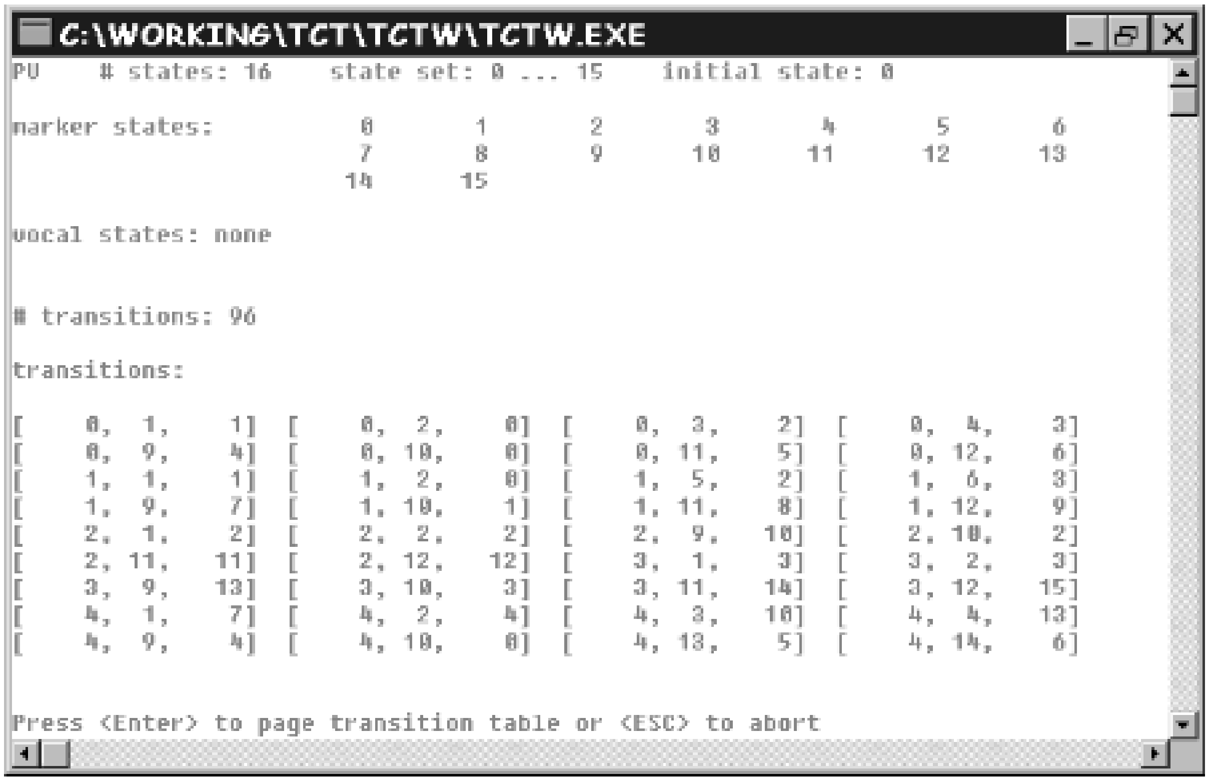


```
C:\WORKING\TCT\TCTW\TCTW.EXE

                          TCT PROCEDURES


  0: Create              H1: Outconsis          N: Nonconflict
  1: Selfloop            H2: Hiconsis           I: Isomorph
  2: Trim                H3: Higen              M: Minstate
  3: Sync
  4: Meet                                       E: Edit
  5: Supcon                                     S: Show
  6: Mutex                                      P: Print
  7: Condat              R1: Supnorm            D: DES file directory
  8: Project
  9: Complement                                 X: Exit to main menu

                         Procedure desired: _

                         # KB free on heap: 15260
                         # KB free on disk: 174976
```

**Figure 4.1:** The TCT main menu

The TCT software relies on the user to interpret lists of states and transitions which can be a time consuming process. It should be noted that all FSMs must be entered as lists of states, marker states, and transitions. Furthermore, TCT does not allow for the labeling of states or transitions. Finally, due to the design of the interactive environment, it is impossible to run scripts of commands. If the user wants to repeat a sequence of calculations (perhaps with some slight modifications to an initial plant), all the work must be done manually.

Object TCT (OTCT) [O'Y92] is a more recent DES software tool written in C++ which, while providing essentially the same DES operation functionality, is designed to process batch files which contain sequences of commands for solving particular DES problems. The OTCT software is also designed to work with DES problems with timing constraints. Unfortunately, it is still necessary to use lists of states, marker states, and transitions (representing DES plants) as input and output.



```
C:\WORKING\TCT\TCTW\TCTW.EXE
PU    # states: 16    state set: 0 ... 15    initial state: 0

marker states:          0       1       2       3       4       5       6
                        7       8       9      10      11      12      13
                       14      15

vocal states: none


# transitions: 96

transitions:

[    0,  1,    1]  [    0,  2,    0]  [    0,  3,    2]  [    0,  4,    3]
[    0,  9,    4]  [    0, 10,    0]  [    0, 11,    5]  [    0, 12,    6]
[    1,  1,    1]  [    1,  2,    0]  [    1,  5,    2]  [    1,  6,    3]
[    1,  9,    7]  [    1, 10,    1]  [    1, 11,    8]  [    1, 12,    9]
[    2,  1,    2]  [    2,  2,    2]  [    2,  9,   10]  [    2, 10,    2]
[    2, 11,   11]  [    2, 12,   12]  [    3,  1,    3]  [    3,  2,    3]
[    3,  9,   13]  [    3, 10,    3]  [    3, 11,   14]  [    3, 12,   15]
[    4,  1,    7]  [    4,  2,    4]  [    4,  3,   10]  [    4,  4,   13]
[    4,  9,    4]  [    4, 10,    0]  [    4, 13,    5]  [    4, 14,    6]


Press <Enter> to page transition table or <ESC> to abort
```

**Figure 4.2:** The TCT FSM output

### 4.1.2 StateTime

The StateTime prototype DES software toolset [Ost97], which has been designed to work with timed DES problems (i.e., timed transition models combined with a real-time temporal logic framework), provides some of the visual state descriptions which allow the user to more easily design and modify inputs, and interpret outputs. This feature has been lacking in both of the previously-discussed software packages. StateTime is designed to work with a type of Statechart [Har87] (with timing information) instead of with FSMs. Statecharts are another type of state-machine which allow for a more compact visual representation of a regular language. It does not appear that the StateTime tool currently has any capability for generating or running scripts or batch files.

## 4.2 A New Approach for DES Software Tools

After using some of the other software tools for solving example problems related to the work presented in this thesis, it quickly became apparent that a new, more flexible tool was required. A new tool should be able to process script files containing (possibly a large number of) basic DES operations. The tool should also be capable of accepting (resp., producing) DES plants as input (resp., output) in a format which is intuitive to the user—in this case, as finite state machines where the states and transitions are displayed graphically, not as lists of data.

These two new high-level DES software requirements effectively determine how the high-level implementation should be done. A core series of DES operations need to be implemented in some well-established language which is reasonably well-suited to solving mathematical problems. If this can be done properly, then this language, in conjunction with the implemented set of DES operations, would provide the required scripting environment. We present a set of functional requirements for such a DES toolkit in Section 4.2.1. We have also included matrix-based algorithms for a subset of the set of DES operations currently available in other DES software tools.

A front–end software tool also needs to be designed to create and interpret DES plant files, and to send commands to the computational engine in an interactive manner. This allows the user to immediately see and understand the structure of DES FSMs which result from using DES operations on an original set of plants. We present a set of visual requirements in Section 4.2.2 which provide a more detailed description of how such a front–end software tool should function.

### 4.2.1 Functional Requirements

In this section, we discuss in more detail what is required in the design of the core computational engine.

#### 4.2.1.1 High-Level Scripting and Batch Processing

In the existing set of DES software tools, it is difficult and time–consuming to process a large number of plants using an identical DES operation or a sequence of DES operations. Essentially, for each plant the user would be required to interactively enter the plant information, and specify the operation or sequence of operations required to process that plant. While this method is suitable for processing a small number of plants, it quickly becomes untenable in cases where the number of plants is large. Such a situation could occur where large numbers of plants are processed to obtain statistical information.

In addition to enabling large batches of plants to be processed, a design which allows for scripting enables the user to define higher-level DES procedures as required to solve specific types of problems. It is conceivable that the user could design a MATLAB procedure which solves a control problem with partial observation, where the user is asked for a specific set of input automata relating to plant and legal languages. The procedure would be composed of basic DES operations which do all the computations required to solve this type of problem.

Finally, by designing a core computational engine that accepts a scripted input, it would be possible to record information about interactive sessions in a log file which would be able to reproduce the set of calculations using only the MATLAB interpreter. This simplifies the process of recording information regarding how specific results were obtained, and of reproducing those results.

#### 4.2.1.2 Extendibility

Discrete Event Systems theory is still an expanding field. As more research is done in this area, it may be desirable to add new procedures and operations to the basic DES software tool. Also, although we only require a subset of the DES operations to be implemented in the prototype tool (i.e., only those functions required for the research presented in this thesis), we want to ensure that when more MATLAB DES operations are implemented, it will be simple to incorporate them in the prototype tool.

#### 4.2.1.3 Capability to Handle Large DES Problems

As we mentioned earlier, models of realistic industrial problems typically use hundreds or thousands of states. Thus, any DES software tool must be able to solve these larger problems in a reliable manner. In the prototype tool developed for this thesis, we took advantage of the sparse matrix functionality available in MATLAB to minimize the amount of information about an automaton which needed to be stored.

In the matrix-based implementation, transitions in automata are represented using adjacency matrices, where a "1" entry represents a transition between states (the specific states are inferred from the row and column of the entry in the matrix), and a "0" entry represents no transition between states. Since in our experience, only a small number of entries contain "1"s representing transitions, it is efficient to store only the information corresponding to "1" entries. Thus for an $n$-state machine with $\delta(q_m, \sigma) = q_{m+1}$ for $m = 1, \ldots, n-1$ being the only defined transitions, storing the whole transition matrix for the $\sigma$ transitions would require $O(n^2)$ space and would appear as

$$\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

where, for example, the transition $\delta(q_1, \sigma) = q_2$ is captured by the "1" entry in row 1 column 2 (i.e., a transition starting at the state corresponding with the row number, and terminating at the state corresponding to the column number).

Storing only those locations in the adjacency matrix which correspond to defined transitions would require only $2 \times n$ space, and would appear as

$$\begin{gathered} (q_1, q_2), \\ (q_2, q_3), \\ \vdots \\ (q_{n-1}, q_n), \end{gathered}$$

where the first entry in the ordered pair represents the state where the transition starts (the row for the "1" entry in the sparse matrix), and the second entry representing the state where the transition terminates (the column for the "1" entry in the sparse matrix). When the number of transitions is small, the space savings can be considerable.

By storing information about the transition structure of plants in the manner outlined above, and by using efficient algorithms for DES operations, we believe that the MATLAB environment will be able to process DES automata which are large enough to model complex problems.

### 4.2.2 Visual Requirements

We now present some details about the front-end software package which has been developed as a first attempt to satisfy the high–level requirement that DES automata should be able to be constructed and viewed by the user in a simple and intuitive manner.

#### 4.2.2.1 The Visual Plant

This section lists a number of desirable features which have been implemented in the prototype software tool.

**State Characteristics**

- Each state in an automaton should be able to be moved to any desired location within the workspace (by the user) so that the automaton may be presented in a readable manner.
- Each state in an automaton should be capable of being labeled in a meaningful manner. It should be possible to modify this information in a simple and direct manner.
- Each state should display information regarding its initial and marked status. It should be possible to modify this information in a simple and direct manner.

**Transition Characteristics**

- Each transition should appear as an arrow originating at a state, and terminating at (pointing to) a state.
- Each transition should be capable of being labeled in a meaningful manner. This label should correspond to an existing alphabet element. If a new label is entered, a corresponding alphabet element should be added. It should be possible to modify this information in a simple and direct manner.
- It should be possible to modify (in a simple manner) the shape of the transition line in order to make the overall automaton simpler to interpret and easier to visualize

**Plant Characteristics**

Some work has been done in the area of drawing directed graphs in an aesthetically pleasing manner, for example [GKNV93]. While the prototype DES tool which we have developed does not provide this function, the data structures have been designed in such a way so that it would be simple to add a MATLAB routine which could arrange visible states in an intuitive and understandable manner. This type of layout function would be useful specifically in cases where a DES function (PROJ, MEET or SYNC for example) generates new state-sets which have a non-trivial relationship to the state sets of their argument FSMs.

#### 4.2.2.2 FSM Interactions

In the same way that it is possible to trace how a variable is calculated in a spreadsheet, it would be useful to be able to trace how an automaton is calculated in our interactive DES software environment. Furthermore, it would be useful to be able to automatically update automata which are derived from other automata when any information regarding the input automata or the type of DES operation performed on the input automata changes. Finally, it would be useful to have a block diagram representing the relationships between all automata currently loaded in the interactive environment.

## 4.3 A Matrix-Based Implementation of DES Operations

The following subsections outline the vector and matrix data structures along with the matrix-based algorithms which were developed as part of this thesis. In order to illustrate these structures and algorithms, we use the two FSMs in Figure 4.3 as running examples. For the remainder of this section, $G_1$ refers to the FSM shown in Figure 4.3(a) and $G_2$ to the FSM shown in Figure 4.3(b).

We point out that although we chose to implement the structures and algorithms described in this subsection in MATLAB, there is no reason why they may not be implemented in other matrix-based mathematical environments. Therefore, in the

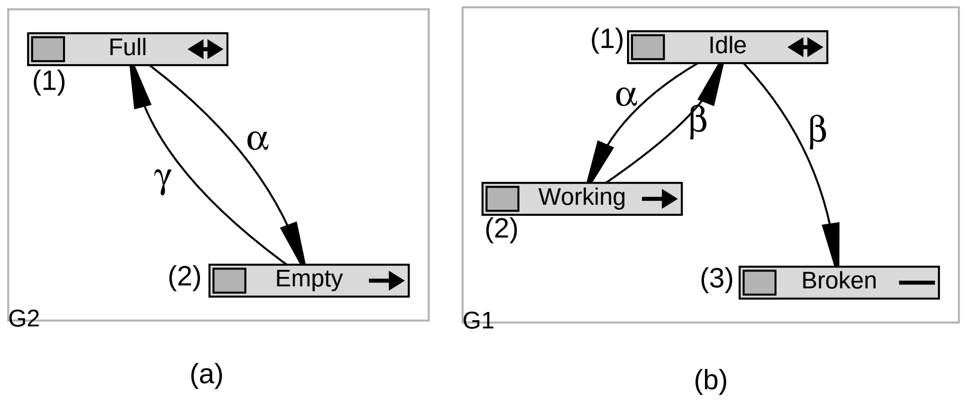


**Figure 4.3:** Two example FSMs

following subsections we focus on the matrix operations and manipulations which comprise the DES operations, and omit many of the MATLAB-specific implementation details.

Also, for clarity we use full matrix representations when describing how the various steps of the matrix algorithms apply to the example FSMs. However, in the MATLAB implementations of these algorithms, all the matrix manipulations are done using the *sparse* matrix form.

### 4.3.1 The File Format

The file format used to store DES plants is essentially a MATLAB .m file. As no MATLAB operations are performed in this .m file, the ordering of the various vectors and matrices which define the FSM is not important. Further, the front-end prototype program has also been designed such that the order of the vectors and matrices is not important. The file contains the vectors and matrices described below.

#### The Plant Name Vector

This vector contains the name of the finite state machine. This name is used as a suffix when naming all the plant vectors and matrices. Thus, for a plant named "G1", the variable would be defined as

$$PN_{G_1} = [\text{“}G1\text{”}]$$

in the plant file.

**The Stacked Transition Matrix**

The Stacked Transition Matrix is essentially an $(m \cdot n) \times n$ matrix, where $n$ is the number of states in the FSM, and $m$ is the number distinct events. Thus, the first $n \times n$ block represents the adjacency matrix for the first event in the event set $\Sigma$, the second $n \times n$ block represents the adjacency matrix for the second event, and so on. Since MATLAB does not easily store lists of information as matrix elements, and the current version does not support $n$-dimensional matrices where $n > 2$, this method of data storage was chosen as the simplest method for storing all the transition information in a single data structure. The following is an example of a stacked transition matrix for the example FSM $G_1$:

$$\left[\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right],$$

which appears in sparse matrix form as

$$\begin{aligned} TR_{G_1} &= \text{sparse}(6,3), \\ TR_{G_1}(1,2) &= 1, \\ TR_{G_1}(4,3) &= 1, \\ TR_{G_1}(5,1) &= 1. \end{aligned}$$

For this example, the first event (say $\alpha$) occurs as a transition between states 1 and 2, and the second event (say $\beta$) occurs as transitions between states 1 and 3, and states

2 and 1. Note that in the sparse matrix representation, a "1" signifies the occurrence of a transition starting at the state indicated by the row number (modulo m) and terminating at the state indicated by the column number, whereas a "0" (i.e., the matrix element is not explicitly defined in a sparse matrix) represents the absence of a transition.

### The Transition Location Matrices

The Transition Location Matrix is used to define the (x,y) screen positions of the spline points for each defined transition occurring between two states. Note that since there needs to be only one physical line/arrow to represent a number of transitions between the same start and termination states, the matrix does not need to be stacked to accommodate an adjacency matrix for each event. However, the matrix is stacked horizontally to accommodate the $x$ and $y$ location information, and can be stacked vertically to accommodate $(x, y)$ locations for splines with multiple points. Thus

$$TS_{G_1} = \begin{bmatrix} \text{[X loc. matrix, spline point 1]} & \text{[Y loc. matrix, spline point 1]} \\ \text{[X loc. matrix, spline point 2]} & \text{[Y loc. matrix, spline point 2]} \\ \vdots & \vdots \\ \text{[X loc. matrix, spline point n]} & \text{[Y loc. matrix, spline point n]} \end{bmatrix}.$$

In the example below (shown in sparse matrix form), the spline associated with the $\delta(1, \alpha) = 2$ transition has (x,y) coordinates (100,120), which correspond to the sparse matrix elements $TS_{G_1}(1, 2)$, and $TS_{G_1}(1, 5)$. Adding the elements for the other transitions in $G_1$ we get

$$\begin{aligned} TS_{G_1} &= \text{sparse}(3, 6), \\ TS_{G_1}(1, 2) &= 100, \\ TS_{G_1}(1, 5) &= 120, \\ TS_{G_1}(2, 1) &= 200, \\ TS_{G_1}(2, 4) &= 220, \\ TS_{G_1}(1, 3) &= 300, \end{aligned}$$

$$TS_{G_1}(1,6) \quad = \quad 320.$$

**The State Label Vector**

The State Label Vector is a list of names corresponding to FSM states. The ordering of the state labels corresponds to the ordering of the transitions in the transition matrices, and in the initial and marked state vectors. Thus, states indicated by the $n^{th}$ row or column of a matrix are labeled by the $n^{th}$ label in the state label vector. An example of a state label vector for the FSM $G_1$ is

$$SN_{G_1} = \begin{bmatrix} \text{“}Idle\text{”} \\ \text{“}Working\text{”} \\ \text{“}Broken\text{”} \end{bmatrix}.$$

**The Initial State Vector**

The Initial State Vector indicates the set of initial states by using a “1” at the locations corresponding to the states in the set. Note that in DES theory, there can be only one initial state, although the software makes no such restrictions. An example of this vector for FSM $G_1$ is

$$SI_{G_1} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix},$$

which indicates that the first state (state “Idle” as defined in the State Label Vector) is the initial state of the FSM.

**The Marked State Vector**

The Marked State Vector is defined in a similar manner to the initial state vector, the two differences being that this vector indicates the marker states of the FSM, and that it is possible in standard DES theory to have multiple marker states. An example of this vector for the FSM $G_1$ is

$$SM_{G_1} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix},$$

which indicates that the first and second states (states "Idle" and "Working" as defined in the State Label Vector) are both marker states of the FSM.

### The State Visibility Vector

The state visibility vector is also defined in a similar manner to the initial and marked state vectors. In this case, the vector indicates those states which are to be displayed using the front-end display program. An example of this vector for the FSM $G_1$ is

$$SV_{G_1} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix},$$

which indicates that all the states are to be displayed.

### The State Location Vector

The state location matrix is an $n \times 2$ matrix with each row containing an (x,y) location for the state which corresponds to the $n^{th}$ element of the State Label Vector. An example of this vector for the FSM $G_1$ is

$$SL_{G_1} = \begin{bmatrix} 382 & 65 \\ 309 & 141 \\ 438 & 183 \end{bmatrix},$$

which indicates that state 1 (or "Idle") is located at screen position (382,65), state 2 (or "Working") is located at screen location (309,141) and so on.

### The Alphabet Label Vector

The Alphabet Label Vector is similar to the State Label Vector in that it provides a list of labels which correspond to the occurrence of the $m^{th}$ $n \times n$ block in the stacked transition matrix. The Alphabet Controllability and Observability Matrices (defined below) also make use of column locations which correspond to this list of alphabet labels. An example of this type of vector for the FSM $G_1$ is

$$AN_{G_1} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix},$$

which indicates that the first $n \times n$ block in the stacked transition matrix corresponds to an event labeled "$\alpha$," and the second block corresponds to an event labeled "$\beta$."

## The Alphabet Controllability Matrix

For both the Alphabet Controllability and Observability Matrices, we introduce the notion of multiple supervising agents. Each supervising agent has its own view of what occurs in a system and a set of events that it can control in that system. While the data-structure defined here allows for multiple agents, the MATLAB implementation of the DES operations currently considers only those cases where a single agents is defined.

The Alphabet Controllability Matrix is a $k \times m$ matrix which contains information about the controllability of each of the alphabet elements ($m$ total) for each of the $k$ supervising agents. This matrix contains a "1" at row $i$, column $j$ if, for supervisor $i$, the $j^{th}$ alphabet element (labeled by the $j^{th}$ entry in the Alphabet Label Vector) is controllable, otherwise it contains a "0" at this location. An example of this matrix for the FSM $G_1$ is

$$AC_{G_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

which indicates that for the first supervising agent, the first alphabet element ("$\alpha$" as defined in the example alphabet label vector) is controllable, and the second alphabet element ("$\beta$") is not controllable. For the second supervising agent, "$\alpha$" is not controllable, while "$\beta$" is controllable.

**The Alphabet Observability Matrix**

The Alphabet Observability Matrix is defined in a manner analogous to the Alphabet Controllability Matrix, the difference being that "1"s in this matrix represent alphabet elements (indicated by the column) which are observable by some supervising agent (indicated by the row). Thus the example matrix

$$AO_{G_1} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

indicates that the first agent cannot observe the first alphabet element ("$\alpha$" as defined in the example alphabet label matrix) but it can observe the second alphabet element ("$\beta$"). For the second supervising agent, the opposite is true.

### 4.3.2 Basic Matrix Operations

Before we can discuss the details of the various DES matrix-based algorithms, it is necessary to define some terminology. First, let adjacency matrix $A$ represent the transitions in a FSM. By computing $A^n$, a matrix which represents the number of distinct "walks" of length $n$ between any two states can be obtained [Epp95]. A *walk* between two states $q_i$ and $q_j$ is a string $s \in \Sigma^*$ such that $\delta(q_i, s) = q_j$. If

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

then

$$A^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

which can be interpreted to mean that there is one walk of length 2 from state 1 to state 1 (in this case via state 2), and similarly one walk of length 2 from state 2 to state 2. There are no walks of length 2 between state 1 and state 2. The $A^n$ matrix can be modified so that it keeps track of accessibility, instead of counting the numbers of walks between pairs of states. First, we define the function *NORM* to be a function which replaces any non-zero matrix element with a 1. If $A$ represents the adjacency matrix for a FSM, then define

$$\begin{aligned} A_0 &= A, \\ A_n &= NORM(A^n + A_{n-1}). \end{aligned}$$

Thus, if $A_n$ has a 1 at location (i,j), then there exists a walk of length less than or equal to $n$ between states i and j.

The matrix operator "$\wedge$" is defined as follows. Given two matrices $B_1$ and $B_2$ with the same dimensions, where the entries of $B_1$ and $B_2$ are 1's or 0's, the resulting matrix $B = B_1 \wedge B_2$ has the same dimensions as matrices $B_1$ and $B_2$ with each element $B(i,j)$ being defined as the logical "and" of elements $B_1(i,j)$ and $B_2(i,j)$.

The matrix operator "$\vee$" is defined as follows. Given two matrices $C_1$ and $C_2$ with the same dimensions, where the entries of $C_1$ and $C_2$ are 1's or 0's, the resulting matrix $C = C_1 \vee C_2$ has the same dimensions as matrices $C_1$ and $C_2$ with each element $C(i,j)$ being defined as the logical "or" of elements $C_1(i,j)$ and $C_2(i,j)$.

The matrix operator "$\cdot$" is defined to be the standard operator for calculating a matrix product.

Finally, the notation $|D|$ indicates the number of entries in a vector $D$.

### 4.3.3 The TRIM Operation

A matrix-based implementation of the trim operation defined in Section 2.2.1.1 is presented in this section. We present a pseudo-code algorithm, followed by a discussion of some of the key steps in the algorithm, and conclude with a simple, illustrative example.

**Algorithm 4.1** : $G_{TRIM} = TRIM(G)$

```
1. input: $G = (PN, TR, TS, SN, SI, SM, SV, SL, AN, AC, AO)$

2. $TX = []$
```

```
3. for $i = 1, \dots, |AN|$
       $TX = TX \bigvee TR[((i-1) \cdot |SN| + 1) \dots (i \cdot |SN|), 1 \dots |SN|]$
   end

4. $R = NORM(SI \cdot TX + SI)$

5. $R_{old} = (1 \times |SI|)$ all zeros vector

6. while $R \neq R_{old}$ do
           $R_{old} = R$
           $R = NORM(R \cdot TX + R)$
   end

7. $SM = SM \bigwedge R$

8. $CR = NORM(SM \cdot TX' + SM)$

9. $CR_{old} = (1 \times |SI|)$ all zeros vector

10. while $R \neq R_{old}$ do
           $CR_{old} = CR$
           $CR = NORM(CR \cdot TX' + CR)$
    end

11. $PN_{TRIM} = PN$
    $SN_{TRIM} = SN$
    $SL_{TRIM} = SL$
    $SI_{TRIM} = SI$
    $SM_{TRIM} = SM$
    $SV_{TRIM} = SV$
    $TR_{TRIM} = TR$
    $TS_{TRIM} = TS$
    $AN_{TRIM} = AN$
    $AC_{TRIM} = AC$
    $AO_{TRIM} = AO$
```

```
12. for each nonzero entry i in $R \bigwedge CR$ do
        remove corresponding rows from $SN_{TRIM}$ and $SL_{TRIM}$
        remove corresponding columns from $SI_{TRIM}$, $SM_{TRIM}$ and $SV_{TRIM}$
        remove corresponding rows and columns from $TR_{TRIM}$ and $TS_{TRIM}$
    end

13. return $G_{TRIM} = (PN_{TRIM}, TR_{TRIM}, TS_{TRIM}, SN_{TRIM}, SI_{TRIM}, SM_{TRIM},$
    $SV_{TRIM}, SL_{TRIM}, AN_{TRIM}, AC_{TRIM}, AO_{TRIM})$
```

Algorithm 4.1 creates a FSM which recognizes the same languages as the input FSM, but which contains only those states which are both "reachable" and "co-reachable". A state $q$ is defined to be *reachable* if there exists a string $s$ such that $\delta(q_o, s) = q$. A state $q$ is defined to be *co-reachable* if there exists a string $s$ such that $\delta(q, s) = q_m$ for some marker state $q_m \in Q_m$.

In order to calculate reachable states, the matrix-based implementation of the TRIM operation needs to know only that there exists some transition between a given set of states. It does not need to know the label of the transition. For that reason, the MATLAB function creates a new $n \times n$ adjacency matrix based on the stacked $m \cdot n \times n$ transition matrix of the original DES. This matrix is the logical OR of each $n \times n$ block (representing the adjacency matrix for a single alphabet element) in the stacked transition matrix. Formally, for some arbitrary FSM with a stacked transition matrix $TR$, a matrix $TX$ is defined as follows:

$$TX = TR(1 \ldots n, 1 \ldots n) \bigvee \ldots \bigvee TR(((m-1) \cdot n + 1) \ldots (m \cdot n), 1 \ldots n).$$

This $TX$ matrix now represents an adjacency matrix where an unlabeled transition is defined between two states $q_1$ and $q_2$ if $\delta(q_1, \sigma) = q_2$ is defined for any $\sigma \in \Sigma$.

The second part of the matrix-based function for TRIM calculates FSM reachability using the $TX$ matrix. Trivially, the initial state (represented by the vector $SI$) is reachable. The algorithm then uses the $TX$ matrix to calculate the set of states, represented by vector $R$, which are reachable from the initial state via at most a single transition as follows:

$$R = \text{NORM}(SI \cdot TX + SI). \tag{4.1}$$

In general, if $TX$ is an adjacency matrix, and $SI$ is a matrix representing a subset of states (with a "1" representing the inclusion of the corresponding state in the subset), then $SI \cdot TX^n$ is a vector where the value of each element corresponds to the number of distinct paths between states in the subset represented by $SI$ and the state represented by the element of the vector $SI \cdot TX^n$. Since we are also interested in states which were already reachable (the initial state vector in this case), we add the previously reachable states to our result. Finally, as we are not concerned about how many ways a state is reachable, only that it is reachable, we use the NORM function to replace any non-zero elements of a matrix with the value "1."

In order to check for states which are reachable via strings of arbitrary length, this process is repeated, substituting the $R$ vector for the initial state vector $SI$ in the previous equation. Thus,

$$R = \mathrm{NORM}(R \cdot TX + R). \tag{4.2}$$

This is repeated until the number of non-zero entries in $R$ matrix does not increase after the application of (4.2). Note that for our simple FSM $G_1$, all the states are reached after the application of (4.1), and therefore, in this particular case, no iterations of (4.2) are required.

We have calculated all the reachable states. The function now calculates the subset of reachable states which are also co-reachable. The algorithm used to calculate this is very similar to the algorithm used to calculate reachable states. First, the function calculates the transpose of the $TX$ matrix (call it the $TX'$ matrix). This matrix represents the adjacency matrix of a directed graph where the direction of all the (observable) transitions is reversed. The function then calculates a modified marker state set by taking the logical AND of the marker state vector $SM$ and the reachable state vector $R$ (i.e., we do not care about marker states which are not reachable):

$$SM = SM \bigwedge R. \tag{4.3}$$

The initial equation used to calculate states which are co-reachable from the new $SM$ vector is

$$CR = \mathrm{NORM}(SM \cdot TX' + SM), \tag{4.4}$$

and the equation which is iterated until the co-reachability matrix $CR$ no longer increases in its number of non-zero entries is

$$CR = \mathrm{NORM}(CR \cdot TX' + CR). \tag{4.5}$$

By removing all elements in all state and transition vectors and matrices which correspond to zero entries in the $R$ and $CR$ vectors, we now have a set of states which are both reachable and co-reachable.

**Example**

To illustrate how this matrix-based algorithm works, we will use the FSM $G_1$ in Figure 4.3 as an input.

For the FSM $G_1$ (i.e., $n = 3$), the $TR$ matrix appears as

$$TR_{G_1} = \left[\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right],$$

and thus, the corresponding $TX$ matrix is:

$$\begin{aligned} TX_{G_1} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \bigvee \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \end{aligned}$$

The reachability vector $R$ is calculated for the FSM $G_1$ as follows:

$$\begin{aligned} R_{G_1} &= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}. \end{aligned}$$

Then the new $SM$ reachable marked state vector is

$$\begin{aligned} SM_{G_1} &= \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \bigwedge \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}. \end{aligned}$$

Finally, the coreachability vector $CR$ is

$$\begin{aligned} CR_{G_1} &= \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}. \end{aligned}$$

We note that as with the reachability calculation, no iterations of (4.5) are required for this example, as the number of non-zero elements in the $CR$ vector does not change after the application of (4.4). The final $CR_{G_1}$ vector tells us that only the first two states are coreachable. The final step in our example is to remove rows and columns which correspond to zero elements in the $R_{G_1} \wedge CR_{G_1}$ vector (i.e. states which are not reachable and not coreachable). Thus, for example, the modified $TR$ matrix would appear as

$$TR_{TRIM} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ \hline 0 & 0 \\ 1 & 0 \end{bmatrix},$$

and the $SI$ and $SM$ vectors would appear as

$$SI_{TRIM} = \begin{bmatrix} 1 & 0 \end{bmatrix},$$

$$SM_{TRIM} = \begin{bmatrix} 1 & 1 \end{bmatrix},$$

and finally, the resulting FSM is shown in Figure 4.4.

### 4.3.4 The MEET Operation

A matrix-based implementation of the meet operation defined in Section 2.2.1.2 is presented in this section. We present a pseudo-code algorithm, followed by a

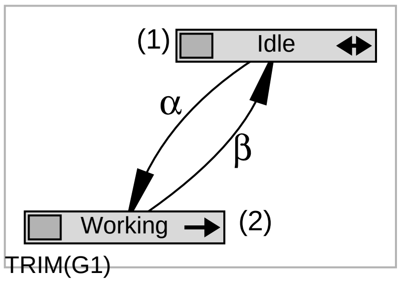


**Figure 4.4:** The TRIM of FSM $G_1$

discussion of some of the key steps in the algorithm, and conclude with a simple, illustrative example.

**Algorithm 4.2 :** $G_{MEET} = (G_1, G_2)$

```
1. input: $G_1 = (PN_{G_1}, TR_{G_1}, TS_{G_1}, SN_{G_1}, SI_{G_1},$
               $SM_{G_1}, SV_{G_1}, SL_{G_1}, AN_{G_1}, AC_{G_1}, AO_{G_1})$
          $G_2 = (PN_{G_2}, TR_{G_2}, TS_{G_2}, SN_{G_2}, SI_{G_2},$
               $SM_{G_2}, SV_{G_2}, SL_{G_2}, AN_{G_2}, AC_{G_2}, AO_{G_2})$

2. $TR_{MEET} = []$
   $TS_{MEET} = (|SN_{G_1}| \cdot |SN_{G_2}| \times 2 \cdot |SN_{G_1}| \cdot |SN_{G_2}|)$ all zeros matrix
   $SN_{MEET} = (1 \times |SN_{G_1}| \cdot |SN_{G_2}|)$ label vector
   $SI_{MEET} = []$
   $SM_{MEET} = []$
   $SV_{MEET} = []$
   $SL_{MEET} = []$
   $AN_{MEET} = []$
   $AC_{MEET} = []$
   $AO_{MEET} = []$

3. for $i = 1 \dots |SN_{G_1}|$ do
       for $j = 1 \dots |SN_{G_2}|$ do
```

```
        if $SI_{G_1}(i) = 1$ and $SI_{G_2}(j) = 1$ then
            $SI_{MEET}((i-1) \cdot |SN_{G_2}| + j) = 1$
        else
            $SI_{MEET}((i-1) \cdot |SN_{G_2}| + j) = 0$
        end
        if $SM_{G_1}(i) = 1$ and $SM_{G_2}(j) = 1$ then
            $SM_{MEET}((i-1) \cdot |SN_{G_2}| + j) = 1$
        else
            $SM_{MEET}((i-1) \cdot |SN_{G_2}| + j) = 0$
        end
    end
end

4. for each $(p, q)$ such that $AN_{G_1}(p) = AN_{G_2}(q)$ do
    $AN_{MEET} = \begin{bmatrix} AN_{MEET} \\ AN_{G_1}(p) \end{bmatrix}$
    $AC_{MEET} = \begin{bmatrix} AC_{MEET} & AC_{G_1}(p) \end{bmatrix}$
    $AO_{MEET} = \begin{bmatrix} AO_{MEET} & AO_{G_1}(p) \end{bmatrix}$
    $A_{\sigma_{G_1}} = TR_{G_1}((p-1) \cdot |SN_{G_1}| + 1 \ldots p \cdot |SN_{G_1}|, 1 \ldots |SN_{G_1}|)$
    $A_{\sigma_{G_2}} = TR_{G_2}((q-1) \cdot |SN_{G_2}| + 1 \ldots q \cdot |SN_{G_2}|, 1 \ldots |SN_{G_2}|)$
    for $i = 1 \ldots |SN_{G_1}|$ do
        for $j = 1 \ldots |SN_{G_1}|$ do
            $A_{\sigma}((i-1) \cdot |SN_{G_2}| + 1 \ldots i \cdot |SN_{G_2}|$ ,
                $(j-1) \cdot |SN_{G_2}| + 1 \ldots j \cdot |SN_{G_2}|) = A_{\sigma_{G_1}}(i, j) \cdot A_{\sigma_{G_2}}$
        end
    end
    $TR_{MEET} = \begin{bmatrix} TR_{MEET} \\ A_{\sigma} \end{bmatrix}$
end
```

*Note: $TS_{MEET}, SV_{MEET}, SN_{MEET}$, and $SL_{MEET}$ all contain information which pertains to the display of the FSM. The information contained in these matrices and vectors is not presented here. However, in the MATLAB implementation, these vectors are computed using heuristics for screen locations and labeling rules.*

5. *return* $G_{MEET} = (PN_{MEET}, TR_{MEET}, TS_{MEET}, SN_{MEET}, SI_{MEET}, SM_{MEET}, SV_{MEET}, SL_{MEET}, AN_{MEET}, AC_{MEET}, AO_{MEET})$

Algorithm 4.2 creates a FSM that recognizes a language composed of strings which are recognized by all of the the FSMs used as the arguments to the operation. Thus for some arbitrary number of FSMs where

$$G_{meet} = \text{MEET}(G_1, G_2, \ldots, G_n),$$

$G_{meet}$ recognizes only those strings, and all those strings, which are recognized by $G_1, G_2, \ldots, G_n$.

The FSM which generates the meet language as defined in (2.1) can be constructed as a $|Q_{G_1}| \times |Q_{G_2}| \times \ldots \times |Q_{G_n}|$ state machine. Since it can be shown that

$$MEET(MEET(G_1, G_2), G_3) = MEET(G_1, MEET(G_2, G_3))$$

(i.e., it is associative), we can simplify our example, without loss of generality, by considering MEET to be the meet of only two FSMs (say $G_1$ and $G_2$).

The matrix-based MEET routine builds the various FSM matrices and vectors as follows. First, the alphabet label vector and the stacked transition matrix for $G_{meet}$ are constructed. Since the language that describes the meet of two input languages contains only those elements which are contained in both the input languages, any strings in either of the two input languages which contain events which are unique to that language will not be included in the meet language. Thus, $\Sigma_{meet} = \Sigma_{G_1} \cap \Sigma_{G_2}$.

Next, the alphabets of the two FSMs are compared, and for each set of common event labels, the event label is added to the alphabet label vector $AN_{meet}$, and a new $|Q_{G_1}| \cdot |Q_{G_2}| \times |Q_{G_1}| \cdot |Q_{G_2}|$ block is added to the new stacked transition matrix

$TR_{meet}$. For some $\sigma \in \Sigma_{G_1} \bigcap \Sigma_{G_2}$, the $|Q_{G_1}| \times |Q_{G_1}|$ adjacency matrix for $\sigma$ in $G_1$ (call it $A_{\sigma_{G_1}}$), and the $|Q_{G_2}| \times |Q_{G_2}|$ adjacency matrix for $\sigma$ in $G_2$ (call it $A_{\sigma_{G_2}}$) are combined to create an adjacency matrix (call it $A_\sigma$) for the new FSM in the following manner. For $i, j = 1 \ldots |Q_{G_1}|$,

$$A_\sigma((i-1)\cdot|Q_{G_2}|+1 \ldots i\cdot|Q_{G_2}|\,,(j-1)\cdot|Q_{G_2}|+1 \ldots j\cdot|Q_{G_2}|) = A_{\sigma_{G_1}}(i,j)\cdot A_{\sigma_{G_2}}. \quad (4.6)$$

Equation (4.6) can also be described in a more graphical manner as follows

$$A_\sigma = \begin{bmatrix} A_{\sigma_{G_1}}(1,1)\cdot A_{\sigma_{G_2}} & A_{\sigma_{G_1}}(1,2)\cdot A_{\sigma_{G_2}} & \cdots & A_{\sigma_{G_1}}(1,|Q_{G_1}|)\cdot A_{\sigma_{G_2}} \\ A_{\sigma_{G_1}}(2,1)\cdot A_{\sigma_{G_2}} & A_{\sigma_{G_1}}(2,2)\cdot A_{\sigma_{G_2}} & \cdots & A_{\sigma_{G_1}}(2,|Q_{G_1}|)\cdot A_{\sigma_{G_2}} \\ \vdots & \vdots & \ddots & \vdots \\ A_{\sigma_{G_1}}(|Q_{G_1}|,1)\cdot A_{\sigma_{G_2}} & A_{\sigma_{G_1}}(|Q_{G_1}|,2)\cdot A_{\sigma_{G_2}} & \cdots & A_{\sigma_{G_1}}(|Q_{G_1}|,|Q_{G_1}|)\cdot A_{\sigma_{G_2}} \end{bmatrix}.$$

The new stacked transition matrix $TR_{meet}$ is then constructed, with each $A_\sigma$ adjacency matrix block corresponding to some $\sigma \in \Sigma_{G_1} \cap \Sigma_{G_2}$:

$$TR_{meet} = \left[\begin{array}{c} A_{\sigma_1} \\ \hline A_{\sigma_2} \\ \hline \vdots \\ \hline A_{\sigma_n} \end{array}\right]. \quad (4.7)$$

The initial state vector is constructed as follows. For each vector element in the initial state vector for $G_1$, $|Q_{G_2}|$ vector elements are added to the new initial state vector by multiplying the vector element for $G_1$ by the entire initial state vector (containing $|Q_{G_2}|$ elements) for $G_2$. Thus, a state in the meet FSM is an initial state if both the corresponding states in $G_1$ and $G_2$ are initial states.

The marked state vector is constructed using the same method that was used for constructing the initial state vector. Thus, a state in the meet FSM is a marked state if both the corresponding states in $G_1$ and $G_2$ are marked states.

The alphabet controllability and observability properties for those alphabet elements $\sigma \in \Sigma_{G_1}$ which form $\Sigma_{meet}$ are taken as the default values for the controllability and observability matrices in the new $G_{meet}$ FSM. State and transition locations are generated automatically according to a simple heuristic algorithm.

As discussed in Section 2.2.1.2, we are primarily interested in trim languages. Thus, the MEET routine includes as a final stage a call to the TRIM routine, before it returns the FSM to the user.

**Example:**

We use the two example FSMs $G_1$ and $G_2$ from Figure 4.3 as inputs to the MEET operation to illustrate how this operation works. Equations (4.6) and (4.7) are used to construct the new $TR_{meet}$ stacked transition matrix. Since $G_1$ and $G_2$ have the $\alpha$ event in common, the adjacency matrix $A_\alpha$ for $G_{meet}$ will contain non-zero elements, and is constructed using (4.6). The adjacency matrix can be thought of as a $3 \times 3$ group of $2 \times 2$ submatrix blocks:

$$A_\alpha = \left[\begin{array}{c|c|c} 2\times 2 & 2\times 2 & 2\times 2 \\ \hline 2\times 2 & 2\times 2 & 2\times 2 \\ \hline 2\times 2 & 2\times 2 & 2\times 2 \end{array}\right].$$

The $2 \times 2$ submatrix blocks are computed as follows. Given that the adjacency matrix for the $\alpha$ event in $G_2$ is

$$\begin{aligned} A_{\alpha_{G_2}} &= TR_{G_1}(1\ldots 2, 1\ldots 2) \\ &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \end{aligned}$$

and the adjacency matrix for the $\alpha$ event in $G_1$ is

$$\begin{aligned} A_{\alpha_{G_1}} &= TR_{G_1}(1\ldots 3, 1\ldots 3) \\ &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \end{aligned}$$

For each zero entry in the $A_{\alpha_{G_1}}$ adjacency matrix, an all-zero $2 \times 2$ submatrix block is inserted in the corresponding entry in the $A_\alpha$ adjacency matrix as follows:

$$A_\alpha = \begin{array}{c} (1,1) \\ (1,2) \\ (2,1) \\ (2,2) \\ (3,1) \\ (3,2) \end{array} \left[ \begin{array}{cc|c|cc} 0 & 0 & \multirow{2}{*}{$2 \times 2$} & 0 & 0 \\ 0 & 0 & & 0 & 0 \\ \hline 0 & 0 & 0 \quad 0 & 0 & 0 \\ 0 & 0 & 0 \quad 0 & 0 & 0 \\ \hline 0 & 0 & 0 \quad 0 & 0 & 0 \\ 0 & 0 & 0 \quad 0 & 0 & 0 \end{array} \right].$$

Note that for each row in the $A_\alpha$ matrix, a state-pair has been included which illustrates how the rows in the matrix correspond to the elements of the Cartesian product $Q_{G_1} \times Q_{G_2}$. A similar labeling applies to the columns of the matrix. Since the top middle entry of the $A_{\alpha_{G_1}}$ adjacency matrix is "1," then the $A_{\alpha_{G_2}}$ adjacency matrix is inserted in the corresponding top middle block of the new $A_\alpha$ adjacency matrix, completing the matrix:

$$A_\alpha = \begin{array}{c} (1,1) \\ (1,2) \\ (2,1) \\ (2,2) \\ (3,1) \\ (3,2) \end{array} \left[ \begin{array}{cc|cc|cc} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right].$$

Since in this case, the only event that $G_1$ and $G_2$ have in common is $\alpha$, then the $TR_{meet}$ stacked transition matrix is simply the $A_\alpha$ adjacency matrix

$$TR_{meet} = [A_\alpha].$$

Had $G_1$ and $G_2$ had more events in common, then the $TR_{meet}$ matrix would be a stack of all the newly calculated adjacency matrices as indicated by (4.7). We also note that in this example, after the final stage when $G_{meet}$ is trimmed, the resulting FSM shown in Figure 4.5 contains only two states.

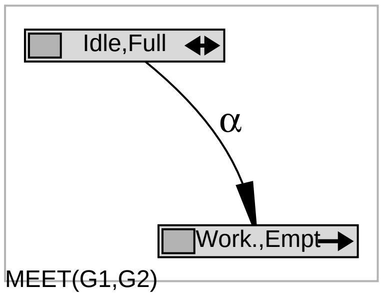


**Figure 4.5:** The MEET of FSMs $G_1$ and $G_2$

### 4.3.5 The SYNC Operation

A matrix-based implementation of the synchronous product operation defined in Section 2.2.1.3 is presented in this section. We present a pseudo-code algorithm, followed by a discussion of some of the key steps in the algorithm, and conclude with a simple, illustrative example.

**Algorithm 4.3 :** $G_{SYNC} = SYNC(G_1, G_2)$

```
1. input: $G_1 = (PN_{G_1}, TR_{G_1}, TS_{G_1}, SN_{G_1}, SI_{G_1},$
                $SM_{G_1}, SV_{G_1}, SL_{G_1}, AN_{G_1}, AC_{G_1}, AO_{G_1})$
          $G_2 = (PN_{G_2}, TR_{G_2}, TS_{G_2}, SN_{G_2}, SI_{G_2},$
                $SM_{G_2}, SV_{G_2}, SL_{G_2}, AN_{G_2}, AC_{G_2}, AO_{G_2})$

2. for each entry $\sigma$ in $AN_{G_1}$ which does not appear as an entry in $AN_{G_2}$ do
       Put 1's along the diagonal of the $A_\sigma$ adjacency matrix
       within the $TR_{G_1}$ stacked transition matrix
   end
   for each entry $\sigma$ in $AN_{G_2}$ which does not appear as an entry in $AN_{G_1}$ do
       Put 1's along the diagonal of the $A_\sigma$ adjacency matrix
       within the $TR_{G_2}$ stacked transition matrix
   end

3. calculate $G_{MEET} = MEET(G_1, G_2)$
```

*4. $G_{SYNC} = G_{MEET}$*

*5. return $G_{SYNC}$*

Algorithm 4.3 combines the MEET operation with an operation (step 2 of Algorithm 4.3) that adds event self-loops at each state of an FSM, to obtain an output FSM that synchronizes on common events, and otherwise allows for all possible interleavings of events, as defined by (2.2). For example, if two FSMs $G_1$ and $G_2$ are used as input, the output FSM $G_{SYNC}$ can be informally described as follows. If $G_1$ is at some state $q_1$, and $G_2$ is at some state $q_2$, then for state $(q_1, q_2)$ in $Q$, which corresponds to $G_1$ being in state $q_1$ and $G_2$ being in state $q_2$, then $\delta((q_1, q_2), \sigma)$ is defined if any of the following are true:

- $\delta_{G_1}(q_1, \sigma)$ is defined and $\delta_{G_2}(q_2, \sigma)$ is defined, or
- $\delta_{G_1}(q_1, \sigma)$ is defined and $\sigma \notin \Sigma_{G_2}$, or
- $\delta_{G_2}(q_2, \sigma)$ is defined and $\sigma \notin \Sigma_{G_1}$.

The matrix-based SYNC routine first goes through the event labels for $G_1$ ($\Sigma_{G_1}$), and adds self-loops of events to each state in $G_2$ if for $\sigma \in \Sigma_{G_1}, \sigma \notin \Sigma_{G_2}$ holds. It does an analogous step for each state in $G_1$. The procedure then computes the meet of these two modified FSMs, and returns it to the user.

**Example:**

We again consider the example where the two input FSMs are $G_1$ and $G_2$. In this case, a $\gamma$ self-loop event is added to $G_1$, and a $\beta$ self-loop event is added to $G_2$, resulting in the two modified automata ($G_1'$ and $G_2'$) shown in Figure 4.6.

The MEET of $G_1'$ and $G_2'$ is computed using the algorithm presented in Section 4.3.4. The resulting $A_\alpha$, $A_\beta$, and $A_\gamma$ matrices represent the adjacency matrices

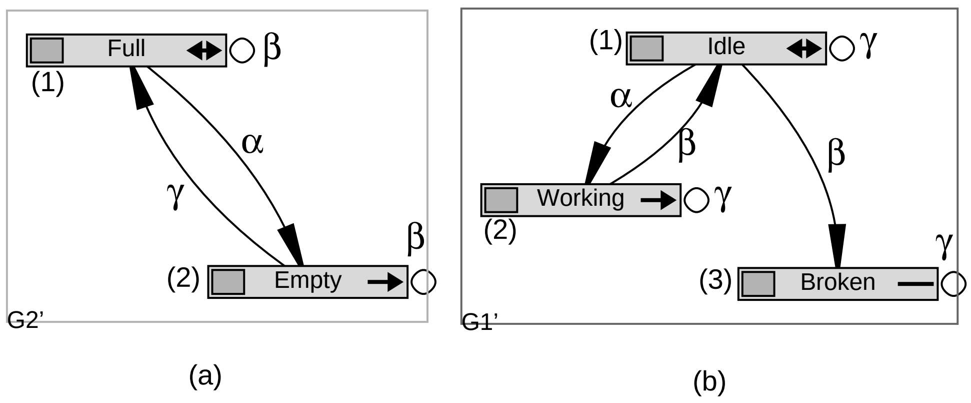


**Figure 4.6:** The example FSMs with self-loops

for $\alpha$, $\beta$ and $\gamma$ in the new FSM $G_{sync}$:

$$A_\alpha = \left[\begin{array}{cc|cc|cc} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right],$$

$$A_\beta = \left[\begin{array}{cc|cc|cc} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right],$$

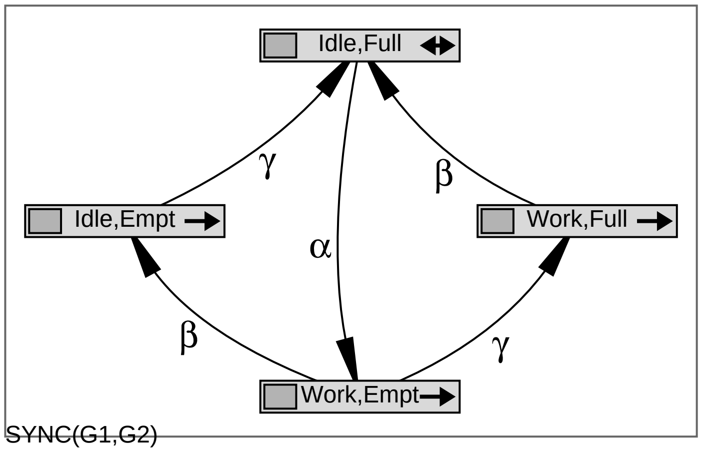


**Figure 4.7:** The SYNC of FSMs $G_1$ and $G_2$

$$A_\gamma = \left[\begin{array}{cc|cc|cc} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right].$$

Therefore, the resulting $TR_{sync}$ stacked transition matrix (before trimming) is

$$TR_{sync} = \left[\begin{array}{c} A_\alpha \\ \hline A_\beta \\ \hline A_\gamma \end{array}\right].$$

The other FSM vectors and matrices are calculated in the same manner as for the MEET operation. Note, however, that for the SYNC operation, $\Sigma_{sync} = \Sigma_{G_1} \cup \Sigma_{G_2}$. The trimmed FSM which is constructed using this procedure is shown in Figure 4.7.

### 4.3.6 The PROJ Operation

A matrix-based implementation of the projection operation defined in Algorithm 2.4 is given in this section. We present a pseudo-code algorithm, followed by a discussion of some of the key steps in the algorithm, and conclude by discussing a simple, illustrative example.

**Algorithm 4.4 :** $G_{PROJ} = PROJ(G)$

```
1. input: $G = (PN, TR, TS, SN, SI, SM, SV, SL, AN, AC, AO)$

2. $TR_{PROJ} = []$
   $TS_{PROJ} = []$
   $SN_{PROJ} = []$
   $SI_{PROJ} = []$
   $SM_{PROJ} = []$
   $SV_{PROJ} = []$
   $SL_{PROJ} = []$
   $AN_{PROJ} = []$
   $AC_{PROJ} = []$
   $AO_{PROJ} = []$

3. for $i = 1 \ldots |AN|$ do
       if $AO(i) = 1$ then
           $AO_{PROJ} = \begin{bmatrix} AO_{PROJ} & AO(i) \end{bmatrix}$
           $AC_{PROJ} = \begin{bmatrix} AC_{PROJ} & AC(i) \end{bmatrix}$
           $AN_{PROJ} = \begin{bmatrix} AN_{PROJ} \\ AN(i) \end{bmatrix}$
       end
   end

4. let $TX_{UO}$ be an all-zero $|SN| \times |SN|$ matrix
   for $i = 1 \ldots |AN|$ do
```

```
    if $AO(i) = 0$ then
      $TX_{UO} = TX_{UO} \bigvee TR(((i-1)\cdot|SN|+1)\ldots(i\cdot|SN|), 1\ldots|SN|)$
    end
  while the number of non-zero elements in $TX$ is increasing do
      $TX = NORM(TX_{UO}\cdot TX_{UO} + TX_{UO})$
  end

5. $MAP = [SI\cdot TX_{UO} + SI, \text{“}new''] $
  while there exists some row (j) labeled “new” in the $MAP$ matrix, do
      for $i = 1\ldots|AN|$ do
          if $AO(i) = 1$ then
            $S = MAP(j, 1\ldots|SN|)\cdot$
                $TR(((i-1)\cdot|SN|+1)\ldots(i\cdot|SN|), 1\ldots|SN|)\cdot$
                $TX_{UO}$
          end
          if $S \neq MAP(k, 1\ldots|SN|)$ for some $k$ then
            $MAP = \begin{bmatrix} MAP & \\ S & \text{“}new'' \end{bmatrix}$
            add a new element to the $SN_{PROJ}, SI_{PROJ}, SM_{PROJ}, SV_{PROJ}$
            and $SL_{PROJ}$ vectors
            add new rows and columns to the $TR_{PROJ}$ and $TS_{PROJ}$ matrices
          end
          change entries in the $TR_{PROJ}$ and $TS_{PROJ}$ matrices to “1”
          to reflect transitions between state sets in $G_{PROJ}$ as required
      end
      remove the “new” flag from the $j^{th}$ row of the $MAP$ matrix
  end
```

*Note: $TS_{PROJ}, SV_{PROJ}, SN_{PROJ}$, and $SL_{PROJ}$ all contain information which pertains to the display of the FSM. The information contained in these matrices* *and vectors is not presented here. However, in the MATLAB implementation, these vectors are computed using heuristics for screen locations and labeling rules.*

6. *return* $G_{PROJ} = (PN_{PROJ}, TR_{PROJ}, TS_{PROJ}, SN_{PROJ}, SI_{PROJ}, SM_{PROJ}, SV_{PROJ}, SL_{PROJ}, AN_{PROJ}, AC_{PROJ}, AO_{PROJ})$

Algorithm 4.4, which is a matrix-based implementation of Algorithm 2.4, constructs a FSM which generates the projection of the language generated by an input FSM.

First, the routine creates a $(n \times n)$ $TX_{UO}$ matrix. In this case, the $TX_{UO}$ matrix is based on the $(m \cdot n \times n)$ Stacked Transition Matrix, but includes only unobservable events. Thus, for $I = \{i \mid AO[i] = 0, i = 1 \ldots m\}$,

$$TX_{UO} = \bigvee_{\forall i \in I} TR(((i-1) \cdot n + 1) \ldots (i \cdot n), 1 \ldots n). \tag{4.8}$$

The $TX_{UO}$ matrix as defined above can be interpreted to be an adjacency matrix for any single unobservable event. What we now require is a modified adjacency matrix which accounts for strings of unobservable events. To accomplish this, we iterate the equation

$$TX_{UO} = NORM(TX_{UO} \cdot TX_{UO} + TX_{UO}) \tag{4.9}$$

until the number of non-zero elements in the $TX$ matrix stops increasing. We have now created an adjacency matrix where a transition is defined between two states if there exists a chain of unobservable events connecting the two states in the original FSM.

The PROJ routine then creates a $1 \times n$ MAP matrix that will contain subset information for all the states in the new FSM. This matrix starts as a $1 \times n$ matrix, but will grow as new states are added to a $k \times n$ matrix, with each of the $k$ rows corresponding to a state in the projection state-space. Row 1 of the MAP matrix (the only row at this stage) is defined as

$$MAP(1, 1 \ldots n) = SI \cdot TX_{UO} + SI. \tag{4.10}$$

This is equivalent to the subset of states defined by $\varepsilon$-CLOSURE($q_o$), or the set of states the input FSM could be in after the occurrence of a (possibly zero length) string of unobservable events (i.e., no observable events have yet occurred in the input FSM). This subset represents the initial state in the output FSM.

This first row of the MAP matrix is flagged as "new." The routine then enters an iterative stage. Here, for each "new" row in the MAP matrix, the routine determines what subset of states $S$ can be reached from the subset of states represented by the "new" row in the MAP matrix via each $\sigma \in \Sigma_o$ followed by a string of unobservable events. The equation is as follows:

$$S = MAP(j, 1 \ldots n) \cdot TR(((i-1) \cdot n + 1) \ldots (i \cdot n), 1 \ldots n) \cdot TX_{UO}. \tag{4.11}$$

This equation uses the subset of states represented by the $j^{th}$ ("new") row in the MAP matrix, and the observable event with an adjacency matrix represented by the $i^{th}$ $n \times n$ block in the stacked transition matrix $TR$. If the $S$ vector does not match any of the existing rows in the MAP matrix, then it is added as a new row to the MAP matrix, and is flagged as "new" as follows:

$$MAP = \begin{bmatrix} MAP & \\ S & \text{“}new'' \end{bmatrix}. \tag{4.12}$$

As the MAP matrix is being constructed, corresponding $TR$, $SN$, $SM$, $SI$, and $SL$ matrices (which represent data for the new FSM) are updated as required with new transitions and states. The states represented by the vector elements in the $SN$, $SM$, $SI$, and $SL$ vectors correspond to the rows in the MAP matrix: that is, the first row in the MAP matrix corresponds to the state represented by the first element in the $SN$, $SM$, $SI$ and $SL$ matrices (and the first row and column in each block of the the $TR$ matrix)

Although in general, it is desirable to obtain a minimum state representation of the language generated by the output FSM, this routine does not do this by default. This allows us to examine the structure and size of the resulting FSM, and make conjectures about how the structure of the input FSM impacts the size of the output FSM (before minimizing).

**Example:**

To illustrate how the PROJ matrix algorithm works, we partition the event set $\Sigma$ of $G_1$ in Figure 4.3 into $\Sigma_{uo} = \{\alpha\}$ and $\Sigma_o = \{\beta\}$. The $TX$ matrix resulting from (4.8) and (4.9) is

$$TX = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

and the first row for the MAP matrix as defined in (4.10) is

$$\begin{aligned} MAP &= SI_{G_1} \cdot TX \\ &= \begin{bmatrix} 1 & 1 & 0 & \text{New} \end{bmatrix}. \end{aligned}$$

Now, we iteratively apply (4.11) to each "new" row in the MAP matrix. For the first iteration, we consider the first row of the MAP matrix, and the $\beta$ event. The $\beta$ event maps the first state to the third state, and the second state to the first state. Note, however that it is possible to reach the second state from the first state via a string (namely $\alpha$) of unobservable events. Thus, $\beta$ also, in effect maps the second state back to the second state. Therefore, the MAP matrix becomes

$$MAP = \begin{bmatrix} 1 & 1 & 0 & \\ 1 & 1 & 1 & \text{New} \end{bmatrix}.$$

We now consider the second row of the $MAP$ matrix, which contains the only "new" flag. After applying (4.11) to the row, no new rows are added to the $MAP$ matrix. We can therefore conclude that the only two state-sets which make up the projection state space are {Idle,Working} and {Idle,Working,Broken}. The adjacency matrices for each of the observable events are constructed as the $MAP$ matrix grows. Thus, if at some stage during the procedure, the $MAP$ matrix contains n rows, then for each observable event, there exists an $n \times n$ adjacency matrix. The FSM which recognizes the projection of $L(G_1)$ is shown in Figure 4.8. For simplicity, we have renamed the state "{Idle,Working}" as "1", and the state "{Idle,Working,Broken}" as "2".

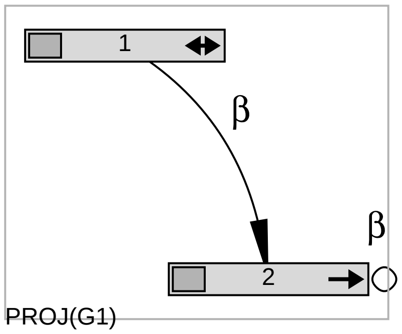


**Figure 4.8:** The PROJ of FSM $G_1$

### 4.3.7 The MINI Operation

The matrix-based MINI operation presented here implements Algorithm 2.1. The algorithm takes an input DFA, and provides an output DFA which recognizes the same language, but which contains the minimum number of states required to recognize that language. The matrix-based implementation of this algorithm has been developed to work with matrix representations of FSMs.

First, the MINI routine constructs a three-column matrix FLAG. The first two columns of each row of this matrix contain unique $i, j$ pairs, $i \neq j$, where $i, j$ represent distinct states in the FSM. Thus, there are as many rows as there are combinations of two distinct states in the FSM (specifically: $(n \cdot (n-1)/2)$ rows). The entry in the third column of each row (with entries $i, j$ in the first two columns) is defined using the Marked State Vector SM as follows:

$$\text{Column 3 entry} = 1 \text{ if } SM(i) \neq SM(j), \text{ or}$$
$$\text{Column 3 entry} = 0 \text{ if } SM(i) = SM(j).$$

The routine then goes through each row of the FLAG matrix, and for each row containing a zero in the third column, it finds the state-pair $(q_{ii}, q_{jj})$ where $q_{ii} = \delta(q_i, \sigma)$ and $q_{jj} = \delta(q_j, \sigma)$, $q_i \neq q_j$, $\sigma \in \Sigma$, and $q_i, q_j$ corresponding to $i, j$ in the first two columns of the FLAG matrix. If the row in the FLAG matrix which corresponds to the state pair $(q_{ii}, q_{jj})$ has a “1” in the third column, then the routine enters a “1” in the third column of the row corresponding to the $(q_i, q_j)$ state pair. This process is iterated until no new 1’s are entered in the third column of any row in the FLAG matrix.

Finally, those pairs of states which have not been “flagged” in the above iterative process, are considered to be equivalent states. The matrix-based routine therefore combines these states, and outputs a minimum DFA to the user.

It should be noted that this implementation of the Myhill-Nerode theorem, while fairly simple to code in MATLAB, is not the most computationally-efficient way to calculate the minimum DFA [Hop71]. Specifically, Algorithm 2.1 (step 2, line 6) uses recursion on lists to efficiently flag unflagged pairs. In contrast, our matrix-based routine cycles through the list of state pairs to test, and in some cases flag, unflaggeded pairs. This cycle continues until a complete test of all the state pairs is done with no further flagging.

# Chapter 5

# Examples

Before we present a series of example, we first need to define the two types of figures which are used to present some of our results. First, we display a form of adjacency matrix which illustrates how states map to other states after the occurrence of an observable event $\sigma$, followed by a string of unobservable events. Both the x-axis and y-axis represent the set of states in these matrices.

We also use an $n\sigma$-reachability matrix (sometimes referred to as a summary matrix), where nonzero (i.e., dotted) elements in the matrix represent occurrences of strings of length $n$ in the plant. Thus, while the y-axis still represents the set of states, the x-axis represents all possible strings of observable events of length $n$.

## 5.1 The Two-Train Problem

For our first example, we chose the simple problem where two trains must share a common length of track [RW89], [Won96]. In this problem, parts of the track have sensors which can detect the passage of the trains, and parts of the track have stop lights which may prevent the trains from entering the following sections of track (Figure 5.1).

The plant language can be modeled by taking the synchronous product of two finite state machines ($G_{V1}$ and $G_{V2}$, each representing the behaviour of a train) which are provided in Figure 5.2. Let $V$ be the language recognized by the resulting automaton $G_V$ $G_V$ recognizes

$$G_V = \text{SYNC}(V1, V2).$$

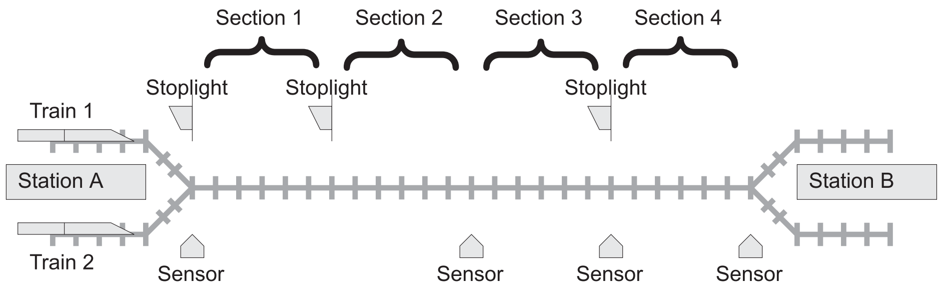


**Figure 5.1:** A block diagram of the two-train problem

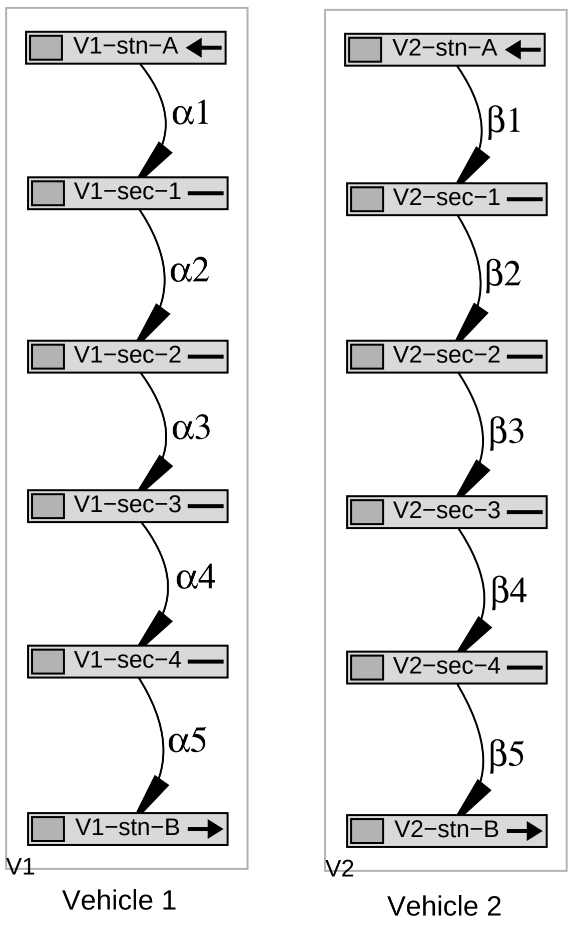


**Figure 5.2:** The component models for the two-train problem

In controlling this system, we require that the two trains not occupy the same segment of track at the same time, thus the legal language is defined as the language $E$ recognized by the automaton $G_V$ after the states $(1,1), (2,2), (3,3)$, and $(4,4)$ have been removed together with all transitions leading into and out of the removed states. This provides a FSM which recognizes a language which does not include strings corresponding to train movements which result in the two trains occupying the same section of track at the same time.

Now, as Figure 5.1 indicates, there is no sensor before section 2 of the track. This means that the unobservable event set is $\Sigma_{uo} = \{\alpha_2, \beta_2\}$. As part of the solution to the control problem, it is useful to take the projection of all those strings which are considered to be illegal. The language $V - E$ represents all the possible strings of events in the plant minus the legal strings, leaving only those strings which are illegal. An automaton $G_{V-E}$ which recognizes the language $V - E$ can be constructed, so that the FSM which recognizes $P(V - E)$ can be calculated:

$$G_p = p(G_{V-E}).$$

As $G_{V-E}$ has 56 states, it is possible that the FSM which generates the projection of $V - E$ could have on the order of $2^{56}$ states.

### 5.1.1 $\sigma$-Reachability Analysis Results

Table 5.1 summarizes the results obtained for the $\sigma$-reachability analysis of the two-train problem. It is interesting to note that while in the worst case, the size of the state-space could be on the order of $10^{16}$, even a $1\sigma$-reachability test indicates that due to the structure of the problem, the upper limit is no greater than 20225. A $3\sigma$-reachability test further reduces this upper limit to 1600 states.

Although $\sigma$-reachability analysis does not allow us to strictly do better than make exponential predictions about the size of the state-space of the projection FSM, the structure of this problem allows us to improve our state-space estimate significantly. Specifically, by using the number and type of transitions coupled with the number

**Table 5.1:** $\sigma$-reachability results for the two-train problem

| | $1\sigma$-reach. | $2\sigma$-reach. | $3\sigma$-reach. | $4\sigma$-reach. | $5\sigma$-reach. |
|---|---|---|---|---|---|
| State Estimate | 20225 | 1839 | 1600 | 4980 | 37659 |
| Max. Subset | 13 | 9 | 9 | 6 | 4 |
| Complexity | $O(\|\Sigma_o\|)$ | $O(\|\Sigma_o\|^2)$ | $O(\|\Sigma_o\|^3)$ | $O(\|\Sigma_o\|^4)$ | $O(\|\Sigma_o\|^5)$ |

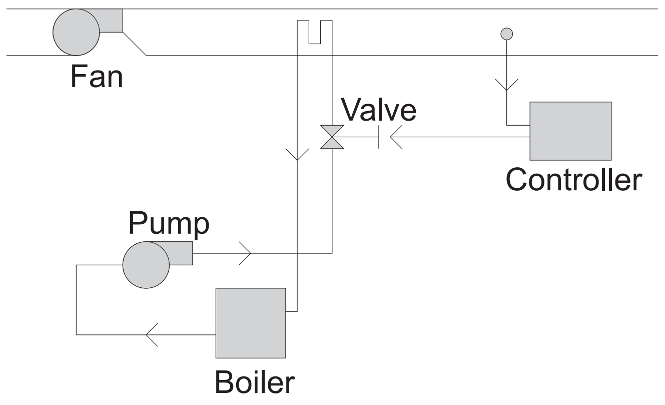


**Figure 5.3:** A block diagram of the HVAC system

of states, instead of simply using the number of states, as the parameter for the exponential estimate we can reduce our state-space estimate by a factor of $2^{43}$. Thus, the $1\sigma$-reachability test indicates that the size of the projection state-space is of order $2^{13}$ versus the state-space of the FSM which by itself indicates that the size of the projection state-space could be of order $2^{56}$. In this case, the $\sigma$-reachability test has reduced the exponent by a factor of 4.

## 5.2 An HVAC System

A heating, ventilation and air-conditioning (HVAC) DES diagnosability problem from [SSL+96] (Figure 5.3) was chosen as an example of a system where a large number of the transitions occurring in the FSM are unobservable. In this case, 118 of 228, or more than 50% of the transitions in the 90-state FSM are unobservable.

In our analysis of the problem, we assume that all the failure events are strictly

**Table 5.2:** $\sigma$-reachability results for the HVAC system

| | $1\sigma$-reach. | $2\sigma$-reach. | $3\sigma$-reach. | $4\sigma$-reach. | $5\sigma$-reach. |
|---|---|---|---|---|---|
| State Est. | 528897 | 269921 | 272966 | 808786 | 913899 |
| Max. Subset | 18 | 18 | 18 | 18 | 18 |
| Complexity | $O(\lvert\Sigma_o\rvert)$ | $O(\lvert\Sigma_o\rvert^2)$ | $O(\lvert\Sigma_o\rvert^3)$ | $O(\lvert\Sigma_o\rvert^4)$ | $O(\lvert\Sigma_o\rvert^5)$ |

unobservable, and that all other events are observable. For simplicity, we do not make use of additional sensors which are used in [SSL+96] when checking for diagnosability.

The plant can be modeled by computing the synchronous product of six component FSMs. These component FSMs represent models of a Pump, a Valve, a Fan, a Boiler, a Load, and a Controller (Figure 5.4(a)–(f), respectively). By taking the synchronous product of these components, a 90-state, 228-transition FSM is obtained. We then note that the set of observable events is

$$\Sigma_o = \{\text{PON,POFF,FON,FOFF,OV,CV,BON,BOFF,SPD,SPI}\},$$

with a combined total of 110 transitions in the composed 90-state FSM, and the set of unobservable failure events is

$$\Sigma_{uo} = \{\text{PFON1,PFON2,PFOFF1,PFOFF2,SC1,SC2,SO1,SO2}\},$$

with a combined total of 118 transitions in the composed 90-state FSM.

## 5.2.1 $\sigma$-Reachability Analysis Results

Table 5.2 summarizes the results obtained for the $\sigma$-reachability analysis of the HVAC problem. In this example, while the best results are obtained using a $2\sigma$-reachability test, the maximum subset size does not reduce past the initial $1\sigma$-reachability value of 18. Thus, a simple $1\sigma$-reachability test results in a reduction by a factor of five of the exponent ($2^{18}$ versus $3/4 \cdot 2^{90}$) used to estimate the size of the projection state-space.

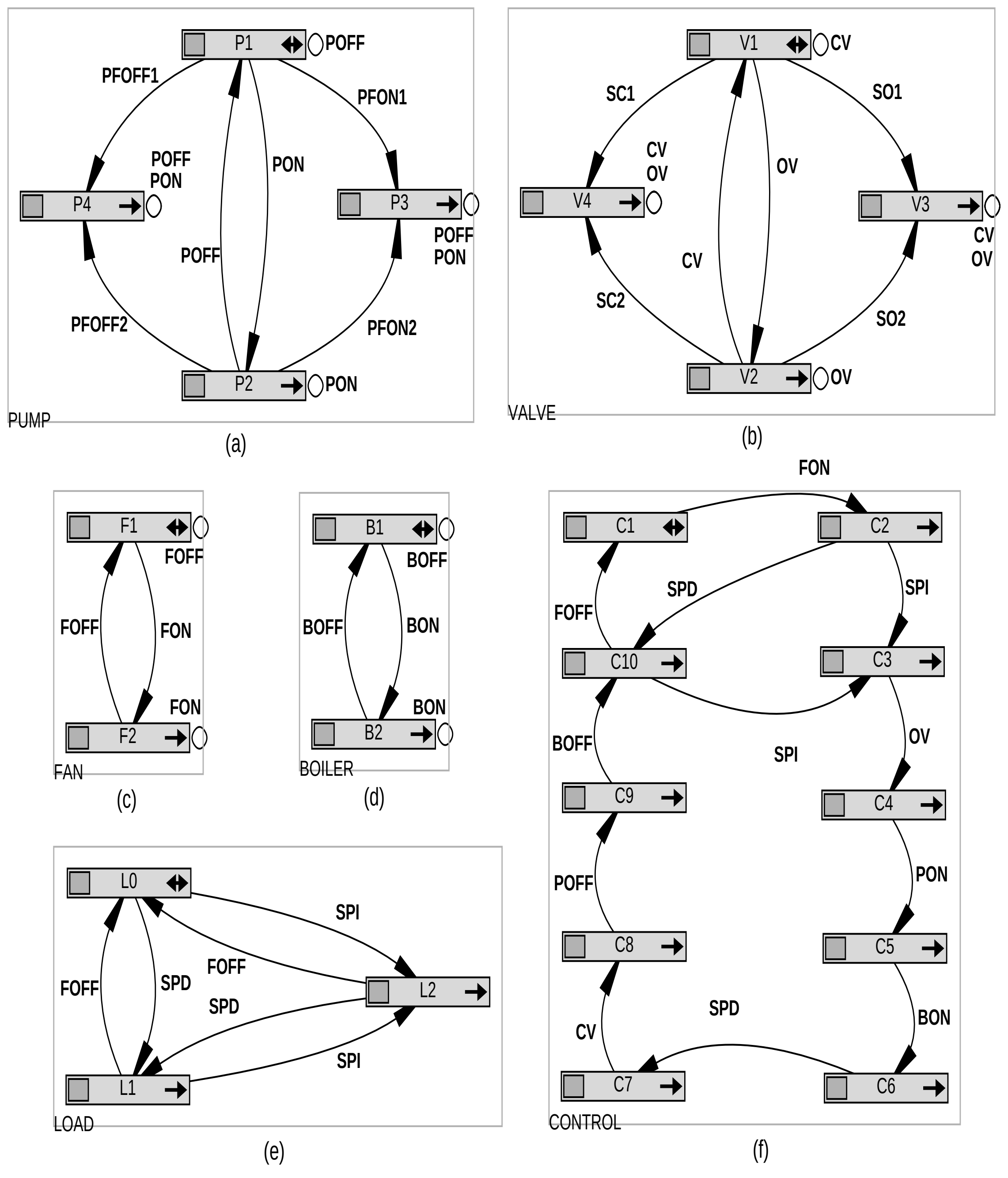


**Figure 5.4:** The component models for the HVAC system

It is also of interest to look at the matrix structure for some of the transition adjacency matrices and some of the summary matrices. Figure 5.5 shows the adjacency matrix representing a NFA containing all the unobservable events as defined in the HVAC problem, plus additional unobservable transitions between states as follows:

$$\delta(q_1, \varepsilon) = q_2 \text{ if } \exists s \in \Sigma_{uo}^* \text{ such that } \delta(q_1, s) = q_2.$$

The adjacency matrix can be interpreted to mean the following. For some state represented by a matrix row, the adjacency matrix shows all the states (represented by the matrix columns) which can be reached by some string of unobservable events. For example, if there is a dot in row 1, column 6 and in row 1, column 8, then there are two strings of unobservable events starting at state 1, with one string leading to state 6 and the other to state 8.

Figure 5.6 summarizes the data obtained while doing the $1\sigma$-reachability test. The matrix rows correspond to states in the FSM in the usual manner. Each column however corresponds to a unique $\sigma \in \Sigma_o$. Thus, the matrix can be interpreted to mean that after observing some event $\sigma$ (followed by some string of unobservable events), the system can be in at most some (not necessarily strict) subset of the states corresponding to the matrix rows containing dots. We can see from Figure 5.6 that the FON event (corresponding to matrix column 1) results in the system being in at most a subset of 18 states in the 42-65 row range, whereas the FOFF event (corresponding to matrix column 2) results in the system being in a subset of 9 states in the 78-90 row range. In fact, since the FON event maps to a subset of 18 events, it is the event which bounds the $1\sigma$-reachability estimate (i.e., since there are no other events which map to more than 18 states, then FON is one of the events which provides the $O(2^{18})$ value).

We have also included the $2\sigma$-reachability matrix (Figure 5.7), which is interpreted the same way as was the $1\sigma$-reachability matrix, with the exception being that each column now corresponds to some unique group of two observable events (i.e., FON,FON or FOFF,CV, etc...). It is interesting to note here that while each of the ten observable events mapped to some nonempty subset of states in Figure 5.6,

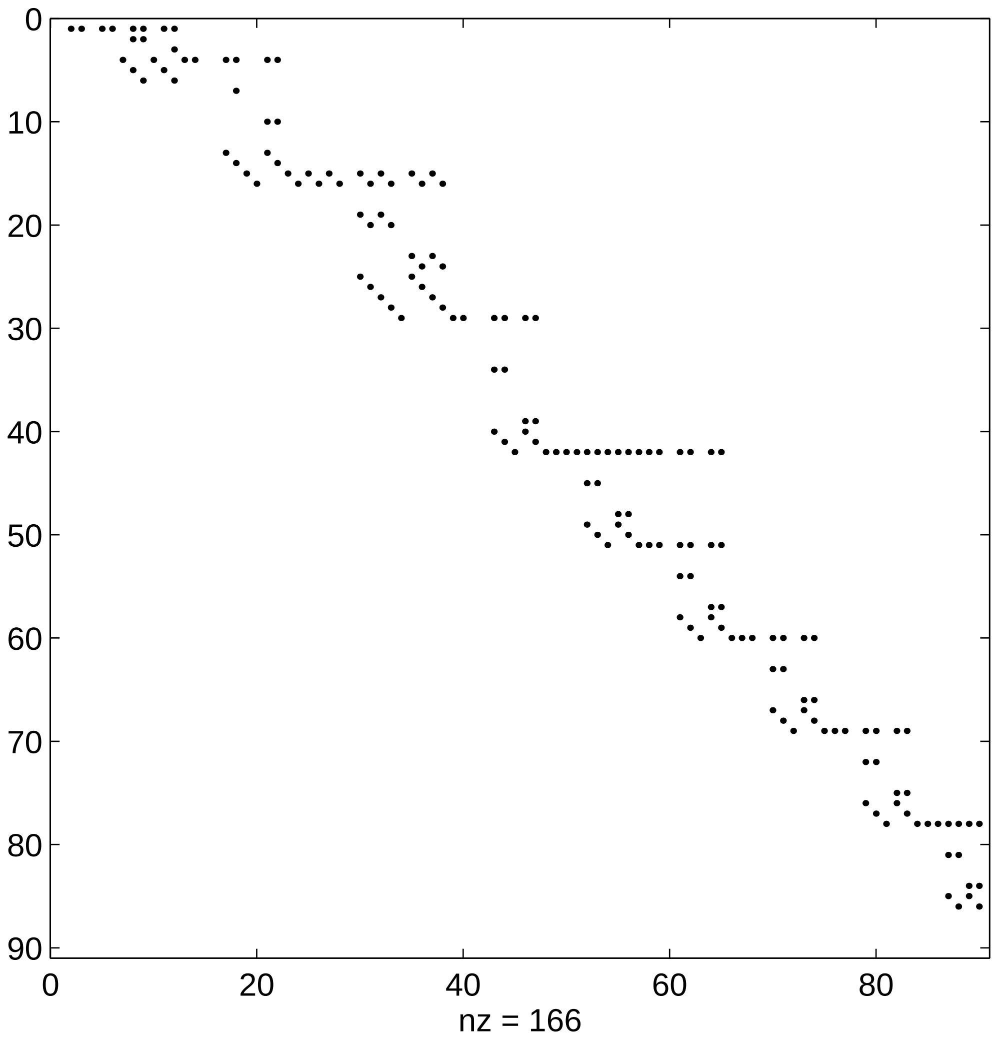


**Figure 5.5:** Matrix structure for the unobservable events

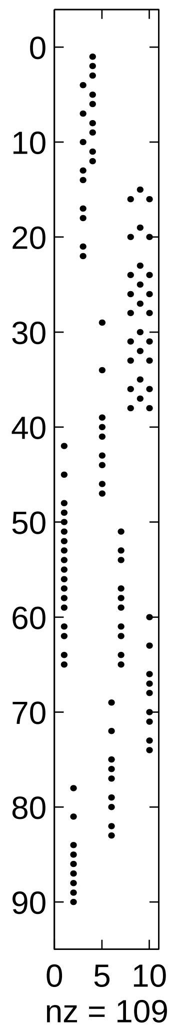


**Figure 5.6:** Matrix structure for the $1\sigma$-reachability matrix

only 16 of a possible 100 groups of double events map to nonempty subsets of states. Unfortunately, the largest of these subsets is still 18 states, and thus, no significant improvement can be expected over the $1\sigma$-reachability projection state-space size estimate. Indeed, this maximum subset size does not drop before the state-space estimate starts increasing due to the double counting effect discussed in Section 3.1.3.2.

Finally, the adjacency matrix for the FON event (Figure 5.8) has been included to illustrate how each event maps to a small subset of states in this system. The adjacency matrix for the other observable events are similar in structure to this matrix.

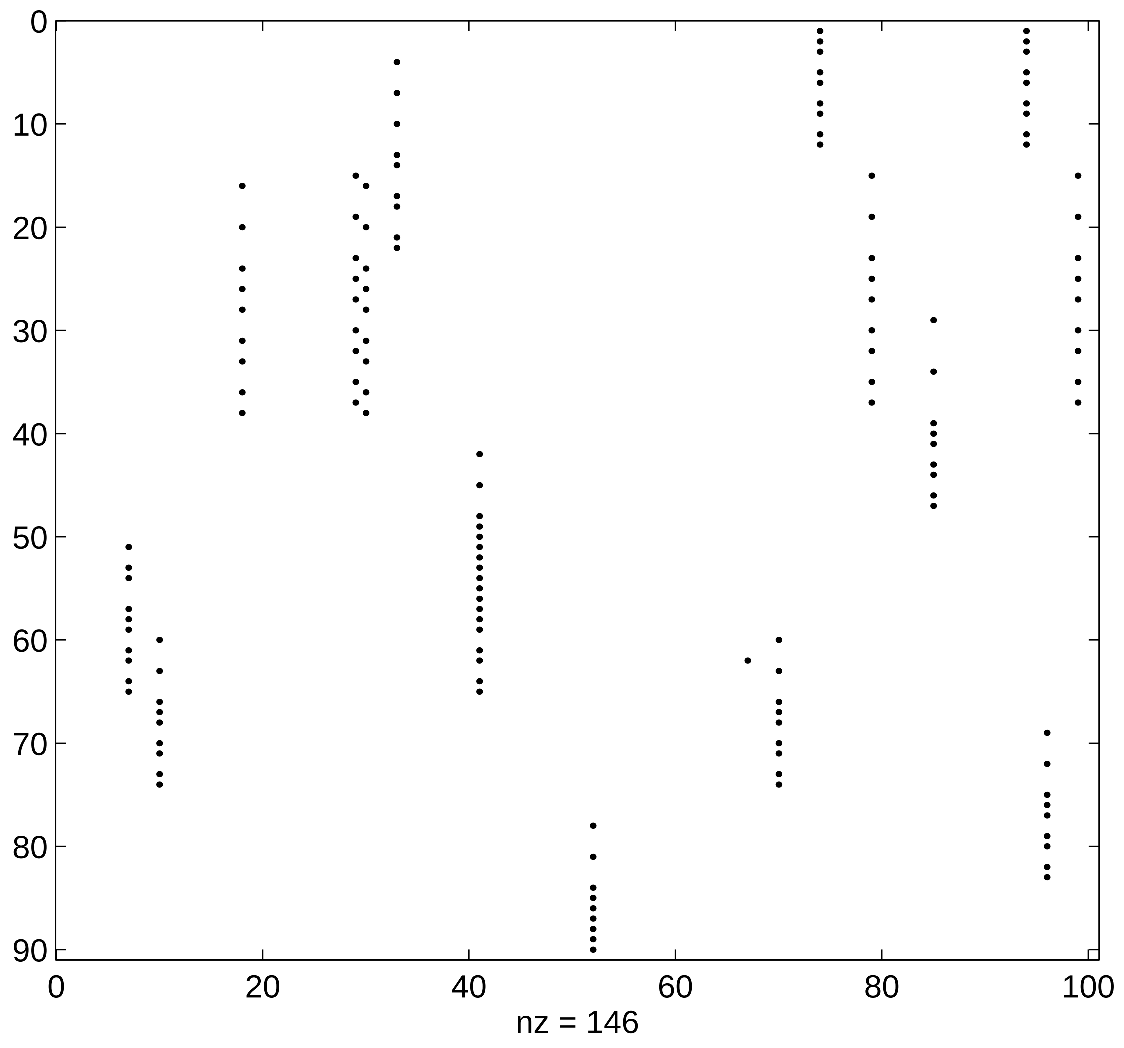


**Figure 5.7:** Matrix structure for the $2\sigma$-reachability matrix

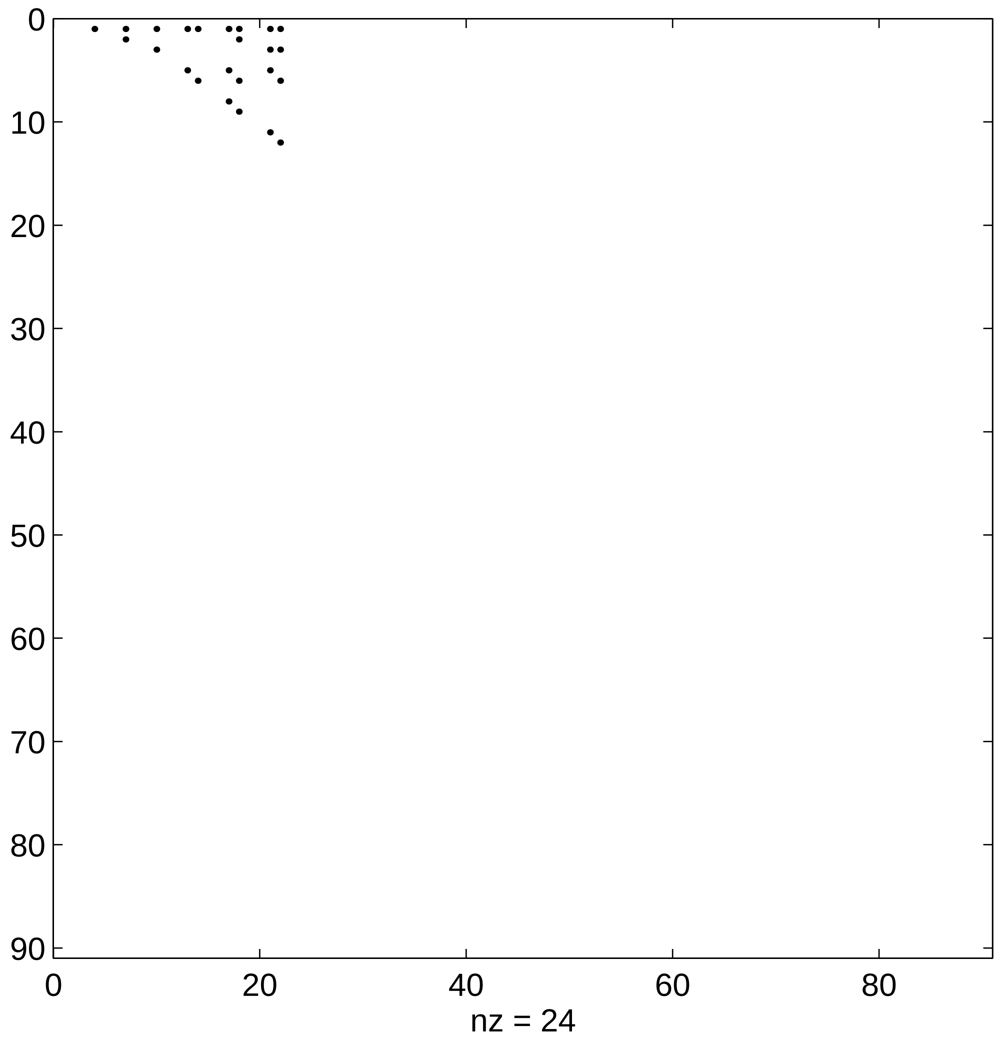


**Figure 5.8:** The adjacency matrix for the FON event

Note that for this example, there is a significant amount of nondeterminism due to strings of unobservable events following the FON event.

## 5.3 The Tsitsiklis Problem

It has been proven [Tsi89] that building supervisors for partially-observable systems can be computationally intractable. In devising the proof for this, an arbitrarily large DFA (Figure 3.7) parameterized by n is constructed, with an unobservable event set $\Sigma_{uo} = \{u1, d1, u2, d2, \ldots, un, dn\}$. The control problem in this example is to disable the events in brackets in Figure 3.7. In order to do this, the supervisor must remember the sequence of 1's and 0's which have occurred. Based on which $\alpha_i$ event it observes, the supervisor then must examine the $i^{th}$ event. If the $i^{th}$ event is a 1 then the supervisor must disable 0 otherwise it must disable 1.

When larger versions of the FSM shown in Figure 3.7 are constructed, the size of the state-space of the FSM grows with $n^2$. However, as the supervisor must at each stage remember the sequence of 1's and 0's which have occurred, the constructed supervisor must be of order $2^n$. This example, while contrived, is of significant interest to us because it provides a scalable example of a case where $\sigma$-reachability does not improve the estimate of the size of the projected state-space significantly.

The 37-state, 54-transition example in Figure 3.7 is constructed by scaling the general problem to $n = 3$. For this example, the number of states in the FSM is $2 \cdot n^2 + 6 \cdot n + 1$ or $O(n^2)$. The proof found in [Tsi89] shows that the size of a supervisor for such a system is $O(2^n)$. Since there are $2 \times n^2 + n$ occurrences of 1 transitions and the same number of occurrences of 0 transitions, the $1\sigma$-reachability test will always return two sets of size $2 \times n^2 + n$. Thus, the $1\sigma$-reachability estimate grows exponentially worse as $n$ increases. We conjecture that the $n\sigma$-reachability tests will also produce estimates which grow exponentially with $n$.

While [Tsi89] shows that the size of a supervisor must be $O(2^n)$, $1\sigma$-reachability analysis shows that the size of the projection state space of the system (upon which the supervisor is based) could be as high as $O(2^{2 \times n^2 + n})$. Thus, it is clear that in this type of example where the number of occurrences of a specific observable event is high, $\sigma$-reachability does not offer any improvements to state-space size estimates.

### 5.3.1 $\sigma$-Reachability Analysis Results

Table 5.3 summarizes the results obtained for the $\sigma$-reachability analysis of this example. It is interesting to note that significant improvements on the size estimates are made up to and including the $3\sigma$-reachability test. We conjecture that due to the

**Table 5.3:** $\sigma$-reachability results for the Tsitsiklis problem

| | $1\sigma$-reach. | $2\sigma$-reach. | $3\sigma$-reach. | $4\sigma$-reach. |
|---|---|---|---|---|
| State Estimate | 4194317 | 1083 | 260 | 5220 |
| Max. Subset | 21 | 8 | 3 | 2 |
| Complexity | $O(\|\Sigma_o\|)$ | $O(\|\Sigma_o\|^2)$ | $O(\|\Sigma_o\|^3)$ | $O(\|\Sigma_o\|^4)$ |

nature of the construction, the best estimate will occur at approximately the $n^{th}\sigma$-reachability test (with some correction for the double-counting effect), where $n$ is the parameter used for the construction of the system.

Figure 5.9 shows how both the 1 and 0 events (corresponding to the first two columns in the matrix) map to a large subsets of states. This corresponds to the large number of occurrences of 1's and 0's in the system. This matrix also shows how each of the $\alpha_i$-events (corresponding to the last 3 columns of the matrix) maps to exactly 2 states, as expected. In addition, the adjacency matrix presented in Figure 5.10 illustrates how the "1" event is defined for a large subset of states, compared to Figure 5.8 in the HVAC example, where the non-zero elements of the adjacency matrix are more localized. It is also of interest to note that unlike the HVAC example, no nondeterminism has been introduced in this matrix, as there are no places where a "1" can occur followed by a string composed of unobservable events.

## 5.4 A 10-state Cycle Problem

The somewhat contrived example DES model shown in Figure 5.11 is an instance of the class of FSMs presented in Figure 3.6 for n=9. It has been included in this section as an example of a case where $\sigma$-reachability offers no improvement over the single $\varepsilon$-transition geometry result presented in Section 3.1.1.1.

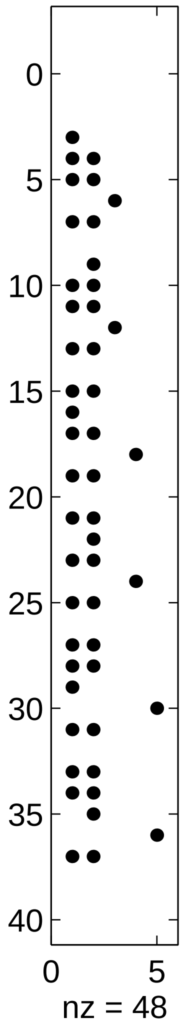


**Figure 5.9:** Matrix structure for the $1\sigma$-reachability matrix

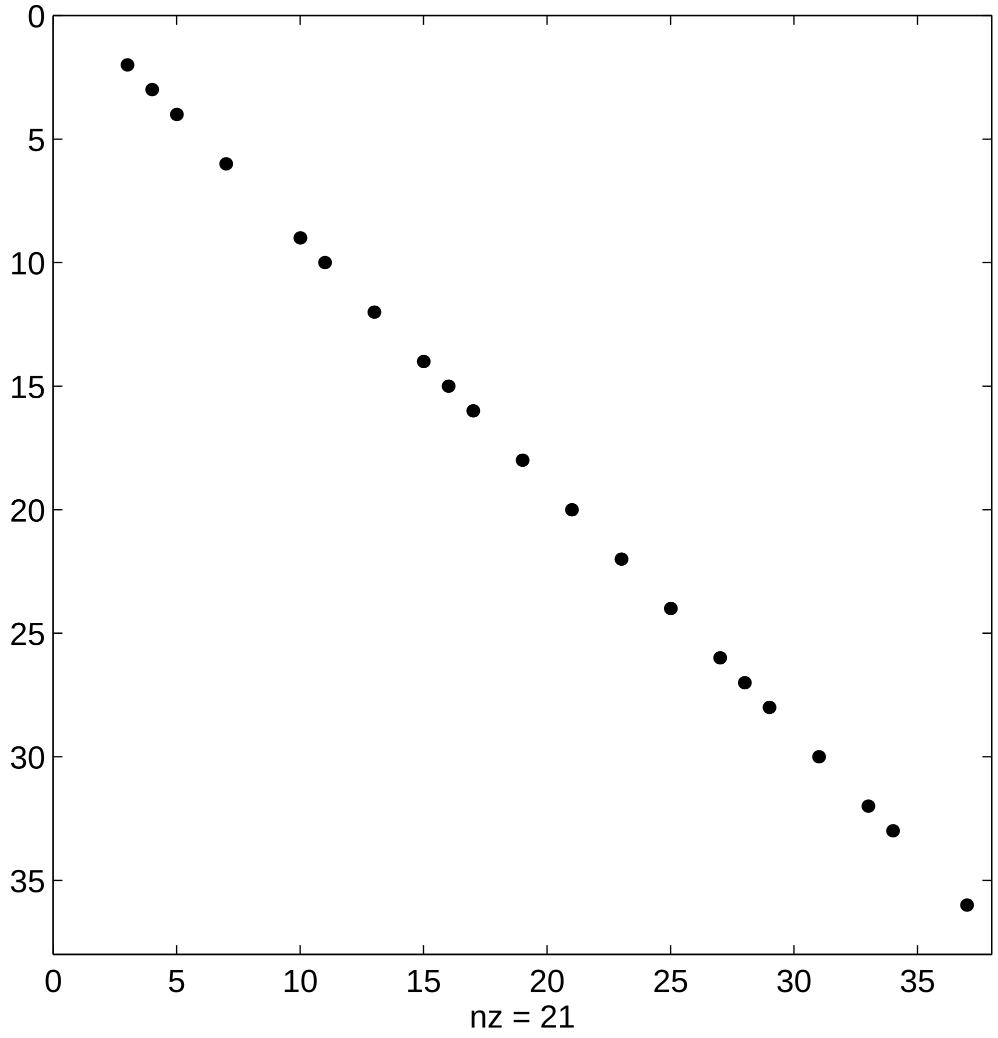


**Figure 5.10:** The adjacency matrix for the "1" event

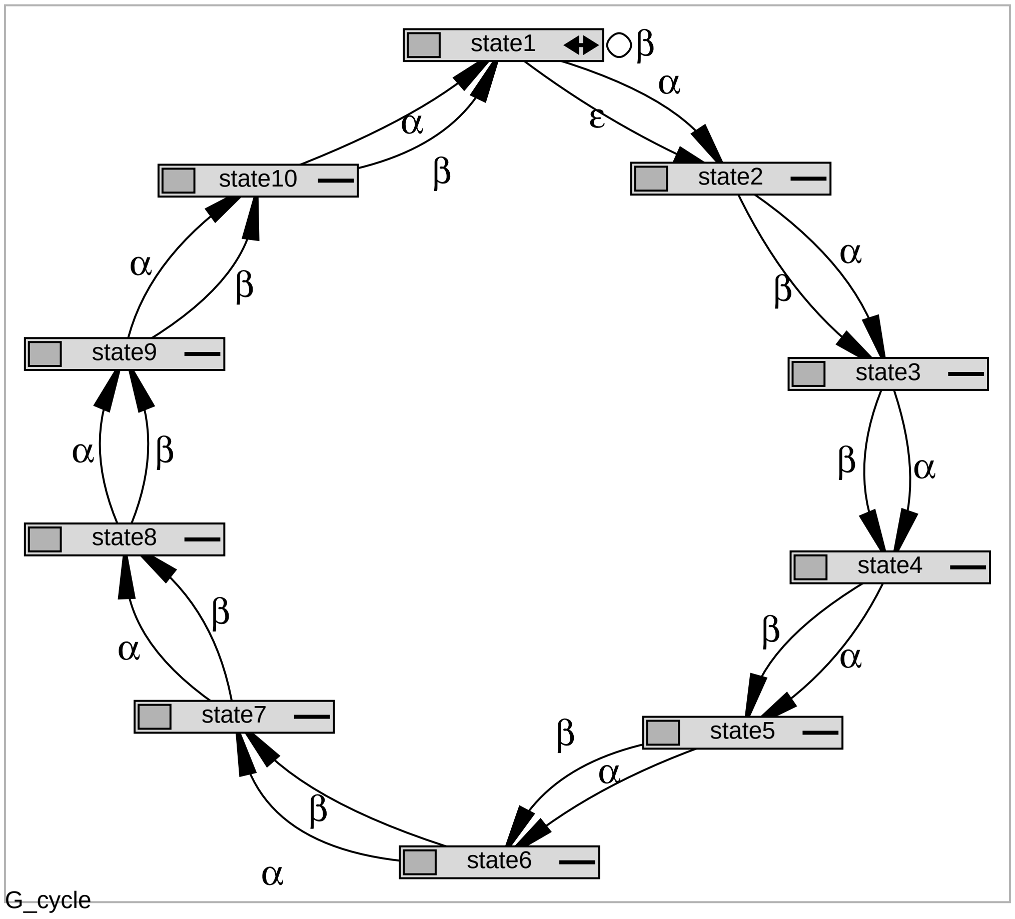


**Figure 5.11:** A cyclic FSM with 10 states

**Table 5.4:** $\sigma$-reachability results for the cyclic example

| | $1\sigma$-reach. | $2\sigma$-reach. | $3\sigma$-reach. | $4\sigma$-reach. | $5\sigma$-reach. |
|---|---|---|---|---|---|
| State Est. | 1537 | 2819 | 4999 | 8911 | 15871 |
| Max. Subset | 10 | 10 | 10 | 10 | 10 |
| Complexity | $O(\lvert\Sigma_o\rvert)$ | $O(\lvert\Sigma_o\rvert^2)$ | $O(\lvert\Sigma_o\rvert^3)$ | $O(\lvert\Sigma_o\rvert^4)$ | $O(\lvert\Sigma_o\rvert^5)$ |

### 5.4.1 $\sigma$-Reachability Analysis Results

Table 5.4 presents the results of the $\sigma$-reachability tests which were done on the FSM in Figure 5.11. We can immediately see that the $1\sigma$-reachability test gives a worse estimate than the $3/4 \cdot 2^{10} - 1 = 767$ upper limit for the number of projection states, and that each subsequent iteration of the test serves only to roughly double the projection state–space estimate.

To try and understand why our $\sigma$-reachability results do not improve the projection state-space estimate, we look at the adjacency matrix for the observable event $\alpha$ (Figure 5.12). Note that as in the previous cases, this adjacency matrix represents not only the occurrence of $\alpha$ events, but also the occurrence all possible unobservable event strings which may follow an $\alpha$ event. Whereas in the previous example, one of the events maps to a large subset of states, in this example Figure 5.12 illustrates the less desirable case when an event (in this case the $\alpha$ event) maps to the entire set of states. It follows that when this type of mapping occurs, we can conclude that no number of iterations of the $\sigma$-reachability test will reduce this set, since the occurrence of an arbitrarily long string of $\alpha$ events will always map to the entire set of states. This is exhibited in Figure 5.13 for $5\sigma$-reachability.

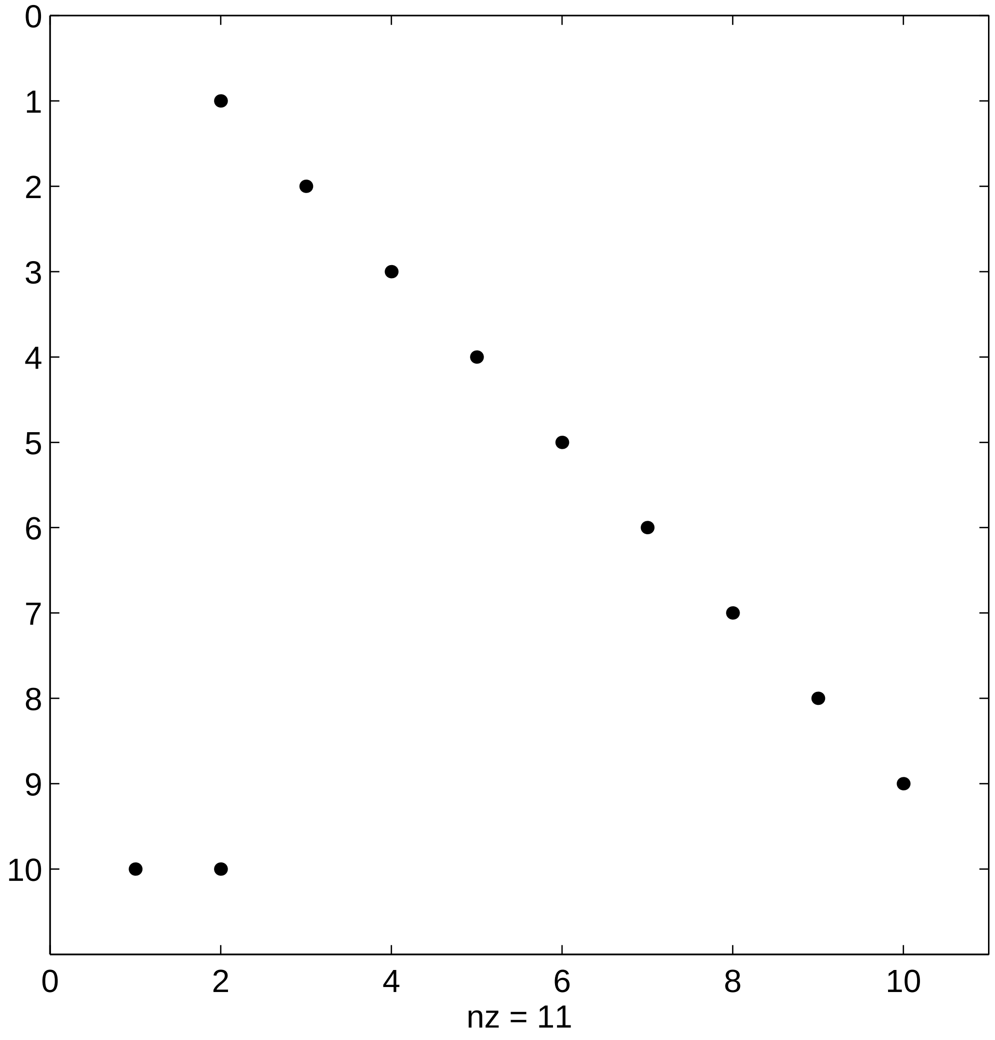


**Figure 5.12:** The adjacency matrix for the $\alpha$ event

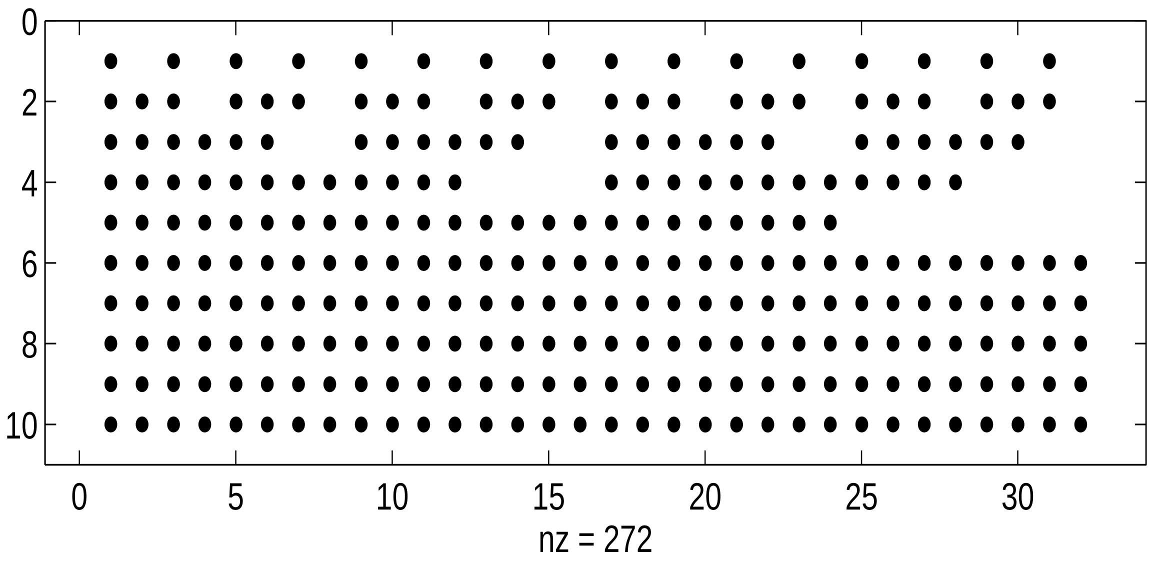


**Figure 5.13:** Reachable state-sets via all 5-length strings

# Chapter 6

# Conclusions and Discussion

The DES software requirements and the set of matrix–based algorithms presented in this thesis constitute a basis for the design and implementation of a DES software toolbox which provides a flexible and visual environment for the design and analysis of discrete–event systems. This design attempts to combine some of the useful features of existing software packages with the proven reliability of a high-level matrix-based computational engine (MATLAB). In addition, the thesis outlines a series of user-interface requirements which enable the user to design, modify and analyze discrete-event systems in a simple and intuitive manner.

We have proposed a method for storing DES models (in the form of FSMs) in a matrix-based computational environment. We have also provided a set of matrix-based DES operations which serve as building blocks for modeling DES problems. The logical continuation of this work would include completing the set of matrix-based DES operations. At a higher level, the software has been designed so that it could be possible to include modules which incorporate additional DES requirements, such as timing or knowledge, into the basic DES toolbox.

Our set of requirements and prototype software implementation served as a tool for investigating the effects of structure on the computational complexity of constructing FSMs which generate projected languages. A number of methods (based on DES structures) which attempt to improve the estimate of the size of the projection state-space have been presented. The effectiveness of this type of analysis is illustrated through a series of simple yet illustrative examples. These examples have been chosen to show cases where our analysis improves and does not improve the estimate of the projection state-space.

It is our belief (confirmed by the work presented in [OW90]) that by analyzing in more detail the FSM properties related specifically to $\sigma$-reachability, the results presented here could be improved upon. Further, we believe that to fully take advantage of the work done here, some work could be done which would identify exactly how problem structures (such as cycles) could be modified so that computational problems can be avoided.

Finally, the computational complexity analysis presented in this thesis could be implemented in the prototype software tool as a package of pre-filtering tests which would provide estimates of the computational complexity of doing projection before the projection operation is run.

# References

[BH93] Y. Brave and M. Heymann. Control of discrete event systems modeled as hierarchical state machines. *IEEE Transactions on Automatic Control*, 38(12):1803–1819, 1993.

[CDFV88] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, March 1988.

[CLL92] S. L. Chung, S. Lafortune, and F. Lin. Limited lookahead policies in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 37(12):1921–1935, 1992.

[CLO95] C. G. Cassandras, S. Lafortune, and G. J. Olsder. Introduction to the modelling, control and optimization of discrete event systems. In *Proceedings of the 1995 European Control Conference*, pages 1–71, Springer-Verlag, September 1995.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Massachusetts Institute of Technology, Massachusetts, 1990.

[Epp95] S. S. Epp. *Discrete Mathematics with Applications*. PWS Publishing Company, Boston, 1995.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–229, 1993.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HL94] M. Heymann and F. Lin. On-line control of partially observed discrete event systems. *Discrete Event Dynamic Systems*, 4:221–236, 1994.

[Hop71] J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computation*, pages 189–196. Academic Press, New York, 1971.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[JR93] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM Journal of Computation*, 22(6):1117–1141, 1993.

[Laf88] S. Lafortune. Modeling and analysis of transaction execution in database systems. *IEEE Transactions on Automatic Control*, 33(5):439–447, May 1988.

[Leu93] H. Leung. Separating exponentially ambiguous NFA from polynomially ambiguous NFA. In *Algorithms and Computation*, Lecture Notes in Computer Science, No. 778, pages 451–472. Springer-Verlag, Berlin, 1993.

[LMMB88] S. C. Lauzon, A. K. L. Ma, J. K. Mills, and B. Benhabib. Application of discrete-event systems to flexible manufacturing. *Algorithmica*, 3(3):451–472, 1988.

[LW88] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44:173–198, 1988.

[LW93] Y. Li and W. M. Wonham. Control of vector discrete-event systems I—the base model. *IEEE Transactions on Automatic Control*, 38(8):1214–1227, 1993.

[LW94] Y. Li and W. M. Wonham. Control of vector discrete-event systems II—controller synthesis. *IEEE Transactions on Automatic Control*, 39(3):512–531, 1994.

[Mat92] The Mathworks, Inc., Natick, MA. *MATLAB User's Guide*, 1992.

[Ost97] J. S. Ostroff. A visual toolset for the design of real-time discrete event systems. *IEEE Transactions on Control Systems Technology*, 5(3):320–337, May 1997.

[Ous94] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[OW90] C. M. Özveren and A. S. Willsky. Observability of discrete event dynamic systems. *IEEE Transactions on Automatic Control*, 35(7):797–806, July 1990.

[O'Y92] S. D. O'Young. Object TCT: User's guide. Systems Control Group Report, Department of Electrical Engineering, University of Toronto, 1992.

[Rud88] K. G. Rudie. Software for the control of discrete event systems: A complexity study. Master's thesis, Department of Electrical Engineering, University of Toronto, 1988.

[RW82] P. J. Ramadge and W. M. Wonham. Supervision of discrete event processes. In *Proceedings of the 21st IEEE Conference on Decision and Control*, volume 3, pages 1228–1229, December 1982.

[RW89] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[RW90] K. Rudie and W. M. Wonham. The infimal prefix-closed and observable superlanguage of a given language. *Systems & Control Letters*, 15(5):361–371, 1990.

[RW92a] K. Rudie and W. M. Wonham. Protocol verification using discrete-event systems. In *Proceedings of the 31st IEEE Conference on Decision and Control*, pages 3770–3777, Tucson, Arizona, December 1992.

[RW92b] K. Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, November 1992.

[RW95] K. Rudie and J. C. Willems. The computational complexity of decentralized discrete-event control problems. *IEEE Transactions on Automatic Control*, 40(7):1313–1319, 1995.

[SSL$^+$95] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.

[SSL$^+$96] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105–124, 1996.

[Thi96] J. G. Thistle. Supervisory control of discrete event systems. *Mathematical Computer Modelling*, 23(11):25–53, 1996.

[Tsi89] J. N. Tsitsiklis. On the control of discrete-event dynamical systems. *Mathematics of Control, Signals, and Systems*, 2:95–107, 1989.

[Won96] W. M. Wonham. Notes on control of discrete-event systems. Unpublished course notes, 1996.

[WR88] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems*, 1:13–30, 1988.

## VITA

| | |
|---|---|
| Name: | Adrian Victor Payne |
| Place and Year of Birth: | Ottawa, 1970 |
| Education: | Queen's University, 1988–93<br>B.Sc. (Honours, Engineering Physics) 1993 |
| | Queen's University, 1995–present<br>M.Sc.(Eng) |
| Experience: | Software Developer<br>Calian Technical Services, 1993–95<br>On contract to the Canadian Space Agency |
| | Teaching Assistant<br>Department of Electrical and Computer Engineering<br>Queen's University, 1996–97 |
| Awards: | James A. Rattray Memorial Scholarship, $500<br>Queen's University, 1990–91 |
| | Queen's Graduate Award, $6000 per annum<br>Queen's University, 1995–97 |