1996

Designing Diagnosable Distributed Programs.

Amit Anil Nanavati
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

DESIGNING DIAGNOSABLE DISTRIBUTED PROGRAMS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Amit Anil Nanavati
B.E. in Computer Science, M.S. University of Baroda, India, 1989
M.S. in Systems Science, Louisiana State University, 1994
December 1996

To Dhabu,
With Love

Acknowledgments

First and foremost, I would like to thank Prof. S.S. Iyengar, without whom none of this would have ever happened. For his kindness and understanding, and for the opportunity to spend a summer at JPL which, in a rather circumlocutous way, lead to the consideration of this problem. I also thank Prof. S. Kundu, for several interesting courses, for adversarial arguments, and for his painstaking, careful criticism. He tried very hard to drill into me the importance of examples. I hope he succeeded. I am indeed very grateful to both of them for all that they have done for me. To Alok, my dear friend, philosopher and guide since childhood, who made all of this possible.

I would like to thank Prof. El-Amawy, Prof. Carver and Prof. Zheng for their warmth, interest and concern for me and my work. I thank them and Prof. Nikitopoulos for being on my committee.

My wonderful friends: Das, for introducing me to celery and western classical music; 'gurudev' Pendse for teaching me to appreciate indian classical music; Ramanas for showing the number of ways of having fun, LP for his very presence, and both of them for a zillion interesting anecdotes ; Sudhakar, Sundar and to a lesser extent Raghu, for bearing bravely the brunt of my culinary skills; Raghu, for several meal pile-ons, KK-appreciation and a wonderful time in California; Harry$^{\text{Annie}}$, for numerous music sessions and a singular poem; Sundar, my friend, philosopher, roomie and guide for awesome dinners, improved living standards, homer-humor, tie, T-shirts, Tamil songs, and all that sort of thing[1]; Vyral(ugrad fpg) and Kd$^{\text{Purvi}}$,

[1]With apologies to P.G. Wodehouse

my pals since undergraduate days, for all the r-fun, r-support, r-chats (r=remote); To each one of them and Harsha, Saip, Rams, Utpal, Debasis, Dhan, for every chat, discussion, and the fun times I shared with them; for all the encouragement I received from them.

I thank the department of Computer Science for supporting me for four years. To Elias, for sharing his love for UNIX, and particularly his hatred for PCs, with me. My heartfelt thanks to Gaby Segarra and Dr. E. Icaza of ITS for providing financial support at a timely juncture, and for all the warmth and kindness they have showered upon me. I blame entirely, Elias and Gaby, for my completely warped notion of the word 'boss'. From them, I learned a few more lessons in humour and patience.

Special thanks to Prof. S. Lafortune (U.Mich.), Prof. F. Schneider (Cornell), Prof. N. Lynch (MIT) and Prof. K. Rudie (Queen's Univ.) for all the references and e-mail exchanges during the early stages of this work.

To my entire family, the Nanavatis and Dholakias, for academic standards that I can only hope to strive for and not meet; and for their enthusiasm and support. My uncle, Prof. Dholakia, for several delightful expositions and engaging discussions during those formative years. From him, I learnt some of my first formal methods on joke creation.

My little sister Dhabu, from whom I took away those years of my life that belonged to her; and who in turn, took away from me, so many of my worries and anxieties. She has been a constant and unbounded source of affection, joy and amusement.

It is my proud privilege to acknowledge my most revered and beloved teachers of my undergraduate years: Prof. S. Ramamohan and Prof. V.D. Pathak, of

M.S.University of Baroda, India. They share the property of being exceedingly patient with intelligences far lesser than their own. They almost deceived me into believing the Holmesian maxim[2], "What one man can invent, another can discover".

Most of all, my parents. About them, there is so much to say, but no *language* to say it in...

[2]Sherlock Holmes in "The Dancing Men".

Table of Contents

List of Tables

List of Figures

Index of Notation

Symbol	Meaning	Page
M, M', M''	Mealy machines (chapter 2) or I/O automata	10
A, B', C''	states of I/O automata	19
a, b, c	actions/events of an I/O-automaton	19
$acts(M)$	input, output and internal actions/events	19
$M \times M'$	composition of M and M'	19
C	finite collection of strongly compatible I/O automata	20
$s[i]$	the local state of component M_i in global state s	21
$e\ell(S)$	event language of a state S in some I/O-automaton	21
$\ell e\ell(S)$	local event language of a state S	21
$S\|M$	local projection of global state S on M	22
M_{-i}	I/O automaton M with input actions removed	25
$ds(S)$	distinguishing set of strings for state S	30
$\langle S_1, \ldots, S_k \rangle$	state uncertainty at a node S in a distinguishing tree	32
$\simeq$	equivalence of state uncertainties	33
$dt(M)$	distinguishing tree of M	33
$\equiv$	semantically equivalent	35
$[a]$	the semantic equivalence class of event a (relabelling)	35
M^R	M with ambiguities resolved	36
M_i^C	augmented form of M_i^R compatible with $M_{j(j\neq i)}^R$	38
$dt_{max}(M)$	maximal distinguishing tree of M in C	39
D, D', D''	debuggers for M, M', M''	46
$D \odot M$	supervision of M by D, denoted by S	46

Abstract

The difficulty in debugging distributed programs motivates the development of formal methods for designing distributed programs that are easier to debug and maintain. We address state identification problem for distributed systems using the finite state I/O automaton model. A state S is identified based on the unique event sequences starting at S, called *distinguishing sequences.*

An automaton is *diagnosable* if every state has a distinguishing sequence. A distributed program may not be diagnosable even if its components are diagnosable. Non-diagnosable automata can, in some cases, be converted to a diagnosable form by relabelling some of its transitions in a way that preserves the semantics of the program. Not all automata can be converted to a diagnosable form in this way. This is due to inherent ill-posedness of specification. Two algorithms to convert a non-diagnosable automaton to a diagnosable form are presented.

Debugging is the controlled execution of one program by another. The latter is called the *supervisor* of the former. The supervision operation is defined so that the debugging of a distributed program by distributed debuggers is reduced to the same as the debugging of a single program by a single debugger. An algorithm to construct a debugger for a diagnosable program is developed. Every diagnosable program has a unique debugger associated with it. This leads to the introduction of the notion of *debugging complexity* of programs.

Chapter 1

Introduction

> *No matter what [programming] language we use, we are bound to make mistakes. Not because we are sloppy and undisciplined [sic], but because we cannot know, at any finite point in time, all the consequences of our current assumptions.* — *E.Y. Shapiro*

The basic problem in programming is the management of complexity. In distributed programs, further complexity arises from the fact that many processors are executing in parallel, with steps interleaved in some undetermined way, implying that there can be prohibitively many different executions, even for the same inputs.

Debugging ordinary programs is hard. Debugging distributed programs, with no shared memory and no global clock, is even more so. This difficulty motivates the development of formal methods for designing distributed programs that are easier to debug and maintain. This investigation was prompted in an attempt to answer the following questions in the context of distributed systems:

1. What makes programs easier to debug ?
2. Is it possible to alter programs so that they become easier to debug ? Is it always possible to do so ?
3. Can the process of debugging be automated ?

1.1 Distributed Debugging

As can be expected, distributed debugging has been approached from several perspectives. These approaches can be broadly classified as follows. *Static analysis*

tries to isolate bugs by analyzing the program statically, before running them [13, 7]. *Event-based* approaches model program behavior, typically by incorporating various types of event filters, which enable the systems to consider only events that meet selective criteria [2, 11]. *Distributed state* computation approaches the issue by detecting or recreating global state in distributed computations [9, 16, 12]. *Formal analysis* techniques include graph theoretic analysis (data path analysis), logic programming, and such [25, 20, 6]. *Visualization* is concerned with presenting execution information in an abstract, intuitive form to aid the user [37, 23]. For an excellent brief overview of the issues involved, the reader is referred to the session summaries of the workshops on distributed debugging [33, 34, 35].

1.1.1 State Identification

As Chandy and Lamport [9] point out, many problems in distributed systems can be cast in terms of the problem of (global) state identification. For instance, *stable property* detection. A stable property is one that persists: once a stable property becomes true, it remains true thereafter. Examples of stable properties are "computation has terminated", "the system has deadlocked" and "all tokens in a token ring have disappeared". Stability must be detected so that one phase can be terminated and the next phase initiated [31]. The termination of a computational phase does *not* necessarily mean cessation of activity – messages may be sent and received; the cessation of activity is only one example of a stable property. Deadlock detection is another special case of stable property detection. State identification can also be used for checkpointing. Cooper and Marzullo approach the problem using *temporal predicates* [12]. They present algorithms for detecting global predicates by introducing the *possibly, currently, definitely* operators.

This work addresses the state identification problem for distributed systems in an automata-theoretic framework, the finite state I/O-automaton model.

The Automata-theoretic Perspective

Once a (finite state) automaton is designed, it is necessary to establish that it functions as intended. What this means is that one must ascertain that the state structure of the automaton matches its specification.

There are many different measurement problems of interest depending upon which parameters of the machine are assumed known, which are assumed unknown, and which ones can be varied in a controlled manner. The *initial state identification* problem deals with the problem of trying to determine the unknown initial state of the machine. This type of problem can occur, for example, while troubleshooting a machine. If one can determine the state of the machine after an error has disrupted the machine's operation, it may be possible to determine the cause of the error [4]. To solve this problem in the case of Mealy machines (details in Chapter 2), a predetermined input sequence is applied and the corresponding output sequence is observed [24]. On the basis of the observed output sequence, it is possible to determine the initial state. Another measurement problem is the *terminal state identification* problem. In this case it is assumed that the machine is in some unknown initial state. A known input sequence is then applied, the resulting output sequence observed, and on the basis of this observation, the terminal state is determined. Such problems have been studied extensively for sequential machines [14, 15, 21, 27, 32].

Thus state identification is a central issue in testing and debugging. This problem is inherently linked to the more general problem of machine identification. Machine

identification is concerned with the problem of determining whether or not a given machine is distinguishable from all other machines. To identify a machine, it is necessary to identify all of its states. This problem is known to be, under certain conditions, equivalent to the problem of determining whether or not a given machine is operating correctly. The problem of designing *fault-detection* experiments is actually a restricted problem of machine identification. An experimenter is supplied with a machine and its state table. The task is to determine from terminal experiments whether the given table accurately describes the behavior of the machine. The experiments are intended to detect the presence of one or more faults.

1.2 This Research

This work is concerned with global state detection in the I/O-automaton model by an event-based approach. The basic idea is to develop systematic methods to design programs that are easier to debug. A formal framework to model distributed debugging is developed. Distributed debugging is analysed from a control-theoretic standpoint. Due to this approach, it is possible to synthesize *automatically*, a debugger for any given diagnosable program.

We develop a theory for designing distributed programs that are *diagnosable*, with the goal of mechanizing this activity. We address the state identification problem for finite state distributed programs in the I/O-automaton model. Programs are expressed as I/O-automata. The aim is to identify the unknown initial state of an automaton based on event sequences generated by it. A program is diagnosable when it is possible to deduce the initial state of the program based upon the event sequences it generates.

The results obtained are quite different from the sequential case. All sequential programs can be converted into diagnosable ones; not all distributed programs can be. This difference leads to major issues such as *ill-posed* problems and *debugging complexity* among others.

1.2.1 The Model

The I/O-automaton model [29], is a model for distributed computation in asynchronous systems. Automaton transitions are labelled with the names of process actions they represent. These actions are partitioned into sets of input and output actions, as well as internal actions representing internal program actions. Input actions have the unique property of being enabled from every state; for every input action there is a transition labelled with this action from every state. Thus, a strong distinction is made between locally-controlled actions (output and internal) and actions controlled by the system's environment (input actions). This gives the model an event-driven flavor. For the analysis, a special subclass of deterministic, minimal automata are considered.

1.2.2 Contributions

Not all programs are diagnosable. Even if the components are diagnosable, the program as a whole need not be. Two algorithms to transform non-diagnosable programs into diagnosable ones are presented. The transformation is done by relabelling some of the transitions in a way that preserves the semantics of the program. Two states that share a common event string prior to relabelling may not do so after the operation. Event strings exclusive to each state are created in this manner.

However, not all programs can be converted to a diagnosable form. Such programs result from inherent ill-posed specifications.

In the second part of this thesis, we introduce a new operation to model debugging in the I/O-automaton model. The process of debugging can be described as the supervised execution of the debuggee by the debugger. If a program is executed uncontrolled, it may follow a path which may make it impossible to distinguish certain initial states of the debuggee. The function of a debugger is to steer the execution of a program by disabling and enabling some of its transitions. We give an algorithm to construct a *unique* debugger for any diagnosable program. Thus, we can associate a unique *debugging complexity* with every diagnosable program. The main contributions of this thesis are:

- A theory of diagnosable *distributed* programs, including
  - Diagnosability Theorem
  - Relabelling Theorem
  - Unresolvability Theorem
  - Ill-posedness of Specifications
- Algorithms to convert a non-diagnosable program to a diagnosable form:
  - Algorithm *max-convert*
  - Algorithm *fast-convert*
- Extension of the I/O-automaton model to support distributed debugging,
  - Supervision operation
  - Commutativity Theorem
- Algorithms for automatic debugger synthesis for diagnosable programs,
  - Algorithm *make-D*
  - Algorithm *make-comp*
- Introduction of the notion of debugging complexity of programs.

1.2.3 Layout

Chapter 2 and 3 describe the previous work done and prepare the background for the contribution of this thesis, which appears in chapters 4 and 5.

Diagnosability has been investigated in the context of Mealy machines [24]. Chapter 2 describes Mealy machines, diagnosability, and preset and adaptive experiments. Every Mealy machine can be converted to a diagnosable form. Chapter 3 describes the I/O-automaton model, including the definition of composition and event sequences. An example to illustrate the model is presented. Chapter 4 introduces the notion of strong equivalence, which is based on local event sequences. Strong equivalence is important because it is impossible to distinguish between strongly equivalent states, and strong non-equivalence is preserved during composition. The finite tell-tale property is required so that the event language of one particular state may not be completely contained in that of another, thereby making distinguishing between the two states difficult. The concepts of distinguishing sequences and diagnosability are defined in the context of I/O-automata. Algorithms to transform, whenever possible, a non-diagnosable automaton to a diagnosable one by relabelling some of its transitions are given. During the process, new events may be added to the event alphabet. Chapter 5 defines the *supervision* operation to support debugging in the I/O-automaton model, and presents an algorithm to construct a debugger for any diagnosable automaton. Chapter 6 concludes this thesis.

Chapter 2

Sequential Machines

The concept of diagnosability is presented in the context of Mealy machines. For further details, the reader is referred to Chapter 13 of Kohavi's book [24]. Mealy machines are finite state machines which output binary strings for binary input. Preset and adaptive diagnosability experiments are described. The main motivation for designing diagnosable machines and studying their properties is the expectation that such machines will prove easier to maintain, and that it will be possible to design fault-location experiments for them. All Mealy machines can be converted to a diagnosable form.

2.1 Preliminaries

For Mealy machines, the next state $S(t+1)$ is determined uniquely by the present state $S(t)$ and the present input $x(t)$. Thus, $S(t+1) = \delta\{S(t), x(t)\}$ where δ is called the *state transition function.* The value of the output $z(t)$ is a function of the present state $S(t)$ and the input $x(t)$, $z(t) = \lambda\{S(t), x(t)\}$ where λ is called the *output function.* M_1 is an example of a Mealy machine figure 2.1.

A machine is assumed to be minimal, strongly connected, and completely specified, and is available to the experimenter as a "black box," which means that the experimenter has access to its input and output terminals, but cannot inspect the internal devices and interconnections. The experiments thus consist of a set of input sequences and their corresponding output sequences.

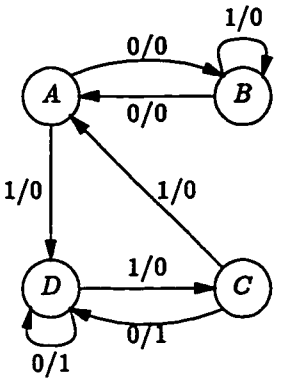

Figure 2.1: A Mealy machine.

The state-identification experiments are designed to identify the unknown initial state of the machine. These experiments are called *distinguishing* experiments. The machine-identification experiments are concerned with the problem of determining whether or not a given n-state machine is distinguishable from all other n-state machines. This problem is known to be, under certain conditions, equivalent to the problem of determining whether or not a given machine is operating correctly. According to performance, experiments are classified as:

1. *Preset* experiments, in which the entire input sequence is predetermined independently of the outcome of the experiment.

2. *Adaptive* experiments, in which the input at any instant of time depends on the previous outputs.

Suppose that a machine M, which is given to the experimenter, can initially be in any one of its n states. In such a case, we say that the initial uncertainty regarding the state of the machine is given by $(S_1 S_2 \cdots S_n)$. The aim is to perform experiments that reduce the initial uncertainty and whenever possible reveal the initial or final state. We shall refer to a collection of uncertainties as an *uncertainty vector*. The individual uncertainties contained in the vector are called the *components* of the

vector. An uncertainty vector whose components contain a single state each is said to be a *trivial uncertainty vector.* An uncertainty vector whose components contain either single states or identical repeated states is said to be a ***homogeneous uncertainty vector.*** For example, the vectors $(AA)(B)(C)$ and $(A)(B)(A)(C)$ are homogeneous and trivial respectively.

Distinguishing experiments are concerned with the identification of the initial state of the machine whose state table is known, but no other information regarding its condition is known.

Definition 2.1 Let M be an n-state machine. An input sequence X is said to be a distinguishing sequence if the output sequence produced by M in response to X is different for each initial state.

Definition 2.2 A distinguishing tree is a successor tree in which node in the jth level becomes terminal when any of the following occur:

1. The node is associated with an uncertainty vector whose non-homogeneous components are associated with some node in a preceding level $i, i \leq j$;
2. The node is associated with an uncertainty vector containing a homogeneous nontrivial component;
3. Some node in the jth level is associated with a trivial uncertainty vector.

2.2 Preset Experiments

A diagnosable machine is one which possesses one or more distinguishing sequences and thus permits us to identify uniquely the states of the machine by inspecting

Table 2.1: Testing Table for M_1.

	0/0	0/1	1/0	1/1
A	B	–	D	–
B	A	–	B	–
C	–	D	A	–
D	–	D	C	–
AB	AB	–	BD	–
AC	–	–	AD	–
AD	–	–	CD	–
BC	–	–	AB	–
BD	–	–	BC	–
CD	–	DD	AC	–

its response to such a sequence. In this section, we shall present a method to modify the design of sequential machines in such a way that they will possess special distinguishing sequences.

Machine M_1 does not possess any distinguishing sequence. However, it can be augmented, by adding to it an additional output terminal, so that the augmented machine will possess several distinguishing sequences. Table 2.1 shows the testing table for M_1.

The state table of M_1 can be written as shown in the upper half of table 2.1. The column headings consist of all input-output combinations, where the pair I_k/O_l corresponds to a combination of input I_k and output O_l. The row headings in the upper half of the table are the states of the machine. The entry in column I_k/O_l, row S_i, is the I_k – *successor* of S_i. If this state is associated with output O_l and is a dash otherwise. For example, the 0-successor of A is B, and the corresponding output is 0. In a similar manner the next-state entries of M_1 are entered in the upper half of the table.

The lower half of the table is derived directly from the upper half. All row headings and table entries of the form S_iS_j are unordered pairs of states. If the entries in rows S_i and S_j, column I_k/O_l, of the upper half are S_p and S_q respectively, then the entry in row S_iS_j, column I_k/O_l, of the lower half is S_pS_q (or S_qS_p). For example, since the entries in rows A and B, column 1/0, are D and B respectively, the corresponding entry in row AB, column 1/0, is BD, and so on. If for some pair of states S_i and S_j either one or both corresponding entries in some column I_k/O_l, are dashes, the corresponding entry in row S_iS_j, column I_k/O_l, is a dash. For example, the entry in row AC, column 1/0, is a dash, since the entry in row C, column 0/0, is a dash. The table thus completed is referred to as a *testing table.*

We shall refer to a pair (S_iS_j) as an *uncertainty pair* and to its successors (S_pS_q) as the *implied pair.* For example, the pair (BD) is implied by (AB). An uncertainty pair that does not imply any other pair, (all the entries in the corresponding row are dashes), can be omitted from the table. Whenever an entry in the testing table consists of a repeated state (DD in row CD), the entry is in bold font. **DD** means that states C and D are *merged* under input 0 and cannot be distinguished by an experiment which starts with a 0 input.

A directed graph, G, called a *testing graph*, is defined in the following way:

1. Corresponding to each row in the lower half of the testing table there is a vertex in G.

2. If there exists an entry S_pS_q, where $p \neq q$, in row S_iS_j, column I_k/O_l, of the testing table, then G has a directed arc leading from the vertex labeled S_iS_j to the vertex labeled S_pS_q. The arc is labeled I_k/O_l. No arc is needed if S_iS_j implies S_pS_p, example, DD in row CD.

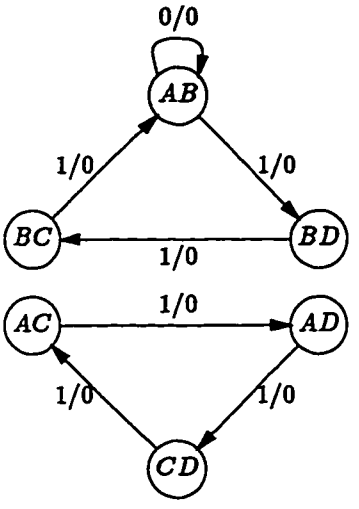

Figure 2.2: Testing graph for M_1.

The testing graph for M_1 is derived directly from the lower half of the testing table and is shown in figure 2.2.

A machine M is defined as a *definitely diagnosable of order* μ , if μ is the least integer so that every sequence of length μ is a distinguishing sequence for M. In other words, a machine is definitely diagnosable if every node at level μ of the distinguishing tree is associated with a trivial uncertainty vector. The distinguishing tree can thus serve as a tool for recognizing definitely diagnosable machines. We present the following theorem and corollary without proof [24].

Theorem 2.1 *A machine M is definitely diagnosable if and only if its testing graph G is loop-free and no repeated state entries exist in the testing table.*

Corollary 2.2 *Let the testing table of machine M be free of repeated state entries, and let G be a loop-free testing graph for M. If the length of the longest path in G is l, then $\mu = l + 1$.*

In order to obtain machine M_1' which contains M_1 and possesses a distinguishing sequence, it is necessary to augment M_1 by adding to it an output terminal and as-

Table 2.2: State Table for M_1 and M_1'.

PS	NS, z	
	$x=0$	$x=1$
A	B,0	D,0
B	A,0	B,0
C	D,1	A,0
D	D,1	C,0

PS	NS, zz_1	
	$x=0$	$x=1$
A	B,01	D,00
B	A,00	B,00
C	D,10	A,01
D	D,11	C,01

signing different output symbols to selected transitions. The addition of one output terminal is sufficient to make M_1' definitely diagnosable. We have to assign different outputs to each transition that may cause a repeated entry in the testing table. In the case of M_1, this is accomplished by assigning an output 10 to the transition from C to D, and as output 11 to the transition from D to D. Such an assignment of output values ensures that the testing table of M_1' is free of repeated entries.

The testing graph of M_1 contains three loops: a self-loop around AB and two other loops, each containing three vertices. These loops must be opened if M_1' is to be definitely diagnosable. In general, a loop is opened by the removal of any of its arcs. To remove an arc, it is necessary to assign different output symbols to the next-state entries represented by the vertex to which that leads. An arc leading from vertex S_iS_j to vertex S_pS_q is eliminated by assigning different output symbols to the transitions from S_i and S_j to S_p and S_q. After the loops marked in figure 2.2 are broken, we have the resulting state table in 2.2.

Since the length of a fault-detection experiment is directly proportional to the length of the distinguishing sequence for the machine, we attempt to open all loops, while simultaneously minimizing the length of the various paths in the graph. In opening the loops in the graph of figure 2.2, all the output entries, with the exception of the entry in row C, column $x = 1$, have been assigned new values. The longest

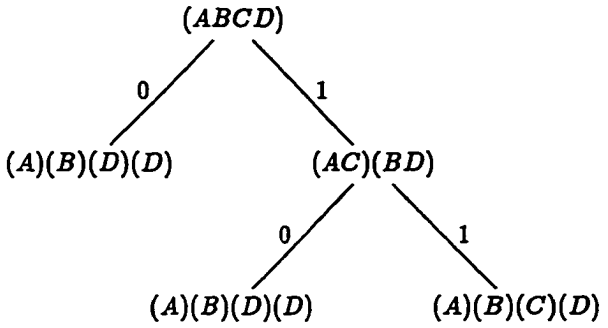

Figure 2.3: Distinguishing tree for machine M_1'.

path in the loop-free graph is of length 2, and consequently the order of the modified machine is $\mu = 3$. This specification actually eliminates the arcs from AC to AD and from BC to AB. As a result, the length of the longest path in the graph is now 1, and M_1' is definitely diagnosable of order 2. The distinguishing tree of machine M_1' is shown in figure 2.3.

For any 2^k-state machine, the addition of k output terminals is sufficient to convert it into a definitely diagnosable machine. The problem of providing an algorithm for finding the minimal number of additional required output terminals is hard. Although the removal of a minimal number of arcs does not necessarily imply the addition of a minimal number of outputs, the elimination of too many arcs does tend to increase the number of necessary output terminals. The following general result holds:

Theorem 2.3 *Corresponding to every minimal machine M, there is a definitely diagnosable machine M', that can be obtained from M by addition of one or more output terminals.*

Table 2.3: State Table for M_2.

PS	NS, z	
	$x=0$	$x=1$
A	C,0	A,1
B	D,0	C,1
C	B,1	D,1
D	C,1	A,0

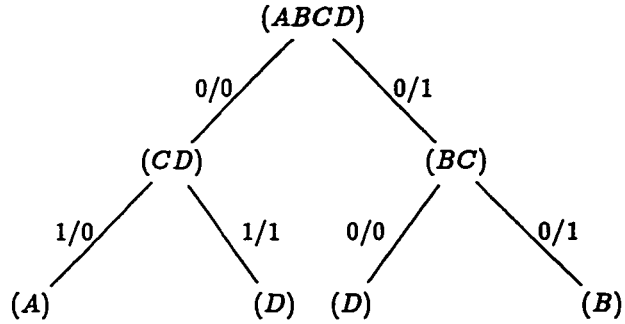

Figure 2.4: Adaptive distinguishing tree for M_2. Each edge is labelled by an input/output pair.

2.3 Adaptive Experiments

In preset experiments the choice of each input symbol is predetermined. and is not influenced by the response of the machine to the preceding symbols. In adaptive experiments, the choice of each input is determined by the machine's response to the previous inputs. The advantage of adaptive experiments is that they are generally shorter than preset ones. The disadvantage is the relative difficulty in designing them and the need to inspect the output after the application of each input.

Figure 2.4 shows an adaptive distinguishing tree for M_2. In effect, each of the uncertainties introduced in course of the experiment can be viewed as an initial uncertainty for a new experiment. The adaptive tree usually lists all possible successor uncertainties, and for each one it specifies the input symbols that can be applied

next and the possible outcomes. A necessary condition for an adaptive experiment to be a distinguishing experiment is that each path that the experiment might follow must terminate on a singleton uncertainty, and that no such path should lead to an uncertainty containing repeated entries. An *minimal adaptive experiment* minimizes the length of the paths leading to singleton uncertainties.

Some machines have both preset and adaptive distinguishing experiments. Clearly, in such a case, the length of the minimal adaptive experiment never exceeds the length of the minimal preset experiment. There are machines which have no distinguishing sequence, preset or adaptive, as demonstrated by M_1. Some may have no preset experiments, but only adaptive ones.

Chapter 3

I/O-Automata

The I/O-automaton model was defined by Lynch and Tuttle [29] as a tool for modelling the components of a concurrent discrete event system. The components operate asynchronously, and continuously receive input from and react to their environment instead of simply computing some function of their input and halting. Each component is modelled as an I/O-automaton.

An I/O-automaton is essentially an automaton with each transition labelled by an action. The actions are partitioned into *input, output* and *internal* actions. Consider a system composed of several automata. For every input action there is a transition from every state, so that an automaton (hence the system) must be able to accept input at any time. This means that an automaton cannot block the progress of another automaton. The output actions of every component are unique. Internal actions represent actions that are internal to a particular automaton and as such do not have any immediate impact on other automata. Communication is modelled by the simultaneous performance of a common action. An input action (with label a, say) in an automaton M_1 must take place if a corresponding output action with the same label occurs in another automaton M_2 in the system or in the environment.

I/O-automata have been a widely used formalism for the specification and verification of concurrent algorithms [28, 38]. In the most general setting, I/O-automata

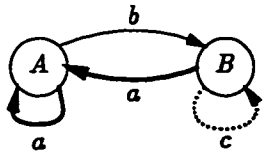

Figure 3.1: An I/O automaton; a is an input action, b output and c internal. Input actions are indicated by heavy lines and internal actions by dotted lines.

may be nondeterministic and infinitary products can be defined; we restrict ourselves to the deterministic, finite case only.

3.1 Definition

The disjoint sets of actions for an automaton M are denoted by $in(M)$, $out(M)$, and $int(M)$, respectively; and $acts(M) = in(M) \cup out(M) \cup int(M)$. The set of *external actions*, $ext(M) = in(M) \cup out(M)$, and the set of local actions, $loc(M) = int(M) \cup out(M)$. An action may also be referred to as an *event*. For the purpose of the current discussion, it suffices to say that an I/O-automaton M consists of:

1. a set $states(M)$ of *states*,
2. a set $start(M) \subseteq states(M)$ of *start states*,
3. a transition relation $steps(M) \subseteq states(M) \times acts(M) \times states(M)$, with the property that for every state S and input action e, there is a transition (S, e, S') in $steps(M)$.

Figure 3.1 shows an I/O-automaton.

3.2 Composition

I/O automata can be composed to yield other I/O automata. When a collection of automata are composed, the same-named actions of different automata are identi-

fied. The composition guarantees that if one automaton has e an an output action, then e may only be an input action for all the remaining automata. As a result, an automaton generating an output action does so autonomously, and this output action is transmitted instantaneously to all other automata having the same action as an input. This synchronization models communication between system components. Two automata cannot be expected to perform an output action simultaneously in an asynchronous system. The requirement that the output actions of every pair of automata in the composition should be disjoint simplifies the notion of composition.

There are certain restrictions on the composition of automata. Since internal actions of an automaton A are intended to be unobservable by any other automaton B, A cannot be composed with B unless the internal actions of A are disjoint from the actions of B, since otherwise one of A's internal actions could force B to take a step. Furthermore, since at most one system component (automaton) controls the performance of any given action, A and B can be allowed to compose only if their output actions are disjoint.

A finite collection $C = \{M_i\}_{i \in I}$ of I/O-automata is said to be ***strongly compatible*** if for all $i \neq j \in I$, the following conditions hold. The components M_i of a system are assumed to be strongly compatible:

1. $out(M_i) \cap out(M_j) = \emptyset$, and
2. $int(M_i) \cap acts(M_j) = \emptyset$.

Figure 3.2 shows the composition of two strongly compatible automata.

The actions of the composition $M = \prod_i M_i$ of a finite collection of strongly compatible I/O-automata are defined to be:

- $in(M) = \bigcup_i in(M_i) - \bigcup_i out(M_i)$

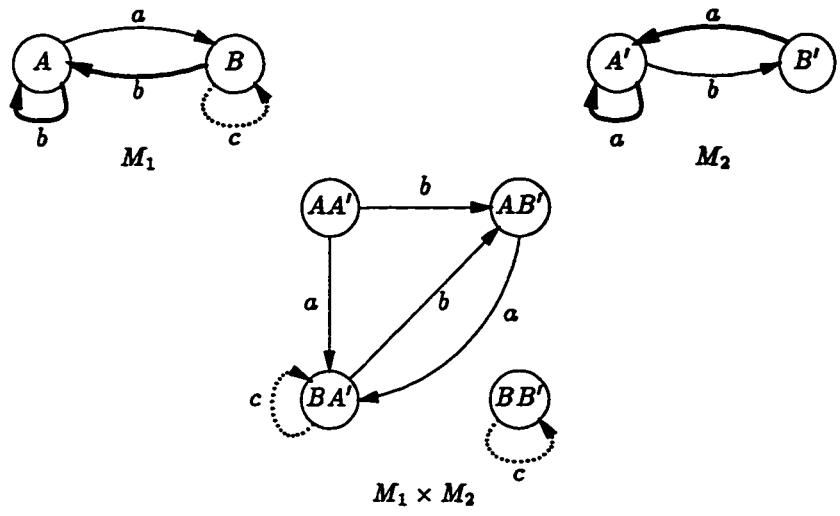

Figure 3.2: Composition of I/O-automata.

- $out(M) = \bigcup_i out(M_i)$ and
- $int(M) = \bigcup_i int(M_i)$.

Thus, $acts(M) = \bigcup_i acts(M_i)$.

The composition $M = \prod_i M_i$ of a finite collection of strongly compatible automata is the automaton defined as follows:

1. $states(M) = \prod_i states(M_i)$,
2. $start(M) = \prod_i start(M_i)$,
3. $steps(M)$ is the set of triplets (s_1, e, s_2) such that, for all $i \in I$, if $e \in acts(M_i)$, then $(s_1[i]^1, e, s_2[i]) \in steps(M_i)$, and if
   (a) if $e \in acts(M_i)$, then $(s_1[i], e, s_2[i]) \in steps(M_i)$, and
   (b) if $e \notin acts(M_i)$ then $s_1[i] = s_2[i]$.

The set of event strings starting at a state S is called the *event language* of S, denoted by $el(S)$. The set of strings comprising only of local events is called the *local event language*, $lel(S)$.

[1] $s_1[i]$ denotes the local state of M_i in global state s of M.

In the composition $M = M_1 \times M_2$, each state of M is of the form A_iB_j, where A_i is a state of M_1 and B_j is a state of M_2. If $S = \{S_i\}$ is a state of M, then $S|M_i = S_i$. If $s|M_1$ denotes the subsequence of the events in s which are events in M_1, then $s|M_1$ forms an event sequence starting at A_i in M_1, and $s|M_2$ forms an event sequence starting at B_j in M_2. Since input actions are defined at each state of an I/O-automaton, it follows that $e\ell(A_iB_j) \supseteq \ell e\ell(A_iB_j) \supseteq \ell e\ell(A_i) \cup \ell e\ell(B_j)$. In fact, $\ell e\ell(A_iB_j)|M_1 = \ell e\ell(A_i)$ and $\ell e\ell(A_iB_j)|M_2 = \ell e\ell(B_j)$.

3.3 Problem Specification

A problem specification is simply a set of allowable *behaviors*, event strings comprising of external actions only. An automaton solves the specification if each of its behaviors is contained in this set. In addition to a set of allowable behaviors, a problem specification must specify the required interface between a solution and its environment. Since the automaton behaviors can be a subset of the specified behaviors, trivial solutions must be avoided by specification.

3.4 An Example

It is also possible to describe I/O-automata based on *precondition* and *effect* specifications for actions. An I/O automata simulator to supports this style has been developed[19].

Let us model the interaction of a candy machine *CM* and a customer *CUST* [29]. *steps(CM)* is described by giving a *precondition* and an *effect* for every action. The triplet (s, e, s') is a step of *CM* if the precondition of e is satisfied by s and s is

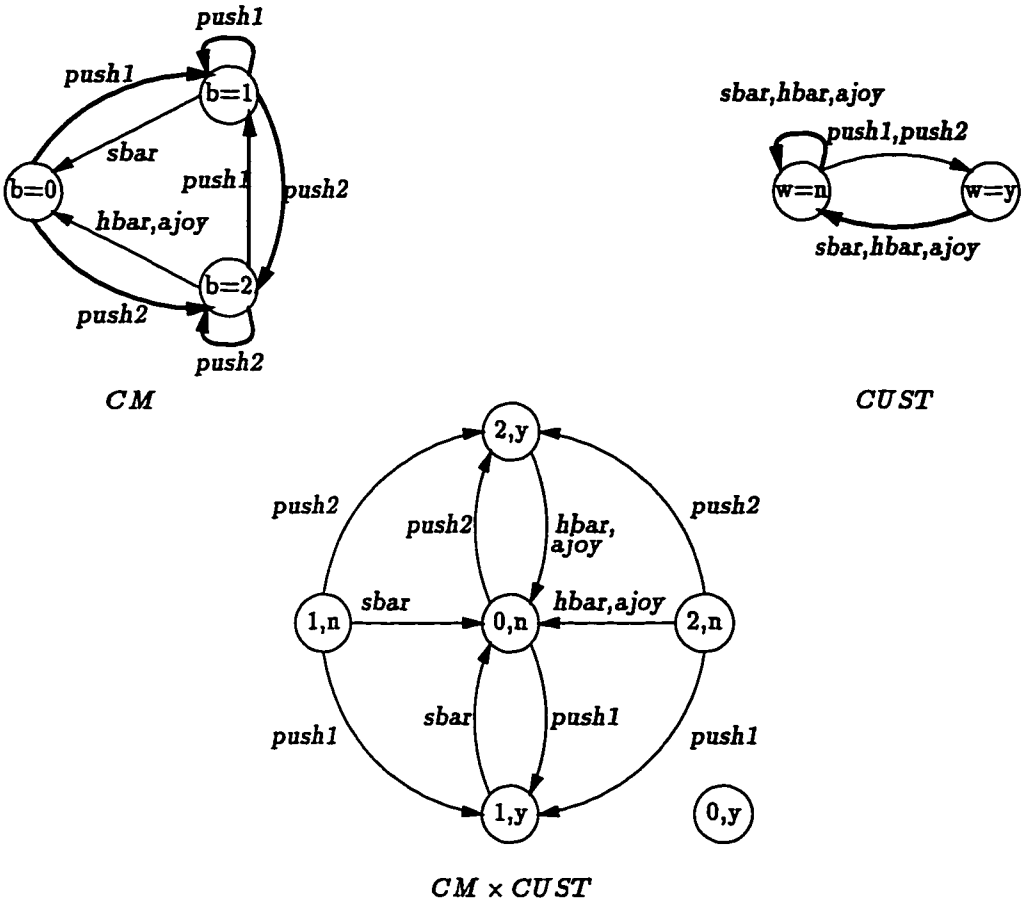

Figure 3.3: Customer and the Candy machine. The states (1,n),(2,n),(0,y) are unreachable if (0,n) is the initial state, but they can be initial states (sbar denotes skybar, hbar denotes heathbar, and ajoy denotes almondjoy).

Table 3.1: Action Table for *CM*.

Action Type	Action Name	Precondition	Effect
input	*push1*	True	button-pushed ← 1
	push2	True	button-pushed ← 2
output	*skybar*	button-pushed = 1	button-pushed ← 0
	heathbar	button-pushed = 2	button-pushed ← 0
	almondjoy	button-pushed = 2	button-pushed ← 0

Table 3.2: Action Table for *CUST*.

Action Type	Action Name	Precondition	Effect
input	*skybar*	True	waiting ← no
	heathbar	True	waiting ← no
	almondjoy	True	waiting ← no
output	*push1*	waiting = no	waiting ← yes
	push2	waiting = no	waiting ← yes

transformed to s' by the effects of e. *steps(CUST)* is defined similarly. Table 3.1 shows the actions for *CM* (no internal actions; states: button-pushed = 0 (waiting), button-pushed = 1, button-pushed = 2). Table 3.2 shows the actions for *CUST* (states: waiting = yes, waiting = no). In the composition $CM \times CUST$, the output action *push1* of *CUST* is identical to the input action *push1* of *CM*. The occurrence of *push1* causes both the candy machine and the customer to perform *push1*, causing button-pushed to be set to 1 in *CM*'s local state and waiting set to yes in *CUST*'s local state. This synchronization models a form of communication from the customer to the candy machine. As a result of this action, *CM* does an output action *skybar* which in turn is an input action for *CUST*, who gladly accepts it. Similarly, when the customer pushes button 2, the candy machine (nondeterministically) dispenses either a heathbar or an almondjoy, but not both. All actions of the composition become output actions in the composition. Figure 3.3 shows *CM*, *CUST* and their composition.

Chapter 4

Diagnosability

This chapter defines diagnosability in the context of I/O-automata. Since equivalent states generate identical event sequences, only minimal automata are considered. The initial state is recognized based upon the event strings it generates, which can be formulated as a problem in formal language learning from positive examples. Hence, the significance of the finite tell-tale property [1].

4.1 Minimality

Two states are *equivalent* if they generate the same event language. An automaton is minimal if and only if no two states are equivalent.

Lemma 4.1 *If AA', BB' of $M \times M'$ are equivalent, then A is equivalent to B in M and A' is equivalent to B' in M'.*

Lemma 4.2 *Minimality is not closed under composition.*

Proof: Figure 4.1 shows a counterexample. □

We now introduce a restricted version of minimality that is closed under composition.

Definition 4.1 If, after deleting all input actions of an automaton M, the resulting automaton M_{-i} is minimal, then M is *strongly minimal.* This is the same as saying that any pair of states are strongly nonequivalent in that $\ell el(A) \neq \ell el(B)$. We say

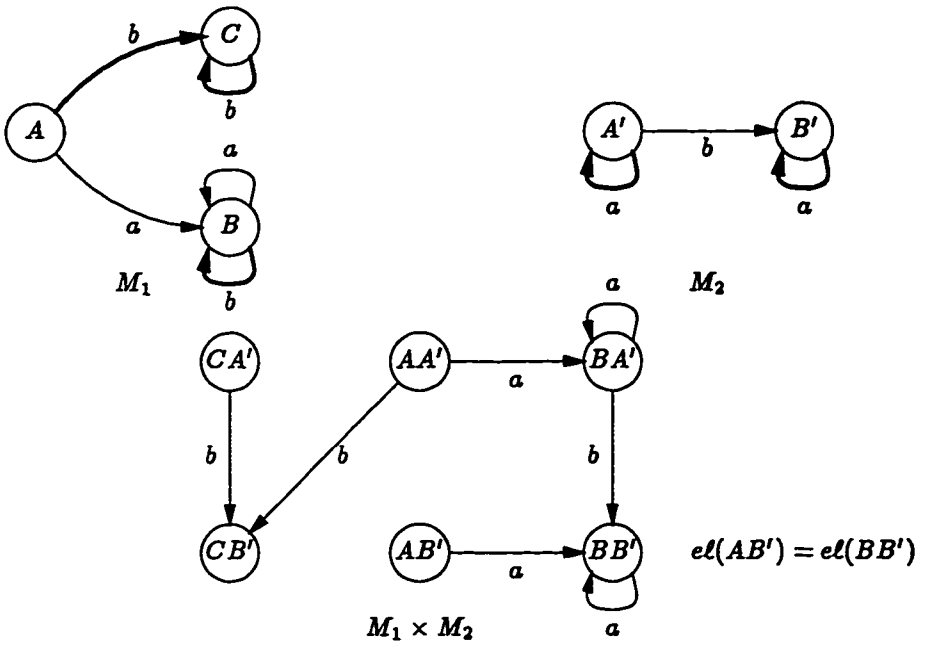

Figure 4.1: A counterexample to show minimality is not closed under composition; both M_1 and M_2 are minimal, but $M_1 \times M_2$ is not.

the event string strongly differentiates A and B if $s \in \ell el(A) \Delta \ell el(B)$, i.e., s is in one of $\ell el(A)$ or $\ell el(B)$, but not in both.

Lemma 4.3 *If M is strongly minimal and A, B are any two states in M, then for any M' and A', B' in M', AA' and BB' in $M \times M'$ are strongly non-equivalent.*

Proof: Let s be a local event sequence that distinguishes A and B; say $s \in \ell el(A) - \ell el(B)$. Then for any two states A', B' (same or different) of $M', e \in \ell el(AA') - \ell el(BB')$. This shows that AA' and BB' are not equivalent. □

Corollary 4.4 *Strong minimality is closed under composition.*

Definition 4.2 If $M \times M'$ is strongly minimal, then M is said to be *co-minimal* with M'.

Theorem 4.5 *If M'' is co-minimal with M and M', then $M \times M' \times M''$ is strongly minimal.*

Proof: Since $(M \times M'')_{-i}$ is minimal, every pair of states of $(M \times M'')_{-i}$ can be differentiated from each other by event strings that have no input actions. These differentiating strings will have to be in $\ell e\ell(M \times M' \times M'')$ as well. Let $AB'C''$ and $DE'F''$ be states in $M \times M' \times M''$. If $A \neq D$, then AC'' and DF'' are non-equivalent in $M \times M''$ and hence by lemma 4.3, $AB'C''$ and $DE'F''$ are not equivalent. Similarly, if $B' \neq E'$. Now, if $A = D$ and $B' = E'$, then AC'' and AF'' are non-equivalent in $(M \times M'')_{-i}$ and $AB'C''$ and $DE'F''$ are non-equivalent. □

Corollary 4.6 *If M is co-minimal with each $\{M_i : i \in I\}$ of a finite collection, $\prod M_i \times M$ is strongly minimal.*

4.2 Finite Tell-Tale

The Finite tell-tale property [1] was introduced in the context of language identification from positive examples. Here, we are concerned with identifying states based upon the (positive) example sequences differentiating them from other states. In the next section, we define distinguishing sequences, which are the positive samples from $e\ell(S)$ for any state S of the automaton. If the languages associated with the states of an automaton do not satisfy this condition, it is impossible to distinguish such states from others based upon the event sequences generated. For example, if A and B are two states of M, such that $e\ell(A) \subseteq e\ell(B)$, then there are no distinguishing strings for A, because every string can be generated if B were the starting state too.

Definition 4.3 An automaton has the finite tell-tale property *(ftt)* if (and only if) for each pair of distinct states A and B, $e\ell(A) \not\subseteq e\ell(B)$. The converse is also true. The *ftt* property says that a finite event string at A can distinguish it from B. In

particular, such an automaton is necessarily minimal. Let $el_k(A) = \{s : |s| \leq k, s \in el(A)\}$, for $k \geq 0$; $el_k(A)$ is a finite subset of $el(A)$. Clearly, $el(A) \not\subseteq el(B)$ implies that for some k, $el_k(A) \not\subseteq el_k(B)$; the same holds for all $k' > k$.

The following theorem shows that there is an apriori bound on k such that $el_k(A) \not\subseteq el_k(B)$.

Theorem 4.7 *If M is an n-state automaton with the ftt property, then for any pair of distinct states A and B, the finite subsets $el_k(A) \subset el(A)$ and $el_k(B) \subset el(B)$ are such that $el_k(A) \not\subseteq el_k(B)$, for some $k \leq \frac{n(n-1)}{2}$.*

Proof: Since A and B are distinguishable from each other, there exists an arbitrarily long event sequence that differentiates them. Let AB denote the initial uncertainty regarding the state. Now, suppose the occurrence of an event $e(\in el_1(A) \cap el_1(B))$ changes the uncertainty to $A'B'$. The number of distinct pairs in an n-state automaton is $\frac{n(n-1)}{2}$. Hence, if A and B are distinguishable from each other, the event sequence that distinguishes them cannot exceed $\frac{n(n-1)}{2}$. □

Lemma 4.8 *If M does not have the ftt property, then for any M', $M \times M'$ does not satisfy the ftt property.*

Proof: Let A and B be two distinct states of M such that $el(A) \subseteq el(B)$. Let A' be any state of M'. It is easy to see that $el(A \times A') \subseteq el(B \times A')$. □

Lemma 4.9 *ftt is not closed under composition.*

Proof: Figure 4.2 shows a counterexample. □

Lemma 4.10 *ftt does not imply strong minimality and vice-versa.*

Proof: Figure 4.3 presents an example to show *ftt* does not imply strong minimality. The converse is obvious. □

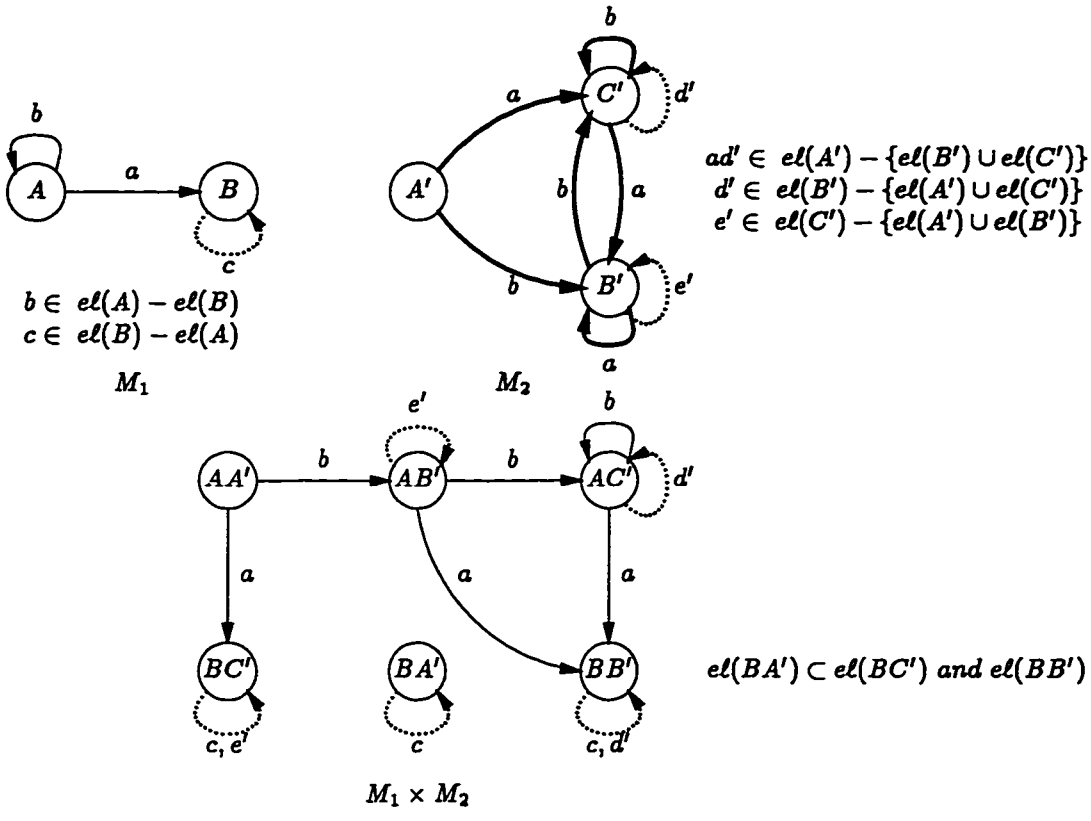

Figure 4.2: A counterexample to show *ftt* is not conserved over composition.

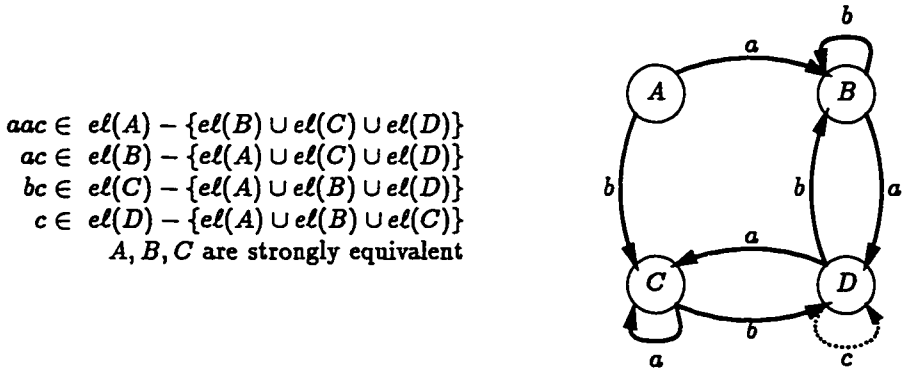

Figure 4.3: Example to show *ftt* does not imply strong minimality.

4.3 Distinguishing Sequences

Definition 4.4 If an event sequence can be generated only from a state S of an automaton, then it is a *distinguishing sequence* for S.

A state can have several distinguishing sequences. The set of distinguishing sequences for S is called the *distinguishing set* for S, denoted by $ds(S)$. Equivalent states cannot be distinguished from each other.

Definition 4.5 M is *diagnosable* if and only if every state of M has a distinguishing sequence.

Theorem 4.11 *(Diagnosability Theorem) Diagnosability is not closed under composition.*

Proof: Figure 4.4 shows a counterexample. □

Perhaps it is not surprising that diagnosability is not closed under composition. The definition of diagnosability involves existential quantification and composition can be thought of as a generalized $\wedge$ (logical AND) operation. It is well-known that the existential quantifier does not distribute over the $\wedge$ operation.

A minimal automaton is not diagnosable when the event language of any one (or more) of its states is a subset of the union of the event language of the rest of its states. For the example in figure 4.4, $el(BB') \subset \{el(AA') \cup el(AB') \cup el(BA')\}$.

Theorem 4.12 *If $M = \prod_i M_i$ is diagnosable, then each M_i is diagnosable.*

Proof: We show that there is a distinguishing sequence for each state A_i in M_i. Consider a state $A = (A_1, A_2, \cdots, A_n)$ in M whose ith component is A_i. Let $s = e_1 e_2 \cdots e_n$ be a distinguishing sequence for A. Then $s_i = s|M_i$ is a distinguishing

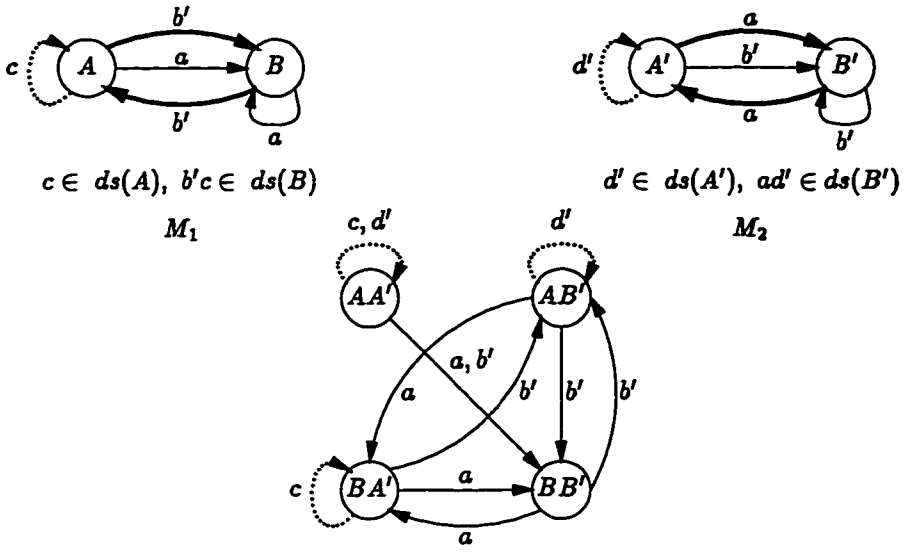

Figure 4.4: A counterexample to show diagnosability is not closed under composition.

sequence for A_i. Clearly, $s_i \in el(A_i)$. If $s_i \in el(A_i')$ for any other state A_i' in M_i, then $s \in el(A')$ where $A' = (A_1, A_2, \cdots, A_{i-1}, A_i', A_{i+1}, \cdots, A_n)$ which contradicts that $s \in ds(A)$. □

4.4 An Issue of Control

In Mealy machines, the application of an input sequence is referred to as an *experiment* on the machine. Every state has a transition defined for 0 as well as 1. Therefore it is possible to design sequences that *enforce* certain transitions on the machine. Whereas I/O-automata do not lend themselves to such complete control. Events *occur* asynchronously and spontaneously. Only input event transitions are defined at every state. Output and internal transitions may not be defined from every state, and at any time, the state of the automaton is unknown, therefore

it is impossible to enforce output or internal transitions. However, it is possible to *prevent* transitions from occurring when so desired. This is the key idea in the development of the debugger in Chapter 5. The debugger disables and enables transitions as the execution progresses so that only desirable sequences are generated.

4.5 Transformation Algorithm

This section presents an algorithm to transform a non-diagnosable automaton to a diagnosable one by augmenting the external event alphabet. The number of states remains unaltered. The idea is to destroy the inclusion of the event language of some states of an automaton by the rest of its states. The inclusion of the event language of one state by the event language of another state is eliminated by the *ftt* property, but the inclusion of the event language of a particular state by the event languages of several other states is not. This section develops an algorithm to achieves this. The following subsection presents some concepts necessary for describing the algorithm.

4.5.1 Uncertainties and Distinguishing Trees

Suppose a machine M can initially be in any one of its n states $\{S_1, S_2, \cdots, S_n\}$. The initial uncertainty regarding the state of the machine is denoted by $\langle S_1, S_2, \cdots, S_n \rangle$, with S_i's arranged in some order. We say that the event sequence s transforms the uncertainty U to $U' = \langle S'_1, S'_2, \cdots, S'_n \rangle$, where $S'_i : (S_i, s, S'_i)$; we write $'-'$ for S'_i in case $s \notin e\ell(S_i)$, i.e., s is not defined for S'_i. Note that U' may contain any of the symbols $'-'$ and the states of M, multiple times.

An uncertainty $U = \langle u_1, \cdots, u_n \rangle$ is called *singleton* (for state S_i), if $u_i \in states(M)$ and $u_j =' -'$ for all $j \neq i$. If U contains repeated entries other than $'-'$, it is called

ambiguous uncertainty. If $U \xrightarrow{e} U'$, where U' is an ambiguous uncertainty and U is not, and e is an input action, it is called *input ambiguity*. *Output ambiguity* and *internal ambiguity* are defined similarly. In figure 4.5, $\langle A, A \rangle$ is an ambiguous uncertainty, and $\langle B, - \rangle$ and $\langle -, B \rangle$ are singleton uncertainties.

Definition 4.6 Two ambiguous uncertainties $U = \langle u_1, \cdots, u_n \rangle$ and $U' = \langle u'_1, \cdots, u'_n \rangle$ are said to be *equivalent* if they contain the same set of distinct states S_i, denoted by $U \simeq U'$. For example, $\langle S_1, S_3, -, S_2 \rangle \simeq \langle -, S_1, S_2, S_3 \rangle$. The order of the states in U and U' is immaterial.

Definition 4.7 A distinguishing tree $dt(M)$ for an automaton M is a finite successor tree which shows the successive uncertainties obtained by application of events $e \in acts(M)$. An uncertainty node U in the jth level of $dt(M)$ is a terminal node if any of the following occur:

1. The uncertainty U is equivalent to an uncertainty U' at some other node in level $k \leq j$. (Only one such node at level j is expanded further).

2. The node is associated with an ambiguous uncertainty.

3. The node is associated with a singleton uncertainty.

Figure 4.5 shows the distinguishing tree for the automaton in figure 3.1. For an ambiguous uncertainty, it is impossible to determine the initial state.

Definition 4.8 A distinguishing tree is *complete* iff there is a singleton uncertainty for each state in the initial uncertainty.

Although $dt(M)$ is obtained in breadth-first fashion, this has no bearing upon the distinguishability of any state from any other. To distinguish between two states,

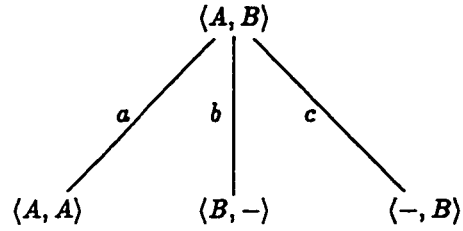

Figure 4.5: The distinguishing tree for the I/O-automaton in figure 1. $\langle B, - \rangle$ is a singleton uncertainty for A.

smaller trees may suffice (Theorem 4.7). The definition of the distinguishing tree leads to the following alternative characterization of diagnosability.

Definition 4.9 An automaton is diagnosable iff its distinguishing tree is complete.

Lemma 4.13 *(Projection Lemma) If* $\langle S_1, S_2 \rangle \xrightarrow{a} S_3$ *is an ambiguity in* $M \times M'$, *then either* $\langle S_1|M, S_2|M \rangle \xrightarrow{a} S_3|M$ *is an ambiguity in* M *or* $\langle S_1|M', S_2|M' \rangle \xrightarrow{a} S_3|M'$ *is an ambiguity in* M'.

Proof: $S_1 \neq S_2$ implies $S_1|M \neq S_2|M$ or $S_1|M' \neq S_2|M'$. In the former case, the ambiguity is in M and in the latter, it is in M'. □

The projection lemma states that every global ambiguity in the composition is the result of an ambiguity in one of its components. This implies that every resolvable ambiguity in the composition can be resolved in the component.

4.5.2 Relabelling

The *relabelling* operation modifies an automaton syntactically, while preserving its semantics. The operation renames some of the transitions of an automaton. Some automata can be transformed into a diagnosable form by relabelling. The effect of relabelling an automaton (figure 4.6) is to reduce the intersection of the event

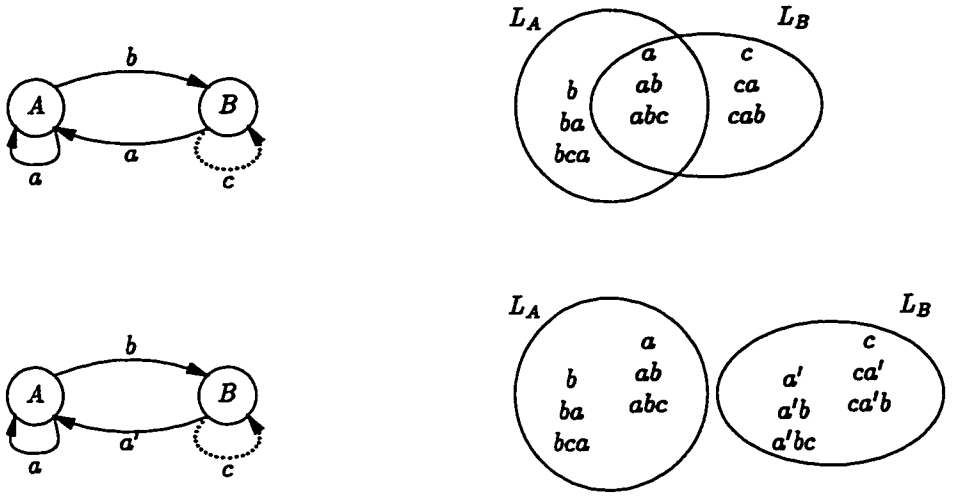

Figure 4.6: The effect of relabelling on an automaton.

languages of states. As it turns out, it is not always possible to make these event languages disjoint (Theorem 4.15).

Definition 4.10 If a transition $A \xrightarrow{a} B$ in M is replaced by $A \xrightarrow{a'} B$, the operation is called *relabelling*. The actions a and a' are considered to be *semantically equivalent*, denoted $a \equiv a'$.

Let M' be the automaton created as a result of such relabellings. Although $acts(M) \subseteq acts(M'))$, the "structure" of M is preserved by M'. The notion of semantic equivalence of actions naturally extends to event strings, event languages and automata. The semantic equivalence of two automata M and M' is denoted by $M' \equiv M$. Let $[a]$ denote the semantic equivalence class of a.

Theorem 4.14 *(Relabelling Theorem) If M has output or internal ambiguity, it can be resolved by relabelling.*

Proof: For a k-fold ambiguity $\langle S_1, S_2, \cdots, S_k \rangle \xrightarrow{a} S_j$, relabel the transition $S_i \xrightarrow{a} S_j$ by $S_i \xrightarrow{a_i} S_j, i \leq k$. The event languages of all states remain semantically unchanged

and the ambiguity is resolved. Clearly, although new events are added to the alphabet, transitions are only *replaced* and no new ambiguity is created. □

Definition 4.11 The automaton that results from the relabelling in Theorem 4.14 is denoted by M^R has no output or internal ambiguities. As a result $acts(M^R)$ may have more events than $acts(M)$.

Input ambiguity cannot be resolved by relabelling because of the constraint that every state must have a transition for all the input actions of an automaton.

Theorem 4.15 *(Unresolvability Theorem) For input ambiguity:*

1. *If M satisfies ftt, it cannot be resolved by relabelling.*
2. *If M does not satisfy ftt, under certain conditions it can be resolved by relabelling.*

Proof: Consider the ambiguity due to $S_1 \xrightarrow{a} S_3$ and $S_2 \xrightarrow{a} S_3$, where a is an input action. For finite state machines, the event languages can be expressed by a grammar:

$$S_1 = aS_3| \cdots |\ell S_i$$
$$S_2 = aS_3| \cdots |qS_j$$
$$\vdots$$
$$S_n = aS_k| \cdots |wS_m$$

where a, b, ℓ, q, w denote events; S_i are states (not necessarily distinct). To resolve the ambiguity, $S_2 \xrightarrow{a} S_3$ is replaced with $S_2 \xrightarrow{a'} S_3$. Since a and a' are input actions, it is required that a transition be defined for them from every state. Each rule in the above grammar has to be augmented to include a transition for both a and a'. For any state S not involved in the ambiguity, the same transition can be defined

for both a and a', $S \xrightarrow{a,a'} S'$. This does not introduce any new ambiguity due to S. For S_1 and S_2, the transition must be differentiated. Without loss of generality, we change S_1 and S_2 as follows:

$$S_1 = aS_3| \cdots |\ell S_i|a'S_p$$

$$S_2 = a'S_3| \cdots |qS_j|aS_q$$

where $S_p(\neq S_3)$ and $S_q(\neq S_3)$ avoid ambiguity for a' in the states S_1, S_2. Since M satisfies *ftt*, $e\ell(S_p) \not\subseteq e\ell(S)$ for any S. Since a and a' are semantically identical, and the new $e\ell(S_1)$ must remain unchanged, this is only possible if $e\ell(S_p) \subseteq e\ell(S_3)$, which is impossible.

2. If M does not satisfy *ftt*, then this is possible if and only if $e\ell(S_p) \subseteq e\ell(S_3)$. □

Input ambiguity is *unresolvable* whereas output and internal ambiguity are *resolvable*. Although the proof relies on the fact that input events must be defined from all states to show that relabelling is ineffective for input ambiguities, there is a more fundamental reason for the failure of relabelling. Suppose M_1 and M_2 are two automata, and $a \in in(M_1) \cap out(M_2)$. Now suppose we relax the requirement that every input action must be defined from every state and relabel some transitions in M_1, there is no way for M_2 to know which state M_1 is in and take an appropriate output action. In figure 4.7, M_2' cannot know whether to take output action a or a' without knowing which state M_1 is in. A diagnosable automaton can have unresolvable ambiguities in its distinguishing tree (Figure 4.5).

Suppose M_1 and M_2 are strongly compatible, $M = M_1 \times M_2$, $C = \{M_1, M_2\}$, and M_1^R denotes the relabelled form of M_1, with all output and internal ambiguities resolved. Suppose that $acts_{12} = \{e \mid e \in out(M_1) \cap in(M_2)\}$ and the symbols $[e] = \{e_1, e_2, \cdots, e_k\}$ were introduced in M_1^R to relabel a set of k transitions with the same label e by the transitions $S_i \xrightarrow{e_i} S$. Similarly, M_2^R denotes the relabelled

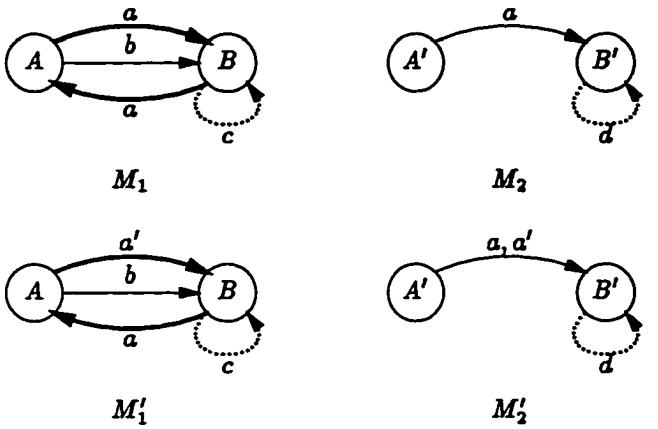

Figure 4.7: Unresolvability of input ambiguities: M_2' (in state A') does not know whether M_1' is in state A or B.

form of M_2 for the common symbols in $acts_{21}$. Since it is necessary to preserve the semantics of M, M_2^R must react identically to $[e]$. M_2^R can be augmented to obtain M_2^C by replacing each transition $A \xrightarrow{e} B$ by k transitions, $A \xrightarrow{e_i} B, 1 \leq i \leq k$, $in(M_2^C) = \bigcup_{e \in acts_{com}} [e] \cup in(M_2)$. M_1^C is similarly obtained from M_1^R. Augmentation neither creates nor resolves ambiguities. This operation is meaningful only with respect to a collection of strongly compatible automata.

Let $C = \{M_1, \cdots, M_n\}$ be a set of strongly compatible automata. M_1^C is obtained by first forming M_1^R and then augmenting it with respect to $M_{j\,(j \neq 1)}^R$. M_1^C does not depend on the order in which augmentation is applied to the automata. M_1^C has no resolvable ambiguities.

Theorem 4.16 *(Preservation Theorem) If* $M = M_1 \times \cdots \times M_n$ *and* $M' = M_1^C \times \cdots \times M_n^C$, *then* $M' \equiv M$, *and* M' *has no resolvable ambiguities.*

Proof: M' has the following properties:

- $in(M') = in(M)$
- $out(M') = out(M_1^C) \cup \cdots \cup out(M_n^C) \supseteq out(M)$

1. For each M_i, resolve every resolvable ambiguity.
2. Compose $M^R = \prod M_i^C$.
3. Construct the maximal distinguishing tree, $dt_{max}(M^R)$.
4. If $dt_{max}(M^R)$ is complete, then print M^R else print NIL; exit.

Figure 4.8: Algorithm *max-convert*.

- $int(M') = int(M_1^C) \cup \cdots \cup int(M_n^C) \supseteq int(M)$

$M' = M^R$ has no resolvable ambiguities. $M' \equiv M$ follows. □

Figure 4.8 presents algorithm *max-convert*, the maximal conversion algorithm. *max-convert* returns the diagnosable version of the automaton whenever it is possible and NIL otherwise. Figure 4.10 shows the diagnosable version of the non-diagnosable automaton in figure 4.4.

Theorem 4.17 *Algorithm max-convert is correct.*

Proof: By the *Projection lemma*, if $\langle S_1, \cdots, S_j \rangle \xrightarrow{a} S_k$ is an ambiguity in M^R, then $\langle S_1|M_i, \cdots, S_j|M_i \rangle \xrightarrow{a} S_k|M_i$ is an ambiguity in some M_i. By step 1 in the algorithm, this can only be due to an unresolvable ambiguity, in which case M^R has unresolvable ambiguities and the algorithm returns NIL. □

Lemma 4.18 *The distinguishing tree $dt_{max}(M^R)$ generated by max-convert is maximal and unique.*

Proof: $dt_{max}(M^R)$ is maximal because there are no resolvable ambiguities in M^R. The uniqueness of $dt_{max}(M^R)$ follows. □

Algorithm *max-convert* adds the maximum number of events to the alphabet. The problem of providing the minimal number of additional required events is hard. In general, *fast-convert* adds fewer events than *max-convert* (Figure 4.9). The

1. Compose $M = \prod M_i$.
2. Construct the distinguishing tree, $dt(M)$.
3. If $dt(M)$ is complete, print M and exit.
4. Let U denote the set of states in M without a corresponding singleton uncertainty.
5. If $U \neq \emptyset$, select $S \in U$:
   (a) If possible, resolve an ambiguity in M_i so that a singleton uncertainty for S is created in $dt(M)$. $U \leftarrow U - S$. Go to step 5.
   (b) Else print NIL and exit.
6. Compose $M^R = \prod M_i^C$.
7. Construct $dt(M^R)$.
8. Print M^R and exit.

Figure 4.9: Algorithm *fast-convert*.

uniqueness of $dt_{max}(M^R)$ plays a pivotal role in the definition of a unique debugger in Chapter 5.

Theorem 4.19 *Algorithm fast-convert is correct.*

Proof: Since relabelling does not introduce new ambiguities, each ambiguity can be resolved sequentially. □

Corollary 4.20 *The distinguishing tree, $dt(M^R)$ generated by fast-convert may not be unique.*

Proof: A state may have several corresponding singleton uncertainties in $dt_{max}(M^R)$. A single corresponding singleton uncertainty for each state in M^R is sufficient for the completeness of $dt(M^R)$. □

The inability to convert some automata to their diagnosable forms suggests that there is something inherent in them that disallows that.

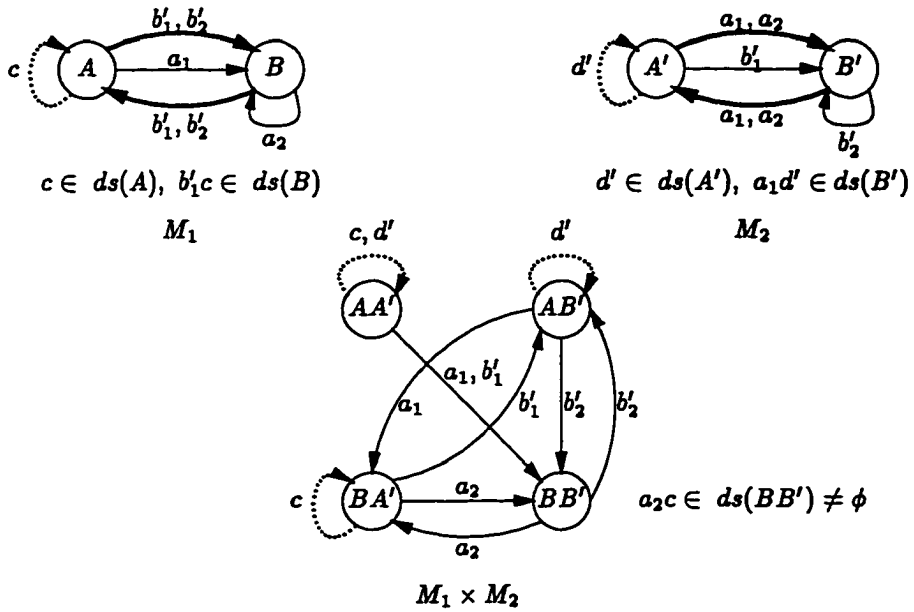

Figure 4.10: The diagnosable version of the automaton in figure 6 as a result of algorithm *convert*.

4.6 Ill-Posedness

A problem specification is a set of event strings comprising of external actions only. An automaton solves the specification if each of its external event strings are contained in this set. The following theorem states that every specification can be associated with a unique implementation.

Theorem 4.21 *Every specification can be implemented by a unique, minimal I/O-automaton with no internal actions.*

Proof: Let M be an I/O-automaton having internal actions satisfying the specification. M', the unique, minimal I/O-automaton with no internal actions, can be constructed from M as follows: Since internal actions are not visible externally, label every internal action of M with ϵ, the empty action. This results in a nondeterministic automaton which accepts the same external language as M. (The external language of any automaton can be obtained by removing the internal actions from

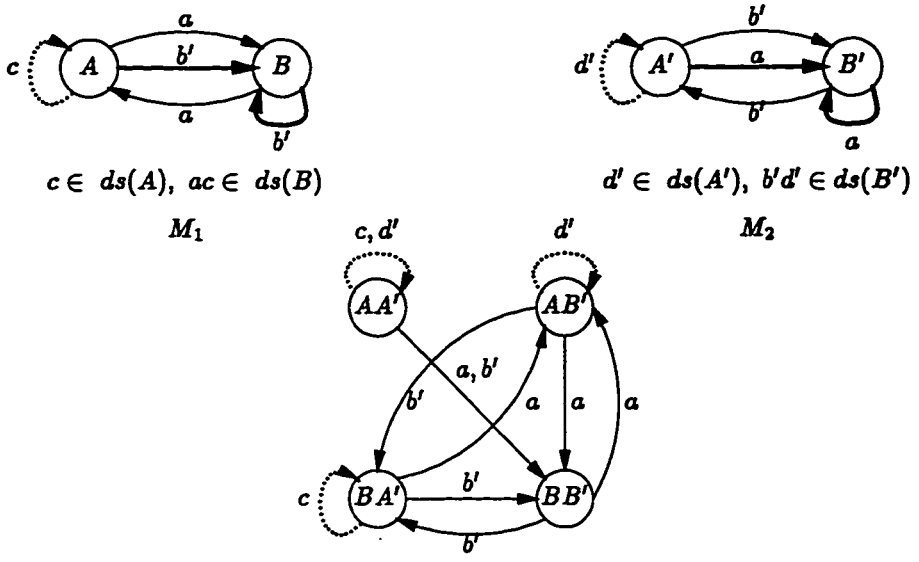

Figure 4.11: An ill-posed problem.

every string of the language of M). Every finite state nondeterministic automaton can be converted to a unique, minimal deterministic one. □

Definition 4.12 *(Ill-posedness of Specifications)* A specification M is ***ill-posed*** if there is no automaton which implements M, is diagnosable. Otherwise, it is said to be *well-posed.* Figure 4.11 shows an example of an ill-posed specification.

Lemma 4.22 *If M is ill-posed, so is $M \times M'$.*

Proof: Suppose S_1 and S_2 are not distinguishable in M due to an unresolvable ambiguity. Then (S_1, S') and (S_1, S') are not distinguishable in $M \times M'$. □

Theorem 4.23 *Well-posedness is not closed under composition.*

Proof: Figure 4.11 shows a counterexample. □

Chapter 5

Supervision

This chapter introduces a new operation, *supervision*, in the I/O-automaton model. The process of debugging can be described as the supervised execution of the debuggee by the debugger. A central task in debugging is to identify the state of the program. In our case, this is done by analyzing certain controlled executions of the program. The possible execution profiles of a program can be represented by its distinguishing tree. The distinguishing tree of an automaton may have paths leading to ambiguities. If the program is executed uncontrolled, it may follow such a path, thereby making it impossible to distinguish certain initial states of the automaton. The function of a debugger is to steer the execution of the automaton by disabling and enabling transitions, so that the transitions which lead to ambiguities are avoided. Supervision and composition commute: The debugging of a distributed program may require simulataneous control of several components. We first construct the supervisor for the composition, and then construct from it a supervisor for each component. These supervisors together form the distributed debugger. The distributed debugging problem is thus solved by reducing the problem to the debugging of a single program by a single debugger. An algorithm (*make-D*) to construct a debugger for a diagnosable automaton is presented. Algorithm *make-comp* extracts the local components from the global debugger constructed by *make-D*. A unique debugging complexity can be associated with every diagnosable program. Before

defining supervision, we describe superposition, an operation which was introduced to support monitoring (no control) of one automaton by another [17].

5.1 Superposition

Superposition, an asymmetric relationship that allows one system component to observe the state of another, appears in a number of models for distributed systems [5, 10]. The I/O automaton model has been extended to support superposition [17]. When one module is superimposed on another, the objective is to prevent the higher level module from interfering with the lower level one. That is, the higher level module should observe (but not modify) the state of the underlying module. In the I/O automaton model, this amounts to ensuring that in the superposition the higher level module does not place constraints on how the lower level module may modify its own variables. Therefore, superposition is defined only when the higher level module is 'unconstrained' for the variables of the lower level module. If X is a set of variables with domain $dom(X)$, A is said to be ***unconstrained*** for X iff A is an I/O automaton such that X is a subset of the state components of A and the transition relation of A places no restrictions on the values of the variables in X following any action. However, the set of locally-controlled actions enabled in a given state of A may depend on the values of the X variables in that state [18].

Let X be a set of variables with domain $dom(X)$, and let A and B be strongly compatible automata such that A is unconstrained for X. Then, the ***superposition of A on B with respect to X,*** denoted $C = \mathcal{S}(A, B, X)$, as follows:

1. $sig(C) = sig(A) \times sig(B)$ (usual signature composition),
2. $states(C) = states(A)$,

3. $start(C) = \{(p, x) \in start(A) : x \in start(B)\}$,
4. $steps(C)$ = all steps $((p', x'), \pi, (p, x))$ such that the following conditions hold:
   (a) $\pi \in sig(C)$
   (b) if $\pi \in sig(A)$, then $((p', x'), \pi, (p, x)) \in steps(A)$
   (c) if $\pi \in sig(B)$, then $(x', \pi, x) \in steps(B)$
   (d) if $\pi \notin sig(A)$, then $p = p'$
   (e) if $\pi \notin sig(B)$, then $x = x'$, and
5. $part(C) = part(A) \cup part(B)$.

Informally, the signature of the superposed automaton C is the composition of the signatures of A and B. The states of C are the same as the states of A, and the set of start states of C is the set of all start states of A such that the values of X agree with some start state of B. Any step of C for an action of A must also be a step of A. Similarly for B, when the projected on the variables in X. Essentially, the actions of A and B are enabled just as before, with automaton B controlling the values of the variables in X. The last two conditions in the definition for a step of C ensure that if a step does not involve an action of $A(B)$, then the state variables (private in case of A) of $A(B)$ are unchanged by the step. Since the superposition of one I/O automaton on another produces a new I/O automaton, all the standard definitions and results for I/O automata carry over to superposed automata. Superposition does not affect the set of executions of the underlying module thus preserving all properties of that module. In the following chapters, automaton should be read as I/O automaton unless explicitly mentioned otherwise.

5.2 Definition

We make the following assumptions about the debugger:

1. The debugger can observe all actions of the debuggee.

2. It can disable (and subsequently enable) the locally controlled actions of the debuggee.

This is the only way that the debugger restricts what a debuggee can or cannot do. The debugger cannot disable the input actions of the debuggee. Naturally, the debuggee cannot observe or place any restrictions on the execution of the debugger.

Definition 5.1 An I/O-automaton M is said to be *supervisable* by another I/O-automaton D, when the following conditions hold:

- $in(D) \supseteq in(M)$
- $out(D) = out(M)$
- $int(D) = int(M)$

A debugger may have extra input actions to maintain relevant non-local state information, but its input actions cannot interfere with the local actions of the debuggee.

Definition 5.2 The *supervision* of M by D, denoted by $\mathcal{S} = D \odot M$, is defined as follows:

1. $states(\mathcal{S}) = states(D) \times states(M)$,
2. $start(\mathcal{S}) = start(D) \times states(M)$,
3. $in(\mathcal{S}) = in(D)$,
4. $out(\mathcal{S}) = out(D)$,
5. $int(\mathcal{S}) = int(D)$,
6. $steps(\mathcal{S})$ equal to triplets $(\{A, B\}, e, \{A', B'\})$ such that:
   (a) $(A, e, A') \in steps(D)$, and $(B, e, B') \in steps(M)$, or
   (b) $e \in acts(D) - acts(M)$, and $B = B'$

Thus, the supervision operation is an asymmetric synchronous composition of the supervisor and the supervised automaton. It is synchronous in that the common actions must occur together. It is asymmetric in that the debugger can take independent input actions. The debuggee can be in any of its states when debugging starts.

Lemma 5.1 *S is an I/O automaton.*

Proof: By definition, all input actions are always enabled. □

We now consider the case where the debuggee has more than one component. Let $\{M_1, \ldots M_n\}$ be a strongly compatible collection of debuggee automata and $\{D_1, \ldots D_n\}$ a strongly compatible collection of their respective debuggers. $\{M_i\}$ are said to be supervisable by $\{D_i\}$ if each M_i is supervisable by D_i and for $i \neq j$, $in(D_i) \cap int(M_j) = \emptyset$.

Theorem 5.2 *(Commutativity Theorem) If $\{M_i\}$ are supervisable by $\{D_i\}$, then* $S = \prod_i (D_i \odot M_i) = (\prod_i D_i) \odot (\prod_i M_i)$.

Proof: Output and internal actions always survive composition. Input actions always survive supervision. Since the operations affect disjoint actions, the order of operations is not important. □

Figure 5.1 shows a schematic view of the distributed debugger.

5.3 Debugger Synthesis

Consider a diagnosable automaton $M = M_1 \times \cdots \times M_n$ and its distinguishing tree, $dt(M)$. Suppose $e \in out(M_i) \cap in(M_j)$ and leads to an ambiguity in M due to M_j, $\langle S_1|M_j, S_2|M_j \rangle \xrightarrow{e} S_3|M_j$. The function of the component debugger D_i is to

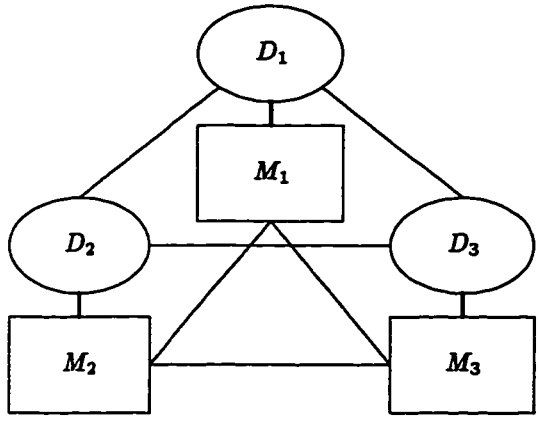

Figure 5.1: A schematic view of the distributed debugger ($M = M_1 \times M_2 \times M_3, D = D_1 \times D_2 \times D_3$.)

prevent an input ambiguity from occurring in M_j by disabling e in M_i if there is an uncertainty about the state of M being $\langle S_1, S_2 \rangle$. Similarly, all transitions that lead to an uncertainty that has been previously encountered in the execution should be avoided.

Thus, if e is locally controlled action in one of the components of M, it can be disabled. If e is an input action of M and leads to an ambiguity, the simulated environment can avoid it.

Figure 5.2 presents algorithm *make-D*, which constructs the supervisor D for M using the transitions in $dt(M)$. $dt(M)$ may have equivalent nodes in it. Let $dg(M)$ denote the digraph obtained by merging the equivalent nodes of $dt(M)$. *make-D* does a depth-first search on $dg(M)$. The debugger D is the tree of all simple (cycle-free) paths from the root of $dg(M)$ (same as the root of $dt(M)$) to the terminal nodes. Each such path in D (and $dg(M)$) is a distinguishing sequence for some state in M. From the point of view of implementation, $dg(M)$ need not be constructed explicitly. Instead, since only one of the equivalent nodes is expanded in $dt(M)$, the rest of the nodes can have a pointer to the expanded node, so $dt(M)$ can be traversed as

1. create start state s_{root} in D, for *root* in $dt(M)$.
2. mark all vertices "new"
3. search(root)

search(v)

1. mark v "old"
2. initialize disable set, $dis(v) = \emptyset$
3. For each child w of v ($v \xrightarrow{e} w$),
   (a) If w is marked "old" and $e \notin in(M)$, then $dis(s_v) = dis(s_v) \cup e$
   (b) Else if w is ambiguous and $e \notin in(M)$, then $dis(s_v) = dis(s_v) \cup e$
   (c) Else, create s_w in D; $search(w)$;
4. mark v "new"

Figure 5.2: Algorithm *make-D*.

$dg(M)$. M is said to be *closed* when it has no input actions. Since input actions cannot be disabled, D is a tree only if M is closed.

It is also assumed that M does not have any internal actions. This restriction is not very binding because given an automaton M' having internal actions, either the internal actions can be converted to output actions (which are not used by any other automaton) or they can be treated as empty non-deterministic transitions and an equivalent deterministic automaton with no internal transitions can be obtained (Theorem 4.21). Every vertex in $dt(M)$ corresponds to a state in D. The transitions in D are built from the transitions in $dt(M)$.

Theorem 5.3 *Algorithm make-D is correct. Also, the debugger D, constructed by make-D has the following properties:*

1. *D is a tree and is unique.*
2. *A vertex labelled s_v (created due to v in $dt_{max}(M)$) occurs at most once in any path from the root to a leaf in D.*

Proof: *make-D* traverses $dt_{max}(M)$ in depth-first fashion. Lines 5a and 2a ensure that no vertex is its own descendant (no loops are created). The effect of marking a

1. For each state S in D, create S_i in D_i.
2. For each state S_i in D_i,
   (a) For every transition $S \xrightarrow{e} S'$,
      i. If $e \in out(D_i)$, add $S_i \xrightarrow{e} S_i'$
      ii. If $e \notin out(D_i)$, add e to $in(D_i)$ and $S_i \xrightarrow{e} S_i'$
3. For every action $e \in in(D_i)$,
   (a) If e is not defined from state S, add a self-loop labelled e.

Figure 5.3: Algorithm *make-comp*.

visited vertex "new" (lines 5c and 2c) after its subtree has been traversed allows the vertex and its subtree to be visited in a different subtree. Every vertex of $dt_{max}(M)$ occurs at most once in any subtree of D. Ambiguous vertices due to output and internal events are avoided. No distinguishing tree $dt(M)$ has more distinguishing sequences than $dt_{max}(M)$. The debugger D created from $dt_{max}(M)$ is therefore maximal in the sense that it is least restrictive. The uniqueness of D follows from the uniqueness of dt_{max}. The debugger D can disable all output and internal actions of M (undesirable input actions can be avoided by simulation), and thus can disable all actions that are *necessary* to disable. The diagnosability of M ensures *sufficiency*. □

As shown in figure 5.3, algorithm *make-comp* constructs component debuggers $D_1, \cdots, D_n$ from D.

5.4 An Example

Figure 5.4 shows a debuggee automaton having two components, $M = M_1 \times M_2$. The maximal distinguishing tree $dt_{max}(M)$ of M, is shown in figure 5.5. The debugger

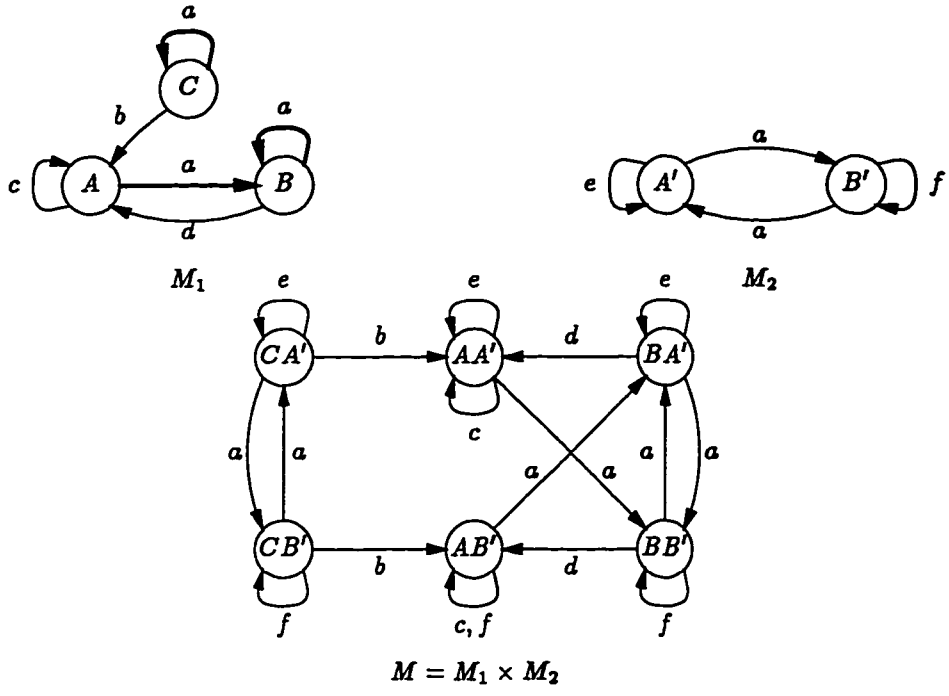

Figure 5.4: Debuggee automaton $M = M_1 \times M_2$.

D constructed by *make-D*, and its two components, D_1 and D_2, constructed by *make-comp* is shown in figure 5.6.

Initially, M can be in any one of its states, as indicated by the root vertex of $dt_{max}(M)$. The debugger D in its initial state, must disable event a because it leads to an ambiguity in M. This can only be done by D_2, since $a \in out(M_2)$. Note that $a \in in(D_1)$, and is therefore defined from every state of D_1. However, disabling a in M_2 means M_1 is presently unable to execute a. Suppose e is the first event that occurs. From figure 5.4, further occurrence of e would leave the uncertainty unchanged. Therefore, following the first occurrence of e, it should be disabled. In general, events may be disabled and enabled several times in the course of an execution. For example, the event sequence *bae* in figure 5.4 successfully identifies the state BA' in M and to allow this sequence, the debugger D can perform the

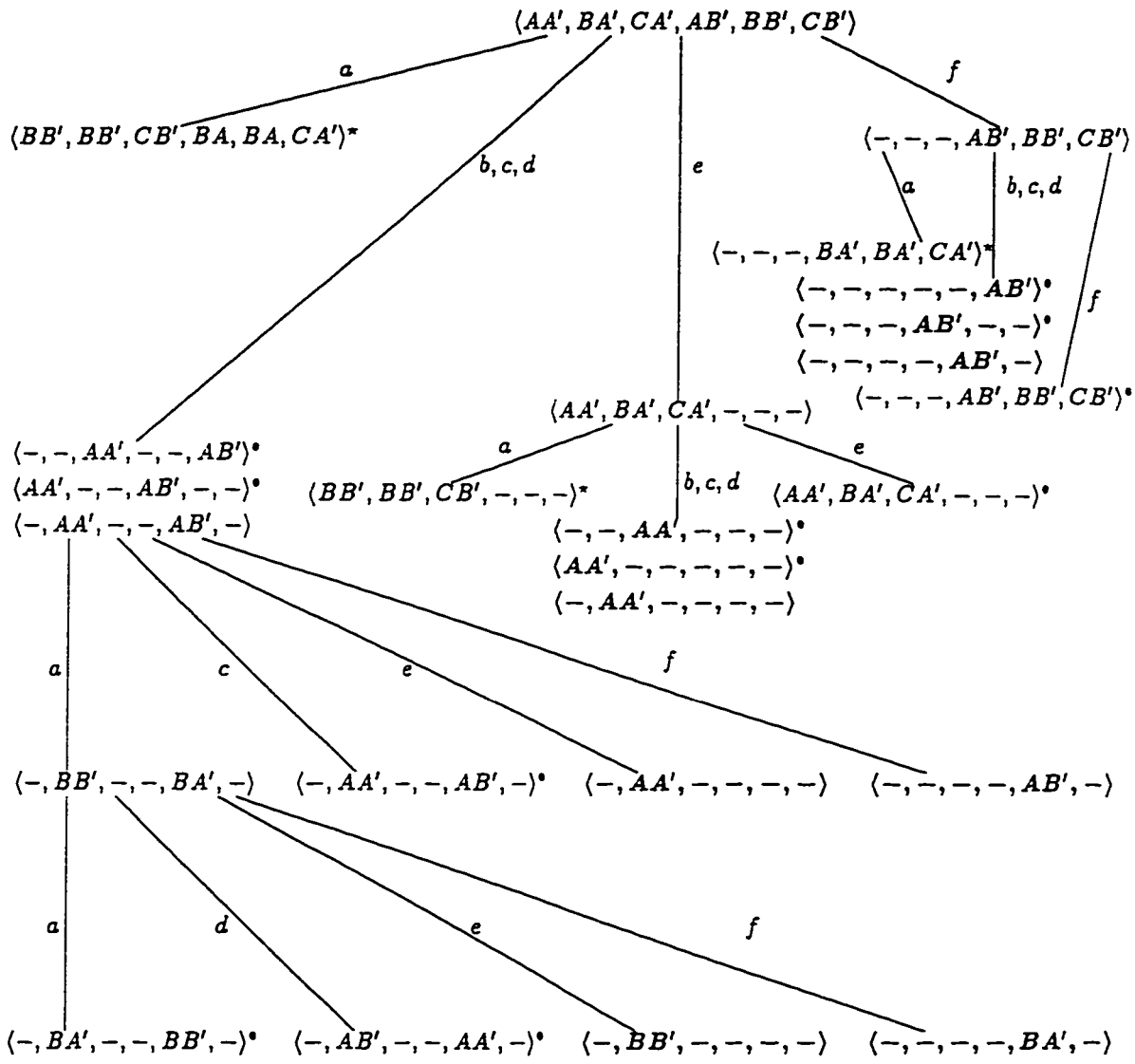

Figure 5.5: The maximal distinguishing tree $dt_{max}(M)$. Ambiguous uncertainties are marked (*), uncertainties which appear elsewhere in the tree are marked (•) and singleton uncertainties are in boldface.

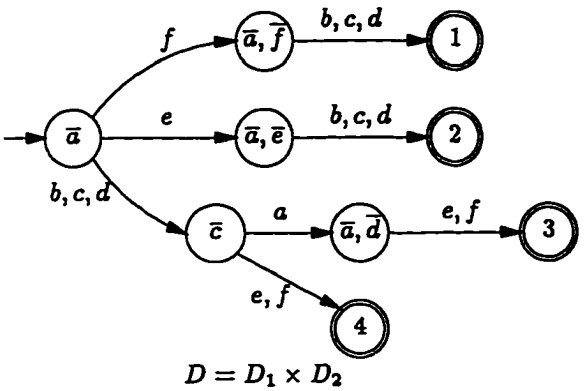

Figure 5.6: The debugger D for $M = M_1 \times M_2$. The start states are marked with an incoming arrow. The final states are encircled. The disabled events are depicted by a bar over them (*e.g.* $\bar{a}, \bar{e}$). The final state 1 identifies states AB', BB', CB'; 2 identifies AA', BA', CA'; 3,4 identifies BA', BB'. (Some edges have been merged for simplicity).

following actions: disable a (leave b enabled); following the occurrence of b, enable a; following the occurrence of a, disable a and d.

5.5 Complexity

Every specification has a unique, minimal automaton implementing it. These automata may be diagnosable or non-diagnosable. There is a unique maximal distinguishing tree for every diagnosable automaton. From the tree, a unique *minimal* debugger can be constructed. The debugger is minimal in that it imposes least restrictions on the debuggee. Every specification which is diagnosable has a unique debugger for it. Thus, a unique *debugging complexity* can be associated with every program in terms of the size of the debugger automaton.

It appears that the debugging complexity of a program is not related to its runtime complexity: The debugging complexity depends upon the acyclic paths in an automaton's graph, whereas the runtime is affected by how many times any cycle

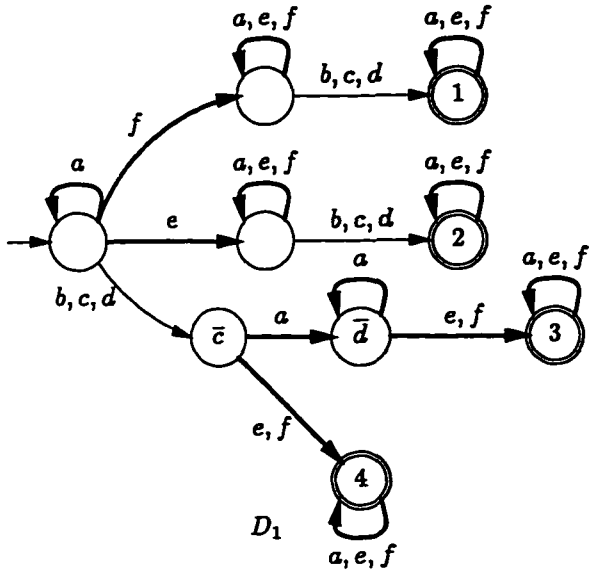

Figure 5.7: Debugger component $D_1(D_1 \times D_2 = D)$.

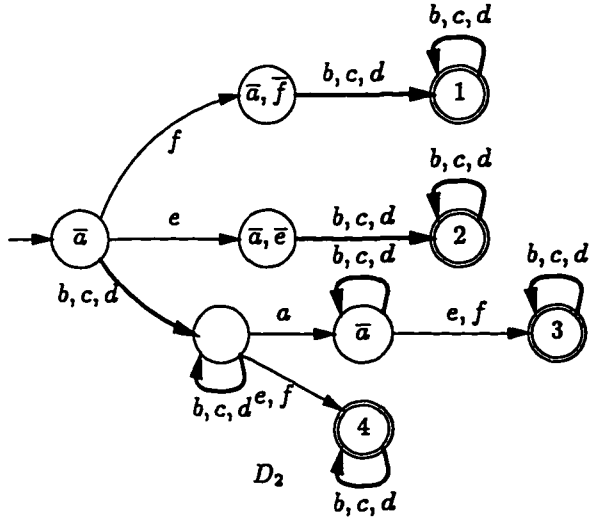

Figure 5.8: Debugger component $D_2(D_1 \times D_2 = D)$.

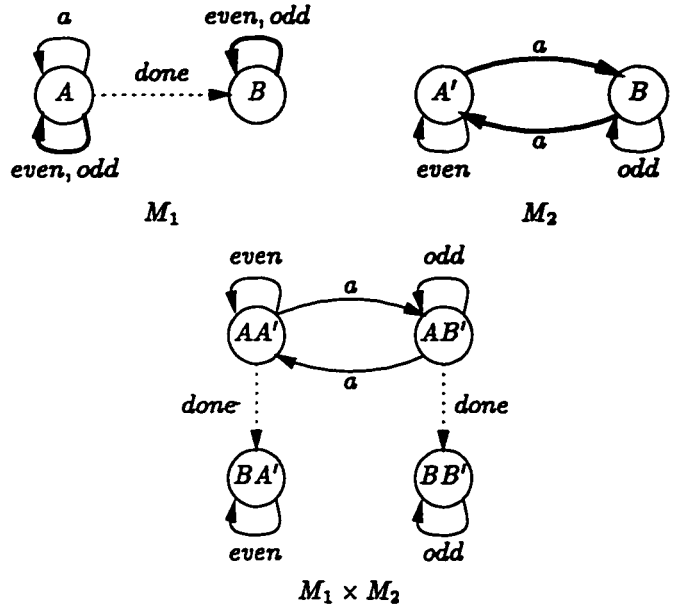

Figure 5.9: M_2 outputs *even* if M_1 executes a an even number of times, and *odd* otherwise.

in the automaton is executed. We have merely *introduced* this notion here; a clearer understanding would require further investigation. Figure 5.9 shows two automata M_1 and M_2; M_2 counts whether M_1 executes event a an odd or an even number of times, and outputs *even* and *odd* accordingly. The debugging complexity depends upon the number of *states*, whereas the execution time depends upon the number of *times* M_1 executes a.

Chapter 6

Towards Automated Debugging

6.1 Contributions

We have developed a framework to model distributed debugging. The notion of diagnosable distributed programs was introduced. Diagnosable programs are easier to debug, test and maintain. We gave algorithms which convert a non-diagnosable automaton into a diagnosable form, whenever such a transformation is possible. The I/O-automaton model was extended to support the process of debugging, which can be described as the controlled execution of the debuggee by the debugger. Algorithms to automatically build a debugger for a diagnosable program were developed. Relabelling guarantees that there are event strings exclusive to each state in the automaton. Supervision ensures that only such strings are executed.

Even though this study has been carried out in the context of the I/O-automaton model, most of the results seem to apply more generally. The definition of diagnosability is independent of the model. It also appears to us that for any non-trivial definition of composition, where some transitions in the component automata are not in the composition or vice-versa, diagnosability will not be closed under composition. The restriction that all output events are disjoint merely simplifies the notion of composition without any loss of generality. If the definition of composition had not required that, the relabelling operation would have explicitly distinguished among

them. The fact that input actions must be enabled from all states is a voluntary choice to disallow one automaton from controlling the output of another.

In Mealy machines, there is complete control because a transition is defined on 0/1 from every state, and the output binary string can be modified without restriction. As a result, every Mealy machine has a corresponding diagnosable form.

In the case of I/O-automata, our findings lead to a different conclusion. Only input actions can be enforced because they are defined from every state. However, this very characteristic makes it impossible to resolve ambiguities by relabelling, because relabelling cannot be done on any one automaton in isolation. Result: Not all automata are diagnosable; even those that are, may not have all ambiguities resolved. Consequence: A debugger is essential.

The debugger acts as the supervisor which prevents (whenever possible) the debuggee from making transitions which lead to ambiguities. A unique debugger can be built for every diagnosable program. This introduces complexity issues in connection with debugging. The size of the debugger gives an estimate of the complexity of controlling a given diagnosable program. Parallel with ordinary complexity theory, another performance metric in terms of diagnosability can thus be used to evaluate a program. Only further investigation will clarify if this metric is of any significance.

The main contributions of this thesis are:

- A theory of diagnosable *distributed* programs,
  - Diagnosability Theorem
  - Relabelling Theorem
  - Unresolvability Theorem
  - Ill-posedness of Specifications
- Algorithms to convert a non-diagnosable program to a diagnosable form whenever possible,

  - Algorithm *max-convert*
  - Algorithm *fast-convert*
- Extension of the I/O-automaton model to support distributed debugging,
  - Supervision operation
  - Commutativity Theorem
- Algorithms for automatic debugger synthesis for diagnosable programs,
  - Algorithm *make-D*
  - Algorithm *make-comp*
- Introduction of the notion of debugging complexity of programs.

6.2 Future Work

As a result of this investigation, several questions arise:

- In the finite state case, what can we say if the machines are incompletely specified ? Perhaps, ideas such as information losslessness, etc. can also be extended to the distributed case.
- The problem of adding the minimum number of events to make a given automaton diagnosable appears to be NP-Complete. Is this indeed true ?
- What are the implications of debugging complexity ? Is it related to the runtime complexity ?
- In the presence of internal (unobservable) events, how can one extend the design of the debugger ?
- Can this approach be extended to the non-finite state case ?
- What makes specifications ill-posed ? Can we make reasonable changes to the model to avoid this ?

We managed to formalize one aspect of distributed debugging: An event based approach for automatic state detection – perhaps a small step towards automated distributed debugging ?

Bibliography

[1] D. Angluin. Inductive inference of formal languages from positive data. Information and Control, **45**, 117–135, 1980.

[2] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **24**(1), 11–22, January 1989.

[3] B. Bloom. Constructing two-writer atomic registers. 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing Vancouver, British Columbia, Canada, 249–259.

[4] T.L. Booth. Sequential Machines and Automata Theory. John Wiley and Sons, Inc., 167–213, 1967.

[5] L. Bougé and N. Francez. A compositional approach to superimposition. Proceedings of the 15th Annual ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, 240–249, January 1989.

[6] B. Bruegge. A Portable Platform for Distributed Event Environments. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **26**(12), 184–193, December 1991.

[7] D. Callahan, J. Subhlok. Static Analysis of Low-level Synchronization. Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **24**(1), 100–111, January 1989.

[8] X. Cao and Y. Ho. Models of Discrete Event Dynamic Systems. IEEE Control Systems Magazine, **10** (4), pp. 69 – 76, 1990.

[9] K.M. Chandy, L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, **3**(1), 63–75, 1985.

[10] K.M. Chandy and J. Misra. A Foundation of Parallel Program Design. Addison-Wesley, Reading, MA, 1988.

[11] J-D. Choi and J.M. Stone. Balancing Runtime and Replay Costs in a Trace-and-Replay System. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **26**(12), 26–35, December 1991.

[12] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **26**(12), 155–166, December 1991.

[13] P. Emrath and D. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **24**(1), 89–99, January 1989.

[14] A. Gill. State-Identification experiments in finite automata. Information and Control, **4**, 132–154, 1961.

[15] A. Gill. Realization of input-output relations by sequential machines. J. Assoc. Comp. Mach., **13**, 33–42, 1966.

[16] A.P. Goldberg, A. Gopal, A. Lowry and R. Strom. Restoring Consistent Global States of Distributed Computations. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **26**(12), 144–154, December 1991.

[17] K. Goldman. A Compositional Model for Layered Distributed Systems. Proceedings of the 2nd Internation Conference on Concurrency Theory, LNCS 527, Springer-Verlag, pp. 220–234, August 1991.

[18] K. Goldman. Separating Structure from Function in the Specification and Design of Distributed Systems. Technical Report, WUCS-92-31, Washington University, MO, September 1992.

[19] K. Goldman. Distributed Algorithm Simulation Using I/O Automata. Ph.D. Thesis, MIT Laboratory of Computer Science, 1990.

[20] G. Goldszmidt, S. Katz and S. Yemini. Interactive Blackbox Debugging for Concurrent Languages. Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **24**(1), 271–282, January 1989.

[21] T.N. Hibbard. Least upper bounds on minimal terminal state experiments for two classes of sequential machines. J. Assoc. Comp. Mach., **8**, 601–612, 1961.

[22] C.A.R. Hoare. Communicating Sequential Processes Prentice-Hall International, 1985.

[23] C. Kilkpatrick and K. Schwan. ChaosMON - Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **26**(12), 57–67, December 1991.

[24] Z. Kohavi. Switching and Finite Automata Theory McGraw-Hill Inc., New York.

[25] Y. Lichtenstein and E. Shapiro. Concurrent Algorithmic Debugging. Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **24**(1), 248–260, January 1989.

[26] F. Lin and W.M. Wonham. Decentralized Supervisory Control for Discrete Event Systems Information Sciences, **44**(3), pp. 199 – 224, 1988.

[27] C.L. Liu. Determination of the final state of an automaton whose initial state is unknown. IEEE Trans. Electron. Computers, **EC-12**, 918–1921, 1963.

[28] N.A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. Proceedings of 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, 1987, 137–151.

[29] N.A. Lynch and M.R. Tuttle. An Introduction to Input/Output Automata. CWI-Quarterly, 2(3), 1989, 219–246.

[30] R. Milner. A Calculus of Communicating Systems. LNCS 92, Springer-Verlag, 1980.

[31] J. Misra and K.M. Chandy. Termination detection of diffusing computations in communicating sequential processes. ACM Trans. Prog. Lang. Syst., **4**(1), 37–43, January 1982.

[32] E.F. Moore. Gedanken-Experiments on Sequential Machines. Automata Studies, Annals of Mathematical Studies No. 34, Princeton University Press, Princeton, N.J., 129–153, 1956.

[33] Workshop summary in SIGPLAN Notices, **24**(1), ix–xxi, January, 1989.

[34] Workshop summary in SIGPLAN Notices, **26**(12), 1–14, December, 1991.

[35] Workshop summary in SIGPLAN Notices, **28**(12), vi–xix, December, 1993.

[36] P.J. Ramadge and W.M. Wonham. Supervisory Control of a Class of Discrete Event Processes. SIAM J. Control and Optimization, **25**(1), pp. 206–230, 1987.

[37] J. Stone. A Graphical Representation of Concurrent Processes. Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, **24**(1), 226–235, January 1989.

[38] M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. Master's Thesis, Massachussets Institute of Technology, MIT/LCS/TR-387, 1987.

Vita

Amit Anil Nanavati received his Bachelor of Engineering in Computer Science from Maharaja Sayajirao University of Baroda, India, in 1989. He joined the Computer Science department at Louisiana State University in Fall, 1990. He received his Master of Science in 1994. His areas of interest include Distributed Systems, Automata Theory, Graph Theory and Artificial Intelligence.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Amit Anil Nanavati

Major Field: Computer Science

Title of Dissertation: Designing Diagnosable Distributed Programs

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

March 8, 1996