REPORT CSG-18 AUGUST 1983

CSL COORDINATED SCIENCE LABORATORY
COMPUTER SYSTEMS GROUP

FAULT TOLERANT BUS COMMUNICATION PROTOCOLS FOR COMPUTER SYSTEMS

LEONARD HOWARD POLLARD

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
FAULT TOLERANT BUS COMMUNICATION PROTOCOLS FOR COMPUTER SYSTEMS

5. TYPE OF REPORT & PERIOD COVERED
Technical Report

6. PERFORMING ORG. REPORT NUMBER
CSG-18

7. AUTHOR(s)
LEONARD HOWARD POLLARD

8. CONTRACT OR GRANT NUMBER(s)
N00039-80-C-0556

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Electronics Systems Command,
VHSIC Program

12. REPORT DATE
August 1983

13. NUMBER OF PAGES
163

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
bus communication, communication protocols, error detection, fault-tolerance, data transmission, time-shared bus

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Bus systems form the communication medium for computers, and a great deal of effort has been devoted to detecting errors which occur as information is transferred from one module to another. In this thesis we look at the problem of not only detecting faults which occur, but continuing to function in the presence of those faults. The lines of a bus are grouped together into two classes: synchronous address and data lines, and the control lines which govern the action of the synchronous lines.

The fault model which is assumed for the synchronous lines includes not only the classical stuck-at fault, but also bridging faults and transient faults. Two algorithms are presented which use time redundancy to guarantee correct transfer of information in the presence of a single fault. One requires a single retry and is applicable for stuck-at faults and adjacent bridging faults. The other algorithm removes the adjacency requirement, but requires either one or two retrys, depending on the type of fault.

The modules comprising the protocol system are represented using state machines, and the action of the system is monitored by observing the states of the modules and the levels of the bus lines. A State Machine Language (SML) is developed to represent a protocol system. SML representations of the modules form the input to a protocol exercise system which simulates the action of the system and identifies errors which occur.

Knowledge of the prescribed behavior of the control lines allows the presence of stuck-at faults to be detected by the use of time-out escape sequences. The knowledge of the behavior of the control lines also permits dual-rail control signals to be used to guarantee continued operation in the presence of single faults. The expected levels of the signals as the protocol sequences through its actions allow identification of lines which are stuck at an incorrect value; the incorrect line can then be ignored as the module continues to function. Three algorithms are presented which will convent state machines for single-rail control signals to state machines which accommodate dual-rail signals. The system cost associated with this technique consists of the additional lines needed for dual-rail control signals and for implementing the time redundant transfer algorithms, and the additional hardware needed to implement the algorithms. For a standard bus this means about a 40% increase in the number of lines and approximately doubling the hardware dedicated to the bus control function.

FAULT TOLERANT BUS COMMUNICATION PROTOCOLS FOR COMPUTER SYSTEMS

BY

LEONARD HOWARD POLLARD

B.S., Utah State University, 1971
M.S., Utah State University, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1983

Urbana, Illinois

ABSTRACT

Bus systems form the communication medium for computers, and a great deal of effort has been devoted to detecting errors which occur as information is transferred from one module to another. In this thesis we look at the problem of not only detecting faults which occur, but continuing to function in the presense of those faults. The lines of a bus are grouped together into two classes: synchronous address and data lines, and the control lines which govern the action of the synchronous lines.

The fault model which is assumed for the synchronous lines includes not only the classical stuck-at fault, but also bridging faults and transient faults. Two algorithms are presented which use time redundancy to guarantee correct transfer of information in the presense of a single fault. One requires a single retry and is applicable for stuck-at faults and adjacent bridging faults. The other algorithm removes the adjacency requirement, but requires either one or two retrys, depending on the type of fault.

The modules comprising the protocol system are represented using state machines, and the action of the system is monitored by observing the states of the modules and the levels of the bus lines. A State Machine Language (SML) is developed to represent a protocol system. SML representations of the modules form the input to a protocol exercise system which simulates the actions of the system and identifies errors which occur.

Knowledge of the prescribed behavior of the control lines allows the presense of stuck-at faults to be detected by the use of time-out escape sequences. The knowledge of the behavior of the control lines also permits dual-rail control signals to be used to guarantee continued operation in the presense of single faults. The expected levels of the signals as the protocol

sequences through its actions allow identification of lines which are stuck at an incorrect value; the incorrect line can then be ignored as the module continues to function. Three algorithms are presented which will convert state machines for single-rail control signals to state machines which accommodate dual-rail signals. The system cost associated with this technique consists of the additional lines needed for dual-rail control signals and for implementing the time redundant transfer algorithms, and the additional hardware needed to implement the algorithms. For a standard bus this means about a 40% increase in the number of lines and approximately doubling the hardware dedicated to the bus control function.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Introduction

In this thesis we treat the following question:

> "How can information be transferred correctly in the presence of faults, between digital systems connected by a bus?"

Bus systems form a central method for information transfer in digital systems, transferring data and addresses between the different modules, and allowing the system to perform useful work. Busing techniques are not only important as an interconnection path between the modules of a computer, but they also play an important part in the communication tasks of the modules themselves. This includes data exchanges on the boards of a system, and also internal communication paths for integrated circuits which are growing ever larger. And as the feature size decreases in integrated circuits, the susceptibility to noise increases. This increase of awareness of electrical faults, and their impact on system behavior, is not limited to the small circuits; striving to increase system performance to the limits of technology increases the probability that faults will be a factor which needs to be handled.

In order to allow continued system operation in the presence of these faults, a set of techniques and policies is needed which will provide tolerance to the effects. This collection of techniques and policies will form a protocol which can be used to prevent system failure and permit operation in

the presence of the faults. This protocol needs to provide the ability to detect the presence of a fault, identify the fault for reporting and repair purposes, and still maintain correct operation, even if the operation is performed at a degraded speed.

1.2. Bus Level Protocols

The word 'protocol' has appeared many times in the literature and has been used to represent many different types of interactions. Perhaps the most pervasive use of the word is to refer to the set of rules governing the interaction of a number of computer systems. Examples of this type of system include ARPANET, DECNET, SNA, ETHERNET, and other related host-to-host communication methods. These protocols govern the action of the various computers which are connected together, and concern themselves with such issues as routing algorithms, retransmission of data assumed to be lost, management of temporary storage in the various computers to permit message exchange, and other interactions which are needed to transfer information from one site to another.

The word 'protocol' has also been used to refer to the rules which have been set up to provide standardization of basic system functions from one system to another. Examples of this are a file transfer protocol [1], a virtual terminal protocol [2], and tape format protocols.

Still another sense of the word represents the specification of the physical elements involved in the communication process [3]. This includes the mechanical and electrical elements of an interconnection scheme, such as identifying the connectors involved in a specific product and specification of the voltage and current levels to be used.

In this thesis the word 'protocol' is taken to mean the set of rules and procedures which specify the interaction of modules which are connected to a bus structure and whose source of system knowledge is limited to the signals which form the bus. Although we will be concerned with identification of the signals which make up the bus, or the portion of the bus required for a specific interaction, we will not identify the technology or the voltage (or current) levels. Rather, we will assume that a signal is 'asserted', 'unasserted', or faulty, either from a 'stuck-at' fault or a fault which is characterized by the logical bridging of two signals. We also do not consider those protocol rules mentioned above which are concerned with more global, multi-machine matters, such as the routing algorithms or storage allocation procedures. Thus, we limit ourselves to that set of rules which concerns the signal level of a bus system.

1.3. Bus Level System Representation

The bus level model of a system as viewed in this thesis is shown in Figure 1.1. This figure shows several independent modules which are connected by a bus structure. The different modules are composed of a functional part and an interface part. The functional portion of a module implements some system specific function which is independent of the protocol which ties the modules together. Examples of functional modules are memories, interfaces to peripheral equipment, and processors.

The interface portion of a module is that hardware which connects the functional part to the actual lines of the bus to communicate with other

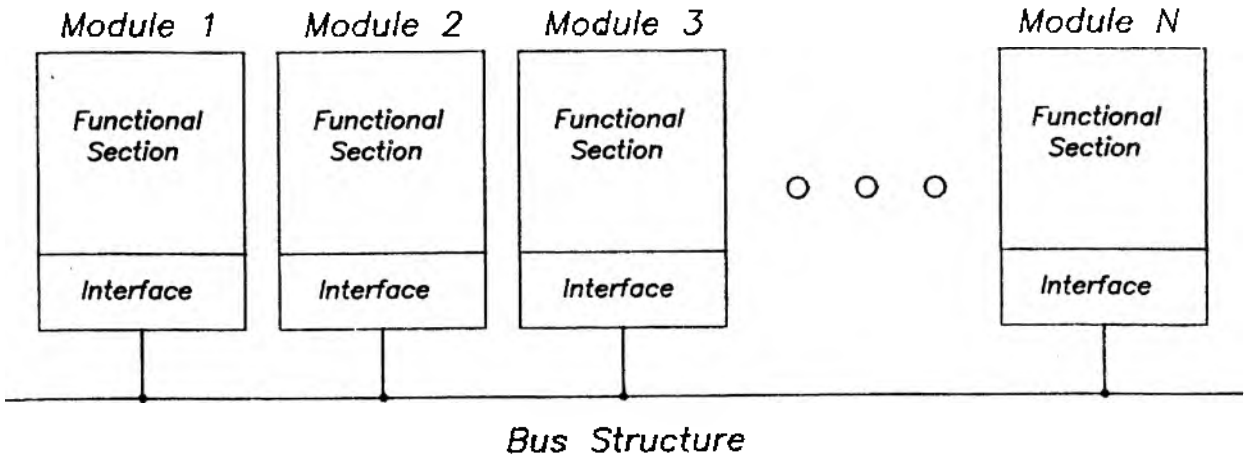


Figure 1.1. Functional Model of Protocol System.

members of the system. The interface portion contains the hardware to implement the algorithms and control procedures which are required by a protocol. Thus the interface portion receives direction from the functional portion as to what type of interaction is needed with the other modules which comprise the system, and the interface portion operates according to the protocol of the system to communicate with the other modules in the system.

The bus itself consists of a number of physical lines which join the different interface modules together. Some lines provide a connection which is common to all of the modules, while others may provide a daisy-chain type of connection between modules. No properties are assumed for these connections; the actual drivers and receivers are part of the different interface modules. Thus, whether a line assumes the properties of a common collector, tri-state, or other interconnection method is determined by the electronics of the interface module.

The principal function of the bus is to transfer information from one module to another. The signal lines which accomplish this are divided into two groups: the lines which carry information, such as address and data lines (collectively called data lines in this thesis), and the lines which control the interaction, such as request and acknowledge lines (called control lines). Although serial data lines have been used to satisfy the needs of some systems, most computers utilize parallel data lines for information transfer to achieve higher data rates. In this thesis we are concerned mainly with the parallel mode of data transfer, although the control methods presented here will also work with serial data lines.

1.4. An Overview of This Research

The division of the bus signals into two different groups gives rise to two different methods of modeling the lines and the algorithms which deal with them. The busing of synchronous information on parallel lines is treated by referring to the lines as a bus with no timing constraints. The information on these lines is assumed to be present when needed by the algorithms and no requirements are placed on the signal arrival time. Single error detection in this case is provided by a single parity bit which is added to the existing data lines by the sending interface to assure that all transfers on the bus have correct parity. Our goal is to provide not only detection, but also correction as well. This is accomplished by adding enough redundancy to assure that single bit errors can be corrected. This redundancy can be supplied by adding sufficient lines to encode this information into each transfer through codes like the Hamming code, or the redundancy could be obtained by

retransferring the data for a time redundancy technique. In Chapter 2 we present two algorithms for the correct transfer of information using time redundancy. These algorithms extend the work previously done by Shedletsky for stuck-at errors [4], as well as that done by Agrawal and Agarwal [5], who extended the fault model to include bridging faults as well as stuck-at faults.

The control signals which govern transfers across the data lines are the links which join the modules of the bus together, and they provide the communication between independent, asynchronous controllers. Systems composed of independent modules have been modeled and studied in order to gain insight into the action of the system as a whole. Protocol systems provide examples of a variety of modeling techniques. The UCLA graph was introduced to model parallel computational systems and applied to higher level protocols [6]. Petri-nets have been applied in diverse studies concerning protocols [7,8]. State machines have been incorporated in models representing the interaction of systems [9,10,11]. Other approaches to the study of protocol interactions include abstract data types [12] and programming languages [13]. In Chapter 3 we examine in detail the various methods used to represent the action of independent modules and present the rationale for selecting state machines for the representation of the signal level bus protocols.

The representation of the action of the various modules which are attached to the bus is presented with the use of state machines which indicate the response of the module to the various signals on the bus. This response action includes the signals asserted by the interface which provide the only

information that the other modules of the system will have concerning the action of the first module. Chapter 3 details this modeling method and introduces steps which can be taken to detect and correct errors which occur on the control lines. These steps include methods which use time redundancy to detect errors which occur on the control lines when detection alone is sufficient and methods to guarantee correct operation in the presence of faults using dual-rail redundant signals when detection and correction are needed. These methods are applicable to any physical level protocol which can be modeled as explained in Chapter 3.

A tool which can be used to examine the properties of protocols is the general purpose computer. Previously computers have analyzed systems by performing reachability analyses on graph model representations [14], by system state examinations [15], and by the use of theorem proving when the protocol is appropriately represented [12]. In order to utilize the power of the general purpose computer to examine the effect of faults on bus level protocol systems, we have developed a language to describe the state machine protocol representation. This language is detailed in Chapter 4. The use of the computer to exercise the protocol from its state machine description is a valuable tool which tests the effectiveness of error detection and fault tolerant techniques.

Various techniques have been employed to detect and correct errors in data transfers, such as Hamming codes and time redundancy techniques like those presented in Chapter 2. However, little is known concerning the action of the signals which control the interactions on the data paths. One tech-

nique which has been used is Triple Modular Redundancy (TMR) where all of the lines are triplicated and voting is done on the resulting signals [16,17]. Chapter 5 presents a new approach to the control signal problem which utilizes time redundancy and dual-rail signals. Knowing the expected behavior of the protocol system allows us to identify a faulty line and continue to operate. Chapter 6 extends the techniques of Chapter 5 to the arbitration problem. Chapter 7 examines an existing bus to determine the cost of implementing the fault tolerant techniques of this thesis. Chapter 8 provides a conclusion to this work and some suggestions for further research.

CHAPTER 2

CORRECT TRANSMISSION OF SYNCHRONOUS DATA USING TIME REDUNDANCY

2.1. Introduction

The major function of a bus in a computer system is to transfer data values from one module to another. Although serial buses are used when circumstances permit, data transfers primarily use parallel lines to achieve high data rates. Detection of single errors on a set of parallel data lines can be accomplished with the addition of a parity bit. If correction is required as well as detection, then sufficient redundancy must be included to locate the error, and this can cost several lines in addition to the data lines. However, another approach is to retry the exchange when an error is detected using redundancy in time instead of redundancy in space to provide correction.

Stuck-at errors can be corrected by using time redundancy with a very simple scheme called Alternate Data Retry, as suggested by Shedletsky [4]. When an error occurs, this method calls for the retransmission of data inverted from the original sense so that the stuck-at fault will be masked by the inverted data value; hence, the retransmitted data are received without error.

Agrawal and Agarwal [5] have proposed a different algorithm which will correctly transfer the information not only in the presence of a stuck-at fault, but also in the case of logically adjacent bridging faults. Their algorithm uses a recursive method of finding and correcting the error caused by the fault. The fault could be permanent or transient, so long as it occurs for the duration of the operation.

The model which is assumed for the algorithms presented in the following sections is shown in Figure 2.1. It is assumed that information is present in Module A for transmission to Module B. The interface between the two modules is a bus structure which has a data path sufficiently wide to transfer a complete word in a single cycle with a parity bit added for single error detection. Also, the interfaces to the bus structure have the capability of encoding/decoding the information as specified by the algorithms and handling any retransmissions which may be required.

The fault model used for these algorithms assumes a single line error, which may be caused by a single line stuck at some logical value, by a

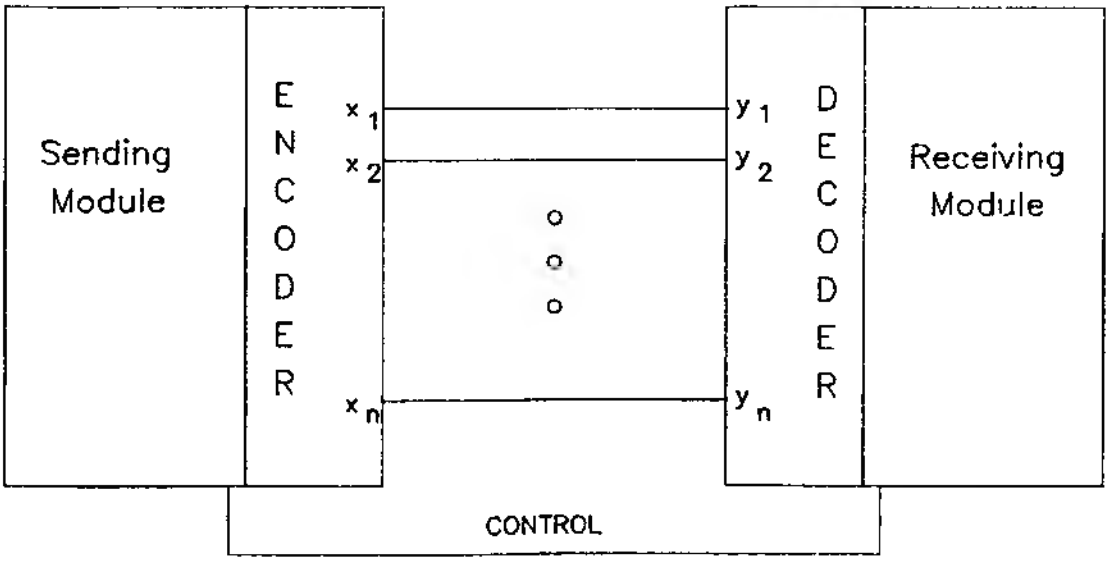


Figure 2.1. Model for Time Redundancy Transfer Algorithms.

transient noise on the line, or by a bridge (electrical short) between two lines. Bridging faults can occur either in the interface between the modules and the bus, or on the bus structure itself. The most obvious situation is that lines which are physically adjacent have created a bridging fault by means of a solder bridge or some other physical or electrical fault. This fault may manifest itself as either the logical-OR of the two lines or the logical-AND of the lines. In those systems which use physically contiguous lines to create the bus system and in which the lines are not only physically but logically adjacent, the algorithm presented in Section 2.2 would be sufficient to produce correct transfers. However, in most systems as well as most chips which are constructed, the lines of a communication path which are physically adjacent are not necessarily logically adjacent. Thus, malfunctions can occur which will cause bridging between two lines which are not logically adjacent. For this case the algorithm of Section 2.3 would be needed for proper information transfer. The faults which can occur include both permanent and transient faults, and the correctness of the algorithms is demonstrated for both cases.

2.2. Correction of Errors Due to Bridging of Logically Adjacent Lines

For those systems in which the bridging faults are confined to logically adjacent lines, correct transfer of information can be guaranteed with a single retry as demonstrated by the following algorithm. The notation "X -> Y" stands for "transmit X to Y" where X is the transmitted data and Y is the received data. The notation x_i is used to indicate the i^{th} bit of X starting from the left. Thus, $X = (x_1, x_2, \ldots, x_i, \ldots, x_n)$.

For those systems in which the bridging faults are confined to logically adjacent lines, correct transfer of information can be guaranteed with a single retry as demonstrated by the following algorithm.

Algorithm 2.1

Step 1: X -> Y. If there is no parity error, then Y is the correct data. Stop.

Step 2: X1 -> Y1, where X1 is formed by rotating $\overline{X}$ one bit to the left.

Step 3: Let Y2 be obtained by rotating Y1 one bit to the right.
Define S = Y ⊕ Y2.

Step 4: For each bit position in Y, call it k, invert the bit y_k to form the correct result if and only if $s_k = 0$ and $s_{k-1} = 1$. (Subscript arithmetic is done modulo word size to make it circular.)

Theorem: Algorithm 2.1 will ensure that the correct result is obtained in one retry at most in the presence of one stuck-at fault, one transient fault, or one AND or OR bridge fault on logically adjacent lines in the data transfer path.

Proof: We will consider the different types of faults separately.

1) Stuck-at Fault. Let $X = (x_1, x_2, \ldots, x_i, \ldots, x_n)$. Suppose that line i is stuck at a_i, a_i in {0,1}, such that $a_i \neq x_i$. Then $a_i = \overline{x_i} = y_i$ so that

$$Y = (x_1, x_2, \ldots, a_i, x_{i+1}, \ldots, x_n).$$

On the retry, $X1 = (\bar{x}_2, \bar{x}_3, \ldots, \bar{x}_n, \bar{x}_1)$. By assumption the only fault on the data path is on line i, so Y1 will be

$$Y1 = (x_2, x_3, \ldots, x_i, a_i, \ldots, x_n, x_1) . \tag{1}$$

Y2 is obtained by shifting Y1 to the right:

$$\begin{array}{l} \qquad\qquad\quad i \quad i+1 \qquad\quad (\text{ <- positional indicators }) \\ Y2 = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_i, a_i, \bar{x}_{i+2}, \ldots, \bar{x}_n). \end{array} \tag{2}$$

S is formed by $Y \bullet Y2$:

$$S = (x_1 \bullet \bar{x}_1, x_2 \bullet \bar{x}_2, \ldots, a_i \overset{i}{\bullet} \bar{x}_i, x_{i+1} \overset{i+1}{\bullet} a_i, x_{i+2} \overset{i+2}{\bullet} \bar{x}_{i+2}, \ldots, x_n \bullet \bar{x}_n). \tag{3}$$

So,

$$S = (1, 1, 1, \ldots, 0, \overset{i}{x_{i+1}} \overset{i+1}{\bullet} a_i, 1, \ldots., 1) \tag{4}$$

since $a_i = \bar{x}_i$ and $b \bullet b = 0$.

The incorrect bit position in Y is marked by the 0 at position i in S. Position i+1 will either be a 1 or a 0 depending on the value of x_{i+1}, and all other bit positions in S will be 1. Therefore, each bit position filled by a 1 in S indicates that the corresponding bit position in Y is correct. It follows that only bit positions indicated by a 0 in S could be incorrect and hence candidates for inversion to correct the error. Assuming position i+1 is a 0, the choice is between position i and position i+1 since only one bit is in error. The bit position to be inverted is that bit position in which S has

a 0 in the position in question and a 1 in the left adjacent position. Thus step 4 of Algorithm 2.1 will correctly identify the faulty bit and produce the proper result.

A fault which has an effect for only one cycle can be considered to be a transient stuck-at fault, and the analysis for such a fault proceeds as described above with few changes. We assume that the fault is detected, initiating a retransmission, but that the fault is not present during the second transfer. In this case, $\bar{y}_{i+1}$ in Equation 1 is not necessarily a_i, but the value it would naturally assume. Thus, Equation 2 becomes

$$Y2 = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_i, \bar{x}_{i+1}, \bar{x}_{i+2}, \ldots, \bar{x}_n).$$

With this change, Equations 3 and 4 assume the values

$$S = (x_1 \oplus \bar{x}_1, x_2 \oplus \bar{x}_2, \ldots, a_i \oplus \bar{x}_i, x_{i+1} \oplus \bar{x}_{i+1}, x_{i+2} \oplus \bar{x}_{i+2}, \ldots, x_n \oplus \bar{x}_n).$$

$$S = (1, 1, 1, \ldots, 0, 1, 1, \ldots, 1).$$

The 0 in S again indicates the incorrect bit in Y, and Algorithm 2.1 functions properly in the presence of a transient fault.

2) AND Bridge Faults. Let $X = (x_1, x_2, \ldots, x_n)$ as above, and suppose that lines i and i+1 have an AND bridge fault between them. Clearly if $x_i = x_{i+1}$, then the output on lines i and i+1 would be $x_i x_{i+1} = x_i = x_{i+1}$, and in this situation no error would be produced by the fault. However, if $x_i \neq x_{i+1}$, then the faulty output would cause a change in the parity. More specifically, when $x_i \neq x_{i+1}$, then $(x_i, x_{i+1}) = (0,1)$ or $(1,0)$, but

$$Y = (x_1, x_2, \ldots, \overset{i}{x_i x_{i+1}}, \overset{i+1}{x_i x_{i+1}}, x_{i+2}, \ldots, x_n)$$

$$= (x_1, x_2, \ldots, 0, 0, x_{i+2}, \ldots, x_n).$$

After detecting the parity error, X1 is sent and Y1 is received, where

$$Y1 = (\bar{x}_2, \bar{x}_3, \ldots, \bar{x}_{i-1}, \bar{x}_i, \overset{i}{\bar{x}_{i+1}\bar{x}_{i+2}}, \overset{i+1}{\bar{x}_{i+1}\bar{x}_{i+2}}, \bar{x}_{i+3}, \ldots, \bar{x}_n, \bar{x}_1).$$

Thus, Y2 would be

$$Y2 = (\bar{x}_1, \bar{x}_2, \ldots, \overset{i}{\bar{x}_i}, \overset{i+1}{\bar{x}_{i+1}\bar{x}_{i+2}}, \overset{i+2}{\bar{x}_{i+1}\bar{x}_{i+2}}, \bar{x}_{i+3}, \ldots, \bar{x}_n).$$

S is formed from Y $\bullet$ Y2, so

$$S = (x_1 \bullet \bar{x}_1, x_2 \bullet \bar{x}_2, \ldots, \overset{i}{0 \bullet \bar{x}_i}, \overset{i+1}{0 \bullet (\bar{x}_{i+1}\bar{x}_{i+2})}, \overset{i+2}{x_{i+2} \bullet (\bar{x}_{i+1}\bar{x}_{i+2})}, \ldots, x_n \bullet \bar{x}_n).$$

Since $a \bullet \bar{a} = 1$, $a \bullet 0 = a$, and $(\bar{b}\,\bar{c}) \bullet c = \bar{b} + c$, then

$$S = (1, 1, \ldots, \overset{i}{\bar{x}_i}, \overset{i+1}{\bar{x}_{i+1}\bar{x}_{i+2}}, \overset{i+2}{\bar{x}_{i+1} + x_{i+2}}, 1, \ldots, 1).$$

The only values in S which are unknown are those in bit positions i, i+1, and i+2. Since $x_i \neq x_{i+1}$, there are only four possible cases, as shown by Table 2.1. First we note that the positions of S indicated in Table 2.1 are the only ones which could contain 0's. This table thus indicates that the bit position in error will always be marked by a 0 in S with a 1 in the left adja-

Table 2.1. Cases for AND Bridge Fault Consideration in Algorithm 2.1.

Case	x_i	x_{i+1}	x_{i+2}	$s_i=\bar{x}_i$	$s_{i+1}=\bar{x}_{i+1}\bar{x}_{i+2}$	$s_{i+2}=\bar{x}_{i+1} + x_{i+2}$	Error in position
1	0	1	0	1	0	0	i+1
2	0	1	1	1	0	1	i+1
3	1	0	0	0	1	1	i
4	1	0	1	0	0	1	i

cent bit position. Therefore, step 4 of Algorithm 2.1 will always produce the correct result, and Algorithm 2.1 will recover the correct result in the presence of a single AND bridge fault between logically adjacent lines.

3) OR Bridge Faults. Again, let $X = (x_1, x_2, \ldots, x_n)$, and assume that lines i and i+1 are connected by an OR bridge fault. As with the AND bridge fault, if $x_i = x_{i+1}$, then no error would be produced by the fault. However, if $x_i \neq x_{i+1}$, then faulty output would result in incorrect parity. That is, when $x_i \neq x_{i+1}$, then $(x_i, x_{i+1}) = (0,1)$ or $(1,0)$, but

$$Y = (x_1, x_2, \ldots, \overset{i}{x_i + x_{i+1}}, \overset{i+1}{x_i + x_{i+1}}, x_{i+2}, \ldots, x_n)$$

$$= (x_1, x_2, \ldots, 1, 1, x_{i+2}, \ldots, x_n).$$

Following the detection of the parity error, X1 is sent and Y1 is received, where

$$Y1 = (\bar{x}_2, \bar{x}_3, \ldots, \bar{x}_{i-1}, \bar{x}_i, \overset{i}{\bar{x}_{i+1} + \bar{x}_{i+2}}, \overset{i+1}{\bar{x}_{i+1} + \bar{x}_{i+2}}, \bar{x}_{i+3}, \ldots, \bar{x}_n, \bar{x}_1).$$

Rotating to form Y2 would result in

$$Y2 = (\bar{x}_1, \bar{x}_2, \ldots, \overset{i}{\bar{x}_i}, \overset{i+1}{\bar{x}_{i+1} + \bar{x}_{i+2}}, \overset{i+2}{\bar{x}_{i+1} + \bar{x}_{i+2}}, \bar{x}_{i+3}, \ldots, \bar{x}_n).$$

S is formed from Y $\oplus$ Y2, so

$$S = (x_1 \oplus \bar{x}_1, x_2 \oplus \bar{x}_2, \ldots, \overset{i}{\bar{x}_i \oplus 1}, \overset{i+1}{(\bar{x}_{i+1} + \bar{x}_{i+2}) \oplus 1}, \overset{i+2}{(\bar{x}_{i+1} + \bar{x}_{i+2}) \oplus x_{i+2}}, \ldots, x_n \oplus \bar{x}_n).$$

Since $a \oplus \bar{a} = 1$, $a \oplus 1 = \bar{a}$, and $(\bar{b} + \bar{c}) \oplus c = b + \bar{c}$,

$$S = (1, 1, \ldots, \overset{i}{x_i}, \overset{i+1}{x_{i+1}x_{i+2}}, \overset{i+2}{x_{i+1} + \bar{x}_{i+2}}, 1, \ldots, 1).$$

Again we examine positions i, i+1, and i+2. The four possible cases are shown in Table 2.2. The positions of S indicated in Table 2.2 are the only ones which could contain 0's for the OR case. The table indicates that the bit position in error will always be marked by a 0 in S with a 1 in the left adjacent bit position. This being the case, step 4 of Algorithm 2.1 will produce the correct result for OR bridge faults between logically adjacent lines.

Table 2.2. Cases for OR Bridge Fault Consideration in Algorithm 2.1.

Case	x_i	x_{i+1}	x_{i+2}	$s_i = x_i$	$s_{i+1} = x_{i+1}x_{i+2}$	$s_{i+2} = x_{i+1} + \bar{x}_{i+2}$	Error in position
1	0	1	0	0	0	1	i
2	0	1	1	0	1	1	i
3	1	0	0	1	0	1	i+1
4	1	0	1	1	0	0	i+1

Consideration of the different cases has shown that Algorithm 2.1 will produce correct results regardless of the type of fault. The correct result will be produced for stuck-at faults, as well as AND or OR bridge faults between logically adjacent lines. This will be true whether the fault is permanent or transient. □

The maximum penalty for the use of this algorithm is one extra cycle per incorrect data transfer. Thus, if the fault remains for only a few cycles, then a momentary degradation in the transfer speed would be the only noticeable change, and the system would continue to function.

2.3. Correction of Errors Due to Bridging of Any Two Lines

The algorithm presented in the preceding section will not function properly for logically non-adjacent bridging faults. This can be illustrated by the following example. Assume that a data path consists of seven bits, and let there be an AND bridge fault between position i=2 and position i=5. Further, let X be defined as:

$X = (1,0,0,0,1,1,0)$.

Then the above algorithm will produce the following series of results:

$Y = (1,0,0,0,0,1,0)$: received data

$\overline{X} = (0,1,1,1,0,0,1)$: bit-wise complement of X

$X1 = (1,1,1,0,0,1,0)$: X rotated left by one bit

$Y1 = (1,0,1,0,0,1,0)$: received data

$Y2 = (0,1,0,1,0,0,1)$: Y1 rotated right by one bit

$S = (1,1,0,1,0,1,1)$: $Y \bullet Y2$

The incorrect bit in Y is located at position i=5, which is identified by Algorithm 2.1 since $s_4 s_5 = 10$. But it also identifies, incorrectly, bit 3 in Y as an erroneous bit since $s_2 s_3 = 10$. However, it is possible to transfer information on this bus system as shown by the following algorithm.

Algorithm 2.2

Step 1: X -> Y. If there is no parity error, then Y is the correct data. Stop.

Step 2: X1 -> Y1, where X1 = $\overline{X}$. If there is no parity error, then $\overline{Y1}$ is the correct data. Stop.

Step 3: X2 -> Y2, where X2 is formed by rotating X one bit to the left.

Step 4: Let Y3 be obtained by rotating Y2 one bit to the right.
Define S = Y $\oplus$ Y1.

Step 5: For each bit position in word Y, call it k, correct the value for y_k if and only if $s_k = 0$. If $s_k = 0$, then make the correction according to the following procedure: if $s_{k-1} = 1$, then $y_k = y3_k$, otherwise $y_k = \overline{y3}_{k-1}$. (Subscript arithmetic is done modulo word size to make it circular.)

Theorem: Algorithm 2.2 will ensure that the correct result is obtained in two retrys at most in the presence of one stuck-at fault, one transient fault, or one AND or OR bridge fault between any two data lines in the data transfer path.

Proof: As with Algorithm 2.1 we will consider the different types separately, the stuck-at, two types of bridging, and transient faults.

1) Stuck-at Fault. Let $X = (x_1, x_2, \ldots, x_i, \ldots, x_n)$. Suppose that line i is stuck at a_i, a_i in {0,1}, such that $a_i \neq x_i$. Then $a_i = \overline{x}_i = y_i$ so that

$$Y = (x_1, x_2, \ldots, a_i, x_{i+1}, \ldots, x_n).$$

On the retry, $X1 = (\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n)$. By assumption the only fault on the data path is on line i, and since $\overline{x}_i = a_i$ this value is transferred correctly, and

$$Y1 = (\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_i, \ldots, \overline{x}_n).$$

With the recognition of correct parity, Algorithm 2.2 halts at step 2 and presents the correct result:

$$\overline{Y1} = (x_1, x_2, \ldots, x_i, \ldots, x_n).$$

Hence, the correct result is obtained in the presence of a stuck-at fault.

2) AND Bridge Faults. Let $X = (x_1, x_2, \ldots, x_n)$ as above, and suppose that lines i and j have an AND bridge fault between them. Without loss of generality we assume that $i < j$. Clearly if $x_i = x_j$, then the output on lines i and j would be $x_i x_j = x_i = x_j$, and no error would be produced by the fault. However, if $x_i \neq x_j$, then the faulty output would cause a change in the parity. More specifically, when $x_i \neq x_j$, then $(\ldots, x_i, \ldots, x_j, \ldots) = (\ldots, 0, \ldots, 1, \ldots)$ or $(\ldots, 1, \ldots, 0, \ldots)$, and

$$Y = (x_1, x_2, \ldots, \underset{i}{x_i x_j}, \ldots, \underset{j}{x_i x_j}, x_{j+1}, \ldots, x_n)$$

$$= (x_1, x_2, \ldots, 0, \ldots, 0, x_{j+1}, \ldots, x_n).$$

After detecting the parity error, X1 = $\bar{X}$ is sent and Y1 is received, where

$$Y1 = (\bar{x}_1, \bar{x}_2, \ldots, \overset{i}{\bar{x}_i}\overset{j}{\bar{x}_j}, \ldots, \bar{x}_i\bar{x}_j, \bar{x}_{j+1}, \ldots, \bar{x}_n)$$
$$= (\bar{x}_1, \bar{x}_2, \ldots, 0, \ldots, 0, \bar{x}_{j+1}, \ldots, \bar{x}_n).$$

This would result in another parity error, and Algorithm 2.2 then calls for the transmission of X2, which is formed by rotating X so that

$$X2 = (x_2, x_3, \ldots, \overset{i}{x_i}, x_{i+1}, \ldots, \overset{j}{x_j}, x_{j+1}, \ldots, x_n, x_1)$$

This value is then transmitted over the lines and Y2 will be

$$Y2 = (x_2, x_3, \ldots, \overset{i}{x_{i+1}}\overset{j}{x_{j+1}}, \ldots, x_{i+1}x_{j+1}, \ldots, x_n, x_1).$$

This value is rotated to the right to form Y3,

$$Y3 = (x_1, x_2, \ldots, \overset{i}{x_i}, \overset{i+1}{x_{i+1}}\overset{j}{x_{j+1}}, \ldots, \overset{j}{x_j}, \overset{j+1}{x_{i+1}x_{j+1}}, \ldots, x_n).$$

S is formed by the Exclusive-OR of Y and Y1,

$$S = Y \oplus Y1$$
$$= (x_1 \oplus \bar{x}_1, x_2 \oplus \bar{x}_2, \ldots, \overset{i}{0 \oplus 0}, \ldots, \overset{j}{0 \oplus 0}, \ldots, x_n \oplus \bar{x}_n)$$
$$= (1, 1, \ldots, 0, \ldots, 0, \ldots, 1).$$

The 0's in S appear in those locations identified by i and j, one of which is incorrect. Thus, all bit locations identified by a 1 in S will need no

correction. Consider the bit locations identified by a 0 in S. If $j > i+1$, then the 0's will not be adjacent, and the correct values of x_i and x_j will be in positions i and j, respectively, of Y3. Therefore, the correct result will be obtained by making those substitutions in Y. If the 0's in S are adjacent, then $y3_i$ will correctly replace y_i, but $y3_{i+1}$ will be a 0, and this is not necessarily the correct value of x_{i+1}. However, since $x_j = \bar{x}_i$, then the correct value of $y3_{i+1}$ will be obtained by substituting $\bar{y}3_i$ for $y3_{i+1}$. With these substitutions,

$$Y = (y_1, y_2, \ldots, y3_i, \ldots, y3_j, \ldots, y_n) \quad (j > i+1)$$
$$= (x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_n)$$

or,

$$Y = (y_1, y_2, \ldots, y3_i, \bar{y}3_i, \ldots, y_n) \qquad (j = i+1)$$
$$= (x_1, x_2, \ldots, x_i, x_j, \ldots, x_n).$$

Therefore, under the assumption of an AND bridging fault, Algorithm 2.2 transfers the correct result, regardless of the location of the bridge in the data lines.

3) OR Bridge Faults. The OR bridge fault case is identical to the AND bridge fault case except that the AND's between x_i and x_j are replaced by OR's in the equations for Y and Y1. Thus, instead of 0's there are 1's in positions i and j in these equations. However, since $1 \bullet 1 = 0 \bullet 0 = 0$, the value of S is the same for the OR case as it is for the AND case, and Algorithm 2.2 produces the correct result for both.

4) Transient faults. Since Algorithm 2.2 is a multiple step process, a transient error could occur during different portions of the process. We

assume that only one error will occur during the transmission process. If the error is not present during step 1 of the algorithm then the correct result occurs, so any transient error will first be detected by step 1. If the fault is no longer active during the second transfer, then no parity error occurs and the correct result is obtained by step 2 regardless of the type of fault, whether stuck-at or bridging. If the fault is active during the first two transfers and inactive during the third transfer, then the fault was a bridging fault and step 3 would transfer correct information over line(s) not affected by the fault. Thus the steps 4 and 5 will function correctly regardless of the presence or absence of a fault during the third transfer. □

We have demonstrated that the algorithm correctly transfers information in the presence of stuck-at or bridging, permanent or transient faults.

2.4. Logic for Implementation

A small amount of logic is required for the implementation of Algorithm 2.1. A register is needed for the storage of the initial transferred value Y. Another register could be used to store Y1, although if sufficient time is allowed for stability of the retransmitted information, then the register for Y1 is not necessary. S is formed by an Exclusive-OR function between Y and Y1 for each bit position. With Y, Y1, and S available as inputs, the logic for implementation of Algorithm 2.1 is shown in Figure 2.2. Each bit position then requires two registers (assuming a register for Y1), two Exclusive-OR gates, an inverter and an AND gate. Since only local information is needed for this function, no additional time is required for propagation from one grouping to another.

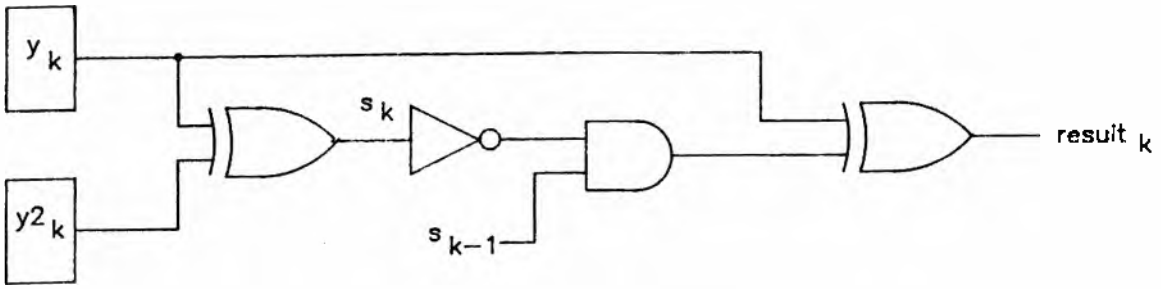


Figure 2.2. Logic for Implementation of Algorithm 2.1.

The transmitting interface which is used for Algorithm 2.1 merely requires a multiplexer to select between the initial data or its shifted inverted representation. There are several ways in which this could be implemented, and the necessary logic is not shown. However, only one control line is needed to select between the normal data and that required for retry. Hence, one additional line from the receiving interface could be used, which when true, calls for a retransmission of data and at the same time selects the correct value for the transfer.

The logic for Algorithm 2.2 is somewhat more complicated since the algorithm itself is more involved. Again a register is required for storage of Y; an additional register is required for Y1. The same register/signal stability problem exists for Y2 in this algorithm as for Y1 in Algorithm 2.1, and each bit position requires an Exclusive-OR gate for the S function. The gating required for step 5 of the algorithm is derived directly from the wording of

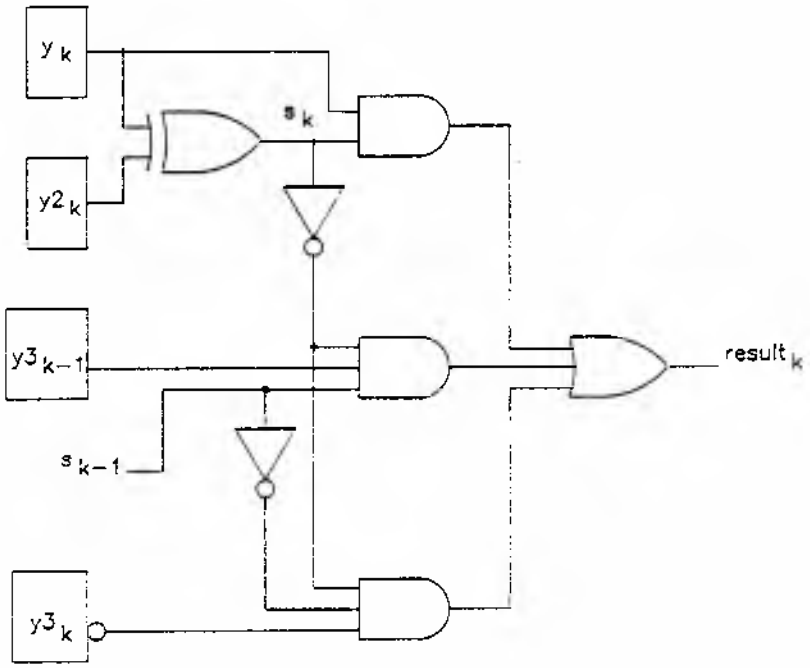
the algorithm, and it is shown in Figure 2.3a. Here each bit position requires three registers (assuming a register for storage of Y3), an Exclusive-OR gate, three AND gates, an OR gate, and two inverters. Again only local information is used so there is no propagation time for information to travel from one side of the bus to the other.

The interface used for transmission of information in Algorithm 2.2 is also quite simple, and could be constructed with an Exclusive-OR and a multiplexer per bit as shown in Figure 2.3b. Two lines are required to select the proper information for transfer, but the control function is still quite simple. In Figure 2.3b, c_1c_0 = 01 corresponds to the first step of Algorithm 2.2, c_1c_0 = 11 corresponds to the second step of retransmission with complemented data, and c_1c_0 = 0x to the third step of retransmission with rotated data. This transfer could be controlled by using additional lines to specify retry values to transfer, or the control could come from sequential logic at both ends.

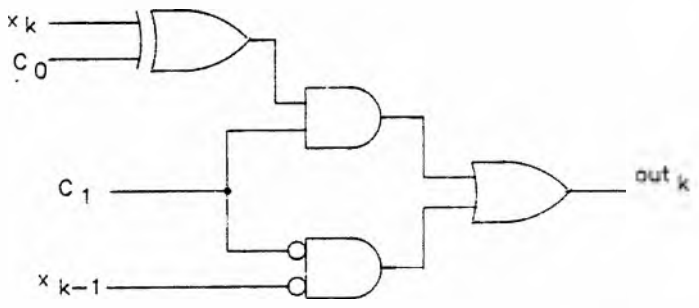
2.5. Concluding Remarks

We have presented two algorithms which will reliably transfer information from one module to another in the presence of a single permanent or transient fault on the bus. Algorithm 2.1 can be used to cover bridging faults when the logical and physical adjacency are the same for the bus, and Algorithm 2.2 can be used for the case when the physical adjacency may be different from the logical adjacency.

The logic for the implementation of the algorithms is relatively simple and could easily be incorporated into the circuitry used to interface to the



(a) Logic for Receiver



(b) Logic for Transmitter

Figure 2.3. Logic for Implementation of Algorithm 2.2.

bus. Algorithm 2.1 needs an additional line which indicates to the source module that an error has been detected and that the retransmission is needed. Algorithm 2.2 can be implemented either with one or two additional lines, with a smaller control complexity needed for the two line implementation. Each algorithm requires that additional logic be added to control the interaction and store intermediate values. Therefore, the cost of implementation is an additional bus line and several gates per bit position. The increased control complexity for Algorithm 2.1 is given for a simple read protocol in Chapter 4. One additional benefit of this logic is the monitoring of errors which occur on the bus and are detected and corrected. This information could be used to report errors to higher levels in the system. By monitoring the transfer medium, the system itself can request repairs when needed.

For permanent faults, the worst-case impact of the proposed methods is to reduce the throughput to 50% for Algorithm 1 and to 33% for Algorithm 2. Faults which occur will sometimes be in agreement with the data transferred, and hence not cause an error. Therefore, under the assumption of random data transfers, the transfer rates should be better than those of the worst-case conditions.

CHAPTER 3

MODELING THE INTERACTION OF THE ASYNCHRONOUS CONTROL SIGNALS OF BUS LEVEL PROTOCOLS

3.1. Introduction

Protocol models are methods of representing events which occur at asynchronous times triggered by independent units. In order to model the random nature of this type of interaction, different models of system behavior have been developed. Each of these modeling methods presents system features in a different manner, and each method was developed to study a different problem or aspect of a problem. Among these methods are Petri-nets, UCLA graphs, flow charts, programming languages, and state machines. The method which will best fit with the goals of this thesis is one which will accurately represent the signals and processes involved in bus level communications.

Petri-nets have been used extensively to study the concurrency or parallelism of the components of a system [18,8]. This technique is applicable to many types of systems, and it is not limited to systems showing parallelism. The Petri-net method represents conditions in the system with special nodes called places, events which occur in the system with nodes called transitions, and the correlation between conditions and events are represented as arcs. Arcs from places to transitions identify the conditions which must be true before the event can occur, referred to as the firing of the transition. Arcs from transitions to places identify those conditions which become true when the transition fires, or the event occurs. A true condition is indicated by

placing a marker called a token in the place representing that condition. An event is enabled when all of the arcs leading to the transition representing the event have tokens in their respective places. When an event takes place a token is removed from each place having an arc leading to the transition, and a token is added to each place which is on an arc leading from the transition. The random nature of the asynchronous events is represented by this model since no timing information is imposed on an enabled transition; thus enabled transitions can fire in any order.

Petri-nets have been used for many different purposes, including design verification and automation [19], systems of software [20,21], systems of concurrent processes [22,23,24], performance analysis of computers [25], and even social systems [26]. Since Petri-nets provide a convenient method for modeling asynchronous concurrent processes they have found application in modeling protocols of various systems [10,7,27]. Of particular interest to this study is the work of Merlin where the time-Petri-net is introduced [28]. In this model timing constraints are added to the firing of a transition, specifying a minimum and a maximum time for the action to occur. This model allows the study of lost messages in a protocol system, where the message is modeled as a token and the entire communication system is represented by a time-Petri-net. This work identifies the requirements that a system must have to be able to recover from the loss of a message in the system [29].

The use of the time-Petri-net has been used by Merlin to develop a methodology for the design of a protocol [14]. In this methodology a designer represents the protocol system under development as a time-Petri-net and then

enters this information into a computer-based design tool. The design tool will then search the states which the system can attain to search for deadlocks and illegal looping conditions. Faulty conditions are made known to the operator who can then take steps to rectify the errors.

Another modeling technique which has been used to represent protocols is the UCLA graph [30,31]. This model is equivalent to the Petri-net representation, yet it has many properties which make it attractive for representing host-to-host communication protocols. Postel [6] uses this method to explore host-to-host protocols, in general, and portions of the ARPANET protocol, in particular.

Protocols are also represented by various forms of finite state machines [32,33,10]. The state machine model can represent the system at different levels, from the most abstract to the most detailed. Different representations are useful for different purposes, each representation providing different insight into another aspect of the system being modeled. The use of the state machine models found in most of current literature represents the system at a higher level of abstraction than is useful for bus level communication, but the state machine model can achieve the degree of detail needed for this study.

The model used for investigating bus level protocols must be capable of representing the system at the lowest level, that level which deals with the signal levels on the lines of the bus and response to the information which is present on those lines. This model must represent the real-world and the hardware which is used to implement the protocol itself. The technique needs

to reflect the modularity with which real world systems are built, and be able to include modules which are added to the bus to satisfy system requirements. This means that the model must be capable of expanding to accept more modules without changing the modeling technique. It must also be able to represent the errors which can occur and their effect upon the system.

The Petri-net model does not satisfy the requirements for this bus level analysis. Although Petri-nets themselves are not limited to two-party protocols, the information which appears in the open literature concerning the analysis of protocols concerns the exchange of information between two units, as exemplified by the work of Merlin [14]. Petri-net analysis requires a model which encompasses the entire system, so to include additional functional units the Petri-net model would have to be reconstructed adding the arcs, transitions, and places necessary to communicate with the new module. This violates the modularity requirement of this analysis.

Also, the Petri-net model is not an adequate physical level representation since the basic method for representation of an error is to assume that a token, becomes lost. The action required to correct the error would be to replace the token. In a distributed system where a byte passing between units on a noisy communications line is represented by a token this model is fairly accurate, but this does not correctly represent the system when the token is defined as the assertion of a signal line. With this model, replacing the token could be accomplished by asserting the line again, but since it is already asserted that would be difficult to do. Another possible sequence is to release the signal and then assert it again, but this also causes problems

since the release of the signal may be interpreted by another module in the system as a different token, initiating an out of sequence action. Thus, the Petri-net is not an adequate model for representation of bus level protocols.

The UCLA graph model is also not an appropriate modeling technique for bus level protocols. Since the UCLA graph is equivalent to the Petri-net, it suffers from the same drawbacks as mentioned above. It does not expand in a modular fashion to accommodate additional functional units, and the error representation does not accurately model the errors which occur on a bus.

The state machine model for a protocol does adequately represent the modularity of a bus system, and it also provides an accurate model of the errors which occur. Adding functional units to the system can be accomplished by adding another state machine to an analysis which already consists of a number of state machines. The communication between these modules is accomplished over the common system bus, and errors are represented by control lines being incorrectly asserted (or released). Thus, the state machine model is superior in both the modularity and the error representation from the models previously discussed.

Another advantage of the state machine model over Petri-nets is that the bus control logic can be directly designed from the state machine description and vice versa; this is not true of Petri-nets. There is an in-depth treatment of state machines in the literature. In particular, algorithms exist for reducing any state machine by removing equivalent states without changing the behavior of the machine. Also, there are algorithms for determining if two state machines are equivalent. There are no known algorithms for reducing a Petri-net, or for proving that two Petri-nets are equivalent.

State machine models have been used for representing various levels of communication protocols [34,35]. In this thesis the level of representation is at the physical or hardware level. The state machines given in this and other chapters deal with the assertion of signals, and the interpretation of signals, which are lines of a bus. We assume that a state machine is level sensitive; that is, it monitors the level of a control signal and not the transition from 0 to 1 or 1 to 0. Thus the levels of control signals and the present state uniquely identify the next state. Each state in a state machine represents a different configuration of the module being modeled, and is not a representation of the state of the system as a whole. The state of the entire protocol system would consist of a collection of states of all of the modules as well as the signal levels on the common bus lines.

3.2. State Machine Representation of a Generic Read Cycle

As shown in Figure 1.1 a bus oriented system consists of several modules communicating with one another over a common set of bus lines. Although some protocols exist where one module may send data to several other modules simultaneously [36], most systems allow communication between only two modules at a time. The problem of arbitration for control of bus lines involves all of the modules, and this problem will be discussed later. Once control of the bus lines has been granted to one module, called the master, that module will control the data interchange. The master will assert the signals necessary to transfer data to or from another module, called the slave; the slave will assert the signals needed to respond to the master. This interaction between

a master and a slave is important because of its extensive use in bus systems, and we will analyze this exchange in detail. The example used for this analysis will be the read cycle, where the master requests a data transfer from a slave.

The lines which are needed to control the action between a master and a slave are the request line ('req') asserted by the master and the acknowledge line ('ack') asserted by the slave. Some other information is also needed to determine whether the cycle will be a read, write, or some other type of cycle, but this information can be encoded as a part of the address and does not need control line interaction. Once the master has control of the bus, it asserts an address, waits for an appropriate delay to allow for propagation delay and signal skew times, and asserts 'req'. All of the slaves on the bus receive and decode the address, but under fault-free conditions only one will respond. This slave performs a read function and asserts the data onto the data portion of the bus. With the data, the slave also asserts 'ack' to acknowledge to the master that the data have been obtained and are now on the bus for the master to accept. When the master recognizes the assertion of 'ack', it waits for a short period to allow for signal skew and accepts the data. Of course, the slave could provide the delay by first asserting the data and, then, after waiting an appropriate amount for skew time, assert the acknowledge. However, in most systems the master provides this delay and, therefore, we will assume that to be the case in our generic protocol. With the acceptance of the data, the master releases 'req', and this release indicates to the slave that the information has been transferred, so the slave releases the data and 'ack'. After the master has released 'req' it delays long enough

to prevent spurious responses and releases the address. With all of the signals released, the cycle is over and another cycle can begin.

The above written description of the interaction which takes place in a simple read cycle can be more clearly represented by a set of state diagrams. The state diagrams for this interaction are given in Figure 3.1. A complete description of the states in Figure 3.1 is given below, with the states for the master presented first, followed by the information for the slave. In the figure the assertion of a signal 'name' is denoted A(name), and the release of a signal is likewise indicated by R(name).

Master State Machine

State 0: Idle state. The master remains in this state until a read cycle is needed and the master has been granted control of the bus. This condition is denoted by assertion of the signal 'read'.

State 1: Assertion of address. The synchronous information needed by the slave is asserted by the master unit (indicated by 'adr' in the diagram). This includes not only the address information, but also any additional lines needed to indicate what type of interaction it is (read, write, read-modify-write, etc). The master moves to the next state a fixed amount of time after entering this state; in this example, the time is 150 nanoseconds (nsec). This time allows for the settling of address lines to a stable value and the propagation delays through th address decoders of slaves.

State 2: Assertion of request signal. The request signal 'req' is asserted to indicate to the slave that the operation should be initiated. The

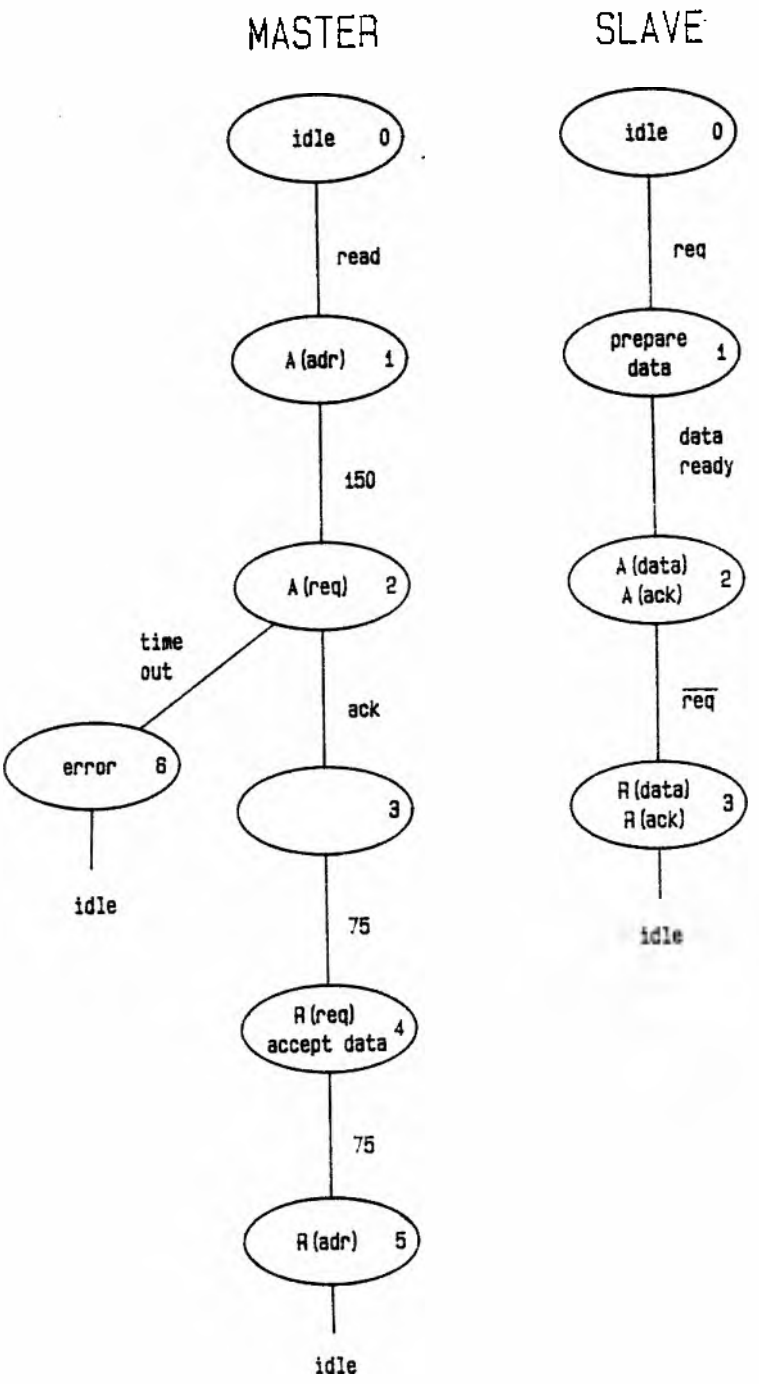


Figure 3.1. State Diagrams for Read Cycle.

master remains in this state until the acknowledge signal from the slave 'ack' has been detected, or until an error time-out has been reached, indicated by the signal 'time-out'. The time-out would occur if no slave responded to the address asserted by the master. Normal operation occurs when 'ack' is received from a slave and the master moves to state 3.

State 3: Delay state. This state is a delay state to allow for skew on the data lines of the bus. In this example, the skew period is assumed to be 75 nsec, so 75 nsec after entry in state 3 the master moves to state 4.

State 4: Accept the data. The master accepts the data asserted by the slave. In addition, 'req' is released to indicate to the slave that the operation has been completed. The master remains in this state for 75 nsec to allow the release of 'req' to propagate as needed before releasing 'adr'. It them moves on to state 5.

State 5: Release address. The synchronous information asserted in state 1 is released. The master then returns to the idle state to await another read request.

State 6: Error state. This state is entered if for some reason the 'ack' signal is not received before 'time-out' occurs. This state detects the condition that an address for a non-existent slave has been asserted. This state prevents the bus from reaching a condition where the only method for continued function is to reset the interfaces.

Slave State Machine

State 0: Idle state. This represents the state of the slave until the time when a response from it is required. A response will be required when

the address matches the assigned address space of the slave ('adrok') and 'req' has been detected. When this condition is true, then the slave enters state 1.

State 1: Prepare data. The slave waits for the functional unit to provide the data to be placed on the data bus. For a memory unit, this may involve a memory cycle; for an interface, this information may be immediately available in registers. When the data are ready for transmission on the data bus, the signal 'data ready' is asserted by the functional portion of the slave. This allows the interface to proceed to state 2.

State 2: Assert data and acknowledge. The slave asserts the data lines and the 'ack' signal. The slave remains in this state until the release of the 'req' signal has been detected, indicating that the master has completed the transfer and that the slave's signals should be released. At this time the slave enters state 3.

State 3: Release data and acknowledge. The slave enters this state when the transaction has been completed, and it releases both the data lines and the acknowledgment line. Then it returns to the idle state to await the next time when an interaction would be necessary.

Normal operation occurs when the master enters state 1 and asserts the synchronous address lines. After the appropriate delay, state 2 is entered and the request line is asserted. The addressed slave will leave the idle state when it detects 'req'. When the functional part of the slave unit provides the requested information, the slave asserts the data bus and the 'ack' signal (state 2 of slave). When the master detects the assertion of 'ack', it moves to state 3 and then to state 4. In state 4, it accepts the data which

have been placed on the bus by the slave and releases the request line. Regardless of what the slave does next, it waits another delay time, enters state 5, releases the address lines, and returns to the idle state. Meanwhile, when the slave detects the release of 'req', it moves to state 3, releases the data bus and the acknowledge line, and returns to the idle state. Protocols of this type are sometimes referred to as four-cycle protocols, after the sequence of four actions for each data transaction: A(req), A(ack), R(req), R(ack).

This simple read cycle provides the basis for studying techniques for detection of errors which occur on the control lines. In addition to detecting the errors, techniques are available to allow continued operation in the presence of the faults. These techniques will be discussed in detail in Chapter 5. First, we consider a more complicated read cycle utilizing additional control lines to implement Algorithm 2.1.

3.3. Read Cycle Using Algorithm 2.1 for Data Correction

Algorithm 2.1 was introduced in Chapter 2 as a means of using time redundancy to correct errors which occur while transferring synchronous information. For most bus systems the synchronous information consists of two transfers, one for the data and one for the address. The "address" may also contain information which indicates the type of transfer, but the timing requirements are the same as for the address itself, so these lines are grouped together. Thus Algorithm 2.1 needs to be applied to the address from the master to the slave as well as the data from the slave to the master.

This appears to be a simple task; the master asserts the address with correct parity, and if the slave detects a parity error, it requests a retry. Likewise, if the master detects a parity error when the slave asserts the data, then the master requests a retry. This simple picture is no longer correct when the location of the fault is unknown. The problem is to make sure that all of the interfaces involved in the transfer will interpret both the original address and the retransmitted address in the same way, and not have a second slave responding to the retransmitted address as if it were a correct address. The answer is to have all of the slave interfaces not only monitor the data lines but also the line requesting a retry so that any retry is recognized as such.

The state diagrams for master and slave units which incorporate Algorithm 2.1 are shown in Figure 3.2. Two more control lines are needed for this transfer than were needed for the read cycle of Figure 3.1. The condition that a parity error has been detected on the address lines is represented by 'errinadr', and a when a slave detects this condition it asserts the control signal 'adrpe'. More than one slave could assert the common 'adrpe' line, but this does not pose any problem. Likewise, the condition that a parity error has been detected on the data lines is represented by 'errindat', and master lets the slave know this by asserting 'datape'. These lines allow the different modules involved with the protocol respond in a predictable way.

Like the simple read cycle of Figure 3.1 this modified read cycle is also used as an example in this thesis. A complete description of the states involved in both master and slave state diagrams for this protocol system is

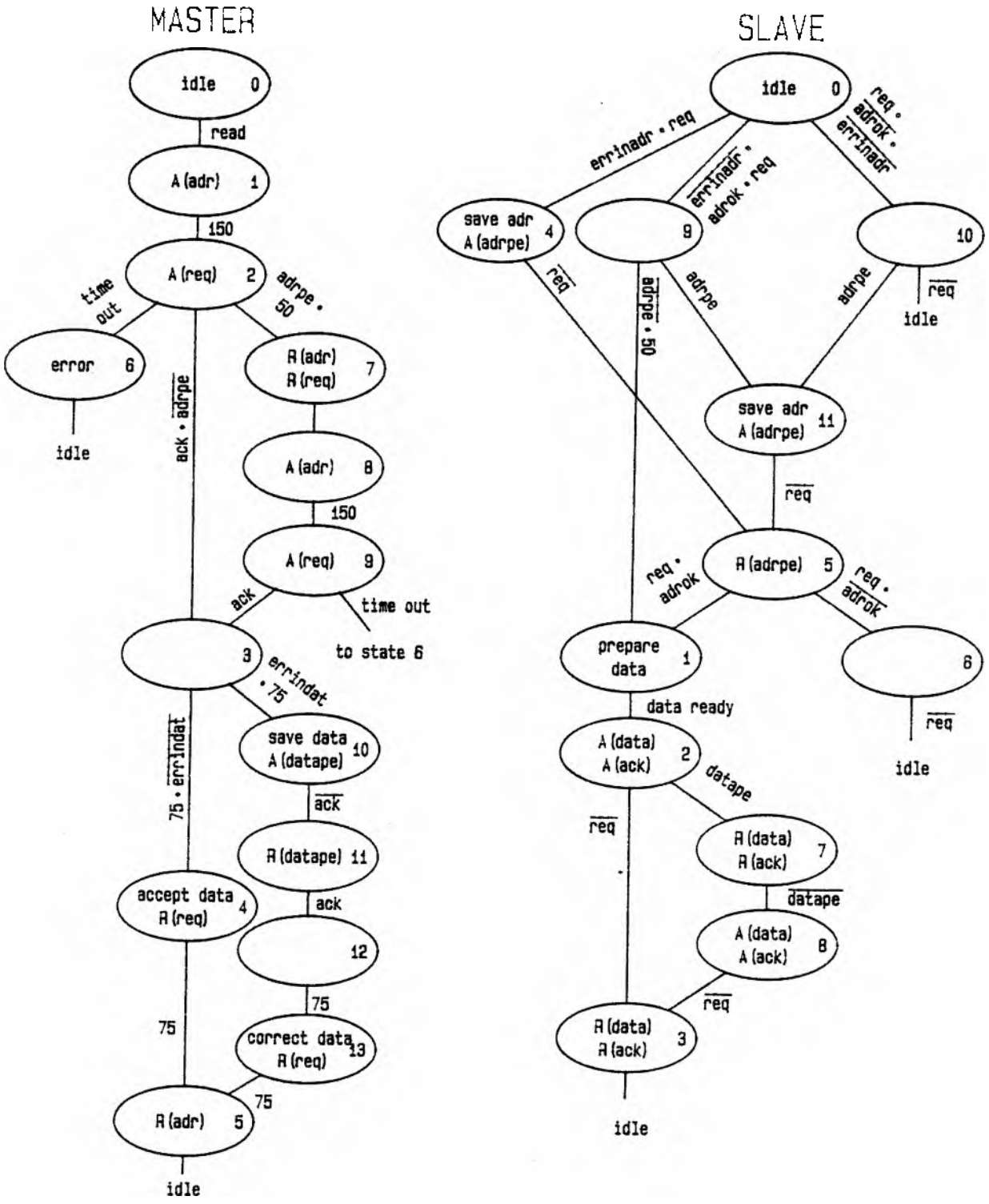


Figure 3.2. Read Cycle Modified to Utilize Algorithm 2.1.

included in Appendix A. We now briefly describe the modifications to the original state diagrams to accommodate Algorithm 2.1.

The effect of adding states to handle the error correction for the master state diagram can easily be seen by comparing Figure 3.1 with Figure 3.2. First we describe the state diagram of the master, on the left in Figure 3.2. States 7, 8, and 9 have been added to respond to a problem with a parity error on the address lines, and states 10, 11 and 12 inform the slave when a parity error has been detected on the data lines. State 7 is entered when a slave has recognized a parity error on the address lines and asserted 'adrpe'. The action of the master is specified by Algorithm 2.1: the old address is released as well as the 'req' signal, the new address is formed and asserted on the address lines (state 8), and after the appropriate delay the 'req' signal is again asserted to inform the slave that the retried address is ready to be accepted. When the slave responds with 'ack', then the master moves on to state 3 as before.

In like manner when the master detects a parity error on the data lines, it leaves state 3 by going to state 10. This captures the value to be corrected and causes the assertion of the 'datape' signal informing the slave that a retry is needed. When the slave releases 'ack', the master moves to state 11 to await the new data. In state 11 it also releases 'datape' which provides a signal to the slave to assert the corrected data. When the new 'ack' is detected, the master waits for a skew time (state 12), accepts the data, and releases 'req' (state 13). The corrected data are formed from the two values according to Algorithm 2.1. With the addition of these seven

states, the master has been modified to correct the synchronous data passed on the bus.

The modification to the slave state diagram is more involved than the modification performed to the master. States 4, 5, 6, 9, 10, and 11 are used to guarantee the correct transfer of the address, and states 7 and 8 are for the data transfer. Each slave will be idle until a 'req' signal is detected, and then the conditions which are present will determine how it proceeds in the state diagram. One of the two conditions detected is the fact that the address is free of parity errors and matches the slave's assigned address space, indicated by 'adrok'. The other condition is that a parity error has been detected by the slave, indicated by 'errinadr'. If 'adrok' is true and no parity error has been detected then the slave will move on to state 9 when 'req' is detected. If 'errinadr' is true, then upon detection of 'req' the slave will move to state 4. However, if neither of these conditions is true, then the slave will move on to state 10 with the detection of 'req'. The result of these three states is that when a 'req' is received each slave will enter either state 4, 9, or 10, and if a parity error is detected anywhere in the address path, either on the bus itself or in the slaves, then a retry address is called for. This is accomplished by state 4 when the error is on an address line common to all units, and by state 11 when the error was detected by another slave unit. The slave units then wait in state 5 and when the retry value becomes available the correct address either matches and the slave proceeds to state 1 or the address does not match and state 6 is entered.

The corrections needed for a fault on the data path are taken care of by entering state 7 when the slave detects the assertion of 'datape', where the slave releases the old data. When 'datape' is released, the slave asserts the value formed according to Algorithm 2.1 and releases the 'ack' signal (state 8), and waits for the master to signal the end of the cycle. This is indicated by the release of the 'req' signal, at which point the slave proceeds to state 3 to release the data value and return to the idle state.

The state machine method of protocol modeling has allowed us to represent the interaction at the lowest level and accurately model the system. We now present a method which enables the use of a general purpose computer to help analyze and test these protocols.

CHAPTER 4

COMPUTER-AIDED ANALYSIS OF PROTOCOL INTERACTION

4.1. Introduction and Previous Work

There are a variety of different approaches to protocol specification and implementation which have led to diversified applications of computers to aid in the evaluation of protocols. Computers have been used to aid in the exercising of protocol behavior according to its definition, and they have also been used to explore the set of assertions that can be proved or inferred from the specifications of the protocol. We have examined a number of procedures and devised a technique consistent with our desires to test the protocol to find latent errors.

One of the most obvious ways to use the computer is to explore the possible states which the system could assume. This type of reachability analysis has been performed by those who use a global graph model for the protocol, such as a Petri-net or a UCLA graph [37,13,6]. Searching the system space in this manner provides insight into he interaction of the component modules. If the known information includes those system states which are improper or for some reason incorrect, then the reachability analysis will indicate the existence of these errors if that combination of conditions is reached when the protocol is exercised. By the introduction of the token machine concept this type of analysis can identify deadlock conditions [37]. Also included in the errors detected are the presence of cycles in the specification and conditions which can lead to an infinite number of states for that representation [13].

Another type of computer interaction allows the exploration of protocols which are specified using limited state machines for each action [38]. The computer is used to find allowable sequences for the interaction of the state machines and create what is called a duologue matrix. This matrix identifies well-behaved and erroneous sequences. The duologue matrix method was initially developed to aid in two-party protocols, but further work relaxed the requirements and allowed multiple units to be modeled and the method to be further automated [15,39]. The extension of this method developed by West is very interesting in that it reduces the number of states examined by performing the system exploration with a perturbation technique. A complete search of successor states is made for each collection of states of the system. In this way the protocol system is tested for deadlocks and other errors, where these errors can be detected by incorrect combinations of states.

Still another approach to checking the validity of protocols comes from representing the protocols in a programming fashion instead of using graphical methods. Brand and Joyner [40] present a method which combines the perturbation technique of West with representation of the protocol in a programming language. Their technique explores the tree of execution paths possible from the initial invocation of the protocol in the language. A different approach is taken by Sunshine, et al. [12]. Here the protocol is expressed as a combination of state transition models and abstract data types, and the AFFIRM system is invoked as a theorem prover to verify different properties of the protocol.

These previous efforts have shown that the computer is a valuable tool to use in investigating the interaction of protocols. We define a proposed protocol in terms of state machines, and use the computer to exercise the protocol to find the errors which may occur. With this method we can also introduce faults on the signals involved so that the vulnerability of the protocol to errors can be investigated. In order to utilize a computer to aid in this process we must represent the protocol in such a way that a computer can understand and exercise it and at the same time have the representation accurately model the errors which can occur.

4.2. SML - A State Machine Representation Language

One of the advantages of the state machine method of representation of a protocol is that it is very easy to understand and follow. The state of a unit involved in a transaction is totally determined by the state that it is in, and all possible successor states are quickly identified. The qualities which allow humans to understand the graphical representation quickly are not automatically communicated to a computer. In order to utilize the power of a general purpose computer in the protocol analysis the state machine must be presented in a regular fashion. We have developed a language for representing state machines which can completely specify the state machine as represented in graphical form and can be used to aid in protocol analysis.

A finite state machine is given by a 5-tuple:

State Machine = < I S O NS-map O-map >

where

I is a set of inputs;

S is a set of states;

O is a set of outputs;

NS-map is a state transition mapping (I X S -> S)

of current state and inputs to next state;

O-map is an output mapping (S -> O)

of current state and inputs to the outputs.

The O-map corresponds to a Moore machine or state assigned output representation, since this is the representation which we use throughout the thesis. With some modifications to the language we can also allow Mealy machine or transition assigned output representation, which will require the O-map from I X S to O. This information will completely specify the action of a device whose behavior can be described with a state machine. However, the model we have used for a protocol system calls for a number of independent modules, the behavior of each to be specified by a state machine. In order to describe state machines which are part of a protocol system we have developed a language which uses two additional fields in representing a state machine:

State Machine = < N D I S O NS-map O-map >

where I, S, O, NS-map, and O-map are defined as above, and

N is a name identifying a particular state machine;

D is an optional set of declarations.

The name identifies the state machine within the system to permit signal identification. The set of declarations is optional and allows the specification of limits or other constants to be used in making decisions in the choice of next state. For example, constants introduced in the set of declarations can define the address range to which a slave will respond.

The language we developed to represent the state machines is called SML - State Machine Language. It is a context-free grammar which can be used to completely specify state machines, and SML descriptions of modules provide input to computer programs which will evaluate the correctness of the protocol. A description of the basic elements of the language is given in Figure 4.1, and a complete description of the language as well as the assumptions made about the signal names and other tokens involved is included in Appendix B. The 'smname' declaration allows the state machine to be uniquely identified. A 'define' statement sets up a constant which will have significance only within that state machine. More than one define statement may be included. The inputs, as well as the outputs, are divided into global and

```
State Machine = N  D  I  S  O  NS-map  O-map ;
            N = smname <identifier> ;
            D =  { define <identifier> = <value> ; }
            I = <inp> { <signal_name> } ;
        <inp> = ginputs | linputs
            S = states <integer> ;
            O = <out> { <signal_name> } ;
        <out> = goutputs | loutputs
       NS-map = <transition> { <transition> }
 <transition> = tran <state_number> -> <state_number> : <condition> ;
  <condition> = <logic_on_inputs> | <delay> |
                      <logic_on_inputs> <logical operator> <delay>
 <output_def> = <assert_def> | <release_def> | <do_def>
  <assert_def> = assert <signal_name> = <value> in <state_number> ;
<release_def> = release <signal_name> in <state_number> ;
     <do_def> = do <signal_name> = <value> in <state_number> ;
```

Figure 4.1. Description of SML Grammar for a State Machine. Variables are inclosed in pointed brackets. Curly brackets {...} call for 0 or more repetitions of the contained item.

local groups. The global signals are common to all modules attached to the bus and are accessible to each module. The local signals are used for communication with the other sections of the module. That is, a local signal is used to allow the functional portion of a module to respond to the interface portion (see Figure 1.1). States of the module are identified by integers, so the 'state' statement establishes the set of states by giving the highest state number. The next state information is given by identifying transitions between states. A transition is identified by the state where it begins, the state where it ends, and the conditions under which the transition occurs. Conditions include logic on the inputs, delays, or a combination of both. The output information is given by identifying states where the signals are asserted and released. The 'do' portion of this process allows specification of a signal which will be asserted while the state machine is in a state and released upon exit from that state.

Using this language each state machine in a protocol system can be completely specified, and computer analysis can accept this description and exercise the system. The following section gives an example of the use of the grammar.

4.3. Representation of State Machines with SML

Figure 4.2 contains modified versions of the master and slave state machines for the read cycle of Section 3.2. The condition for the slave to leave the idle state has been modified to add an address specification and to check that the transaction is a read cycle. Figure 4.3 gives the SML descriptions for these two state machines.

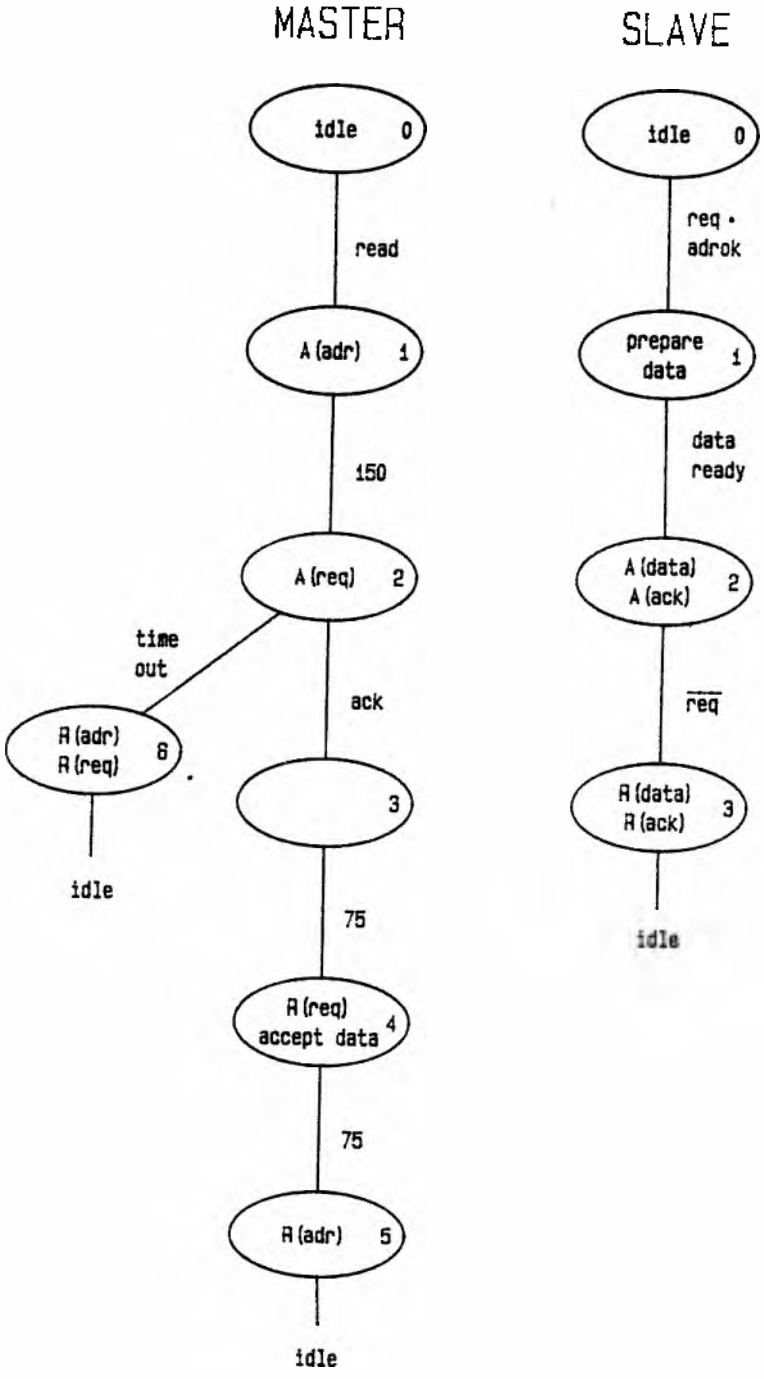


Figure 4.2. Master and Slave State Machines for a Read Cycle.

SML Description of Master:

```
smname master;              # master state machine
ginputs data, ack;          # global inputs (from the bus)
linputs read;               # local (internal) inputs
states 6;                   # highest number for states
goutputs adr, req;          # global outputs (to the bus)
loutputs readdone;          # local (internal) outputs
tran 0 -> 1 : read == 1;    # next state map
tran 1 -> 2 : delay(150,150);
tran 2 -> 3 : ack == 1 ;
tran 2 -> 6 : acc_delay(2000);
tran 3 -> 4 : delay(75,75);
tran 4 -> 5 : delay(75,75);
tran 5 -> 0 ;
tran 6 -> 0 ;
;
assert adr = mkadr(2000,4000) in 1 ;  # output map
assert req = 1 in 2 ;
release req in 4;
release adr in 5;
release req in 6;
release adr in 6;
;
```

SML Description of Slave:

```
smname slave;               # slave state machine
define amax = 3000;         # maximum address allowable
define amin = 2000;         # minimum address
ginputs req, adr;           # global inputs (from the bus)
linputs data_ready, ldata; # local (internal) inputs
states 3;                   # highest number of a state
goutputs data, ack;         # global outputs (to the bus)
loutputs prepare_data;      # local (internal) outputs
tran 0 -> 1 : (adr < amax) && (adr >= amin) && req ;
tran 1 -> 2 : data_ready == 1;  # next state map
tran 2 -> 3 : req != 1 ;
tran 3 -> 0 ;
;
do prepare_data = 1 in 1 ;  # output map
assert data = ldata in 2 ;
assert ack = 1 in 2 ;
release data in 3 ;
release ack in 3 ;
;
```

Figure 4.3. SML Descriptions of the Master and Slave State Machines.

As can be seen from the two figures, names are assigned to the state machines. These names do not have any significance within the state machine itself, but they do allow the protocol system as a whole identify them for its own purposes. This permits these machines to be known by a unique name throughout the system, and local signals have this name prefixed to them in the actual simulation.

The names which are used in the example fall into three categories: constants, global signals, and local signals. The constants are identified in the slave description by the 'define' statements. In this case the two constants ('amax' and 'amin') identify the assigned address space of this slave. Declarations of this type are optional, as can be seen by their absence from the SML representation of the master. The global signals are the address bus ('adr'), the request signal of the master ('req'), the data lines ('data'), and the acknowledge signal of the slave ('ack'). These are identified as inputs or outputs of the state machines as appropriate. If the state machines were expanded to include a write capability as well as a read capability, it would be appropriate for the data lines to be listed as both a global input and a global output. Local signals provide communication within functional units: 'read' initiates the read sequence when the functional unit requires a read cycle and bus ownership has been granted to the master; 'prepare_data' instructs the slave functional unit to perform a read cycle; 'data_ready' indicates that 'ldata' contains the requested information; and 'readdone' lets the master functional unit know that the data has been acquired.

The set of states for each state machine is identified by the 'state' statement: 6 for the master and 3 for the slave. States are numbered consecutively from 0, and by convention the 0 state in each diagram is the idle state.

Each transition is listed by giving the states involved and the condition under which the transition can be made. If the transition is always made the condition portion of the statement can be omitted, as seen by the transitions to idle in both diagrams. The asserted value for signals is 1, and logic on the signal lines is tested against this value, such as 'read == 1'. The logic equations required are expressed in the syntax of the C language. These logic equations check conditions of signal lines, data values on groups of synchronous lines, or delays.

There are two types of delays which are used in the SML descriptions: delays which must occur before moving to another state and delays which provide an alternative path after a specified time. A delay which forces action to remain in a state for a specified action until a certain period has elapsed is exemplified by the 150 nsec delay between state 1 and state 2 in the master. The statement which specifies this time uses the 'delay' function. However, when a delay is specified which identifies an alternative to some other action the 'acc_delay' function is used, as shown by the error exit from state 2 in the master. The 'acc_delay' function was named for an accumulated delay. This delay need not be completed before the state machine moves on to another state. The 'delay' function allows specification of a random value, giving only the minimum and maximum values, while the 'acc_delay' function identifies only one value.

The action imposed on the signals is specified by the 'assert', 'release', or 'do' statements. Signals can be asserted in one state and released in another, as shown by the 'req' signal which is asserted in state 2 and released in state 4 of the master. Note that the error exit must also provide a release of signals previously asserted. If a signal is to be asserted upon entering a state and released upon entering any succeeding state, then a 'do' specification can be used instead of separate 'assert' and 'release' statements. This is exemplified by the 'prepare_data' signal in the slave diagram. The assumed asserted value for signals is 1, but the 'assert' statement allows the signal value to be set at 0 also. The names which represent collections of lines which pass data synchronously, such as 'adr', can be asserted to any desired value. The 'mkadr' function which appears in the master SML description is one of several functions which return a random value. The arguments to the function establish the permissible range of the request, and the value returned will fall somewhere within that range. This example shows the master capable of generating addresses which cannot be responded to by the slave.

SML provides a convenient method of presenting with a regular grammar the information contained in a state machine. A complete SML description of a complex state machine is given in Appendix C. By representing the state machines with SML the computer can exercise the protocol system to check for error conditions. We now present an example of creating an exercise system using SML descriptions.

4.4. Using SML Descriptions with the Protocol Exercise System

The previous section presented the method of representing state machines with SML. In this section we combine SML descriptions of the several state machines which comprise a protocol system into a protocol exerciser which will check the protocol for error conditions. We assume that each state machine which is used in the protocol description is contained in its own file, and our intent is to combine the information given in these files with some other information and emerge with a program which will exercise the system. Figure 4.4 gives the set of commands necessary to make a protocol exercise system of the two SML descriptions presented in the previous section. The descriptions are contained in the files 'master' and 'slave'. It is convenient to give the files the same name as the state machine which they represent, but this is not necessary.

The protocol system is built around routines which have been developed using the C programming language, so the first step in this process is to convert the SML descriptions into routines in C. The 'mfsm' commands which are shown accept the SML descriptions from the files listed and create a number of

```
mfsm < master
mfsm < slave
makenames master slave
makedoup master slave
makeit master slave
cc -o r1 -g whole.c /mnt/dln/pub/sim.new.a
```

Figure 4.4. Example of Creating a Protocol Simulation System.

other files: a file containing a complete C language routine which represents the state machine, files containing global and local variables and other names involved, and a file with information about the delay times involved. The 'mfsm' command is invoked once for each SML description, and this can be done as many times as needed to include all of the modules of the protocol system.

Once all of the SML descriptions have been worked on by 'mfsm', it is necessary to gather all of the names together which will be incorporated in the total system. The command 'makenames' constructs temporary files containing these names, and it assumes that 'mfsm' has been applied to the files which are included as its calling parameters. These names include the local and global variables which make up the signal structure involved.

The protocol system is capable of revealing information about state machines and signals involved in the interaction, but it needs to know the information which is of interest. Therefore, a routine is called at every tick of the system clock to see if an output of information is needed, and if so to print out the information. The 'makedoup' command, named for 'make do_an_update routine', creates a routine which will be combined with the other C routines created by 'msfm' to accomplish this. The command creates a routine to print out variables identified in its calling parameters. Names which are files containing SML descriptions will result in printing out the state of that state machine; names which do not match files of SML descriptions are assumed to be signal names, and these signals are printed out. Run time options indicate if this information is to be given at every tick of the system clock, at each change of the printed variables, or not at all. The rou-

tine which results from execution of 'makedoup' is stored in a file to be combined with the others for the final system.

The 'makeit' command assembles all of the information gathered by the previous commands into a complete C program which will represent the protocol system. The local and global variable names are included, cross-reference tables are set up, process names are added where necessary, the C language routines are included, and other utilities are added to complete the program. In addition to the normal system libraries a simulation system created by Norton [41] is used extensively. If the SML descriptions are complete, then this system will exercise the protocol to check for errors. The errors which are detected include deadlock, where no action is possible, assertion of variables which have already been asserted (two units driving the bus at the same time or incorrect specification of a state machine), and release of a signal which is not asserted (signal had previously been released or never asserted).

When all of the commands have been completed and the C language file is prepared, then the C compiler is invoked to create the protocol system itself. A complete description of these commands and the protocol system they result in is given in Appendix D. The run time options of the protocol system allow specification of the output mode, the number of cycles of what signal to check, specifying signals as stuck-at signals, probability of stuck-at signals, and signals to be printed out in addition to those included in the 'makedoup' command.

As shown above, the preparation of a computer program to exercise the protocol system requires several steps. SML is used to describe the state

machines which make up the system, and then C language routines representing the action of these modules are prepared. Additional routines are created, and then all of these routines are combined into a simulation system. The protocol can then be exercised to check for the existence of errors. Errors result in a print-out of information identifying the states of the modules of the system when the error occurred. This tool allows us to monitor the action of protocols and verify correct behavior.

CHAPTER 5

ALGORITHMS FOR DEALING WITH ERRORS IN CONTROL SIGNALS

5.1. Introduction

In Chapter 2 we presented algorithms for dealing with errors which had been detected by testing parity across synchronous data lines. The algorithms required to control the transfers were mentioned with little comment in Chapter 3, but a question which needs to be dealt with is how to detect errors which occur on the control lines. The technology used for both control lines and data lines is the same in most bus systems, and so the same types of errors can occur on both. We now examine the error-detecting capabilities of the protocols themselves and give algorithms for detecting errors and continuing operation when errors have been detected.

For all of these algorithms we represent the action of different modules with state machines, as presented earlier. State machines have been used for various levels of protocol representation, from showing a few states with little detail to a very detailed representation of the interaction involved. Once the state machine representation has been obtained, the correctness of the protocol can be checked in one of several ways. The method of the previous chapter is to create an SML description of the state machines involved and exercise the system until satisfied that the interaction is correct. The perturbation technique presented by West [15] operates on directed graphs and is similar to the exercise technique presented above. The principal difference is that West's method calls for global system knowledge of correct and

incorrect state combinations, while the exercise system described here uses only the bus signals to detect erroneous behavior.

In this chapter we present guidelines and algorithms that deal with detecting faults which occur on the control lines and, if possible, continuing operation in the presence of the faults. Unless otherwise stated, we assume that there is at most one fault at any one time. The fault model is the stuck-at fault, where the line is stuck-at-1 or stuck-at-0. We assume that the stuck-at-X fault occurs at some point in time when the line was legally at the faulty value, then the fault occurred and the line remained at that value. That is, the line became "stuck" at a value in the normal course of operation, and it was not the result of an out-of-sequence transition to the faulty value.

The signals, which are used to allow the master and slave modules to move from one state to another, include signals which are local to the modules and signals which are common between them. It is assumed that the signals which are local are not susceptible to faults and will remain fault free during these operations. This means that the delays, 'read', and 'data_ready' will not be candidates for faults in the analysis which follows. The bus lines, however, are susceptible to faults, and therefore faults on the 'req' and 'ack' lines will be considered. If further protection of local signals is desired then techniques such as TMR can be used to guarantee correctness. This increases the logic required to implement the module but does not impact on the bus protocol.

We also assume that the algorithms, which are presented in this chapter, described by state machines, will operate correctly at all times. That is, we are limiting the faults to the bus itself and prescribing action which will identify the faults, and, if sufficient redundancy is provided, continue to operate. The task of dealing with faults within a state machine has been treated elsewhere [42,43,44], and will not be discussed here. We simply mention that techniques exist for dealing with faults within the state machines themselves.

Although we will not present the SML descriptions of the interactions involved, the state machines presented in this chapter have been exercised with the system described in Chapter 4. We will first treat the detection of errors in control signals with time redundancy and methods of preventing improper operation. Then we examine the use of dual-rail signals and methods to guarantee correct operation in the presence of a single fault.

5.2. Control Signal Error Detection with Time Redundancy

Normal operation for the four cycle read protocol was described in Section 3.2. The master and slave state machines for this protocol are reproduced in Figure 5.1. We now examine the effect of stuck-at errors on the control lines. Under normal operation (control lines are not faulty) the error escape from state 2 is provided for the case of the master requesting a non-existent slave. The error exit from state 2 results in state 6 of the master. This state will also detect two faults on the control lines which have the same symptom as a non-existent slave. These faults are 'req' stuck-at-0 and 'ack' stuck-at-0. If the request line is stuck-at-0 then as far as the slave

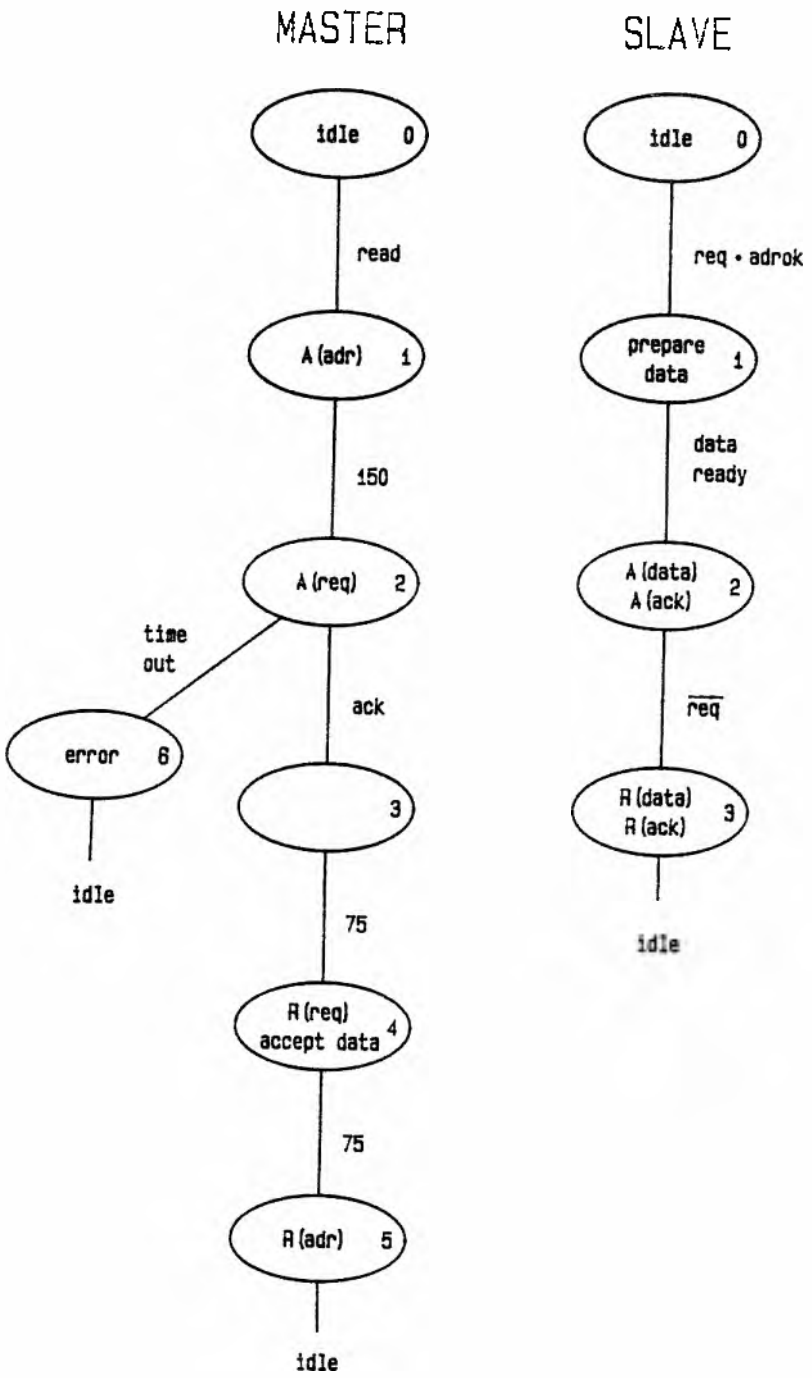


Figure 5.1. Master and Slave State Machines for a Read Cycle.

is concerned, no request is ever issued. Therefore, no read cycle is performed and the slave doesn't respond. The action of the master is then to just go through the error states as though the slave didn't exist. A similar action occurs if the acknowledge is stuck-at-0. The master issues the address and then the request. The slave detects the address and the request, obtains the data, and then asserts the data lines and the acknowledge signal. Since the acknowledge signal is stuck-at-0 the master does not detect it and state 6 is again the outlet. Assuming that the error state causes the release of asserted signals, the 'req' signal is released and the slave continues its operation, returns to the idle state, releasing the data and 'ack' signal. Thus, state 6 in the master will detect two of the errors of the control interaction.

If request stuck-at-1 occurs, then the following sequence is initiated: The master interface asserts 'adr' (state 1) and waits the appropriate amount of time before asserting 'req' (state 2). However, since the request line is stuck-at-1, as soon as the slave detects that the address has been asserted, it initiates the read cycle of its functional portion (state 1). The data is made available and asserted along with the acknowledge line (state 2). This may all be accomplished before the master actually asserts the 'req' line. When the master goes to state 2 and detects assertion of the 'ack' line, it proceeds on to state 3 and completes the cycle, releasing 'req' in state 4. However, since 'req' is stuck-at-1, the slave remains in state 2 and has no way to return to the idle state. Thus, the system cannot recover from the error, and the bus remains inoperative (since 'req' appears to be asserted and 'ack' is asserted).

The acknowledge stuck-at-1 results in a different type of faulty operation. The master goes through the normal cycle to the point of waiting for the 'ack' signal. At this point it assumes that the data is available because it detects a valid 'ack' signal even though the slave has not asserted it. So the master continues according to its state diagram and accepts whatever data is on the bus when it reaches state 4. However, unless the slave is very fast, nothing will have been asserted on the data lines by the slave unit, and the data will be incorrect. Thus, both the master and the slave will go through their cycles, neither detecting that anything is wrong, but invalid data will have been passed.

As can be seen from the above discussion, the error state in the four cycle read operation will detect only two of the four possible stuck-at errors on the control lines. However, the protocol can be modified by the addition of more states to detect the existence of the other error conditions. These modifications are shown in the master and slave state diagrams in Figure 5.2. The master state machine has been modified by the addition of state 7, which has been added to allow the detection of the condition acknowledge stuck-at-1. It is entered from state 1 when a time-out period has passed since the address has been asserted and the 'ack' signal has not been released. In this case 'ack' is in error and assumed to be stuck-at-1; the master unit detects this condition and returns to the idle state through state 7. The modification to the slave state diagram is the addition of state 4. This state is added for the detection of the condition 'req' stuck-at-1. It is entered from state 2 when a time-out period has passed and the 'req' signal has not been released. When this condition exists the signal is assumed to be in error and the slave returns to the idle state through state 4 to indicate the error.

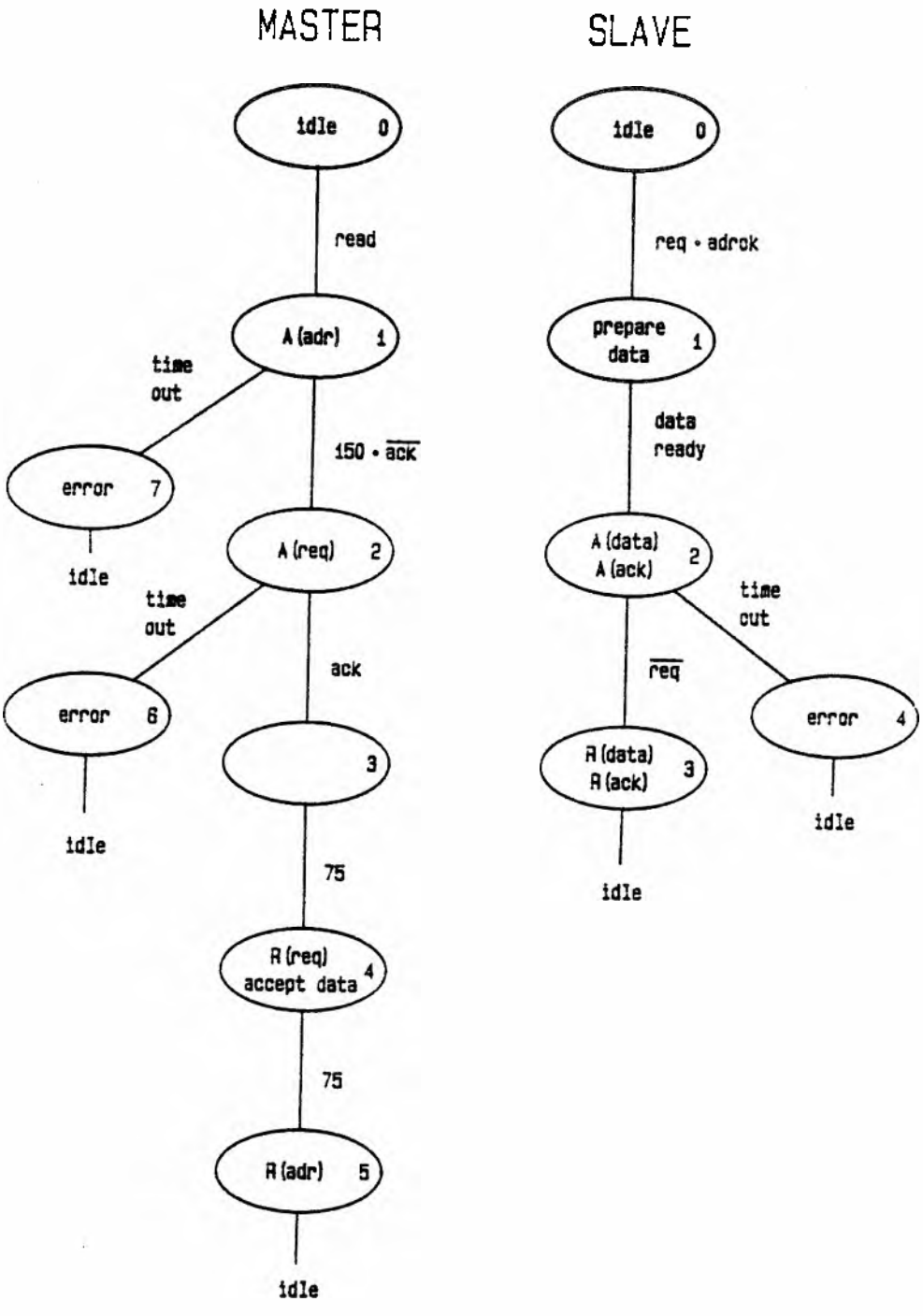


Figure 5.2. Master and Slave State Machines for Error Detection.

The addition of these two states, one to each unit, allows the detection of the two errors missed by the original arrangement. With this modification all four possible errors will be detected. However, the slave unit will still respond when the request line is stuck-at-1. For a read cycle this may result in obtaining the wrong data item, but no data modification occurs so no permanent damage is done. This is not the case with for a write cycle, as shown by the write protocol in Figure 5.3.

The simple write protocol of Figure 5.3 is patterned after the read cycle which has been used for an example. The master differs from a master in a read cycle since the 'data' lines are asserted by the master, and this is done at the same time as the 'adr' lines, in state 1. The master then waits an appropriate amount of time and asserts the 'req' line (in state 2). When the slave detects the 'req' signal and an address within its assigned address space it responds by moving to state 1. This state differs from its counterpart in the read cycle since it is waiting for the slave to accept the data before moving to the next state, instead of waiting for data to become available. When the functional portion of the slave has performed whatever functions are necessary to accept the data, the signal 'data_accepted' is asserted and the slave moves to state 2 to assert the 'ack' signal. When the master detects the assertion of 'ack', it moves to state 3 and releases 'req'. No skew time is needed since the data transfer has already been completed. The delay from state 3 to state 4 prevents spurious actions on a noisy bus, and the master releases the 'adr' and 'data' lines in state 4 before returning to

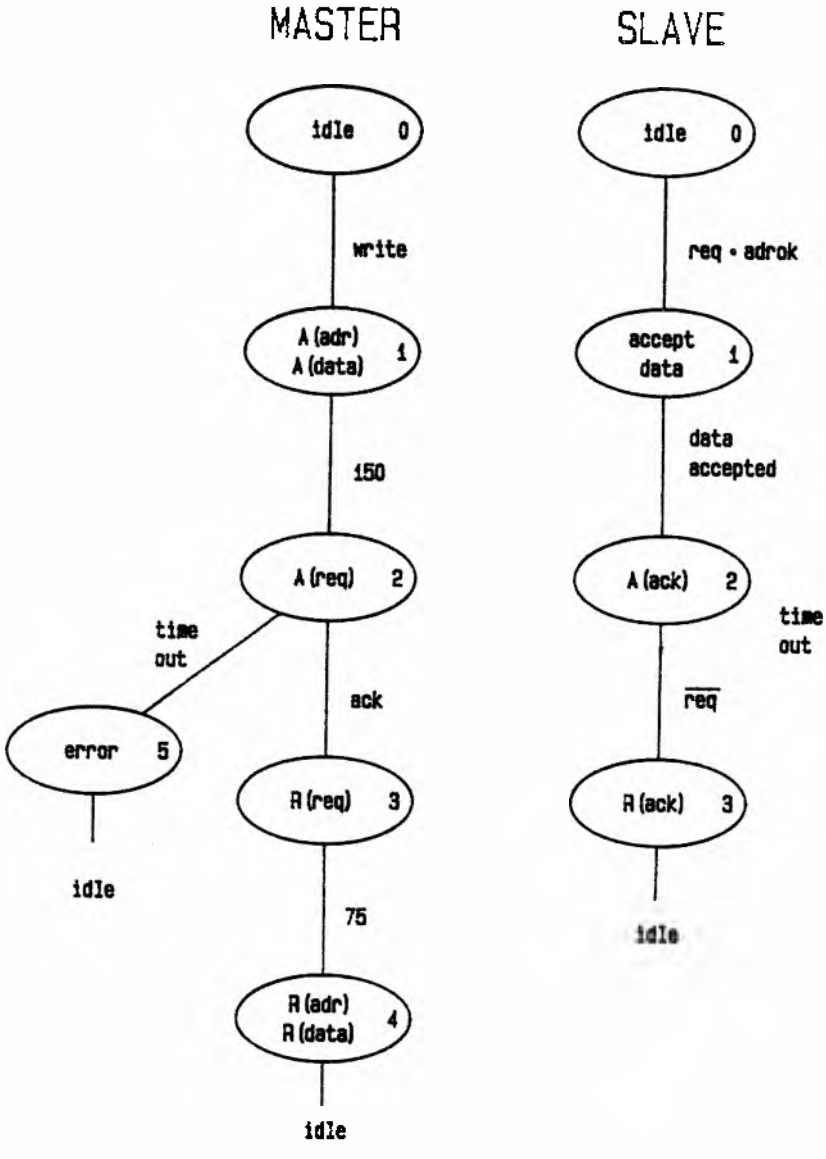


Figure 5.3. Master and Slave State Diagrams for a Simple Write Protocol.

the idle state. The slave responds by moving to state 3 and releasing 'ack' before returning to the idle state.

The master state 5 provides the same type of error handling as previously mentioned for the generic read protocol. The 'req' stuck-at-1 fault will cause problems as shown by the following scenario. The master assumes control of the bus and moves to state 1, asserting the 'adr' lines and the 'data' lines. Since the 'req' signal is stuck-at-1 the slave will respond as soon as an address is detected which is within its assigned address space. This leaves no settling time for noisy signals, so the address which is accepted could well be incorrect; in fact, in this situation more than one slave could respond to the address. The slave then moves to state 1 to accept the data and write it into the address which was detected. If the address which is used is the incorrect address then the data will be written into a wrong location, effectively destroying whatever was contained in that location previously. This improper operation of the slave can be prevented by requiring the request signal to be unasserted at the beginning of the cycle, which will be discussed in further detail in the context of a read protocol.

The assertion of the 'data' lines by the master at the beginning of the cycle is the only basic difference in the read and write protocols. Both cycles require the transfer of synchronous data across the 'data' lines and the 'adr' lines. Both cycles utilize interaction between the modules controlled by the 'req' and 'ack' control lines. Therefore, the techniques presented here for read cycles will apply equally well to write cycles, as do the techniques which are described in the next section. For that reason the

examples which are used in the remainder of this thesis will consist of read cycles.

Figure 5.4 gives a further modification to the read protocol of Figure 5.2, which prevents the slave from responding in the case of request stuck-at-1. This can be accomplished by recognizing the fact that the request line must go through a complete cycle for each transfer, starting at an unasserted level and becoming asserted to initiate the cycle, and then being released again to complete the cycle. Therefore, the request signal must start the cycle in an unasserted state. Detection of this condition is accomplished by the addition of states 5, 6, and 7 in the slave state diagram. The slave will enter state 5 from the idle state when conditions are present for initiation of action on the slave's part, in this case when the address is meant for the slave. This condition must be maintained for a minimum period (called 't[a]' in the state diagram); this prevents noisy address lines from incorrectly causing the slave to begin action. Then, if the request signal is unasserted, the slave will enter state 6. However, if a time-out occurs before the request signal is unasserted then error state 7 is entered and the unit returns to the idle state.

The slave will enter state 6 when the address is correct and the request is unasserted, proper conditions for the initiation of a transaction. For a normal transaction the request line will be asserted and the slave will detect this condition and enter state 1, operating as previously described. If the address or the read command is changed then the unit returns to the idle state. If a time-out occurs before anything happens, then the request signal

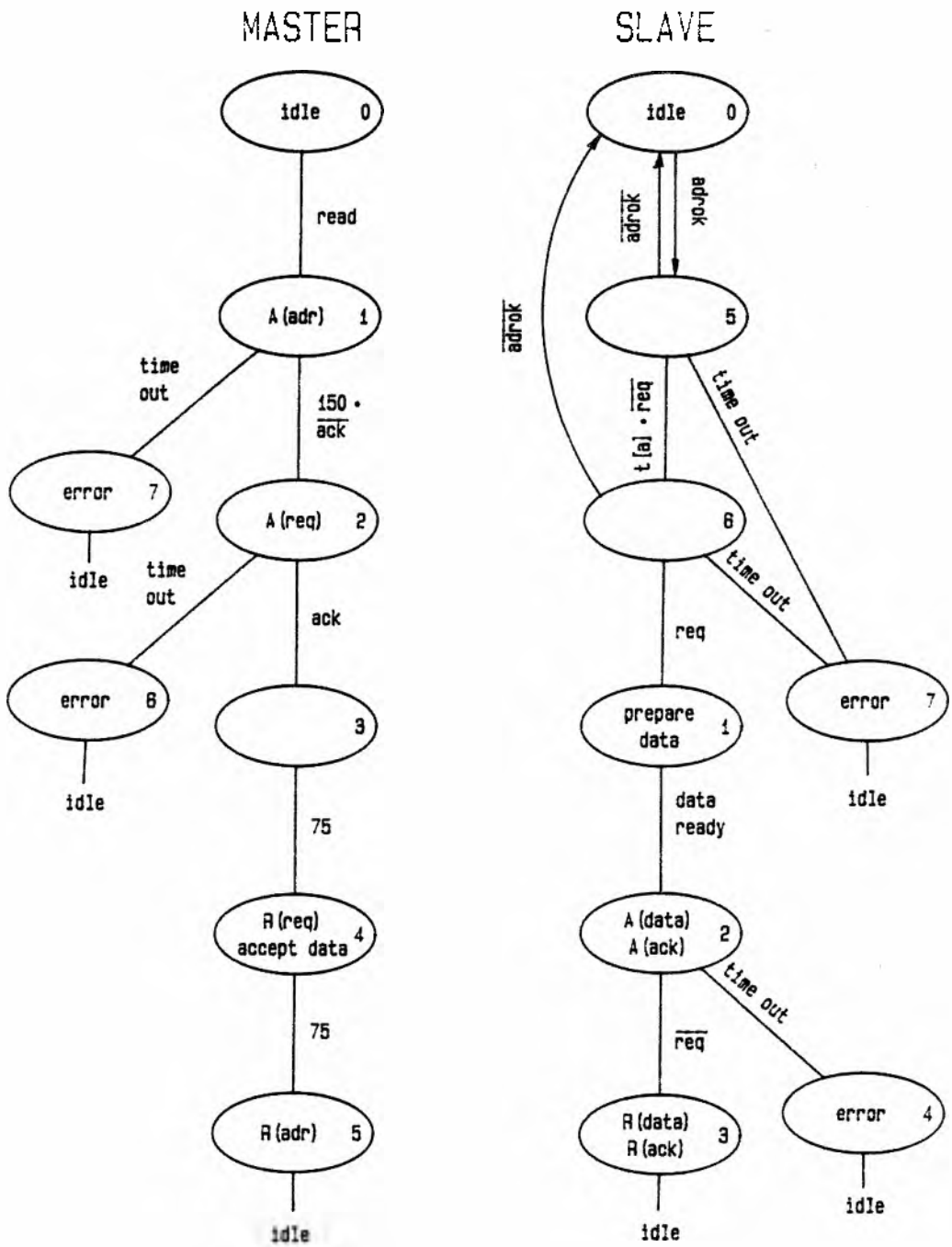


Figure 5.4. Read State Diagrams for Error Detection and No Improper Operation.

is assumed to be stuck-at-0, and the unit returns to the idle state through error state 7. State 7 will be entered by the slave if there is a stuck-at fault on the 'req' line. If the fault is 'req' stuck-at-1, then the entry will be from state 5. If the fault is 'req' stuck-at-0, then the entry will be from state 6. In either case the slave will be prevented from requesting any action by its functional unit.

The system, as described by the state diagrams of Figure 5.4, has two features which enhance system operation over the previously described protocols. First, neither the master nor the slave completes transactions when there is a faulty line. Second, the master detects the occurrence of all four stuck-at errors. The systems of Figures 5.1 and 5.2 do not have either of these features.

That the master unit detects all four errors of the control lines is evident from the following statements. The time-out transition from master state 1 to state 7 will occur when the 'ack' signal is stuck-at-1. The time-out transition from master state 2 to state 6 will occur when the 'ack' signal is stuck-at-0, or the 'req' signal is either stuck-at-1 or stuck-at-0, as well as the improper address as in the original specification.

The protocols represented by the state machines of Figures 5.1 to 5.4 show that there is a correlation between the composition of the state machine and the error detection capability of the protocol. In order to detect stuck-at errors on control lines the following conditions should be incorporated in the state machine:

1. Where a state change requires the assertion or release of a bus con-

trol signal provide time-out escapes to an error state. A bus control signal is one which comes from the bus; other control signals represent the status of local variables. The signals 'req' and 'ack' are bus control signals, while 'read' and 'data_ready' are local signals.

2. Require full transitions of signals before returning to the idle state. Except when leaving the idle state, require a false signal value one state before the true signal is needed, with appropriate time-out escape sequences.
3. Where a bus control signal is needed to initiate action (i.e., get a slave out of idle) and there is a precondition (i.e., the address must match), create a new state where the precondition can be tested first, and then a second state for checking for the unasserted bus control signal. Add time-out escapes to these new states to check for stuck-at faults.

Theorem 5.1: A protocol represented by state machines satisfying conditions 1 and 2 above will detect any stuck-at fault on the bus control lines.

Proof: We assume that the error detection mechanism consists of one of the state machines traversing through an error state. Step 2 assures that bus control signals must make a complete transition from unasserted to asserted to unasserted before the state machine can return to the idle state. If a signal is stuck-at-X, where X is 0 or 1, then the signal will not make a full transition. When the signal does not make the required transition, then step 1 will guarantee that there is a time-out sequence through an error state. Thus, any

stuck-at fault will result in a state machine returning to the idle state through an error state, detecting the fault. □

Theorem 5.2: A protocol represented by state machines satisfying conditions 1, 2, and 3 above will detect any stuck-at fault on the bus control lines and prevent improper operation of the bus. Improper operation of the bus is defined as performing an incorrect cycle.

Proof: Theorem 5.1 has shown that stuck-at faults will be detected by state machines satisfying steps 1 and 2. What remains to be shown is that when a stuck-at fault exists, then improper operation does not occur. An improper operation will be avoided when the state machine cannot complete an incorrect cycle. An incorrect cycle is prevented by not proceeding in the state machine unless conditions are correct for the cycle. The conditions will be correct if and only if the sequencing required by step 3 is in effect: the precondition must be satisfied, and then the required control signal assertion can occur. If the order is reversed and the signal assertion is true before the precondition, then the order is incorrect and the action prescribed by step 3 guarantees that the state machine will not complete the read (or write) cycle. □

Theorems 5.1 and 5.2 indicate that time-out sequences can be utilized effectively in detecting the presence of a fault on the control lines, but these procedures fall short of the goal of continued correct operation in the presence of a single fault. In order to continue operation when a control signal is stuck-at-X, redundancy of that signal is required. In the following section we show that dual-rail redundancy of the control signals can be used to guarantee correct operation when one of the signals is stuck-at-X.

5.3. Continuous Operation with Dual Rail Redundant Signals

If a bus control signal is stuck-at a value, then the use of time-out escapes can detect the presence of that fault, but in order to continue operation some other techniques are required. By using dual-rail control, that is, duplicating the control lines, the correct information is available on one of the duplicated bus control lines, assuming a single line failure. The problem is to decide which of the duplicated lines is in error and then utilize the other line to control the protocol functions. The assumptions about the type of fault remain the same: a single stuck-at-X fault on one of the control lines.

We assume that the duplicated control lines carry the same logical value. Some dual-rail systems implement the signals such that one is the complement of the other; this detects the presence of a bridging fault between the two lines, since the signals would be correct only if the two lines are at opposite logic values. However, when such a bridge exists, the effective logic value of both lines remains the same and no transitions occur. In this situation no information is available as to the intended operation of the signal. If the two lines are asserted to the same value, then bridging faults are not detected, but correct operation will continue.

Duplicating the bus control signals changes the protocol and the state machines which define it. We now present three algorithms for deriving a new state machine utilizing dual-rail signals from the original state machine. The algorithms generate a new state machine based on the characteristics of

the original state machine and the signals which control it. The signals fall into two categories: bus control signals and local signals. Bus control signals are those which communicate information between modules over the bus lines; local signals deal only with the module itself and are not affected by the bus. We are assuming that the faults are contained in the bus control signals. In addition to the control signals, state machines often use conditions to control transitions from one state to another. Conditions reflect the status of bus lines, and one obvious example is the condition of a parity error on a set of synchronous bus lines.

The algorithms specify the number of states needed in the new state machine to represent a state in the original state machine, based on sets of bus control signals. For the following definitions of these sets, refer to the example of Figure 5.5, which shows the state machine for the master involved in a read cycle using Algorithm 2.1.

The set K_s is the set of bus control signals needed to make the next state decision for state s in the original state machine. For the state machine of Figure 5.5, the set K_3 is empty since '75' is a local timing signal and 'errindat' is a condition; the set K_2 contains 'ack' and 'adrpe' since both of these bus control signals are used to determine the next state.

The set L_s is the set of bus control signals which are required to change their logic values in order to cause a transition from a state immediately preceding state s to state s. In Figure 5.5, the set L_2 is empty since only the local timing signal '150' is used on the input arc; the set L_3 consists only of 'ack' since a change is not required in the value of 'adrpe'.

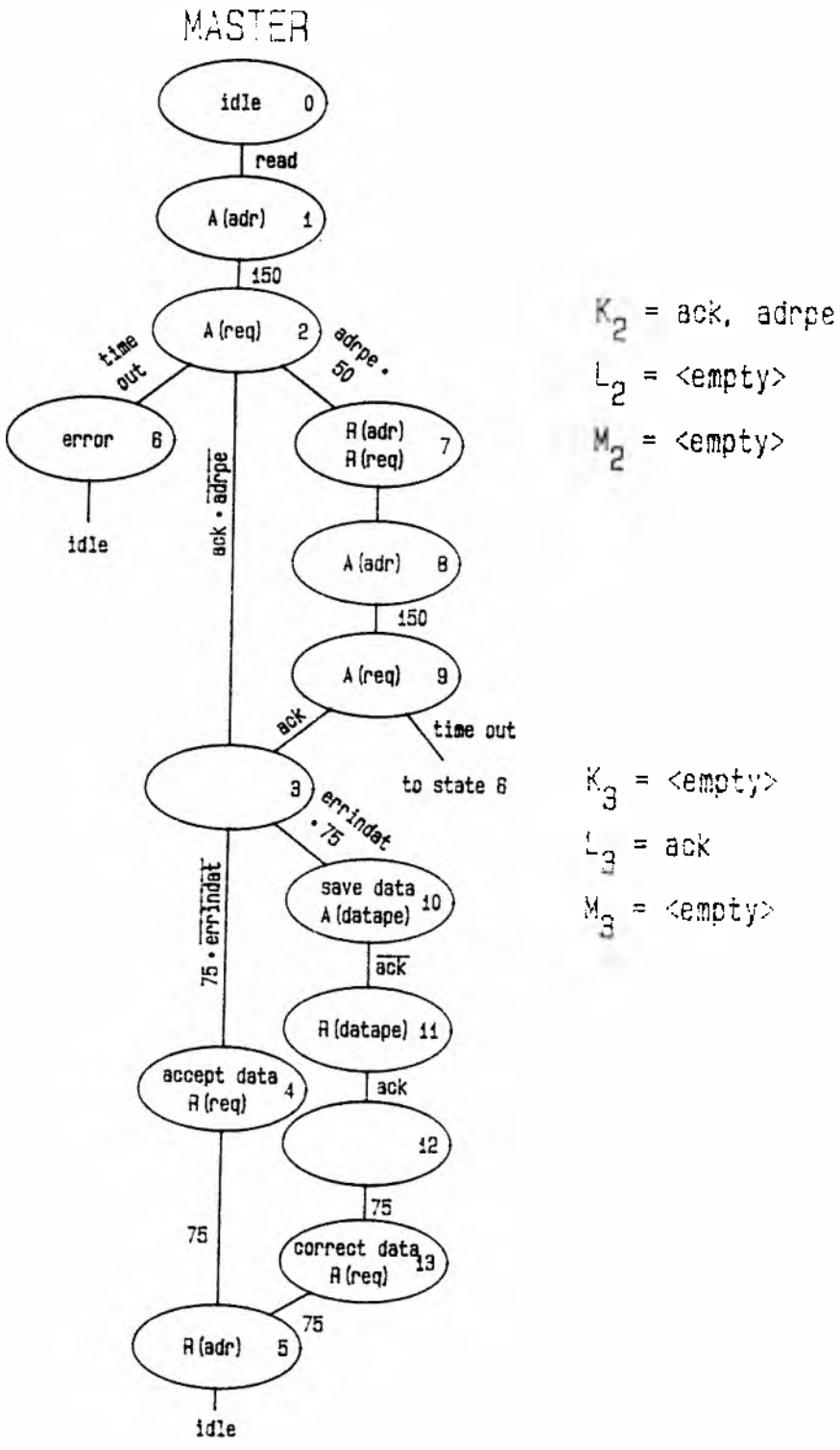


Figure 5.5. Identification of Signal Sets for Dual Rail Algorithms.

The set M_s is the union of sets L_r, where r in an immediate predecessor of state s. M_2 and M_3 are both empty since no bus control signals are required to arrive in state 1 (predecessor to state 2) or states 2 and 9 (predecessors to state 3). Examples of states with non-empty M sets are state 8 and state 13. M_8 contains 'adrpe,' and M_{13} contains 'ack'.

To accommodate the dual-rail signals the new state machine must contain sufficient states to resolve ambiguities caused by a fault on any bus control line. Thus, each state in the original state machine must expand into enough states to identify faulty conditions on the signals it deals with. Resolution must be provided for the signals in the set K_s to detect faults in which the signals in the set are stuck-at the value needed to advance to the next state. Resolution must be provided for the signals in the set L_s to detect faults in which the signals in the set are stuck-at the value opposite that required to arrive in state s. By including the signals in the set M_s allows the resolution of signal skew between the dual-rail lines. The number of states involved in resolving these ambiguities is n, where

$$n = 1 + 2\,|K \cup L \cup M|.$$

$|Q|$ = cardinality of the set Q

$Q \cup R$ = union of sets Q and R

These set definitions allow succinct expression of the algorithms for generating new state machines.

Algorithm 5.1

Step 1: Except for the idle state, replace each state in the original state machine with n states in the new state machine, where

$n = 1 + 2\,|K \cup L \cup M|$. The idle state is a special case treated in the next step.

Step 2: The idle state is acted upon in one of three methods, depending upon the signals needed to exit the idle state:

a) Local signals: If the original state machine exits idle under control of local signals, then no additional states are required; represent the idle condition in the new state machine with a single idle state.

b) Bus control signals ANDed with conditions: If the original state machine exits the idle state by the transition of bus control signals which have been ANDed with conditions, then represent idle as a single state. Create m new states as successors to the idle state for each arc leaving the idle state; $m = 1 + 2\,|\text{set of bus control signals on that arc}|$. For each state in these groups of m states, place an arc from idle to the state and a return arc back to idle. The arcs from idle will be labeled with the condition ANDed with the unasserted control signals, one arc labeled with the fault free control lines and the other m-1 arcs labeled such that one control line is stuck-at an incorrect value. The arcs returning to the idle state will be labeled with the complement of the condition, so if the condition ceases to be valid, then the state machine returns to the idle state.

c) Bus control signals only: If the exit from the idle state is determined only by bus control signals, and no condition is ANDed with these bus control signals, then the idle state must be replaced by m

separate states, where m = 1 + 2 |set of bus control signals which control the exit from idle|. Of the m states, one is for error-free conditions, and the other m-1 states each represent the condition that one of the signals is stuck-at an incorrect value.

Step 3: Let I represent the set of states in the new state diagram, corresponding to state i in the original state diagram (or the set of states representing an arc from idle created by step b above). Let J represent the set of states in the new state diagram, corresponding to state j in the original state diagram. If there is an arc from i to j in the original state machine, then place an arc from each state in I to each state in J where

a) control signal requirements can be met for leaving I and

b) control signal levels will be appropriately matched in J.

Application of Algorithm 5.1 to a state machine will result in a new state machine which will detect stuck-at faults on control lines which it uses as inputs, and continue to function in the presence of those faults. Examples of the application of this algorithm are shown in Figure 5.6 and Figure 5.7.

Figure 5.6 shows the correlation between the master state machine for a simple read cycle and the state machine which has been expanded to utilize dual-rail signals. States 0, 1, 5, and 6 all map to single states in the new state machine. The condition for leaving the idle state is a local signal so step 2a applies, and this requires no expansion of the idle state. However, since K_2 contains 'ack', making n = 3, state 2 in the original state machine expands to states 2, 7, and 8 in the new version. In a fault-free simple sys-

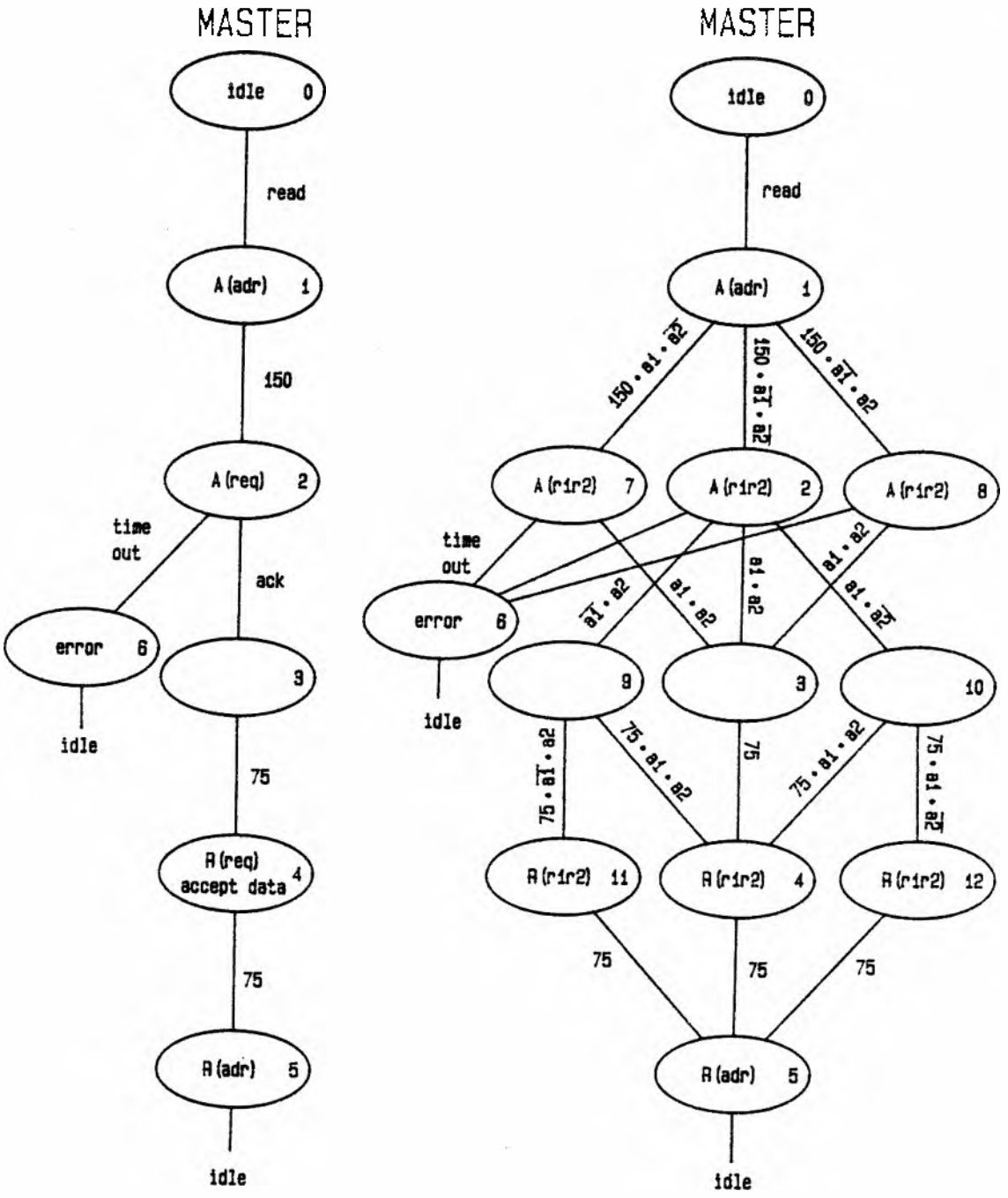


Figure 5.6. Application of Algorithm 5.1 to Simple Read Master.

tem 'ack' will be unasserted when the master moves into state 2, and assertion of 'ack' will cause the system to move to state 3. The single-rail signal 'ack' in the original system has been replaced by two signals ('a1a2') in the dual-rail system. In the new system the addition of states 7 and 8 allows one of the signals to be stuck-at-1; when the fault-free line is asserted then the action continues in spite of the fact that the line was faulty. However, since the execution of the state machine visited state 7 (or state 8), the existence of the faulty line is revealed. State 3 of the original state machine expands to include states 9 and 10 in the new version. If one of the dual-rail lines is stuck-at-0, then either state 9 or state 10 will be visited, but the same action will result if there is skew on the 'a1a2' lines. Therefore, these states will not guarantee the existence of a stuck-at-0 fault. However, states 11 and 12, obtained by expanding state 4 in the original diagram, will indicate the existence of a stuck-at-0 fault on one of these lines.

Figure 5.7 shows the expansion of the slave state machine for a simple read to accept dual-rail signals. The bus control signal in the single-rail version is 'req,' and this has been duplicated to 'r1r2' in the new version. Since leaving the idle state of the original state machine is governed by a bus control signal ANDed with a condition, this is a good example of the addition of states, according to step 2b of Algorithm 5.1. Since one bus control signal is needed to force the state machine to leave the idle state, three states are needed in the new state machine to represent that arc in the original system. The 'adrok' condition is ANDed with the appropriate request signal levels to have the slave go to state 4, 5, or 6, state 5 being the error-

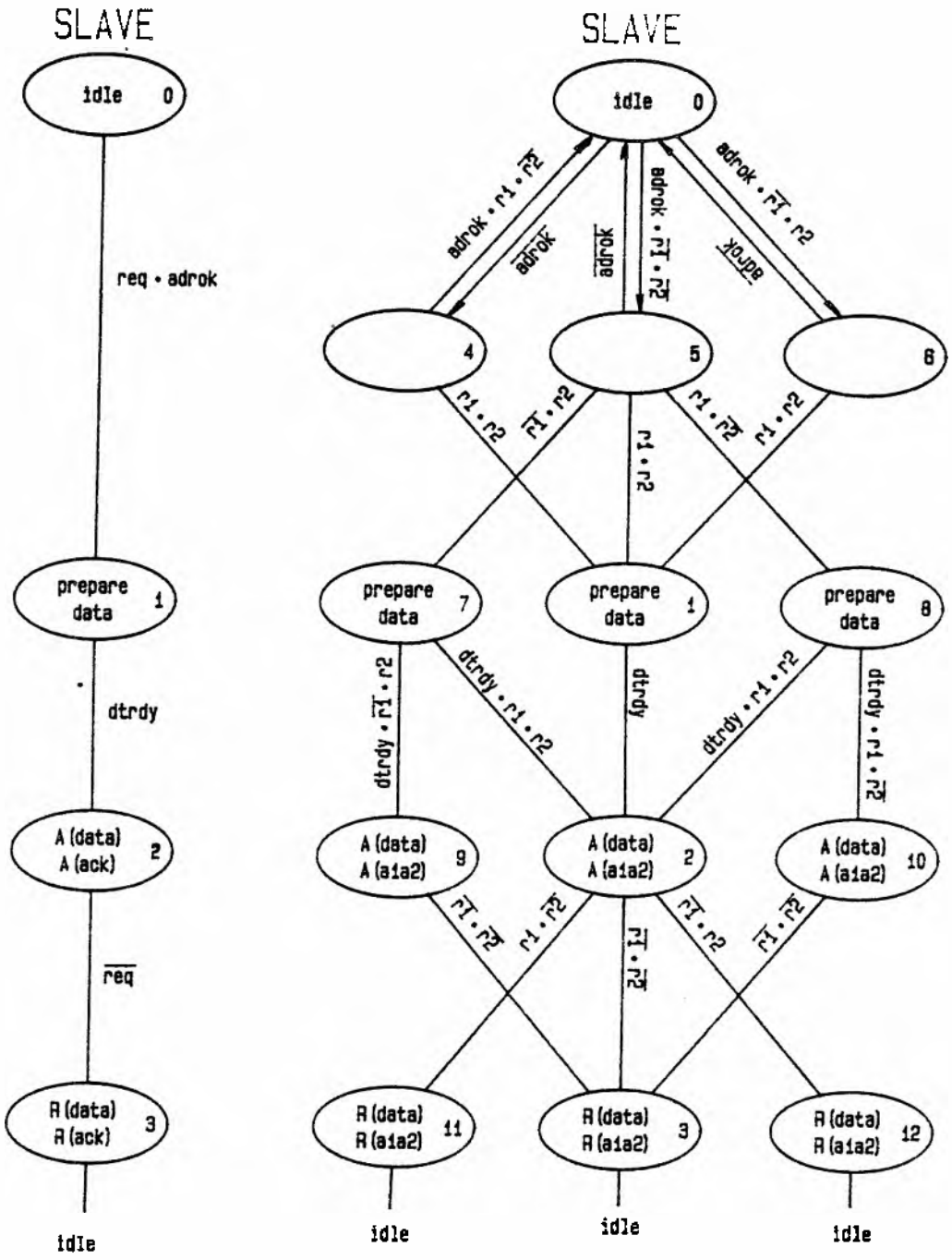


Figure 5.7. Application of Algorithm 5.1 to Simple Read Slave.

free condition and states 4 and 6 indicating a stuck-at-1 fault on a request line. The state machine returns to state 0 if the 'adrok' condition ever becomes false; this prevents spurious addresses on the address bus from causing a slave to respond incorrectly. State 7 or 8 will be the result of either a stuck-at-0 fault or signal skew, so they will not reliably indicate a fault. States 9 and 10 will indicate the presence of a stuck-at-0 fault. States 11 and 12 result from the expansion of state 3; however, like states 7 and 8, they will be used for both faulty conditions and when there is skew between the request lines. Thus, they cannot be used to reliably indicate the presence of an error.

Algorithm 5.1 will result in state machines which detect the presence of single faults on their inputs, but this can result in quite large state machines. This algorithm has been applied to the master and slave state machines for the read cycle incorporating Algorithm 2.1, and these diagrams are given in Appendix E. To reduce the size of n, the number of states required for resolving ambiguities, and to deal with signal skew on dual-rail lines, we now introduce Algorithm 5.2. The number of states involved is reduced by removing the set M from the calculation of the number of states needed in a new state machine to represent states in the original state machine:

$$n = 1 + 2\,|K \cup L|.$$

Although the numbers are different, the basic technique is much the same. The principal difference is the introduction of states to resolve signal skew between the two rows of a dual-rail signal, before entering the correct state in the new version.

Algorithm 5.2

Step 1: Except for the idle state, represent each state in the original state machine with n states in the new state machine, where $n = 1 + 2 |K \cup L|$. The idle state is a special case treated in the next step.

Step 2: The idle state is acted upon in one of three different ways, depending upon the signals needed to exit the idle state:

a) Local signals: If the original state machine exits idle under control of local signals, then no additional states are required; represent the idle condition in the new state machine with a single idle state.

b) Bus control signals ANDed with conditions: If the original state machine exits the idle state by the transition of bus control signals which have been ANDed with conditions, then represent idle as a single state. Create m new states for each arc leaving the idle state, where $m = 1 + 2 |\text{bus control signals on arc}|$. For each state in these groups of m states, place an arc from idle to the state and a return arc back to idle. The arcs from idle will be labeled with a condition ANDed with the unasserted control signals, one arc labeled with the fault free control lines and the other m-1 arcs labeled such that one control line is stuck-at an incorrect value. The arcs returning to the idle state will be labeled with the complement of the condition, so if the condition ceases to be valid then the state machine returns to the idle state.

c) Bus control signals only: If the exit from the idle state is determined only by bus control signals, and no condition is ANDed with these bus control signals, then the idle state must be represented by m separate states, where m = 1 + 2 |set of bus control signals which control exit from idle|. One of the m states is for error-free control lines, and the other m-1 states each represent the condition that one of the bus control signals is stuck-at an incorrect value.

Step 3: Let I be the set of states in the new state diagram corresponding to state i in the original state diagram (or a set of states representing an arc from idle created by step b above). Let J be the set of states in the new state diagram corresponding to state j in the original state diagram. If there is an arc from i to j in the original state machine-then place an arc from each state in I to each state in J where

a) control signal requirements can be met for leaving I and

b) control signal levels will be appropriately matched in J.

Step 4: To allow for the signal skew between dual-rail signals, place additional states in the new state machine where an arc entering a state could be traversed during fault-free operation if the two values have a race condition between them. Place an additional arc from this new state to the state which would have been entered had no race condition been present.

Like Algorithm 5.1 this algorithm will detect single stuck-at faults which occur on the dual-rail inputs. However, unlike Algorithm 5.1, Algorithm

5.2 will guarantee that if the state machine visits one of the n-1 states of the new state machine, which represent error conditions, then the signal involved is stuck-at the indicated value. The contrast between the two techniques can be seen by comparing Figure 5.6 with Figure 5.8, and Figure 5.7 with Figure 5.9. Figure 5.8 shows the application of Algorithm 5.2 to the master of the simple read cycle, and Figure 5.9 demonstrates its application to the slave.

Figure 5.8 shows that state 4 for the master of the simple read cycle does not expand to three states, since the set M of bus control signals is not considered in determining the number of states needed for the new representation. However, two states are required to resolve skew ambiguities between state 2 and states 9 and 10. States 11 and 12 are added according to step 4 of Algorithm 5.2 to resolve these ambiguities. With these additions a stuck-at-1 fault in one of the acknowledge lines will result in state 7 or 8, and a stuck-at-0 fault in one of the lines will result in state 9 or 10.

Figure 5.9 shows a similar result for the slave of the simple read cycle. States 13 and 14 have been added according to Step 4 of Algorithm 5.2 to resolve signal skew on assertion of the request lines, and states 15 and 16 will resolve skew problems on the release of the request lines. This guarantees that states 7, 8, 11, and 12 will detect stuck-at faults on the request lines.

Algorithm 5.2 can also be applied to more complex state machines with similar results. The master and slave state machines for the read cycle incorporating Algorithm 2.1 are given in Appendix E.

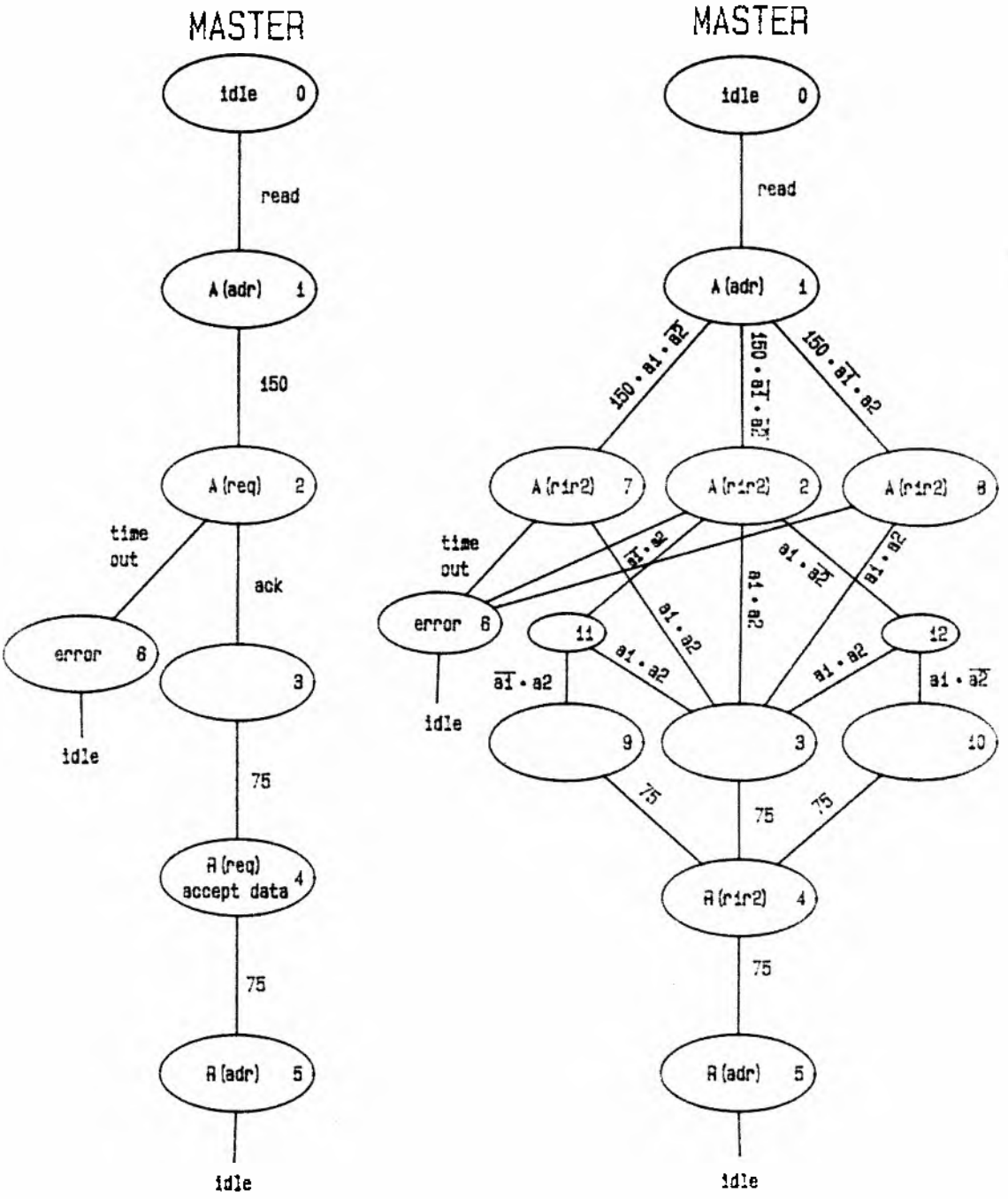


Figure 5.8. Algorithm 5.2 Applied to the Master of the Simple Read Cycle.

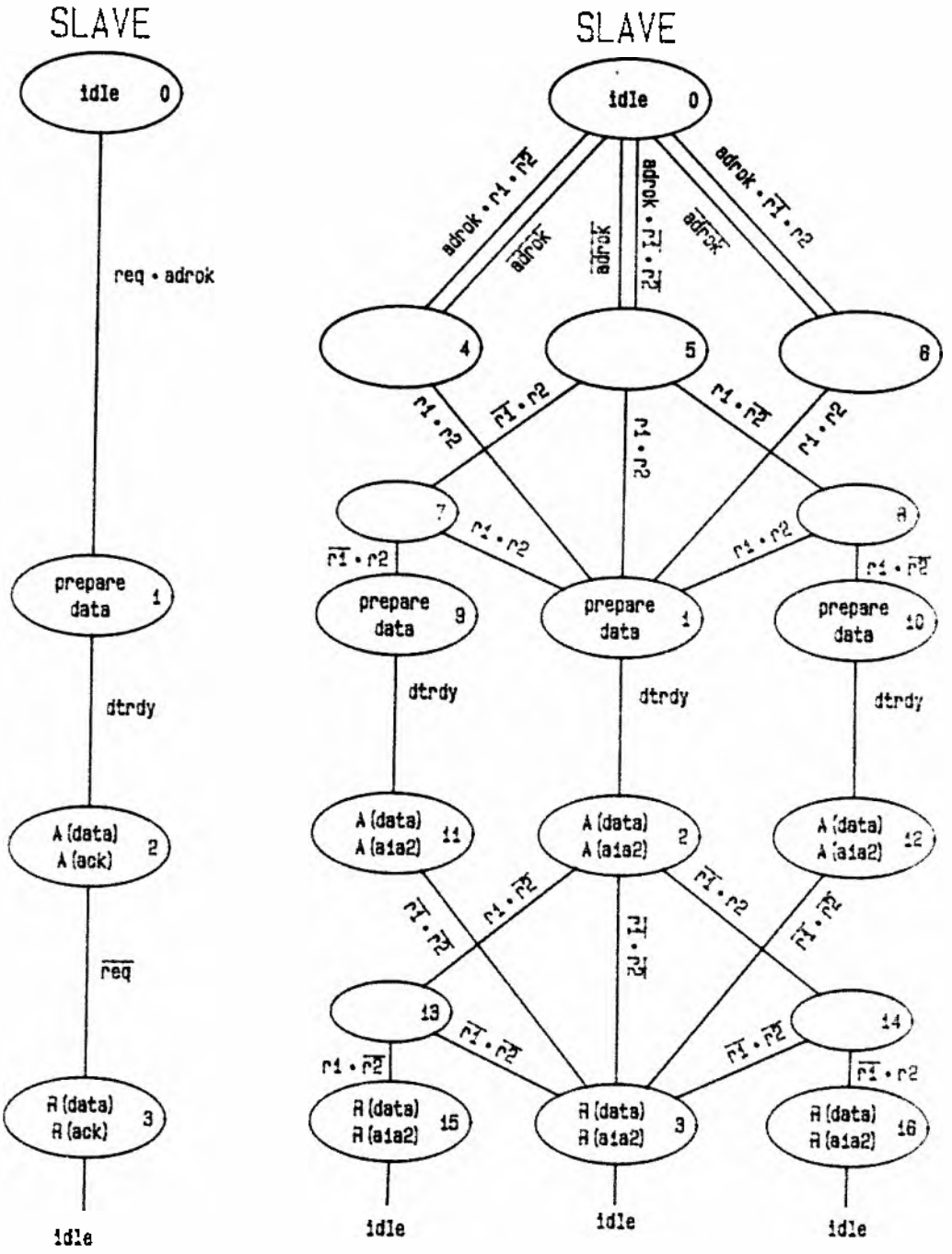


Figure 5.9. Algorithm 5.2 Applied to the Slave of the Simple Read Cycle.

The number of states which is required in the new state machine to represent a state of the old state machine will be fewer for Algorithm 5.2 than for Algorithm 5.1, but the states saved in this manner will generally be added back in to account for the signal skew problems in the state diagram. However, using Algorithm 5.2 will result in a system where the detection of faults at each step of the way is guaranteed.

If detection of a fault is the prime motivation for the application of the algorithms, and specification of the exact line which is at fault is not necessary, then the number of states involved can be reduced by considering the Exclusive-OR of the control signals instead of providing a separate state for each possible error. In this case only one state is provided to indicate an error in a pair of dual-rail lines, instead of providing separate states to uniquely identify the error. Therefore the number of states needed to replace a state in the original state machine changes somewhat, with n being calculated as

$$n = 1 + |K \cup L|.$$

The application of this technique is presented as a modification of Algorithm 5.2, but it could also be applied to Algorithm 5.1.

Algorithm 5.3

Step 1: Except for the idle state, replace each state in the original state machine with n states in the new state machine, where $n = 1 + |K \cup L|$. The idle state is a special case treated in the next step.

Step 2: The idle state is acted upon in one of three different ways, depending upon the signals needed to exit the idle state:

a) Local signals: If the original state machine exits idle under control of local signals, then no additional states are required; represent the idle condition in the new state machine with a single idle state.

b) Bus control signals ANDed with conditions: If the original state machine exits the idle state by the transition of bus control signals which have been ANDed with conditions, then represent idle as a single state. Create m new states as successors to the idle state for each arc leaving idle, where m = 1 + |set of control signals on the arc|. For each state in these groups of m states, place an arc from idle to the state and a return arc back to idle. The arcs from idle will be labeled with the condition ANDed with the unasserted control signals, one arc labeled with the fault-free control lines and the other m-1 arcs labeled such that one of a pair of bus control lines is stuck-at an incorrect value. This is detected by the Exclusive-OR of the two lines. The arcs returning to the idle state will be labeled with the complement of the condition, so that if the condition ceases to be valid, then the state machine returns to the idle state.

c) Bus control signals only: If the exit from the idle state is determined only by bus control signals, and no condition is ANDed with these bus control signals, then the idle state must be replaced by m separate states, where m = 1 + |set of bus control signals

determining the exit from idle!. One of the states is for the fault-free conditions, and the other m-1 states, each represent the condition that one of the lines of a dual-rail bus control signal is stuck-at an incorrect value. Unlike Algorithms 5.1 and 5.2, this algorithm uses only one state to indicate that one member of a dual-rail signal pair is incorrect. This is detected by the Exclusive-OR of the two signals.

Step 3: Let I be the set of states in the new state diagram corresponding to state i in the original state diagram (or a set of states representing an arc from idle created by step b above). Let J be the set of states in the new state diagram corresponding to state j in the original state diagram. If there is an arc from i to j in the original state machine, then place an arc from each state in I to each state in J where

a) control signal requirements can be met for leaving I and

b) control signal levels will be appropriately matched in J.

Step 4: To allow for the signal skew between dual-rail signals, place additional states in the new state machine where an arc entering a state could be traversed during fault-free operation, if the two values had a race condition between them. Place an additional arc from this new state to the state, which would have been entered, had no race condition been present.

Algorithm 5.3 results in a smaller number of states than either Algorithm 5.1 or 5.2, yet the fact that a fault has occurred is detected and the infor-

mation exchange continues as before. The fault is identified to a pair of lines, but which line of the pair is not known. The master and slave state machines for the simple read cycle and the application of Algorithm 5.3 to them are shown in Figures 5.10 and 5.11.

The change in the state machines becomes evident when Figure 5.10 is compared with Figure 5.8. States 0 and 1 remain the same, but states 2 and 3 expand to only two states instead of three. In the modified state diagram there are only two exit paths from state 1. The first is the normal fault-free exit, with both 'a1' and 'a2' unasserted. The second path is followed when either 'a1' or 'a2' is stuck-at-1, which is detected by a1 ⊕ a2. When this is true then state 7 is entered instead of state 2, signaling the existence of the fault. Similarly, there are two possible exits from state 2 which are due to assertion of the 'a1a2' group; state 3 is the normal exit and state 8 is the result if a1 ⊕ a2 is true. This will be true if there is a fault or if there is signal skew. The signal skew problem is resolved by waiting for one more state transition; if at that time both lines are not asserted, then one of them is stuck-at-0, and state 9 is entered. Thus, state 7 will indicate a stuck-at-1 problem and state 9 will indicate a stuck-at-0 problem on one of the 'a1a2' lines, although the resolution is not sufficient to indicate which one. In either case the control algorithm continues in the presence of the fault.

The slave state machine of Figure 5.11 can also be compared to its counterpart, Figure 5.9. Two states are used to represent the normal and error conditions, in contrast to the three states used in Figure 5.9. States 4 and

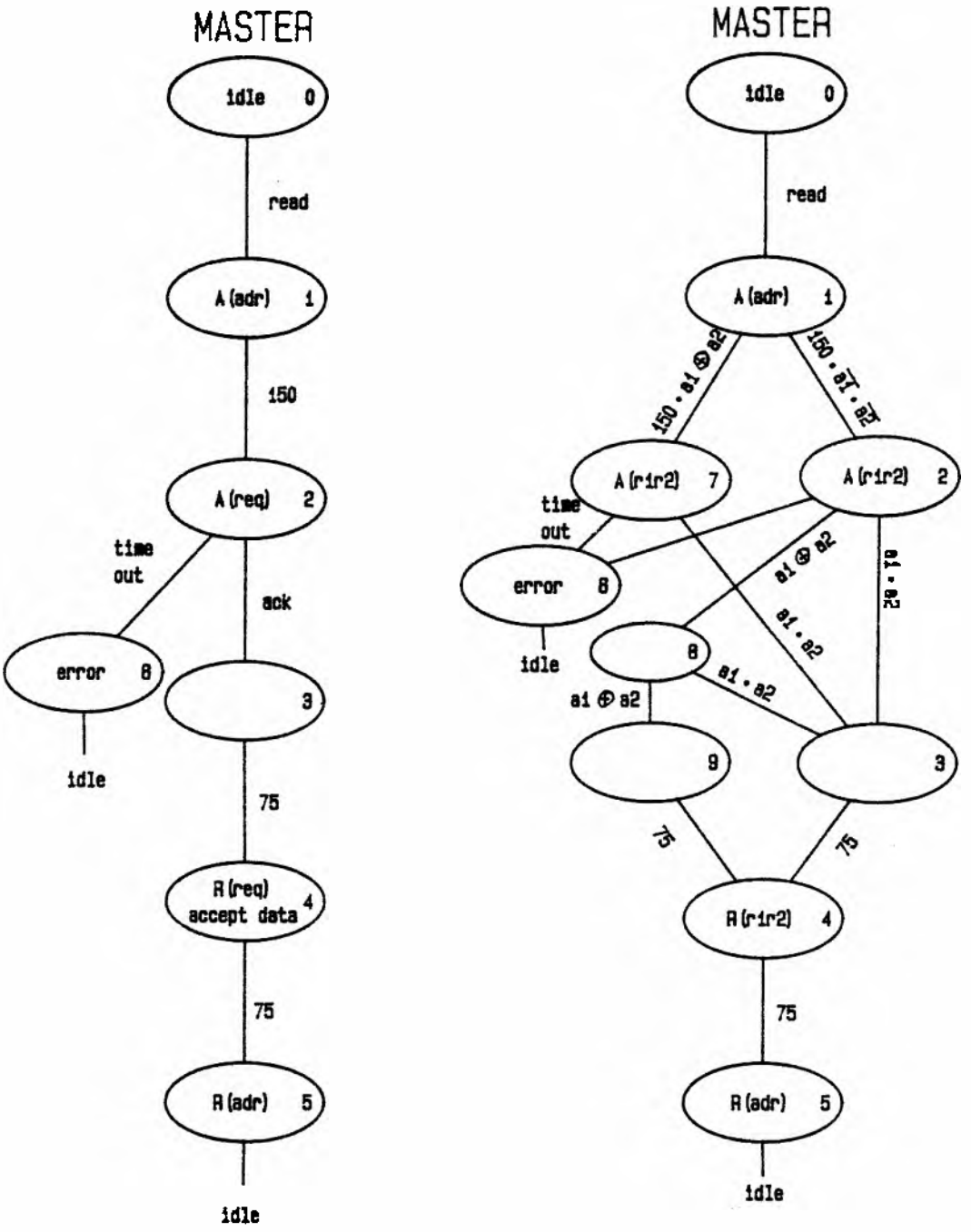


Figure 5.10. Application of Algorithm 5.3 to Master of Simple Read Cycle.

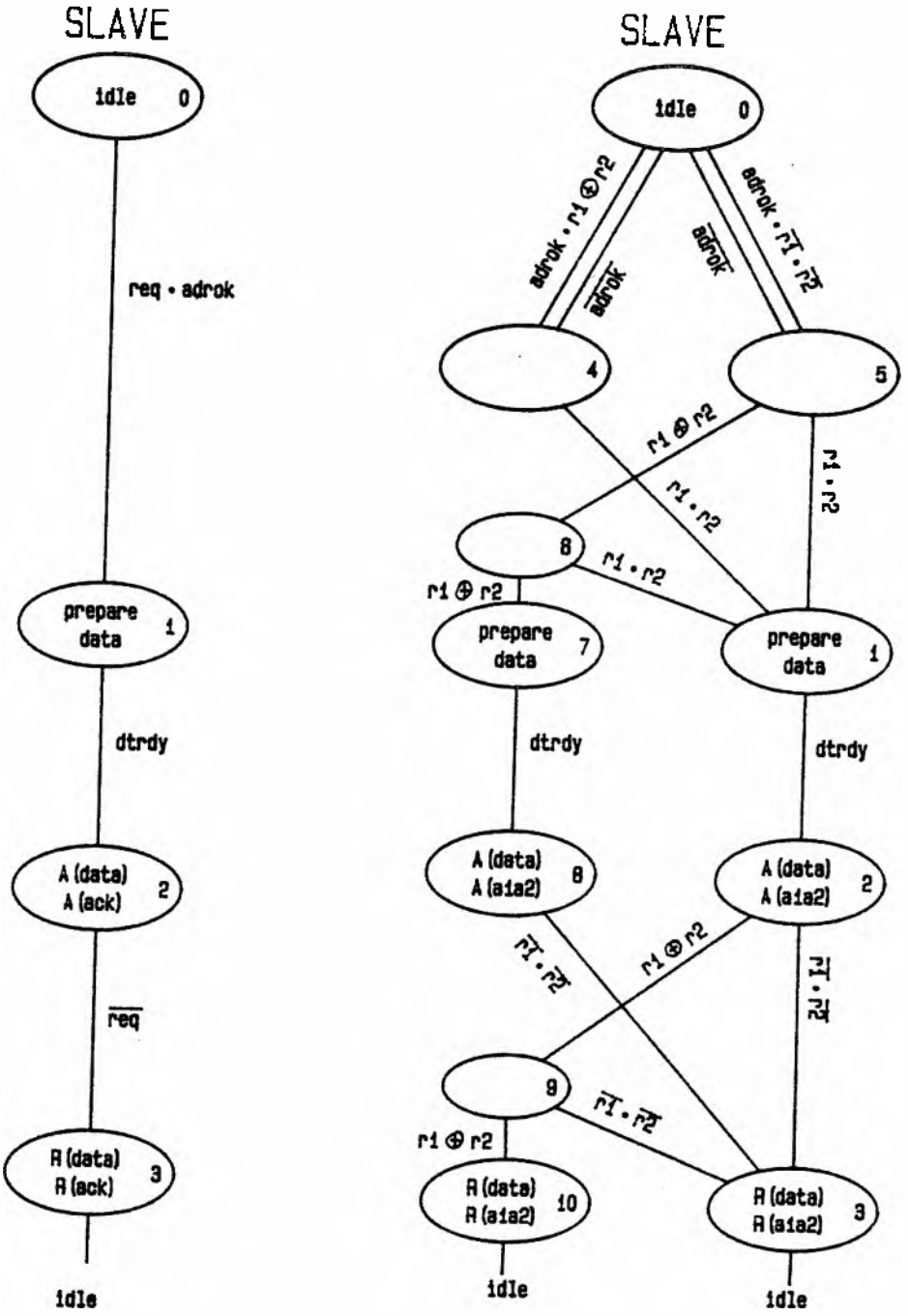


Figure 5.11. Application of Algorithm 5.3 to Slave of Simple Read Cycle.

10 indicate the presence of a stuck-at-1 fault on one of the 'r1r2' lines, and state 7 indicates the presence of a stuck-at-0 fault. These states do not indicate which of the pair of lines is at fault. States 6 and 9 are included to resolve any timing differences caused by signal skew on the two lines. States 3, 9, and 10 could be merged without changing the operation of the slave, since a stuck-at-1 fault on a request line will be reported by state 4 of the next cycle.

The simple examples given in these two figures demonstrate a technique whereby the Exclusive-OR of the two signal lines can be used to determine the presence of a stuck-at-fault. This reduces the number of states required to represent a protocol, thus making implementation more efficient. The loss of this method over the previous method is that the identification as to which of the dual-rail lines is faulty is no longer possible. Nevertheless, the correct operation of the data transfer continues in the presence of the fault.

5.4. Summary of Detection and Correction Techniques

The techniques presented in this chapter have demonstrated that faults which occur on the control lines can be detected, and when enough redundancy is provided operation can be continued in the presence of the faults. If only single-rail control signals are provided, then the presence of the faults will be indicated by the fact that an expected signal does not arrive within a prescribed maximum time interval. This is indicated by a time-out signal. In order to make use of the time-out error indications, guidelines have been provided which identify the characteristics that the state machine representation of a protocol must have.

Another technique which allows the detection of the presence of faults is the use of dual-rail signals, introducing space redundancy in the realm of control signals without imposing the penalties of the three lines needed for TMR. The use of time properties were again utilized in order to allow signal skew to exist between the lines of a dual-rail pair. Allowing the signals to settle by introducing additional states makes the protocol slightly more complicated, but it does allow the system to correctly identify faulty lines at each step of the process.

Three algorithms have been presented which will allow continued operation of transfer algorithms in the presence of faults. The presence of faults is indicated by entering states during the operation of the protocol which identify the problem. Depending on the number of states involved in setting up the system, this will locate the problem either to the specific line which is faulty or to a pair of lines, one of which is faulty. In either case the operation of the system continues.

The master and slave algorithms which have been presented are ideal examples of the application of the algorithms. In both cases the signal sequences are very well defined by the total protocol system: the master proceeds from the idle state knowing that the acknowledge lines must be false, so faults on those lines can be quickly identified. Likewise, the slave will be assured from the protocol definition that the request lines cannot be asserted until a certain time after the address has become stable. This permits the slave to examine the request lines when an address is stable and locate a stuck-at fault if one of the lines is at an unexpected value. These synchronizing

qualities of the protocol sequences are not always true, as shown by the arbitration system problems discussed in the next chapter. However, the expected signal behavior and the use of dual-rail signals will enable us to operate in the presence of faults.

The fault model which has been assumed for these algorithms is rather restrictive since it requires that under fault a line become stuck-at-1(0) when the line is legally at logic 1(0). This precludes any out-of-sequence transitions. If an out-of-sequence transition occurs during the action prescribed by the protocol, then the current operation may not be executed correctly. However, the next cycle will be correctly executed and the error will be identified.

CHAPTER 6

ARBITRATION FOR CONTROL OF BUS LINES

6.1. Introduction

The algorithms presented in the previous chapter deal with the interaction between a module controlling the bus lines and a second module responding to the commands of the first. This interaction is very important in bus communications because of the prevalence of the master-slave type of transfer. However, these interactions can only take place when all of the modules are in agreement as to which module is controlling the bus lines. This decision making process is called bus arbitration and is very important, since only one module can control the bus lines at any time. It is also important because all bus systems are faced with the arbitration problem. Some rather complete discussions of the arbitration problems faced in bus-oriented systems are available [45,46,47]. In this chapter we do not deal with the complexities which arise in the logic for implementation of the various arbitration mechanisms, but rather we are concerned with methods of making fairly standard arbitration schemes tolerant to stuck-at faults on bus control lines.

The consequence of an error which allows two masters to assert bus control lines at the same time can cause incorrect data transfers and possibly damage the circuits which are involved. If the technology utilized on the bus is essentially a open-collector type of implementation, then the assertion of a signal by more than one source will not cause any damage to the circuits, but the resulting data will be a logical combination of more than one source.

For example, two masters attempting to complete a read cycle simultaneously will both assert the address lines, and the result will be a logical combination of the addresses asserted by both units. However, if the technology involved is exemplified by tri-state circuitry, physical damage will result when one unit drives a line to a high level while the other unit is driving the same line to a low level. Thus, an arbitration error allowing two master units to assume control of bus lines concurrently can result in destruction of data and physical damage to the circuitry involved.

Another problem becomes obvious when considering the algorithms for dual-rail signals presented in Chapter 5. These algorithms utilize known information about the master/slave interaction in making assumptions about the expected level of the control signals. For example, when the master unit depicted in Figure 5.8 moves from state 1 to assert the request lines in one of the succeeding states, it is assuming that the acknowledge lines will not be asserted. Therefore, if one of the acknowledge lines is asserted, then that line is in error and a state is entered to recognize that fact and essentially ignore the faulty line. Similarly, the slave depicted in Figure 5.9 chooses one of states 4, 5, or 6 assuming that the request lines are not asserted. If one of the request lines is at an asserted level when the 'adrok' condition becomes true then it is assumed to be faulty and ignored. Both of these assumptions are based on a foreknowledge of the behavior of the signals involved: a slave cannot assert an acknowledge line before the master asserts the request lines, and the master cannot assert the request lines until an address has had time to settle on the address lines. These assumptions cannot be made if the arbitration unit will allow more than one master to assume

control of the bus lines at one time. Thus, the arbitration function is vitally important to the system.

Although the arbitration of bus ownership can be accomplished in many ways two basic methods are used: parallel arbitration and serial or daisy-chain arbitration. These two methods are shown in block diagram form in Figure 6.1. In systems where expansion is limited and speed is of major importance parallel arbitration is used. This method calls for each master module to assert a bus request signal which is not available to other modules on the bus; the system arbiter accepts all bus request signals and asserts a bus grant signal to only one of the modules. This guarantees that only one module at a time assumes control of the bus lines. An example of this type of arbitration system is the Synchronous Backplane Interconnect (SBI) of the VAX11/780 computer system by Digital Equipment Corporation. The MULTIBUS by Intel also has this type of an arbitration system as an option.

The second method of bus arbitration depicted in Figure 6.1 is the daisy-chain or serial type of decision making. In this type of a system a priority signal is passed serially from one module to the next. Whereas the priority system of a parallel system can be dictated by whatever algorithm the central arbiter chooses to implement, the priority of a serial system is determined by the physical position of the different modules on the priority line. In a serial type of system the arbitration decisions are made by a distributed arbitration network; each module contains that portion of the arbitration system needed for making the bus ownership decision for that module. Examples of systems with serial arbitration include the IBM channel and the

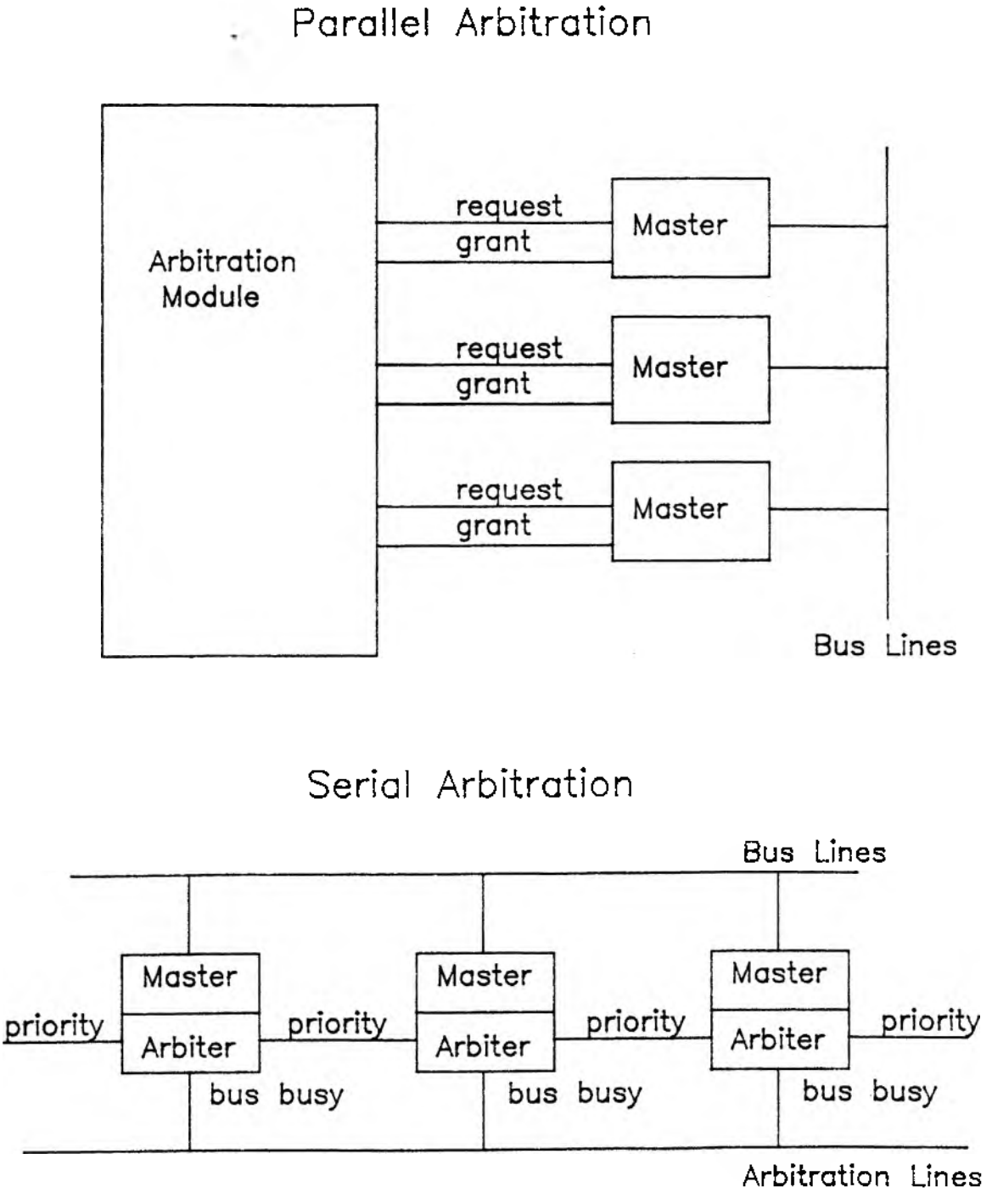


Figure 6.1. Arbitration Mechanisms in Bus Systems.

Motorola VME bus. The MULTIBUS by Intel can operate under either serial or parallel schemes, depending on how the system is constructed. Another option is to combine the two methods, using a parallel scheme between different levels of modules, and within each level use a serial scheme. The UNIBUS by Digital Equipment Corporation is of this type.

The arbitration of the bus ownership is important not only because of the consequences of incorrect data and damaged signal circuits, but because of the universality of the problem. All bus systems which have more than one module capable of controlling the interaction must have some type of an arbitration system. Independent of the method of implementation, whether parallel or serial, the goal is the same for both: permit only one master to control the bus control lines and data lines at one time.

6.2. Parallel Arbitration System

At least two signals in addition to those which have already been discussed are needed to allow a master to obtain control of the bus lines to carry out an information transfer. One signal is used by the master to inform the arbitration system that it needs the bus; this line is called the bus request line, and it is labeled 'br' for short in the diagrams. The other line is a signal from the arbitration system to let the master know that his request is being honored and that he can assume control of the bus; this line is called the bus grant line, and it is labeled 'bg' in the diagrams.

In a fault-free system the action assumed for these lines is that when the master needs the bus, it asserts the bus request line. When the bus is available, following whatever priority algorithm the arbitration system

chooses to implement, the arbitration system asserts the bus grant line allowing the master to assume control of the bus lines. This interaction sequence is added to the simple read sequence of Figure 3.1 and is shown in Figure 6.2, which also shows its expansion into dual-rail signals using Algorithm 5.1. The bus request line 'br' becomes 'b1b2' in the dual-rail version, and the bus grant line 'bg' becomes 'g1g2'.

The 'read' signal for the master of Figure 6.2 differs from the 'read' signal of Figure 3.1 in that no assumption is made concerning the ownership of the bus. In Figure 6.2 the signal merely indicates that the functional portion of the module needs a read cycle. This signal causes the master to enter state 1 and assert the 'br' line; this in turn indicates to the arbitration system that a cycle is needed. When the arbitration system resolves whatever requests have been made for control of the bus, it allows one master module to assume control of the bus lines, and it signals this condition by asserting the 'bg', line of that master which will control the bus. When the master detects the assertion of 'bg' the action as defined by the state machine will continue, the master controlling the lines to complete the needed cycle. When the cycle is complete, the master relinquishes control by releasing the 'br' signal. When the arbitration system detects the release of 'br', it will then grant control to another module as needed.

As can be seen from the state machines of Figure 6.2 the same techniques, which were used to make the master-slave interaction tolerant to single faults on the dual-rail signal lines are used to make the interaction of the arbitration system tolerant to single faults. This system assumes that the parallel

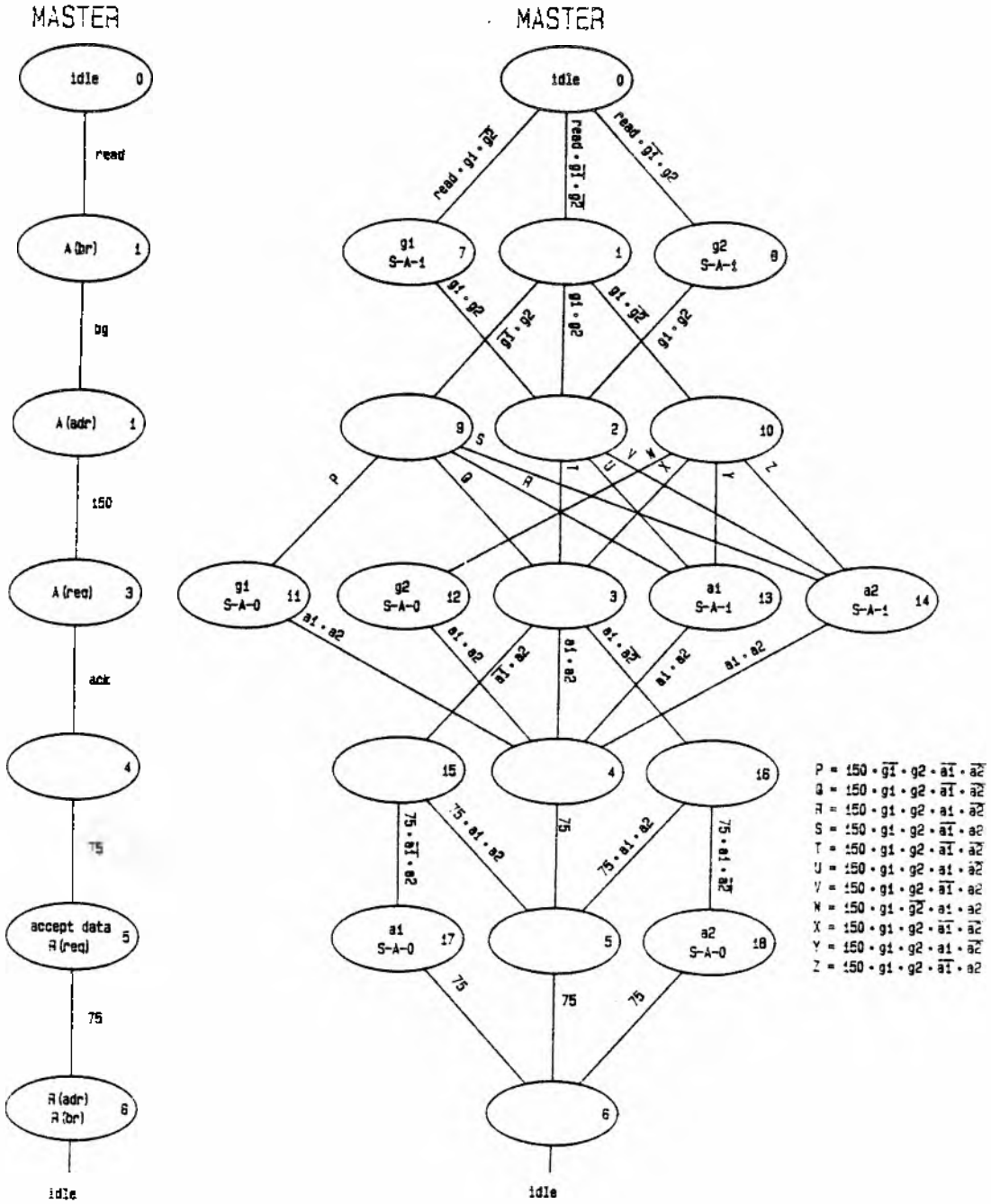


Figure 6.2. Master of Simple Read Cycle with Arbitration Lines Added.

arbiter is free of faults, and that bus lines which interface between the arbitration system and the master modules have at most one stuck-at fault. When these assumptions are correct, then the system will tolerate any single stuck-at fault on data or control lines, as well as bridging faults on data lines.

6.3. Serial Arbitration System

As seen by the previous section, a parallel arbitration scheme is easily incorporated into the bus system which has been discussed in previous chapters. However, there are some drawbacks to parallel arbitration which prevent its use in a variety of systems. In order to perform the central arbitration function of the parallel system, a fairly complex arbiter must be implemented, which becomes entirely responsible for the ownership decisions of the system. In addition to being complex, this system is also limited in its expansion to a fixed number of modules unless the arbitration module is expanded. Thus, many systems use a serial system to accomplish the arbitration function.

A serial arbitration system also has drawbacks, one of which is the time needed to identify a new master module. The time required for arbitration is proportional to the number of master modules on the bus; each module requires some time to respond to arbitration signals.

One simple form of serial arbitration involves three bus control signals: priority-in ('prin'), priority-out ('prout'), and bus-busy ('bbsy'). The bus request and bus grant signal between a module and its serial arbiter are local signals and do not form part of the bus; therefore, we will not consider them

in the bus protection schemes. The 'prin' signal for a module is obtained from the 'prout' signal of the next higher priority module, and the 'prout' for a module becomes the 'prin' of the next lower priority module. The signal 'bbsy' is bidirectional, and it is asserted by the current bus master. The arbitration algorithm grants the bus to the highest priority module seeking ownership when the 'bbsy' signal is released by the current owner. A module desiring to gain mastership of the bus makes a request to its local arbiter. Upon receiving this request, the local arbiter releases its 'prout' signal, effectively preventing lower priority modules from obtaining mastership of the bus. The arbiter then checks its 'prin' signal to see if it can gain mastership of the bus. The arbiter can grant mastership of the bus if its 'prin' signal is asserted and no other module is using the bus, which will be indicated by an unasserted 'bbsy' signal. If the 'prin' signal is unasserted, then the arbiter waits until it becomes asserted. When the 'prin' signal is asserted then the arbiter checks the 'bbsy' signal, and if the bus is not busy, the arbiter asserts the 'bbsy' signal and grants mastership to the module. If 'bbsy' is asserted when the arbiter needs the bus, then the arbiter must wait until 'bbsy' is released by the current bus master before it can assume control of the bus. These three lines allow the arbitration system to be distributed between the modules which require mastership of the bus.

Error detection and fault-tolerance of the control signals involved in the serial arbitration process introduce another problem. Unlike the request-acknowledge protocol between a master and a slave, or the request-grant protocol between a master and a parallel arbiter, there is no handshake or four cycle protocol between the local arbiters of a serial scheme.

Specific values of the bus signals are not expected and, therefore, correct logic values of the signals cannot be distinguished from the incorrect values. Thus, although the presence of a single error can be detected with dual-rail signals, tolerance of a fault cannot be achieved without additional redundancy. Therefore, we propose the use of triplication for the priority and bus busy signal lines. This provides the ability to not only detect faults which occur at any time on the lines but also to identify the faulty line and ignore it.

CHAPTER 7

A CASE STUDY - THE UNIBUS

7.1. Introduction

The UNIBUS was introduced by the Digital Equipment Corporation in 1970 as an integral part of the PDP/11 family of computers [48]. This family of computers was designed to overcome some of the inadequacies of the computers available up to that point. Among the reasons given for the new computer line was a desire for increased structural flexibility or modularity. With the modularity provided by a bus structure users would be able to configure a system optimized for their application, based on cost, performance, and reliability. The majority of this system would be made up of standard modules which would fit into the bus structure, and whatever customized hardware was needed by the user would be added to this same structure. A block diagram of a 'typical' system is shown in Figure 7.1. Another advantage of a system of this type is that as the needs of the user community grow then additional modules, such as memory or peripherals, can be added to meet increased system demands. The UNIBUS became the backbone of the PDP/11 family, and although different timing constraints were associated with the different models the protocol of bus interaction remained the same for all units [49].

Since the introduction of the PDP/11, the number of modules available for use on the UNIBUS has grown to include a multitude of interface and special purpose devices. Users have developed customized interfaces knowing that, if necessary, their system can be upgraded to a higher performance computer

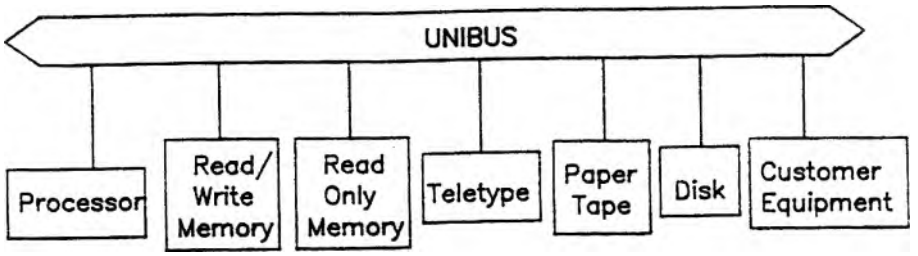


Figure 7.1. Block Diagram of a UNIBUS System.

merely by placing their interface in a more capable machine. These concepts of machine independence and modular construction based on a bus system have been incorporated in other designs since that time. The Lockheed SUE is an example of a system similar to the PDP/11 which has been built using modular construction and a bus protocol for communication. Others include the MULTIBUS by Intel and the VERSABUS by Motorola. We now examine the UNIBUS to ascertain the changes needed to add fault tolerance to the protocol

7.2. Additional Bus Lines for Tolerating Faults

Communication over the lines of the UNIBUS begins when a master signals the arbitration system that it needs to transfer data to/from another module. When the master receives control of the bus, it asserts the address lines to identify the desired partner in the transaction as well as the type of transaction to be performed. The two modules then use the handshake lines to

exchange the data over the data lines. Finally, the bus is released for use by other modules. The lines of the bus which are used to implement this transfer mechanism can be divided into four groups totaling fifty-six lines: the synchronous lines, the handshake lines, the arbitration lines, and the miscellaneous lines.

Information transfer from one module to another occurs in parallel across forty synchronous lines. These include the data lines (18), the address lines (18), and the read/write lines (2). The read/write lines have the same timing specifications as the address lines and identify the transaction as one of four possibilities: read, read-pause, write, or write-byte. The data lines are the only lines on the bus which include any type of error detection; a single parity bit is added to sixteen data bits to assure constant parity on the data path. Thus, only seventeen lines, or only 30% of the bus, are currently covered for error detection and none are covered for error correction. A second parity line has been defined as a member of the 56 bus lines, but it is reserved for future use. Fault coverage can be provided for all the synchronous lines by an additional parity bit to be used for the address and read/write lines. However, in addition to the parity bit, we have also included the two lines needed to implement Algorithm 2.1, so three additional lines are required to detect single bit errors on the parallel data paths and apply the algorithms of Chapter 2 to correct the errors. These additional lines increase the total bus signals to 59, but the fault coverage rises to 69% of the bus, adding correction as well as detection. This information is shown in Table 7.1 along with other configurations of the bus.

Table 7.1. Error Coverage of Different UNIBUS Configurations.

	Synchronous Lines		Control Lines				
	Address and Read/Write	Data	Handshake	Arbitration	Misc	Total	Error Coverage
Standard UNIBUS	20	16(2)	2	12	4	56	30%
Protect Address	20(3)	16(2)	2	12	4	59	69%
Protect Handshake Lines	20(3)	16(2)	2(2)	12	4	61	74%
Detection on Arbitration Lines	20(3)	16(2)	2(2)	12(12)	4	73	95%
Detection on Misc Lines	20(3)	16(2)	2(2)	12(12)	4(4)	77	100%
With Serial Arbitration Only	20(3)	16(2)	2(2)	3(6)	4(8)	66	100%

The signals in the handshake group are the request line (called MSYNC) and the acknowledge line (called SSYNC). Using dual-rail signals for the handshake lines doubles the number needed for that function, but it allows the implementation of the protocols introduced in Chapter 5. The two additional lines bring the total to 61 lines, and the error coverage rises to 74%. The use of dual-rail signals for the handshake functions allows both detection and correction of stuck-at faults.

The arbitration function of the UNIBUS utilizes both parallel and serial functions. The parallel decision is between one of five different levels, each level representing a different priority to the system. In this way the system is capable of ignoring requests for bus interaction from lower level devices until all of the higher level functions have been satisfied. Within each level the priority is a daisy chain arrangement, with each device responsible for passing on any priority signals when not actually requesting bus service. The protection schemes presented in Chapter 6 can be implemented by supplying dual-rail signals for each line in the parallel arbitration group and triplicating the serial priority lines. Fault detection can be achieved by doubling the number of lines involved in the arbitration function. The total number of lines is increased to 73, and the error coverage to 94%. The triplication of the serial priority lines would increase the number of lines further but give assurance of tolerating the faults.

The miscellaneous lines of the UNIBUS include: the interrupt line which is asserted when a master requests interaction with the processor, the initialization signal, and two power supply warning lines. Error detection can be achieved by making these lines dual-rail signals also, bringing the total number of lines to 77 with total fault coverage. Again, in order to tolerate faults on these lines, triplication would be needed.

The number of lines on the bus has increased by 37% in order to provide fault detection on all lines. Triplication of the lines involved in arbitration and the miscellaneous lines in order to tolerate all single faults would increase the number by 66%. However, this is not the only cost to the system

of the additional capability. Not only does the number of lines increase, but also the complexity of the protocol governing those lines. We now examine this increased protocol complexity.

7.3. Additional Protocol Complexity

The action of the UNIBUS protocol is fairly complex and requires many pages of explanations, tables, and graphs to describe [49]. This information can be condensed into state machine representations of the modules involved. The interaction of these state machines describes the control interaction needed to transfer information across the synchronous lines of the bus, etc. Figure 7.2 gives a state machine representation of the master unit and its interaction with the slave. Figure 7.3 is a similar representation of the slave unit. The signals 'msync' and 'ssync' correspond to the request and acknowledge signals of the read protocols discussed in previous chapters. The signal 'sack' is used in the arbitration system. The type of cycle (read, write, etc.) is identified by 'c1c0'. The other signals have names which have been introduced earlier. The number of states in these diagrams gives an indication of the complexity of the protocol, the master requiring 19 states and the slave only 13 states. In order to implement the control required for Algorithm 2.1 and dual-rail signals such as needed for Algorithm 5.2, the number of states in the master increases to 72 and the number of states of the slave increases to 65. However, the amount of logic required to implement these protocols does not increase at the same rate. For example, less than 30% more bits are required to represent the number of states in the new master as opposed to the old one.

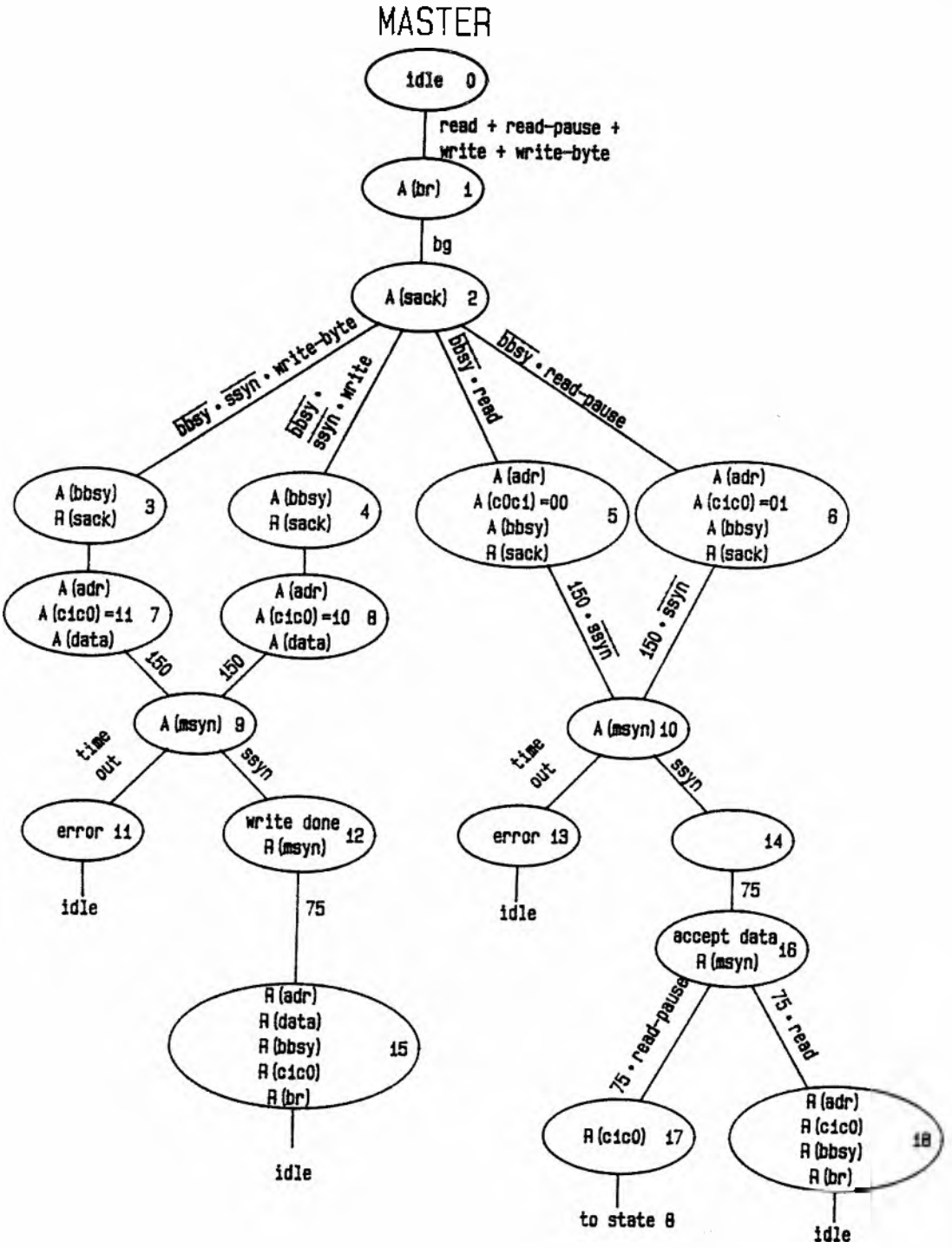


Figure 7.2. State Machine Representation of UNIBUS Master.

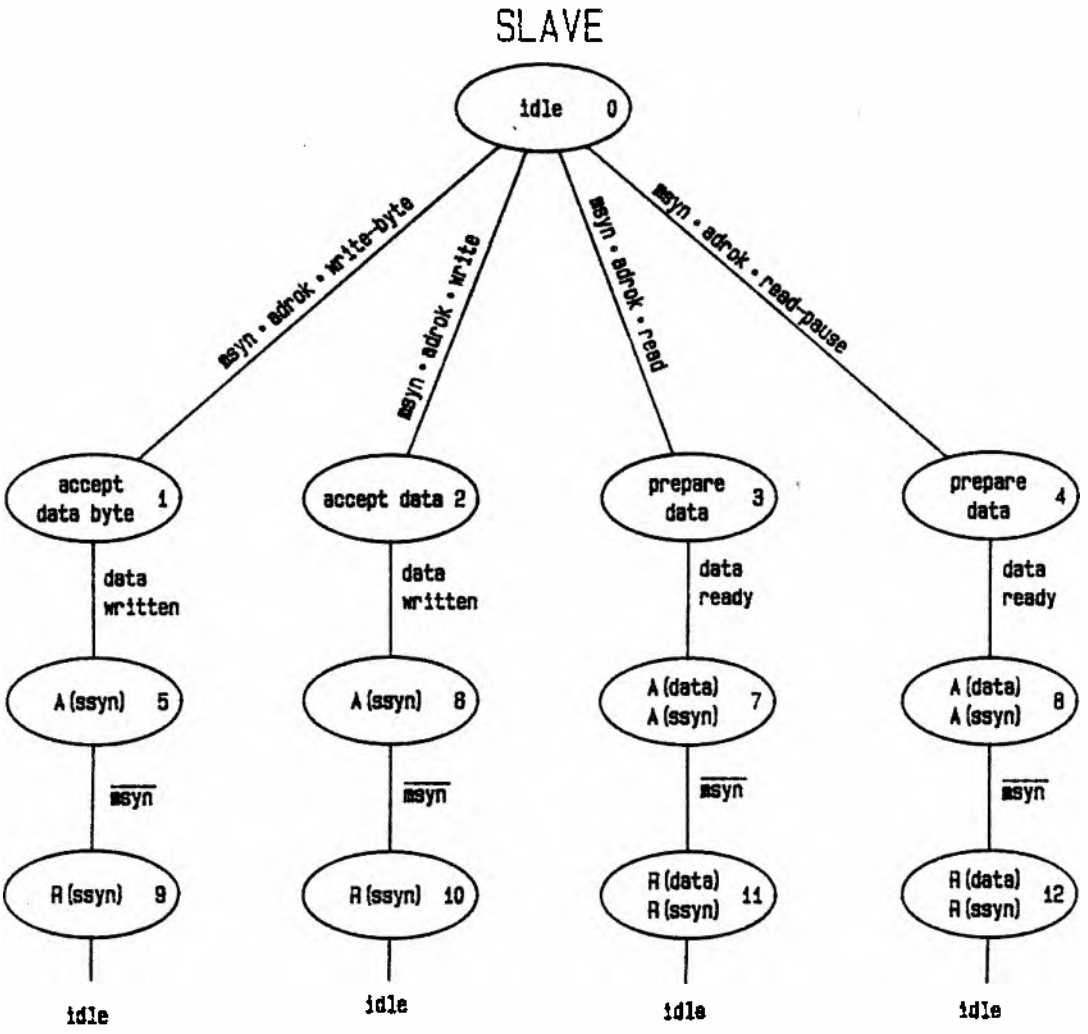


Figure 7.3. State Machine Representation of UNIBUS Slave.

The other functions of the bus, such as the arbitration function, will have the same type of a complexity increase as the functions are implemented with dual-rail signals. The actual cost impact will depend on the actual type of arbitration system used, whether it is a parallel/serial combination or simply one of the methods. The cost of the hardware necessary to implement the protocol has been decreasing steadily as more logic can be included on an integrated circuit chip.

7.4. Total System Cost

By adding lines required to detect faults and continue operation in the presence of those faults, the number of lines needed for the full UNIBUS protocol increased by 23, but the fault coverage (for detection) went from 30% to 100%. In addition to the increased number of lines for the full protocol, the amount of hardware needed for the interface function approximately doubles. Thus, the full UNIBUS protocol can be implemented in a fairly reasonable manner. Typical boards for the UNIBUS devote only about 20% of their logic to the interface function; so the hardware increase for the system would only be on the order of 20%.

A heavy price was paid for the complexity of the protocol, and if a simpler arbitration scheme is acceptable, then the number of lines required for the bus could be reduced. Using only the serial scheme introduced in Chapter 6 would reduce the number of lines involved in arbitration from 24 to 9, and at the same time maintain triplication of arbitration lines for total fault tolerance. Using triplication on the miscellaneous lines as well brings the total number of lines to 69. Therefore a protocol can be established

which has only three more lines than the UNIBUS which will detect the presence of single faults and continue to function. The hardware reduction of this scheme would be principally in the arbitration system, since the parallel arbitration unit is not needed.

The cost of this type of an implementation can be contrasted for comparison with a TMR type of implementation. A TMR implementation of the entire bus would require 168 bus lines, as opposed to the 77 lines of the dual-rail system. If, instead of using TMR for the synchronous data paths, the algorithms of Chapter two are implemented, then only the control lines need be triplicated. However, this still brings the number of bus lines to 95. The TMR implementation also requires voters for each input, so the hardware required is not less than that of the dual-rail system. Therefore, the dual-rail implementation is preferable to the TMR.

As can be seen from the above discussion, there is an entire spectrum of possibilities available to the designer of a protocol. The decisions regarding the 'best' implementation remain a system issue, and so a single best protocol is not possible. But the algorithms and methods are available to use parity, time redundancy, and dual-rail control signals for most of the bus functions to implement a protocol which will tolerate single faults.

CHAPTER 8

CONCLUSION

8.1. Contributions of This Thesis

This thesis deals with the addition of fault tolerance to protocols of bus level communications. We have presented methods and techniques for dealing with faults which occur, not only faults on the data lines but also faults which occur on the control lines. Utilizing these techniques, a protocol designer can incorporate into a bus communication system a high level of fault tolerance.

For the transfer of synchronous information, two algorithms have been presented which will guarantee correct address and data exchange in the presence of single faults. These algorithms operate correctly not only for stuck-at faults but also for faults characterized by a logical bridge between two data lines. Algorithm 2.1 accepts bridging faults between logically adjacent lines, and Algorithm 2.2 extends the model to include a bridging fault between any two lines. The information necessary to locate and correct a fault comes from the use of time redundancy. Additional cycles are supplied when an error is detected; thus, the penalty is incurred only when faults are detected.

To accurately represent the protocols studied in this thesis, a state machine representation of the module interaction was utilized. This also allowed the introduction of errors into the system. In order to use general purpose computers to aid in the analysis of these protocol machines, a State

Machine Language was developed. This language allows complete specification of the protocol modules. Files consisting of SML descriptions of these modules can be utilized in a system simulator which will exercise the action of the entire protocol system, including the introduction of errors to check out the fault tolerant functions.

The use of state machine models, with the protocol simulation system, permits an accurate study of the control interaction used in bus systems. We have shown that the steps taken to detect out-of-bound addresses for bus systems will also detect some of the signal faults which occur. By the addition of states judiciously placed in the state machines of the protocols, the use of timeouts can also detect faults which occur on control signal lines and prevent incorrect operation. But the use of time checking alone cannot permit continued operation in the presence of a fault.

We have shown that by the introduction of dual-rail signals for most of the control lines single stuck-at faults can not only be detected and located, but that the correct operation of the system can continue. Knowledge of correct operation of the protocol allows the module to anticipate the correct level of a control signal. If one of the two lines representing that signal is not at the correct level, then it is in error, and the module is then able to recognize the error and continue to function. We have presented three algorithms which allow conversion from a system using single-rail control signals to a dual-rail system.

Implementation of the techniques presented in this thesis is not without cost. The algorithms presented in Chapter 2 require additional time when an

error is detected: Algorithm 2.1 uses one additional transfer and Algorithm 2.2 uses either one or two additional transfers, depending on the type of fault detected. However, in both cases the additional cycles occur only when faults have been detected, so there is no constant system overhead. The algorithms for dual-rail control signal lines presented in Chapter 5 require doubling the number of control lines and increased complexity in the protocol definition. The increased complexity of the protocol will increase the total system hardware only about 20%.

8.2. Suggestions for Further Work

The use of SML to describe the modules of the protocols enabled us to solicit the aid of the computer in analyzing the interaction. The simulator used in this analysis is capable of providing a great deal of variety in the exercise of the protocols. However, when a range of values will satisfy a particular system call, the actual value returned is produced with the use of a random number generator, so the action of the system is random. In order to traverse the entire decision tree associated with the protocol, all conditions, from using the minimum for each decision to using the maximum, must be exercised. In order to accomplish this the simulator would have to be expanded to traverse that tree and try all combinations.

The State Machine Language is not limited to use with protocol analysis, but it can be used to represent state machine models of systems of independent modules. The simulator presented in this thesis can then be used to model the interactions of other types of systems as well. However, extensions would have to be made to allow the system to detect other types of faults than the ones needed for the protocol analysis.

This work did not address the issue of making a finite state machine tolerant to faults. The fault-tolerance achieved in this work is that of the signals between independent finite state machine. Very little work has been done on a fault-tolerant, finite state machine [43,50]. Similarly, very little work exists in the design of fault-tolerant arbiters. These two aspects must be further investigated in order to achieve an overall fault-tolerant system.

APPENDIX A

READ CYCLE INCORPORATING ALGORITHM 2.1

A simple read cycle is introduced in Section 3.2, and Section 3.3 deals with the additions needed to implement the controls required for Algorithm 2.1. The state machines representing these protocols are given in Figure 3.1 and Figure 3.2, and an explanation of the simple read protocol is contained in Section 3.2. The state machine for implementation of Algorithm 2.1 is duplicated here for reference in Figure A.1. This appendix contains a detailed description of the states involved in this protocol.

Algorithm 2.1 uses time redundancy to correct errors which occur while transferring synchronous information. For most bus systems this synchronous information consists of two transfers: data and address. The address may also contain information which indicates the type of transfer, but the timing requirements are the same as for the address itself, so these lines are grouped together. Thus the read cycle needs to apply Algorithm 2.1 to the address from the master to the slave as well as the data from the slave to the master.

Two more control lines are needed for this transfer over the read cycle of Figure 3.1. The condition that a parity error has been detected on the address lines is represented by 'errinadr', and upon detection of this condition a slave asserts the control signal 'adrpe'. Likewise, the condition that a parity error has been detected on the data lines is represented by 'errindat', and master lets the slave know this by asserting 'datape'. These lines

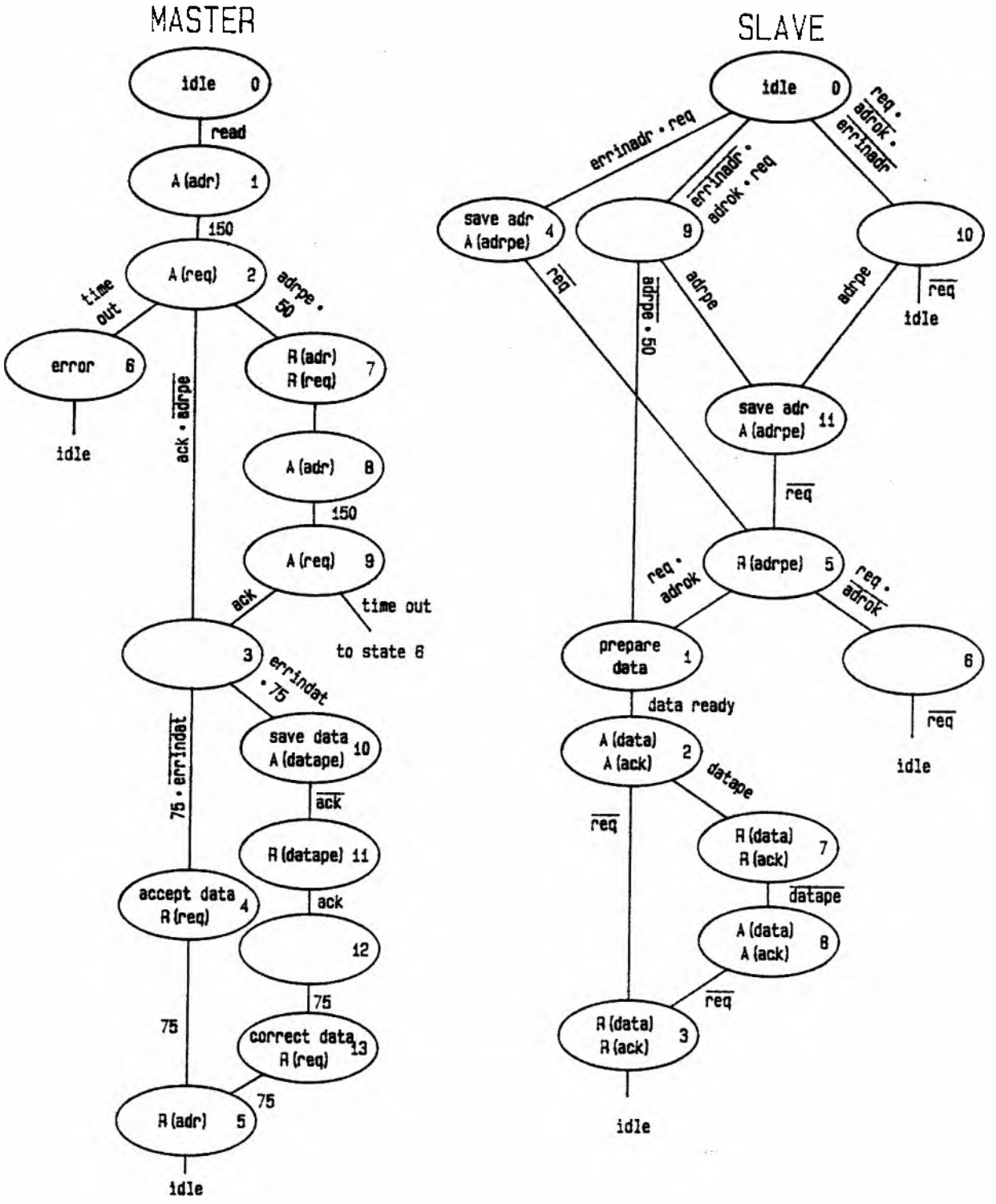


Figure A.1. Read Cycle Modified to Utilize Algorithm 2.1.

allow the different modules involved with the protocol to carry out Algorithm 2.1. In the following description, where the states are essentially identical to their counterparts in the simpler read cycle contained in Section 3.2, the comments are minimal.

Master State Machine - Modified Read Cycle

State 0: Idle state. Master exits when 'read' becomes true.

State 1: Assert address. Synchronous information is asserted. Move on to state 2 after delay.

State 2: Assert request. This state differs from state 2 of the original read cycle only in its exit criteria. When the master detects the assertion of the address parity error signal 'adrpe', it waits long enough to be sure that all the slaves have also detected the assertion of the signal and proceeds to state 7. If no 'adrpe' signal is detected, then normal operation will proceed to state 3 upon detection of 'ack', as before. If too much time passes, then state 6 is the error exit.

State 3: Delay state. Again, this state differs from state 3 of the original read cycle only in its mode of exit. If there is a parity error on the data which was received, then after the 75 nsec delay the master moves to state 10. Otherwise, normal operation will take the master to state 4.

State 4: Accept data. The data is accepted from the bus lines and the 'req' signal is released.

State 5: Release address. The synchronous lines are released and the master returns to the idle state.

State 6: Error state. If no 'ack' signal is received before the time-out

period passes, the slave is delinquent and this state permits continued operation of the bus.

State 7: Release address, request. If a slave requests a retransmission of address, this state is entered. The synchronous information asserted in state 1 is released, and the 'req' signal is released also. The master then moves on to state 8.

State 8: Assert address for retransmission. The modification to the address specified by Algorithm 2.1 is performed, and the modified address value is asserted. After the appropriate time to allow for skew and propagation delay, the master moves to state 9.

State 9: Assert request. Like state 2, this state is used to assert the 'req' signal and wait for a response from the slave. When the 'ack' response arrives, then the master moves on to state 3. If the address was for a non-existent slave unit, then no 'ack' signal will arrive and the master will move to state 6 after 'time-out'.

State 10: Assert data parity error. When the master unit has detected a parity error on the data lines, this state is entered from state 3, and the master asserts the signal 'datape' to ask the slave for a retransmission of the value. At the same time it stores the erroneous data so that Algorithm 2.1 can make use of it later. When the slave has detected the presense of 'datape', it responds by releasing the data and 'ack'. When the master detects this, it will proceed to state 11.

State 11: Release data parity error. When the master unit has arrived in this state, it releases 'datape' and waits for a new 'ack' signal to indicate that the retransmitted data is available. When this condition

is detected, it moves on to state 12.

State 12: Delay state. This state forces the delay needed to account for signal skew on the data lines. When the skew time has passed, control moves on to state 13.

State 13: Accept corrected data, release request. The correct data value will be available by combining the valued stored in state 10 and the value currently on the data lines as indicated by Algorithm 2.1. The resulting data are accepted by the master. The 'req' signal is released to let the slave know of the acceptance of the data, and the master moves on to state 5.

Slave State Diagram - Modified Read Algorithm

State 0: Idle state. Each slave will be idle until a 'req' signal is detected, and then the conditions it detects will determine how it proceeds in the state diagram. One of the two conditions detected is the fact that the address is free of parity errors and matches the slave's assigned address space, indicated by 'adrok'. The other condition is that a parity error has been detected by the slave, indicated by 'errinadr'. If 'adrok' is true and no parity error has been detected, then the slave will move to state 9 when 'req' is detected. If 'errinadr' is true, then upon detection of 'req' the slave will move to state 4. However, if neither of these conditions is true, then the slave will move on to state 10 with the detection of 'req'.

State 1: Prepare data. When the slave reaches state 1, it waits for the functional part of the unit to prepare the data, then it moves on to

state 2.

State 2: Assert the data and 'ack'. The data is asserted onto the bus lines and the 'ack' control signal is asserted. There are two possible successor states; in normal operation the 'req' signal will be released and the slave will move on to state 3. However, if the master has detected a parity error, then 'datape' will be asserted, and when the slave detects this signal it will move on to state 7.

State 3: Release data and 'ack'. The slave will release the data lines and the 'ack' control signal before returning to the idle state.

State 4: Address parity error detected. When the address lines contain a parity error when 'req' is detected, then this state is entered. The effect is to save the value on the address lines for use in Algorithm 2.1 and to assert 'adrpe'. The assertion of this control signal lets the master and all other slaves know that the error has been detected. When 'req' is released, then the slave moves on to state 5.

State 5: Release 'adrpe'. This state is entered when the 'req' signal has been released. This lets the slave know that the 'adrpe' signal has been detected and can be released. When the 'req' signal is asserted again, the slave will proceed to one of two states, depending on the value of the address constructed according to Algorithm 2.1. If the address agrees with the slaves assigned address space, then state 1 is entered; otherwise state 6 is entered.

State 6: Wait state. This state is entered when a transaction occurs which does not involve this slave, but two transfers of the address were required to establish this fact. When the 'req' signal is released, then

the slave proceeds to the idle state.

State 7: Release the data and 'ack'. The slave enters this state when it detects the 'datape' signal, which indicates that the master needs a retransmission of the data to correct an error which has been detected. The slave releases the data lines and the 'ack' control line and moves to state 8 when the 'datape' line has been released. The release of that line synchronizes the actions of the slave and the master.

State 8: Assert data, re-assert 'ack'. Now that the data lines have been released, the slave can assert the data value modified according to Algorithm 2.1 for transmission on the data lines. It also asserts the 'ack' line to indicate to the master that new value is on the data lines. When the release of the 'req' signal is detected, then the slave moves on to state 3.

State 9: Correct address detected. When the 'req' signal has been received by the slave while in the idle state, the address on the address lines falls within the assigned address space of the slave, and no parity error has been detected, this state is entered. No action is taken, and the choice of successor states is made on the basis of the 'adrpe' line. If the 'adrpe' line has not been asserted within 50 nsec, then the slave moves on to state 1 and normal operation. If, however, the 'adrpe' line is asserted, then some other slave unit detected a parity error and the slave moves to state 12 to save the address and wait for the redundant address value.

State 10: Address not in address space. If the address on the bus does not have a parity error and is not in the assigned address space of the

slave, then when the 'req' signal is detected the slave moves to this state. If the 'adrpe' signal is asserted then another slave has detected a parity error and the slave moves to state 12. If no 'adrpe' signal is received, then the slave returns to the idle state when the 'req' signal is released. This decision synchronizes the action of all of the slave modules so that each slave knows if the value on the address lines is a valid address or an address formed by application of Algorithm 2.1 and transmitted a second time.

State 11: Save address, assert 'adrpe'. When this state is entered, another slave has detected a parity error on the address lines. This slave also asserts the 'adrpe' line and saves the address so that Algorithm 2.1 can be applied to correct the fault. When the slave detects the release of 'req', it moves on to state 5 to decide if the address matches its assigned address space.

APPENDIX B

BNF DESCRIPTION OF STATE MACHINE LANGUAGE

Chapter 4 introduces the State Machine Language SML and its application to represent state machines involved in a protocol system. We now give a complete Backus Naur Form (BNF) representation of the grammar. In the production rules which follow, the '::=' symbol calls for the replacement of the non-terminal on the left by the terminals and non-terminals on the right. Non-terminals appear below marked with '<...>' characters, while these symbols are absent from terminals. The '|' symbol represents that the 'OR' function, indicating that one of the entries may be selected. A symbol included in double quotes must be accepted literally as it appears. All other symbols appearing in the production rules are terminals and must remain in the substitution. Also included in this appendix are descriptions of the operators of the language and the naming conventions used.

BNF Description of SML

<machine> ::= <name> <arg> <I> <S> <O> <transitions> <outputs>;

<name> ::= <idname> <identifier>;

<idname> ::= smname | hostname | arbname

<identifier> ::= <letter><char><char><char><char><char><char><char>

<letter> ::= <ucletter> | <lcletter>

<ucletter> ::= A | B | C | ... | Z

<lcletter> ::= a | b | c | ... | z

<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<char> ::= <ucletter> | <lcletter> | <number> | <empty>

<empty> ::=

<arg> ::= <empty> | <arg> define <identifier> = <integer>

<value> ::= <integer> | mkadr(<min_adr>, <max_adr>) | mkdata(<max_value>)

<integer> ::= <number> | <number> <integer>

<min_adr> ::= <integer>

<max_adr> ::= <integer>

<max_value> ::= <integer>

<I> ::= <inp> <signal_name_list> ;

<inp> ::= ginputs | linputs | inputs

<signal_name_list> ::= <signal_name> | <signal_name> <, signal_name_list>

<signal_name> ::= <identifier>

<S> ::= states <integer> ;

<O> ::= <out> <signal_name_list> ;

<out> ::= goutputs | loutputs | outputs

<transitions> ::= <transition spec> ;

<transition spec> ::= <transition> | <transition> <transition spec>

```
<transition> ::= tran <state_number> -> <state_number> : <condition> ; |
                 tran <state_number> -> <state_number> ;

<state_number> ::= <integer>

<condition> ::= <logic_on_inputs> | <DLY>
                | <logic_on_inputs> <logical operator> <DLY>

<logic_on_inputs> ::= <value> | <expression> | ( <expression> )

<expression> ::= <signal_name> = <value> | <signal_name> |
                 <expression> <logical operator> <expression>

<logical operator> ::=  == | != | <= | >= | "||" | && | > | <

<DLY> ::=  delay(<minimum_wait_time>, <maximum_wait_time>)  |
          acc_delay(<maximum_accumulated_time_before_forcing_state_transition>)

<minimum_wait_time> ::= <integer>

<maximum_wait_time> ::= <integer>

<maximum_accumulated_time_before_forcing_state_transition> ::= <integer>

<outputs> ::= <output spec> ;

<output spec> ::= <output_def> | <output_def> <output spec>

<output_def> ::= <assert_def> | <release_def> | <do_def>

<assert_def> ::= assert <signal_name> in <state_number> ; |
                 assert <signal_name> = <value> in <state_number> ; |
                 assert_oc <signal_name> in <state_number> ; |
                 assert_oc <signal_name> = <value> in <state_number> ;
```

<release_def> ::= release <signal_name> in <state_number> ; |
release_oc <signal_name> in <state_number> ;

<do_def> ::= do <signal_name> in <state_number> ; |
do <signal_name> = <value> in <state_number> ; |
do_oc <signal_name> in <state_number> ; |
do_oc <signal_name> = <value> in <state_number> ;

State Machine Names: Three types of state machines are identified by names: smname, hostname, and arbname. The 'smname' designation is for the state machine representing the protocol machine itself. This is different from the functional unit portion of the system, which is represented by a state machine designated by 'hostname', and if an arbitration module is associated with the module that is represented by a state machine identified by 'arbname'. As for the names themselves there are no restrictions except as imposed by the operating system. The reason for this restriction is that the name called out in the '<name>' portion of the grammar is used as a root name for files created by the protocol exercise system. Therefore, the names should not be too long, and no two state machines should have the same name.

The define statement: The identifier of a 'define' statement will identify a constant which should be set to a value. The restrictions on the identifier are those associated with naming of variables in the C language. The value should be an integer.

Values: Values in the protocol system are integers. Where an integer is known then the integer itself can be used. Where an integer can assume a value between two different addresses, then the 'mkadr' function can be used.

The name 'mkadr' represents 'make an address'. There are two calling parameters to the function: <min_adr> which represents the minimum legal address, and <max_adr> which identifies the upper limit of address range. The number generated by this function is an integer somewhere between the two limits. The function 'mkdata' also allows the specification of a value which is a random number. The calling parameter to 'mkdata' is a single integer which is the maximum allowable value that the function can assume. The value which is returned will be a positive integer less than the calling parameter.

Inputs: There are two types of inputs: global inputs and local inputs. Global inputs are signals which originate from the bus, and as such extend between protocol modules. Local inputs are signals which are used between the units which comprise a module, the protocol machine, the arbitration machine, and the functional unit. For the arbitration and protocol modules the 'ginputs' designation is required to identify the global inputs, and the 'linputs' name identifies the signals which are local. Since the functional unit portion deals only with local signals it can use the 'inputs' designation, which defaults to local inputs. The names used for inputs and outputs can carry information concerning their expected values, as explained below.

Signal Names: A convention which has been adopted concerning the names of the signals concerns the terminal character of the signal. It is not imperative that names conform to this convention; the system can operate with signals which do not ascribe to the rules presented here. A name should end in one of the letters 'h','l','b', or 'p'. A signal whose name ends with 'h' is a single line considered to be active in the high state, and a name which ends

with 'l' is a single line considered to be active in the low state. A name which ends in a 'b' is considered to be a bus value, such as a data bus or an address bus, which does not have parity error detection associated with it. A name which ends in a 'p' is considered to be a bus which does have protection of a parity bit. When space is allocated on the output lines for data values more room is left for the bus values than for the signal values, so following the convention aids the format of the output data.

States: States in the state machine system are identified by integers, and so the set of states comprising the state machine can be identified by giving the highest number of a state. By convention, states are numbered consecutively from 0, so a state machine contains n+1 states, where n is called out in the 'states' statement. When a state is identified in a transition statement or an output statement, it is in the set of states of the machine when the state in question has a number less than or equal to the number called out by the 'state statement'.

Outputs: Like the inputs, the outputs are grouped into global and local signals. The global signals are used to communicate between modules on the bus, and local names are used within functional units. For the arbitration and protocol machines the 'goutputs' statement is require to identify the outputs which are global, and 'loutputs' identifies the signals which are used within the module. For the state machine of the functional unit the 'outputs' designation is allowed, since all outputs of the functional portion are local to the module.

Transitions: Transitions are identified by giving the initial state of the transition, the final state, and the conditions under which the transition will be made. The condition portion of the statement is optional; if it is not present then the transition to the next state is made one state time later than entry into the state. The conditions involved in the transition statements can consist of logic on the input values, delays, or a combination of logic and delays. The logic involved is comparing the signals to known values or to values established by the define statements. For example, the logical statement 'testh != 1' will be true when the signal 'testh' is not asserted. The logical operators available are those involved in the C language. The delays involved are represented by two functions: 'delay' and 'acc_delay'. The 'delay' function accepts two parameters, both of which are integers. The first establishes a minimum time, and the second identifies a maximum time. The actual delay time, which will be a random number between the two values, must transpire before control can pass to another state. That is, this time must occur before the logical expression containing the delay can be true. The 'acc_delay' statement has one parameter which establishes the time after which the function becomes true. There is no random nature involved with 'acc_delay'.

Controlling the outputs: There are three types of statements which deal with controlling the output signals. The 'assert_def' statements are used to specify the states in which a signal is asserted. Signals asserted in this way will need to be released in a later state with a 'release_def' type of statement. The assertion statement must identify the signal to be asserted, the value which it will be given, and the state number in which this action

occurs. If the naming conventions outlined above have been followed and single signal line names end in either an 'h' or an 'l', then the value portion of the statement is optional. If it is not present, then the value indicated by the name is assumed for the assertion value. Signals which are open collector signals are asserted by 'assert_oc' statements; 'assert' statements are used for all other signals. A signal which has been asserted in one state must be released in a later state, and this is done by the appropriate 'release_def' statement. Open collector signals are released by using the 'release_oc' statement. All other signals are released with the 'release' statement. When a signal is to be asserted in one state and released when that state is exited, regardless of the next state, then it is possible to use the 'do_def' type of statement. The same conventions about values and open collector restrictions apply to the 'do' statements as for the 'assert' statements. Signals identified by the 'do_def' statements are true only for the duration of the state identified by the statement.

APPENDIX C

SML DESCRIPTION OF UNIBUS MASTER

SML can be used to describe arbitrarily complex state machines to enable the simulation system introduced in Chapter 4 to exercise a system composed of multiple state machines. The UNIBUS Master from Chapter 7 is shown in Figure C.1, with the signal names given below:

abusb - the address bus; no parity protection

addressb - the address from the functional section of the module indicating the target address of the transaction

bbsy - bus busy - asserted by master while using bus lines

bg - bus grant - asserted by arbiter when control of bus can be obtained

br - bus request - asserted by a master when it needs control of bus

c1,c0 - the control lines - same timing as the address lines and used to indicate the type of transfer

dbusp - the data lines, including parity

msyn - master sync - asserted by master to communicate with slave

readh, read_pauseh, writeh, write_byteh - local signals from the functional portion of the module to start a transaction

read_datab - path to functional portion of module for passing data from a slave

sack - selection acknowledge - asserted by master to acknowledge that it will assume control of bus when bbsy goes false

ssyn - slave sync - asserted by slave to communicate with master

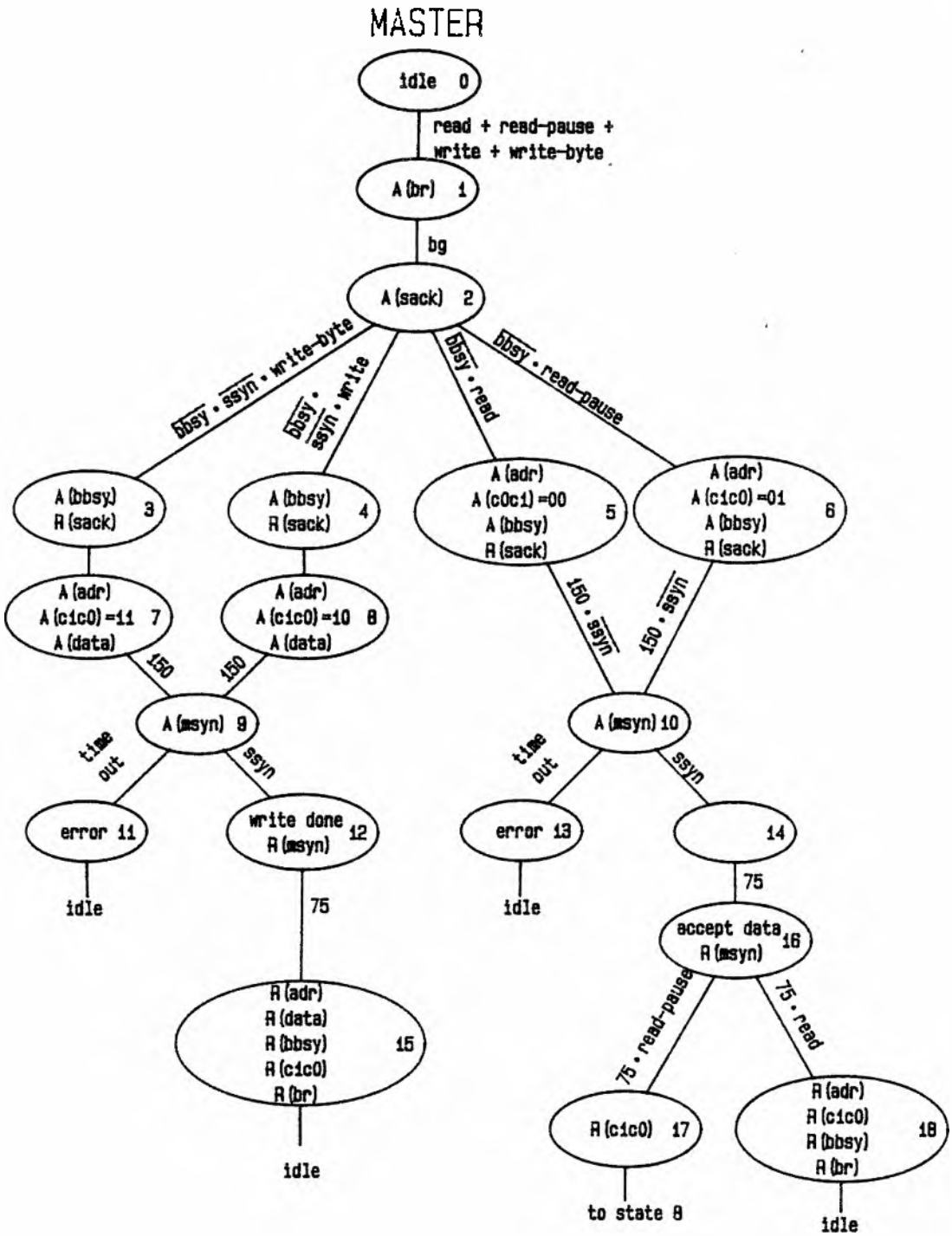


Figure C.1. State Machine of the UNIBUS Master.

write_datab - path from functional portion of module carrying data to be sent to slave

write_doneh, read_doneh - local signals to the functional portion of the module to indicate completion of a transaction

Note that the states 12 and 13 now contain the signals needed to be released to accomodate the error conditions of the states.

The text of the file describing the state machine is as follows:

```
smname master;          # master of UNIBUS
ginputs dbusp, ssyn, bg, bbsy;
linputs writeh, readh, read_pauseh, write_byteh, addressb, write_datab;
states 17;
goutputs dbusp, msyn, c0, c1, abusb, br;
loutputs write_doneh, read_doneh, read_datab;
                        # beginning of next state section
tran    0 -> 1 : readh == 1 | read_pauseh == 1 | writeh == 1 | write_byteh == 1;

tran    1 -> 2 : bg == 1;

tran    2 -> 3 : bbsy != 1 && ssyn != 1 && write_byteh == 1;
tran    2 -> 4 : bbsy != 1 && ssyn != 1 && write == 1;
tran    2 -> 5 : bbsy != 1 && read == 1;
tran    2 -> 6 : bbsy != 1 && read_pause == 1;

tran    3 -> 7 ;

tran    4 -> 8 ;

tran    5 -> 10 : delay(150,150) ;

tran    6 -> 10 : delay(150,150) ;

tran    7 -> 9 : delay(150,150) ;

tran    8 -> 9 : delay(150,150) ;

tran    9 -> 11 : ssyn == 1 ;
tran    9 -> 12 : acc_delay(20000);

tran    10 -> 13 : acc_delay(20000) ;
tran    10 -> 14 : ssyn == 1 ;
```

```
tran    11 -> 15 : delay(75,75) ;

tran    12 -> 0 ;

tran    13 -> 0 ;

tran    14 -> 16 ; delay(75,75) ;

tran    15 -> 0 ;

tran    16 -> 17 : delay(75,75) && read_pause == 1 ;
tran    16 -> 18 : delay(75,75) && readh == 1 ;

tran    17 -> 8 ;

tran    18 -> 0 ;

;                       # beginning of output section

assert br = 1 in 1 ;
assert sack = 1 in 2 ;
assert bbsy = 1 in 3;
release sack in 3;
assert bbsy = 1 in 4;
release sack in 4;
assert abusb = addressb in 5;
assert c1h = 0 in 5;
assert c0h = 0 in 5;
assert bbsy = 1 in 5;
release sack in 5;
assert abusb = addressb in 6;
assert c1h = 0 in 6;
assert c0h = 1 in 6;
assert bbsy = 1 in 6;
release sack in 6;
assert abusb = addressb in 7;
assert c1h = 1 in 7;
assert c0h = 1 in 7;
assert datab = write_datab in 7;
assert abusb = addressb in 8;
assert c1h = 1 in 8;
assert c0h = 0 in 8;
assert datab = write_datab in 8;
assert msyn in 9;
assert msyn in 10;
release msyn in 11;
do write_doneh = 1 in 11;
release msyn in 12;
release c0 in 12;
```

```
release c1 in 12;
release abusb in 12;
release bbsy in 12;
release sack in 12;
release datap in 12;
release br in 12;
release msyn in 13;
release c0 in 13;
release c1 in 13;
release abusb in 13;
release bbsy in 13;
release sack in 13;
release br in 13;
release abusb in 15;
release c0 in 15;
release c1 in 15;
release bbsy in 15;
release br in 15;
do read_datab = datap in 16;
do read_doneh = 1 in 16;
release msyn in 16;
release c0 in 17;
release c1 in 17;
release abusp in 18;
release c0 in 18;
release c1 in 18;
release bbsy in 18;
release bg in 18;
;
```

APPENDIX D

USE OF PROTOCOL EXERCISE SYSTEM

The protocol exercise system explained in Chapter 4 allows a computer to simulate the action of a system described as a number of state machines. There are several steps involved in the process of building a computer program to simulate the system, and an overview of this process is provided by Figure D.1. As shown in the figure, the state machine describing each module is represented in SML, and these descriptions form the input to a lexical analyzer called 'mfsm'. This analyzer converts file containing the SML descriptions into several files, one of which is a C language routine describing the action of the module. When this conversion has been performed on each module, three other programs are invoked to create the system. The first program is called 'makedoup', and it is used to create a C language routine which monitors the states and variables of interest and outputs the information when called for by the execution of the system. The second program is called 'makenames', and it is used to create lists of global and local names for the system being simulated. The third program is called 'makeit', and it is responsible for taking the information created by the other programs and generating a file containing a valid C program which can be compiled with the standard C compiler. In addition to these routines there is another utility called 'ch' which can be used to check the syntax of the SML file. These routines, their calling parameters, and the inputs and outputs expected by each are explained below.

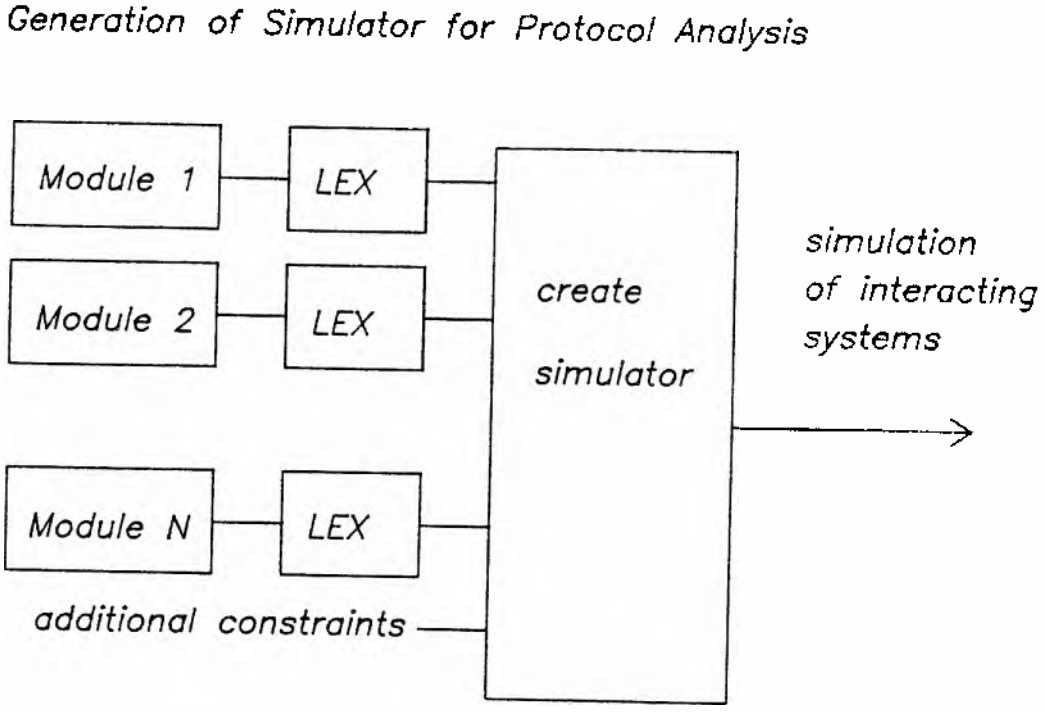


Figure D.1. Overview of Protocol Exercise System.

mfsm: The program 'mfsm' converts SML descriptions of state machines into a number of output files. The input is provided via the standard input facility of UNIX. The output appears in several files, the names of which are derived from the name of the state machine specified in the opening statement of the SML description. This name is used as the '<state machine name>' in the following description of files generated by 'mfsm'.

<state machine name>.b : This file contains the largest delay called for in the state machine. It allows the system to set a limit when checking for deadlock conditions.

<state machine name>.c : This file contains a C language routine which represents the state machine defined by the SML description.

<state machine name>.d : This file contains the name of the C routine which is created to represent the module. This name is needed by the simulation system in order to initiate the action of the module.

<state machine name>.gi : This file contains the names of the global inputs called out in the 'ginputs' statement of the SML description. It is combined with the other global symbol names of other modules to create a list of the global signals. This list defines the lines on the bus in question.

<state machine name>.go : This file contains the names of the global outputs specified by the 'goutputs' statement of an SML description. It is combined with other global symbol lists to define the names of the lines of the bus being simulated.

<state machine name>.li : This file contains the names of the local inputs called for by the SML description. The signals specified here are not

available to other state machines in the system, but they can be identified for output or error specification if needed.

<state machine name>.lo : This file contains the names of the local outputs identified in the SML description. The signals specified here are not available to other state machines in the system, but they can be identified for output or error specification if needed.

<state machine name>.n : This file contains the names containing the state variables of the state machine. That is, the variables named in this file are the present state register and the next state register in the representation of the action of the module. The integer in the present state register identifies the current status of the state machine.

<state machine name>.p : This file identifies the name by which the process can be referred to for output statements.

makedoup: The routine 'makedoup' is used to create a C language routine which updates temporary variables to keep track of the action of the system. The variables which are of interest to the system are identified in the calling parameters. Those names which identify state machine names which have been processed by the 'mfsm' routine result in the output of the state of that module on an output line. Names which do not coincide with names of state machines are assumed to be names of auxiliary variables identified by the operator as of interest, and the signals so named also appear in the output. In addition to this information specified by the calling parameters, the time of the simulation system clock and the global variables are also printed out. The C routine which results from the execution of 'makedoup' is sent to the standard output facility of UNIX and redirected to the file 'do_an_upda.c'.

makenames: The routine 'makenames' is used to create a list of the global and local names used by the system. The calling parameters to 'makenames' identify the state machines which will jointly make up the system, and the routine collects the signal names from the files created by 'mfsm'. These names are placed into the appropriate files and the duplicate entries removed. The files are then ready for 'makeit' to create the system as a whole.

makeit: The simulation system as a whole is created by the routine 'makeit'. The calling parameters to 'makeit' identify the names of the state machines which will make up the system. The files created by 'mfsm', 'makedoup', and 'makenames' are all combined into one file which represents the entire system. This file is then input to the C compiler to create an executable version of the system. Other versions of the system using fewer or more modules can be created by adjusting the names used as calling parameters to the above routines.

Invoking the action is accomplished in the same manner as starting any other executable program on the system. However, the action of the simulation system can be modified by the calling parameters with which it is started. These parameters are as follows:

-d name n The -d option is used to identify a drive signal. The signal identified by 'name' is monitored, and after 'n' cycles of the signal the action of the system is terminated.

-f name or -F name The -f option is identifies the signal 'name' as a signal which will be susceptible to stuck-at faults. The inclusion of a second signal for stuck-at faults is allowed by the -F option. This allows

specification of a fault on the data path and a fault on the control lines.

-% n The probability of a signal sticking at an asserted value is 'n' percent.

-# n The probability of a signal which has been stuck-at a value being released is 'n' percent.

-p This flag inhibits the output information from being provided at state changes. In this way only error messages will be output.

-r Action of the system is often controlled by the action of a pseudo-random number generator. The seed of that number generator is initially the same from one run to the next, allowing duplication of a run. The '-r' flag causes the seed of the number generator to be based on the time, resulting in a random start for the random number generator.

-w name The signal 'name' is added to the information printed at every output time.

ch: The routine 'ch' provides a syntax checker for the SML files. The checker can be fooled into thinking that a description is valid when in reality it is not, but it does provide a vehicle for checking for naming errors and operator malfunction. The input to 'ch' is an SML description and the output is the same file with whatever error messages are identified by the syntax checker. The absence of error messages indicates that there is a fairly good probability that the syntax is correct.

Once a set of files has been established with the SML descriptions of the state machines of the protocol to be exercised, then a simulation system can

be created by using the commands listed above. The commands needed to build a system from four SML files named 'master', 'slave', 'master_host', and 'slave_host' are as follows:

```
mfsm < master
mfsm < master_host
mfsm < slave
mfsm < slave_host
makenames master master_h slave slave_h
makedoup master master_h slave slave_h > do_an_upda.c
makeit master master_h slave slave_h > whole.c
cc -o r1 -g whole.c /mnt/dln/pub/sim.new.a
```

Once the executable file has been created by the C compiler, then it can be invoked with the the appropriate options as shown below. This example uses the executable 'r1', created by the operations shown above, and the calling parameters specify only two cycles of the protocol. The output which appears after the command line is supplied by the protocol system, and identifies the drive signal, the number of cycles, and the type of printout. The numbers on each line are the time, the states of the four state machines (master, master_host, slave, and slave_host), and the global variables ('ack', 'adr', 'data', and 'req').

```
% r1 -d req 2
Drive signal - req
Number of cycles - 2
Print out state changes and globals
     1 -  0  1  0  0    -1    -1    -1    -1
   101 -  0  2  0  0    -1    -1    -1    -1
   102 -  0  3  0  0    -1    -1    -1    -1
   103 -  1  3  0  0    -1  1474    -1    -1
   253 -  2  3  0  0    -1  1474    -1     1
   255 -  2  3  1  0    -1  1474    -1     1
   257 -  2  3  1  1    -1  1474    -1     1
   657 -  2  3  1  2    -1  1474    -1     1
   658 -  2  3  2  2     1  1474  1599     1
```

660 -	3	3	2	0	1	1474	1599	1
735 -	4	3	2	0	1	1474	1599	-1
737 -	4	3	3	0	-1	1474	-1	-1
738 -	4	3	0	0	-1	1474	-1	-1
810 -	5	3	0	0	-1	-1	-1	-1
811 -	0	3	0	0	-1	-1	-1	-1
812 -	1	3	0	0	-1	1329	-1	-1
962 -	2	3	0	0	-1	1329	-1	1
964 -	2	3	1	0	-1	1329	-1	1
966 -	2	3	1	1	-1	1329	-1	1
1366 -	2	3	1	2	-1	1329	-1	1
1367 -	2	3	2	2	1	1329	4833	1
1369 -	3	3	2	0	1	1329	4833	1
1444 -	4	3	2	0	1	1329	4833	-1
1446 -	4	3	3	0	-1	1329	-1	-1
1447 -	4	3	0	0	-1	1329	-1	-1

Total time 1519

APPENDIX E

STATE MACHINES FOR PROTOCOLS UTILIZING ALGORITHM 2.1 AND DUAL-RAIL SIGNALS

The state machines presented in Chapter 5 use the generic read protocol to introduce the algroritms for conversion from single-rail control signals to dual-rail control signals. These algorithms can be applied to more complex protocols as well with similar results. This appendix contains the state machines derived from application of the algorithms of Chapter 5 to the read protocol which has been modified to utilize Algorithm 2.1. The state machines for this protocol appear as Figure 3.2.

In these figures the states which will be reached in the presence of single stuck-at faults are labeled with the appropriate fault. The format of the label is xx/f, where the signal name is xx and the value at which the signal is stuck is f. In addition, the correlation between the initial state machine and the new state machine is indicated by the state numbers of the original state machine which appear near the appropriate group of states in the new state machine.

Figure E.1 presents the application of Algorithm 5.1 to the master of the read protocol, and Figure E.2 presents the slave. The application of Algorithm 5.2 to the master and slave is shown in Figure E.3 and Figure E.4. The application of Algorithm 5.3 is shown in Figure E.5 and Figure E.6.

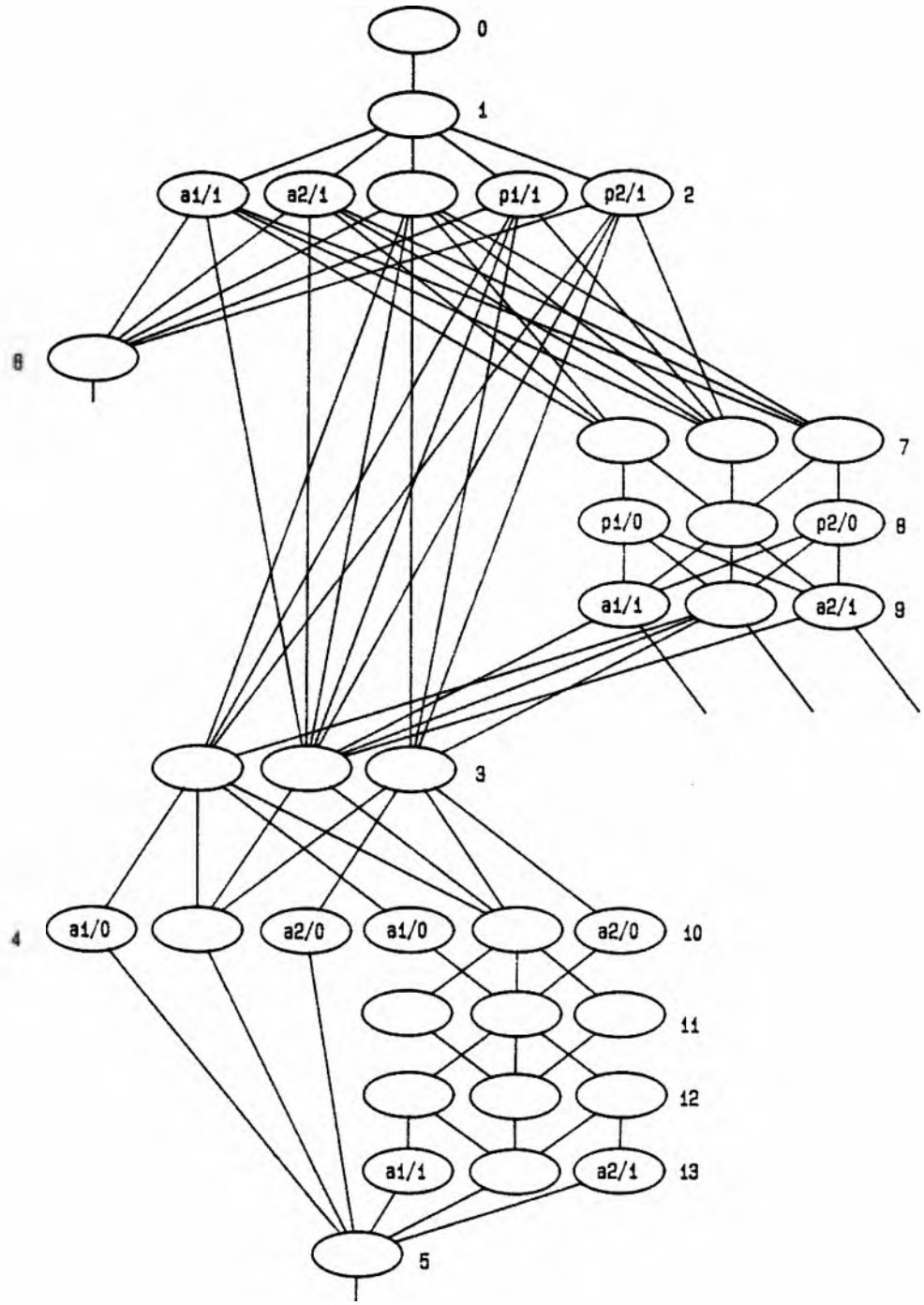


Figure E.1. Application of Algorithm 5.1 to Read Protocol Master.

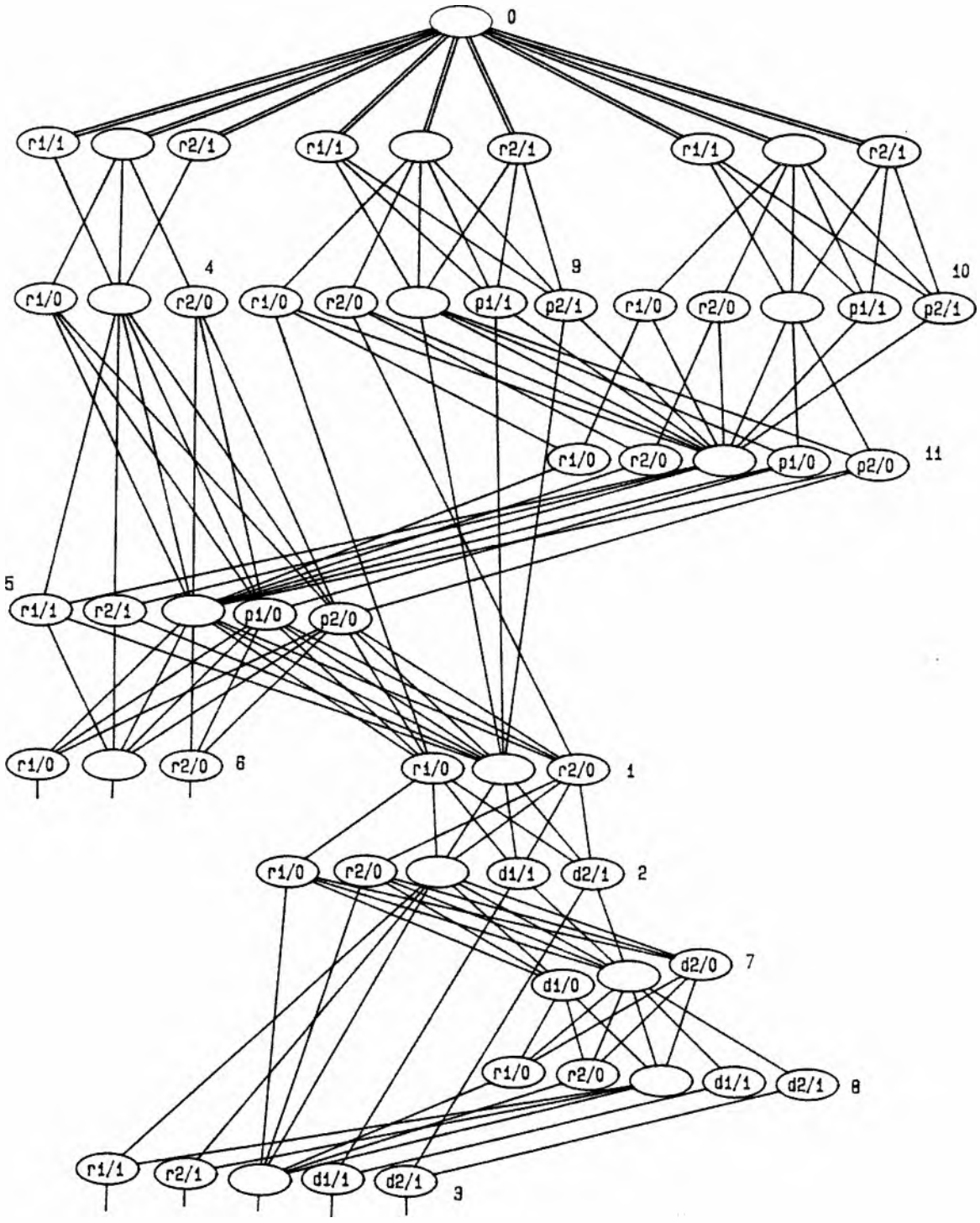


Figure E.2. Application of Algorithm 5.1 to Read Protocol Slave.

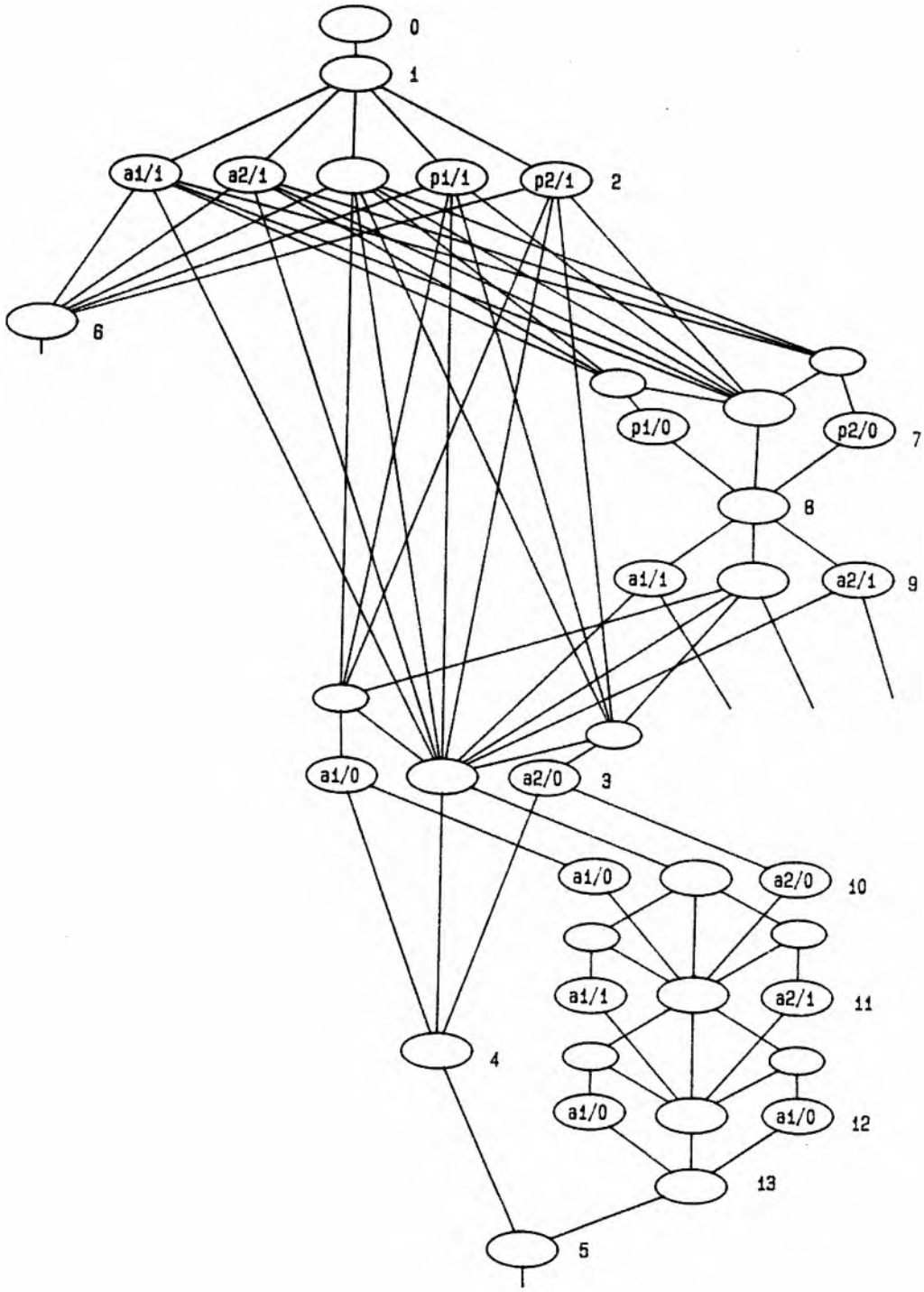


Figure E.3. Application of Algorithm 5.2 to Read Protocol Master.

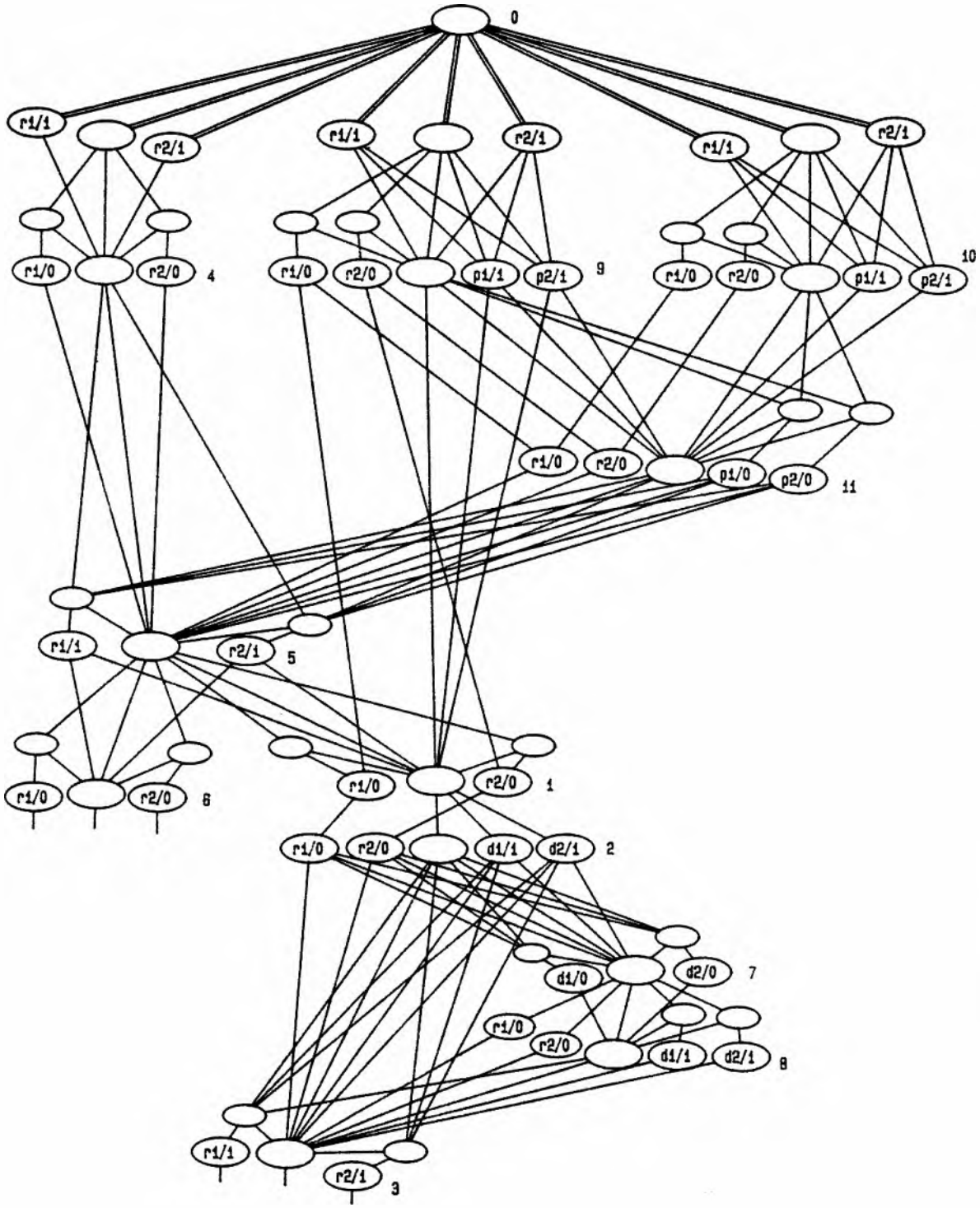


Figure E.4. Application of Algorithm 5.2 to Read Protocol Slave.

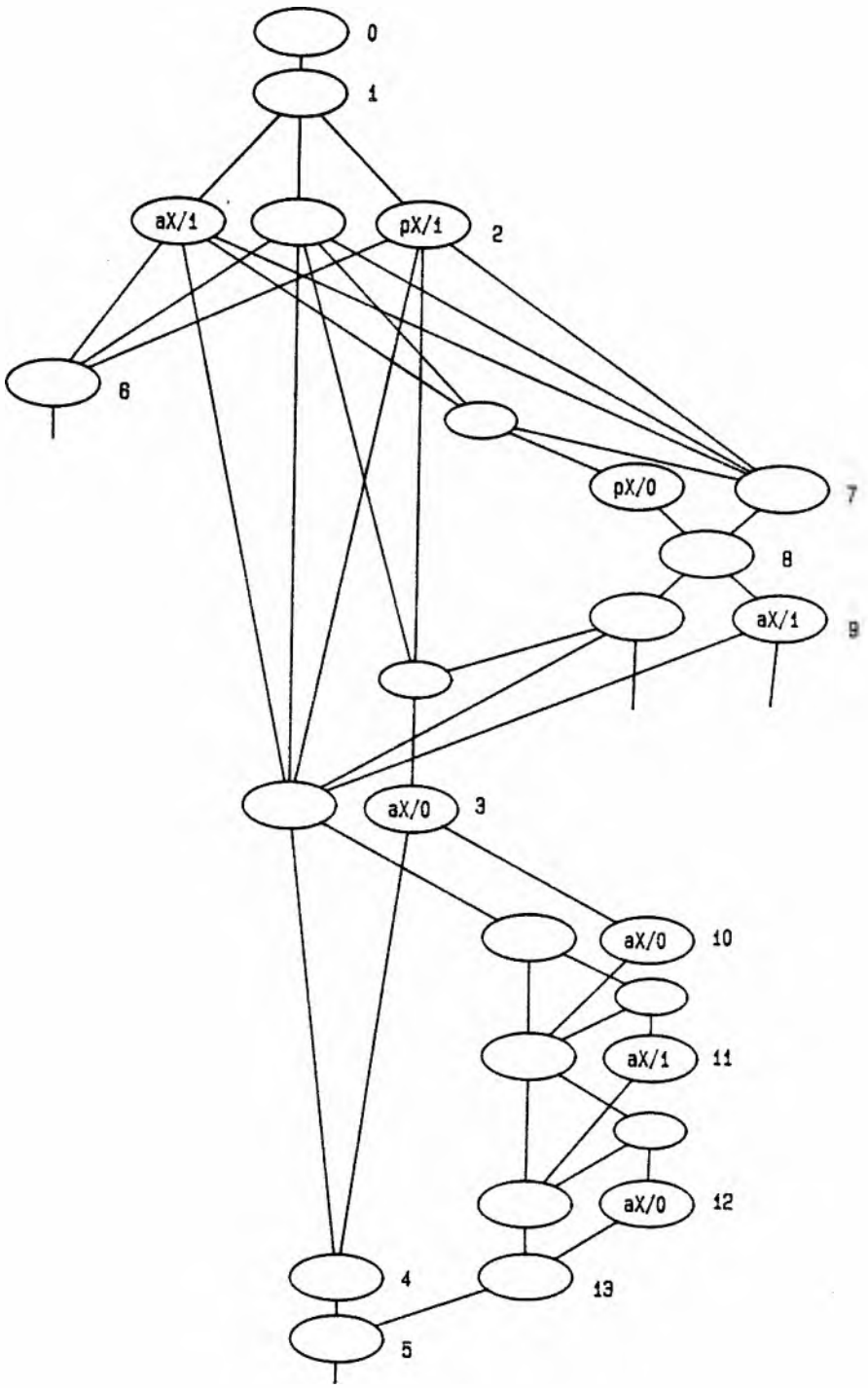


Figure E.5. Application of Algorithm 5.3 to Read Protocol Master.

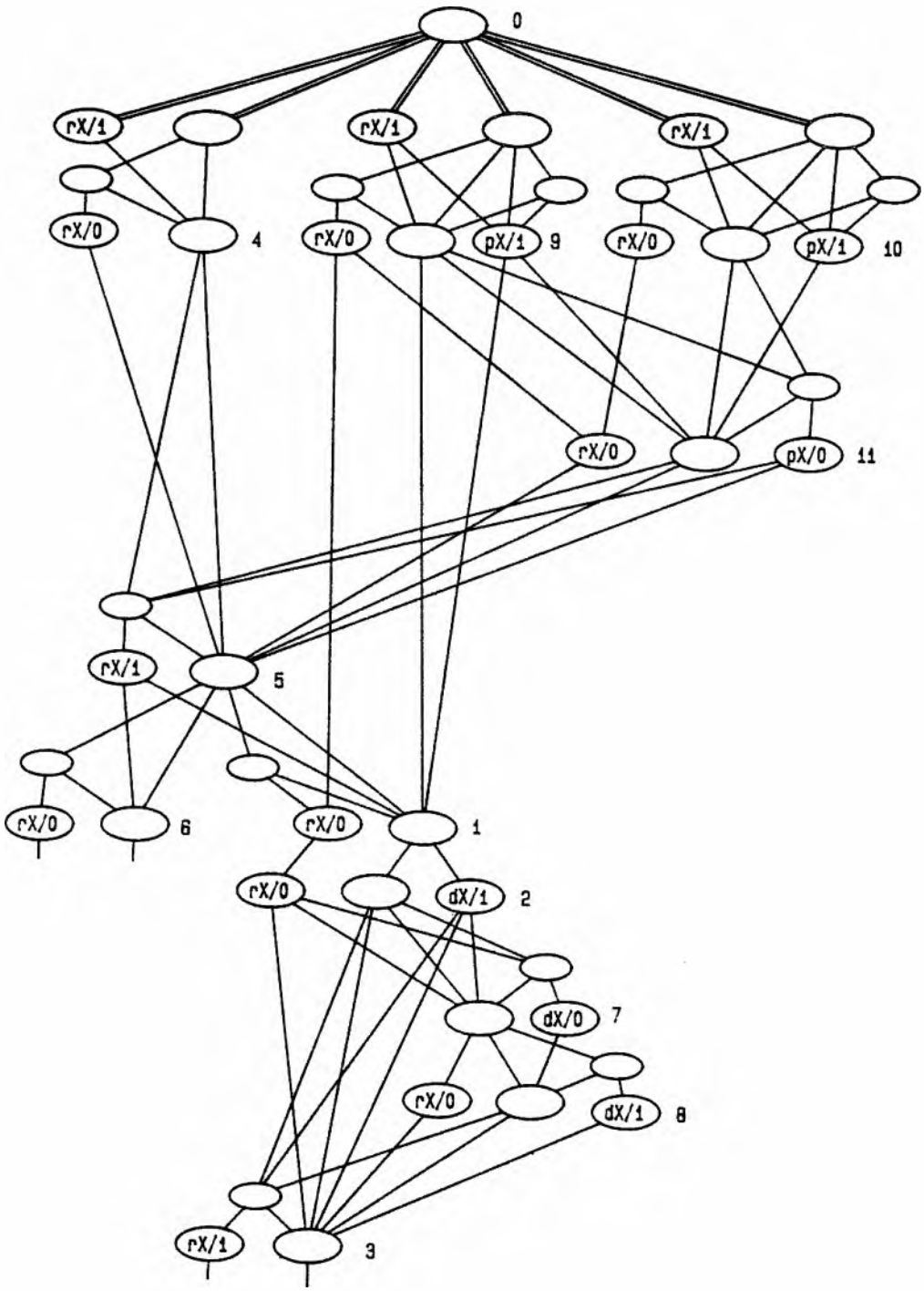


Figure E.6. Application of Algorithm 5.3 to Read Protocol Slave.

REFERENCES

1. M. Gien, "A file transfer protocol," Computer Networks, Vol. 2, pp. 312-319, Sept./Oct. 1978.

2. G Schulze and J. Borger, "A virtual terminal protocol based upon the 'communication variable' concept," Computer Networks, Vol. 2, pp. 291-296, Sept./Oct. 1978.

3. H. V. Bertine, "Physical level protocols," IEEE Transactions on Communications, Vol. COM-28, pp. 433-444, Apr. 1980.

4. J. J. Shedletsky, "Error correction by alternate data retry," IEEE Transactions on Computers, Vol. C-27, pp. 106-112, Feb. 1978.

5. D. P. Agrawal and V. K. Agarwal, "On-line bus fault diagnosis in microprocessor systems," J. Digital Syst., Vol. 4, pp. 337-391, Winter 1980.

6. J. B. Postel, "A graph-model analysis of computer communications protocols," Ph.D. dissertation, Comput. Sci. Dep., University of California, Los Angeles, CA, UCLA ENG-7410, 1974.

7. P. M. Merlin, "Specification and validation of protocols," IEEE Transactions on Communications, Vol. COM-27, pp. 1671-1680, Nov. 1979.

8. J. L. Peterson, Petri net theory and the modeling of systems, Prentice-Hall, Inc., Englewood Cliffs, NJ 1981.

9. G. V. Bochmann and J. Gecsei, "A unified model for the specification and verification of protocols," Proceedings, IFIP Congress, pp. 229-234, 1977.

10. A. S. Danthine, "Protocol representation with finite-state models," IEEE Transactions on Communications, Vol. COM-28, pp. 632-643, Apr. 1980.

11. H. K. Knudsen,, "Linked state machines," Los Alamos National Laboratory Report LA-9770-MS, June 1983.

12. C. A. Sunshine, D. H. Thompson, R. W. Erickson, S. L. Gerhart, and D. Schwabe, "Specification and verification of communication protocols in AFFIRM using state transition models," IEEE Transactions on Software Engineering, Vol. SE-8, pp. 460-489, Sept. 1982.

13. R. R. Razouk and G. Estrin, "Modeling and evaluation of communication protocols," in Computer Networks and Simulation II, ed. S. Schoemaker, North-Holland, pp. 167-190, 1982.

14. P. M. Merlin, "A methodology for the design and implementation of communication protocols," IEEE Transactions on Communications, Vol. COM-24, pp. 614-621, 1976.

15. C. H. West, "General technique for communications protocol validation," IBM J. Res. Develop., Vol. 22, pp. 393-404, July 1978.

16. D. Siewiorek, M. Canepa, and S. Clark, "C.vmp: the architecture and implementation of a fault tolerant multiprocessor," in 7th Annual International Conference on Fault-Tolerant Computing, Los Angeles, Calif., 28-30 June 1977, IEEEpp. 37-43, , New York 1977.

17. W. W. Knight, "Fault-tolerance with standard computer modules," Proceedings of the Distributed Data Acquisition, Computing, and Control Symposium - 1980, pp. 99-106, 1980.

18. T. Agerwala, "Some applications of Petri nets," Proceedings of the 1978 National Electronics Conference, Vol. 23, pp. 149-154, Oct. 1978.

19. P. Azema, R. Valette, and M. Diaz, "Petri nets as a common tool for design verification and hardware simulation," Proceedings 13th Design Automation Conference, pp. 109-116, June 1976.

20. J. Baer and C. Ellis, "Model, design and evaluation of a compiler for a parallel processing environment," IEEE Transactions on Software Engineering, Vol. SE-3, pp. 394-405, Nov. 1977.

21. R. Shapiro and H. Saint, "A new approach to optimization of sequencing decisions," Annual Review in Automatic Programming, Vol. 6, pp. 257-288, 1970.

22. K. Lautenbach and H. Schmid, "Use of Petri nets for proving correctness of concurrent process systems," Information Processing 74, Proceedings of the 1974 IFIP Congress, North-Holland, pp. 187-191, Aug. 1974.

23. F. Ramming, "Petri-net based description, analysis and simulation of concurrent processes," Proceedings 14th Design Automation Conference, IEEE, June 1977.

24. P. Thomas, "The Petri net: a modeling tool for the coordination of asynchronous processes," M.S. thesis, University of Tennesee, Knoxville, TN, June 1976.

25. L. Cox, Jr., "Predicting concurrent computer system performance using Petri net models," Proceedings of the 1978 ACM National Conference, ACM, pp. 901-913, Dec. 1978.

26. J. Meldman, "A Petri-net representation of civil procedure," IDEA - The Journal of Law and Technology, Vol. 19, pp. 123-148, 1978.

27. C. A. Sunshine, "Formal modeling of communication protocols," in Computer Networks and Simulation II, ed. S. Schoemaker, North-Holland, pp. 53-75, 1982.

28. P. M. Merlin, "A Study of the recoverability of computing systems," Ph.D. dissertation, University of California, Irvine, CA, 1974.

29. P. M. Merlin and D. J. Farber, "Recoverability of communication protocols--implications of a theoretical study," IEEE Transactions on Communications, Vol. COM-24, pp. 1036-1043, Sept. 1976.

30. V. Cerf, "Multiprocessors, semaphores, and a graph model of computation," Ph.D. dissertation, Computer Science Dept., University of California,

Los Angeles, CA, Apr. 1972.

31. K. Gostelow, "Flow of control, resource allocation and the proper termination of programs," Ph.D. dissertation, Computer Science Dept., University of California, Los Angeles, CA, Dec. 1971.

32. G. V. Bochmann, "Finite state description of communication protocols," Computer Networks, Vol. 2, pp. 361-372, Sept./Oct. 1978.

33. D. Bjorner, "Finite state automation - definition of data communication line control procedures," Fall Joint Computer Conference, AFIPS Conference Proceedings, Vol. 37, pp. 477-490, 1970.

34. G. V. Bochmann and C. A. Sunshine, "Formal methods in communication protocol design," IEEE Transactions on Communications, Vol. COM-28, pp. 624-631, Apr. 1980.

35. A. S. Danthine and J. Bremer, "Modeling and verification of end-to-end transport protocols," Computer Networks, Vol. 2, pp. 381-395, Oct. 1978.

36. "IEEE standard digital interface for programmable instrumentation," IEEE Std 488-1975, The Institute of Electrical and Electronics Engineers, Inc., Oct. 1975.

37. P. M. Merlin, "A methodology for the design and implementation of communication protocols," Report RC-5541, IBM T. J. Watson Res. Center, Yorktown Heights, NY, June 1975.

38. H. Rudin, C. H. West, and P. Zafiropulo, "Automated Protocol Validation: One Chain of Development," Computer Networks, Vol. 2, pp. 373-380, Sept./Oct. 1978.

39. C. H. West, "An automated technique of communications protocol validation," IEEE Transactions on Communications, Vol. COM-26, pp. 1271-1275, Aug. 1978.

40. D. Brand and W. H. Joyner, Jr., "Verification of protocols using symbolic execution," Computer Networks, Vol. 2, pp. 351-360, Oct. 1978.

41. D. Norton, "A process based general purpose system simulator," Internal Report, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, 1983.

42. G. K. Maki and D. H. Sawin, "Fault tolerant asynchronous sequential machines," IEEE Transactions on Computers, Vol. C-23, pp. 651-657, July 1974.

43. D. H. Sawin, III and G. K. Maki, "Fail-safe asynchronous sequential machines," IEEE Transactions on Computers, Vol. C-24, pp. 675-677, June 1975.

44. J. A. Teeter and G. K. Maki, "Multiple fault tolerant design of asynchronous sequential machines," in 1975 International Symposium on Fault-Tolerant Computing. Digest of Papers, Paris, France, 1-20 June 1975, IEEEpp. 257, , New York 1975.

45. R. C. H. Chen, "Bus communications systems," Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburg, PA, 1974.

46. K. J. Thurber, E. D. Jensen, L. A. Jack, L. L. Kinney, P. C. Patton, and L. C. Anderson, "A systematic approach to the design of digital bussing structures," AFIPS Conference Proceedings - Fall Joint Computer Conference 1972, pp. 719-740, 1972.

47. K. J. Thurber and G. M. Masson, "Bus structures," in Distributed-Processor Communication Architecture, Lexington Books, pp. 131-174, Lexington, Massachusetts 1979.

48. C. G. Bell, R. Cady, H. McFarland, B. Delagi, J. O'Laughlin, R. Noonan, and W. Wulf, "A new architecture for mini-computers - the DEC PDP-11," Proceedings SJCC, pp. 657-675, 1970.

49. "The PDP/11 peripherals handbook," Chapter 5, Digital Equipment Corporation, 1973.

50. D. K. Pradhan and S. M. Reddy, "Fault-tolerant asynchronous networks," IEEE Transactions on Computers, Vol. C-22, pp. 662-669, July 1973.

VITA

Leonard Howard Pollard was born July 21, 1947 in Logan, Utah. He graduated Cum Laude from Utah State University in 1971 with a B.S. in Electrical Engineering. He received his M.S. in Electrical Engineering in 1977, also from Utah State University.

While at Utah State University he won the statewide IEEE Student Paper Contest (1971) and was named Most Valuable Graduating Electrical Engineer (1971). He was also inducted into the Phi Kappa Phi and Sigma Tau honorary societies.

He was employed as a Research Engineer with Electro-Dynamics Laboratories at Utah State University until 1973, when he became an engineer of the systems group of Reticon, Inc. (Sunnyvale, CA). In 1975 he joined the Signal Processing Systems group of Lockheed Missiles and Space Company (Sunnyvale, CA), where he was a research engineer until 1980. At that time he began a Ph.D. program at the University of Illinois, where he has been a member of the Computer Systems Group of the Coordinated Science Laboratory.

Publications:

L. H. Pollard, "Multiprocessing with the TI 9900," Proc. Eleventh Annual Asilomar Conference on Circuits, Systems and Computers, Nov 1977.

L. H. Pollard, "A Multiprocessing Approach to Data Processing," TI-MIXER (Texas Instruments, Inc.) Vol 4, No 5, Nov 1977.

L. H. Pollard and J. H. Patel, "Correction of Errors in Data Transmissions Using Time Redundancy," Proc. Fault Tolerant Computing Symposium, Jun. 1983.